1988 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
Vol. I Architecture Briggs

PENN
STATE

INTERNATIONAL
CONFERENCE ON
PARALLEL
PROCESSING

PROCEEDINGS

OF THE

1988 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 15-19, 1988

Vol. I Architecture
Fayé A. Briggs, Editor

Sponsored by

THE PENNSYLVANIA STATE UNIVERSITY
1855

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

THE PENNSYLVANIA STATE UNIVERSITY PRESS
UNIVERSITY PARK AND LONDON

PROCEEDINGS

OF THE

1988 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 15-19, 1988

Vol. I Architecture

Fayé A. Briggs, Editor

Sponsored by

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

THE PENNSYLVANIA STATE UNIVERSITY PRESS
UNIVERSITY PARK AND LONDON

The papers appearing in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, Penn State Press, or the Institute of Electrical and Electronics Engineers, Inc.

Library of Congress Catalog Card Number 79-640377
ISSN 0190-3918
ISBN 0-271-00654-4
IEEE Computer Society Order Number 889
IEEE Catalog Number 88CH2625-2

Printed in the United States of America

Additional copies may be obtained from:
Penn State Press
215 Wagner Building
University Park, PA 16802

PREFACE

Interest in the field of parallel processing continues to climb. This trend is evidenced by the sharp increase in papers submitted to the International Conference on Parallel Processing during recent years:

Year	Papers Submitted	Papers Accepted	Percent
1980	170	65	57
1983	240	136	57
1986	400	170	43
1987	487	174	36
1988	590	173	29

Although the number of submissions continues to increase, the number of accepted papers this year and in the past two years has remained relatively unchanged. This is due to the limitation imposed by the fixed number of hours available for the conference. As a result, a record number of papers had to be rejected. This year, the conference proceedings is being published in three volumes according to the subject category. The breakdown of submissions and acceptances in the three main categories of this conference is as follows:

Category	Papers Submitted	Papers Accepted	Percent
Architecture	264	74	28
Software	144	43	30
Algorithms and Applications	182	56	31

Of the 173 papers that were accepted, 79 were accepted as regular papers and 94 were accepted as short papers. Many papers that normally would have been accepted as long papers were accepted as short papers in order to meet the maximum number of paper-sessions allotted for the conference.

Finding sufficient numbers of qualified reviewers to evaluate the record number of submissions this year was a particularly challenging task. Over 1,000 professionals in the field participated in this process. This year the process of selecting referees was simplified by the use of questionnaires, which were mailed to previous participants in the conference. The information on the completed questionnaires was entered into databases, which then allowed the conference chairmen to select reviewers qualified in fairly specialized fields. Even so, numerous papers were so highly specialized that custom selection of referees was still required. It appears that an even more detailed breakdown of specializations will be needed for these questionnaires in the future. Greater effort will also be required in the future to find additional reviewers to adequately evaluate the increasing numbers of submissions.

I am grateful to Sun Microsystems Inc., for the support and in particular, to Wayne Rosing (Vice President of Advanced Development) for giving me the opportunity to co-chair the ICPP88 program. I am very grateful to the reviewers for their timely and thorough evaluation, and the many other persons who assisted in the program effort this year. Many thanks are due to administrators, Janice Barnes and Marianne Witkop, in helping to make the proceedings a reality. In particular, I would like to express my appreciation to Alex Kwok, Michel Cekleov and Roland Lee who assisted in selecting referees and in handling the correspondence. Special thanks are due to Alex Kwok for developing user-friendly author and referee databases, and for automating the generation of correspondence for handling the papers. Finally, I wish to thank Prof. Tse-yun Feng for his guidance, support and encouragement in this effort.

Fayé A. Briggs
1988 Program Co-chair
Sun Microsystems, Inc.
Mountain View, CA 94043

LIST OF REFEREES

Aboelaze, M.	Purdue U.
Abonamah, A.	U. of Wisconsin
Abraham, S.	U. of Michigan
Abu-Sufah, W.	Virginia Polytechnic Inst.
Adams, S.	C.S. Draper Labs.
Adams, J.	The U. of Pittsburgh
Aggarwal, J.	U. of Texas at Austin
Agrawal, D.	North Carolina State U.
Alnuweiri, H.	
Ammar, H.	Clarkson U.
Anderson, R.	Lawrence Livermore National Lab
Archibald, J.	Brigham Young U.
Atwood, J.	Concordia U., CANADA
Azimi, M.	Michigan State U.
Baer, J.L.	U. of Washington
Bagherzadeh, N.	U. of Ca., Irvine
Bandyopahyay, S.	U. of Windsor, CANADA
Banerjee, U.	Control Data Corporation
Banerjee, P.	U. of Illinois
Barad, H.	Tulane U.
Barbour, A.	U. of Illinois, Chicago
Batcher, K.	Goodyear Aerospace Corp
Baumgartner, T.	A T&T Bell Laboratories
Baxter, B.	Intel
Bayoumi, M. A.	U. of Southwestern Louisiana
Beetem, J. F.	U. of Wisconsin, Madison
Bermond, J. C.	Simon Fraser U., CANADA
Bhavsar, V. C.	U. of New Brunswick
Bhuyan, L. N.	U. of Southwestern Louisiana
Bic, L.	U. of California, Irvine
Bodnar, B.	AT&T Bell Labs, Naperville
Bounds, P.	Allied/Signal Oceanics Div.
Brooks, E. D. III	Lawrence Livermore Nat. Lab
Brown, R. H.	Hartford Graduate Center
Brule, M.	Syracuse U.
Bucher, I. Y.	Los Alamos National Lab
Burns, J.	Georgia Inst. of Tech.
Butner, S. E.	U. of Santa Barbara
Calahan, D.A.	U. of Michigan, Ann Harbor
Cappello, P. R.	U. of California, Santa Barbara
Carlson, D. A.	Institute for Defense Analyses
Casavant, T. L.	Purdue U.
Cekleov, M.	Sun Microsystems, Inc
Chalasani, S. B.	U. of Southern Calif., LA
Chandna, A.	Case Western Reserve U.
Chang, Y.	Penn. State U.
Chao, P.	
Chen, M.S.	U. of Michigan, Ann Arbor
Chen, S. S.	U. of North Carolina
Chen, D. J.	The U. of Texas at Arlington
Cheng, K. H.	U. of Houston
Cherkassky, V.	U. of Minnesota
Chesley, G.	Sun Microsystems, Inc
Chiang, Y. P.	Washington State U.
Chiarull, D. M.	U. of Pittsburgh
Chin, C.Y.	General Electric Company
Chow, Y.C.	U. of Florida
Chow, E. T.	Jet Propulsion Lab
Chronopolos, A.	U. of Minnesota
Chuang, H. Y. H.	U. of Pittsburgh
Cok, R. S.	Eastman Kodak Co.
Coletti, N.	Institute for Defense Analyses
Daasch, R.	Portland State U.
Daghi, A.	Sun Microsystems, Inc.
Das, C. R.	Pennsylvania State U.
Davis, N. J.	Air Force Institute of Tech
Deering, M. F.	Schlumberger Palo Alto Res.
DeGroot, D.	Texas Instruments
Desai, B. C.	Concordia U., CANADA
Dickey, S.	Courant Inst. of Mathematical Sci.
Dubois, M.	U. of Southern California
Dunne, R. C.	Eaton Corporation, AIL Division
Emberson, D. R.	Sun Microsystems, Inc.
Eshaghian, M. M.	U. of Southern California
Fang, Z.	
Faroughi, N.	Cal. State Sacramento
Fellman, R. D.	
Fernandez, E. B.	Florida Atlantic U.
Foo, S. Y.P.	U. of South Carolina
Fortes, J. A.	Purdue U.
Fu, J.	U. of Illinois, Urbana
Gait, J.	Teltronix
Ganesan, S.	Oakland U.
Garcia, A. B.	Wright-Patterson AFB
Garner, R.	Sun Microsystems, Inc
Ghafoor, A.	Syracuse U.
Ghosal, D.	U. of Southwestern Louisiana
Ghosh, J.	
Ghozati, S.	Queens College
Goldstein, J. D.	The Analytic Sciences Corp.
Gooley, M.	U. of Illinois, Urbana
Guharoy, B.	
Gupta, R.	Philips Labs.
Hac, A.	AT&T Bell Labs, Naperville
Hai, A.	Bell Labs, Middleton
Hall, R. W.	U. of Pittsburgh
Hare, D.	Sun Microsystems, Inc.
Harper, D. T. III	The U. of Texas At Dallas
Hayes, A. H.	Los Alamos National Laboratory
Heath, J.R.	U. of Kent.
Ho, C.T.	Yale U.
Houle, J.L.	Ecole Poly. de Montreal, CANADA
Hsu, Y.	IBM Yorktown Heights
Hsu, W.	U. of Illinois
Hsu, W.J.	Michigan State U.
Humphreys, S. L.	Sandia National Laboratories
Hurson, A. R.	The Pennsylvania State U.
Hwang, F. K.	AT&T Bell Labs, Murray Hill
Hyde, D. C.	Bucknell U.
Iacoponi, M. J.	Harris Corporation

Ibrahim, H. A. H.	Columbia U.
Jager, W. J.	U. of Waterloo
Jain, R.	U. of S. Calif.
Jayakumar, R.	Concordia U., CANADA
Jayasimha, D. N.	U. of Illinois, Urbana
Jin, L.	Pennsylvania State U.
Jou, J. Y.	AT&T Bell Labs, Murray Hill
Juang, J.Y.	Northwestern U.
Kale, L. V.	U. of Illinios at Urbana-Champaign
Kermaani, K.	Sun Microsystems, Inc.
Kichul, K.	USC
Kim, S. M.	R.P.I.
Kim, M. H.	Michigan State U.
Kim, K.	Univ of S. Calif.
Kim, D.	U. of Southern California
Kim, D.W.	U. of Texas at Austin
Kimura, T. D.	Washington U.
King, C.T.	Michigan State U.
Konstantinidou, S.	U. of Washington
Kothari, S. C.	Iowa State U.
Kowalik, J. S.	Boeing
Krishnamurthy, B.	Tektronix Laboratories
Kumar, V. K. P.	U. of Southern California
Kung, S.Y.	Princeton U.
Kunkel, S. R.	IBM, Encott
Kuszmaui, B. C.	Thinking Machines/MIT
Kwok, A.	Sun Microsystems, Inc.
Ladkin, P. B.	Kestrel Institute
Lakhani, G.	Texas Tech U
Lakshmivaraham, S.	U. of Oklahoma
Lam, H.	U. of Florida
Lam, M.	Carnegie Mellon U.
Lan, Y.	IEEE Computer Society
Landis, D. L.	Penn State U.
Lang, T.	U. of California, Los Angeles
Larson, B. R.	Unisys Corporation
Lastra, A.	Duke U.
Lee, G.	U. of Southwestern Louisiana
Lee, C.	U. of California, Berkeley
Lee, C.	U. of Florida
Lee, D.	U. of Illinois
Lee, T.C.	U. of Tennessee
Lee, K.	Ohio State U.
Lee, D.L.	York U., CANADA
Lee, R.	Sun Microsystems, Inc.
Li, H. F.	Concordia U., CANADA
Li, Q.	Florida International U.
Lien, Y.N.	Ohio State U.
Lillevik, S. L.	Intel Corp.
Lin, T.C.	U. of Texas, Arlington
Lin, Y.B.	U. of Washington
Lin, T. C.	The U. of Texas at Arlington
Lin, T.Y.	Calif. State U., Northridge
Lin, W.	USC
Liou, D.M.	Illinois Inst. of Tech.
Little, R. R.	Clemson U.
Lopresti, D.	Brown U.
Loucks, W. M.	U. of Waterloo
Lougheed, R. M.	ERIM
Loui, M. C.	U. of Illinois, Urbana
Lovcks, W. M.	U. of Waterloo

Lubachevsky, B. O.	A T&T Bell Laboratories, Murray Hill
Lumpp, J. E.	Purdue U.
McElvany, M. C.	Allied Bendix
McGahee, K. L.	Rockwell International
McGuire, P.	Hewlett-Packard
McMillen, R.	Hughes Aircraft
McMillin, B. M.	Michigan State U.
Majumdar, A.	USC
Mak, V.	Bell Comunications Res.
Malony, A. D.	U of Illinois at Urbana-Champaign
Marquardt, D.	Sequent Computer Systems
Martin, A. J.	Caltech
Mathieson, I.	LaTrobe U., AUSTRALIA
Maurer, P. M.	Univ of Florida
Mayer, H. G.	
Mazina, M.	Rice U.
Mazumder, P.	U. of Michigan, Ann Arbor
Melhem, R.	U. of Pittsburgh
Meybodi, M. R.	Ohio U.
Midkiff, S. F.	Virginia Polytechnic Inst. & State U
Miller, R.	SUNY-Buffalo
Mittal, V.	Ohio State U.
Moreno, J. H.	U. of California, LA
Mossaad, K.	U. of Texas, Austin
Mudge, T.	U. of Michigan, Ann Arbor
Mukkamala, R.	Old Dominion U.
Murata, T.	U. of Illinois at Chicago
Najjar, W. A.	USC Inform. Sciences Inst.
Nakazawa, S.	MARC Anal. Res. Corp.
Nation, W. G.	Purdue U.
Neches, P. M.	Teradata Corporation
Nelson, V. P.	Auburn U.
Ni, L. M.	Michigan State U.
Nicol, D. M.	College of William & Mary
Nolan, J.	Department of Defense
O'Hallaron, D. R.	General Electric Co.
O'Keefe, M.	Purdue U.
Otto, S. W.	Caltech
Padmanabhan, K.	AT&T Bell Lab, Murray Hill
Pakzad, S.	Penn. State U.
Pargas, R. P.	Clemson U.
Patton, P. C.	Consortium for Supercomputer Res.
Peir, J.K.	IBM, Yorktown
Peterson, J.C.	Jet Propulsion Lab
Place, J.	U. of Missouri-Kansas City
Polychronopoulos, C.	U. of Illinois, Urbana
Poplawski, D. R.	Michigan Technological U.
Pramanik, S.	Michigan State U.
Prins, P. R.	Calvin College
Probst, D. K.	Concordia U., CANADA
Przytula, K. W.	Hughes Res. Labs
Putcha, K.	
Raghavendra, C. S.	U. of Southern California
Rajopadhye, S. V.	U. of Oregon
Ramachandran, U.	Georgia Institute of Technology
Ramkumar, B.	U. of Illinois
Rancourt, D. R.	
Rau, D.	
Ravi, S.S.	SUNY at Albany
Reed, D. A.	U. of Illinois, Urbana
Reeves, A. P.	U. of Illinois, Dept. of Comp. Sci.

Reinhardt, S.	Cray Res., Inc
Ribeiro, J.	Syracuse U.
Ricket, N. W.	Northern Illinios U.
Saad, Y.	U. of Illinois, Urbana
Salfi, R. E.	
Sanz, J. L. C.	IBM, San Jose
Sarma, D.	U. of Cincinnati
Scherson, I. D.	Princeton U.
Scheurich, C.	U. of Southern California
Schwederski, T.	Purdue U.
Sehr, D.	U. of Illinois, Urbana
Seidel, S. R.	Michigan Technological U.
Sengupta, A.	U. of South Carolina
Seow, C. H.	MIT Lab.
Serlin, O.	ITOM International Co.
Shaffer, P. L.	General Electric Co.
Shang, W.	Purdue U.
Sharma, R.	A T&T Bell Labs, Murray Hill
Shaw, W. H.	AFIT/ENG
Shea, D. G.	IBM Res., Yorktown
Shepard, T.	Royal Military College, CANADA
Sheth, A. P.	UNISYS West Coast Res. Cntr.
Shih, Y. L.	Ametek
Shin, K. G.	U. of Michigan, Ann Arbor
Shirazi, B.	Southern Methodist U.
Shu, R.	
Siegel, H.J.	Supercomputing Res. Center
Silberschatz, A.	U. of Texas Austin
Simmons, M. L.	Los Alamos National Lab.
Sinclair, J. B.	Rice U.
Singhal, M.	Ohio State U.
Slaney, M.	Schlumberger Palo Alto Res.
Smiarowski, A.	Tennessee Technological U.
Smith, J.	Astronautics Tech Center
Smith, S. P.	MCC
Smitley, D. L.	Supercomputing Res. Center
So, K.	IBM Res., Yorktown
Somani, A. K.	U. of Washington
Sterling, T. L.	Harris Corp.
Strik, C. W.	BDM Corp.
Stormon, C.	Syracuse U.
Subramanian, R.	AT&T Bell Lab, Columbus
Sung, Y.	Memphis State U.
Szymanski, T. H.	Columbia U.
Tai, H.M.	U. of Tulsa
Takefuji	U. of S. Carolina
Tang, J.H.	U. of Illinois
Tantawi, A. N.	IBM, Yorktown
Tao, L.	U. of Penn.
Tarbet, D.	
Testa, J.	Sun Microsystems, Inc.
Thakkar, J. D.	Sequent Computer Systems
Thomas, D. R.	Harris Corp.
Thomasian, A.	IBM, Yorktown
Trimble, G. M.	Lockheed Missiles & Space Co.
Tsai, W. T.	U. of Minnesota
Tseng, P.S.	Carnegie Mellon U.
Tsin, Y. H.	U. of Windsor, CANADA
Tzeng, N.F.	U. of Southwestern Louisiana
Varadarajan, R.	U. of Florida
Varma, A.	IBM, Yorktown
Varma, A.	IBM T. J Watson Res. Center
Vranesic, Z.G.	U. of Toronto
Wah, B. W.	U. of Illinois
Walicki, J.	Colorado State U.
Wallace, R. M.	AFWAL/AADE
Wang, Y. X.	Purdue U.
Wang, W.H.	U. of Washington
Warter, N.	U. of Illinois
Weems, C.	U. of Massachusetts
Widlicka, R.	New Mexico State U.
Wiltsie, W. F.	AT&T, Basking Ridge
Wing, O.	Columbia U.
Winsor, D. C.	U. of Michigan
Witten, M.	U. of Louisville
Wolf, J. J.	Colorado State U.
Wu, C.L.	U. of Texas Austin
Wu, K.L.	U. of Illinois, Urbana
Yalamanchili, S.	Honeywell Systems & Res. Cntr.
Yang, C.	Naval Postgraduate School
Yasrebi, M.	IBM Corp., Austin
Yew, P.C.	U. of Illinois, CSRD
Yoon, H.	Ohio State U.
Youn, H. Y.	U. of Mass.
Young, W.	AT&T Bell Labs, Holmdel
Young, H. C.	IBM Almaden Res. Center
Young, B. B.	Cray Res.
Yu, C.T.	USC
Van Zandt, J.	RCA
Zhang, C. N.	North Carolina A & T U.
Zhu, C.Q.	U. of Illinois, Urbana
Zipf, M. E.	U. of Pittsburgh
Zubair, M	Old Dominion U.

AUTHOR INDEX

TABLE OF CONTENTS

SESSION 1A: Distributed Systems

SESSION 2A: Systolic Arrays I

SESSION 3A: Logic Design and Tools

SESSION 3C: Meshes

SESSION 4A: Multiprocessor Issues

SESSION 5A: Multiprocessor Cache Coherence

SESSION 6A: Multiprocessor Performance

SESSION 7A: Networks

SESSION 13A: Fault Tolerance

SESSION 14A: Systolic Arrays II

SESSION 14B: Reliability

SESSION 15A: Logic Programming and Pipelined Systems

SESSION 15B: Reconfigurable Systems

PROCESSOR AND LINK ASSIGNMENT IN MULTICOMPUTERS USING SIMULATED ANNEALING

S. Wayne Bollinger and Scott F. Midkiff

Bradley Department of Electrical Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

ABSTRACT

In the design of multicomputer systems, the scheduling and mapping of a parallel algorithm onto a host architecture has a critical impact on overall system performance. In this paper we develop a graph-based solution to both aspects of the mapping problem using the simulated annealing optimization heuristic. A two phase mapping strategy is formulated: 1) *process annealing* assigns parallel processes to processing nodes, and 2) *connection annealing* schedules traffic connections on network data links so that interprocess communication conflicts are minimized. To evaluate the quality of generated mappings, cost functions suitable for simulated annealing are derived that accurately quantify communication overhead. Application examples are presented using the hypercube as a host architecture, with host graphs containing up to 512 nodes.

I. INTRODUCTION

Multicomputers are a form of parallel processing system composed of many processing elements (PE's), each with its own local memory. Individual PE's are connected to other PE's by point-to-point links that allow the bidirectional transfer of data. The cost of connecting every processor to every other processor is typically prohibitive, so links connect only selected processors, forming an interconnection topology such as a mesh, tree, or binary hypercube. Each processor executes a task or process. Local references by a process are efficient, since the PE contains local memory, but communication with processes executing on other PE's can significantly limit system throughput if data must be transferred over many links or if links are congested due to excessive traffic. To realize the full potential of a multicomputer's capabilities, it is essential that the distance between communicating processes be minimized and that link traffic is minimized to reduce delay.

The mapping problem maps an image architecture, a set of processes and their communication requirements, onto a multicomputer or host architecture. The problem consists of two components: 1) assignment of processes to processors, and 2) assignment or scheduling of interprocess communication traffic over network links. This paper presents a new approach to processor and link assignment in multicomputers based on the simulated annealing heuristic. The procedure has been implemented for binary hypercube host architectures. Results indicate that the technique produces good, and often optimal, mappings within reasonable computation times.

The paper first discusses the mapping problem, recent research, and communication overhead cost functions that can be used in an objective function. Simulated annealing is then applied to the mapping problem to find processor and link assignments that minimize the objective function for given host and image architectures. The final section presents results for mapping two image architectures, each containing up to 512 processes, onto a binary hypercube multicomputer.

II. THE MAPPING PROBLEM

The assignment and scheduling problem concerns the mapping of an arbitrary image architecture onto a general-purpose host or target architecture in a manner that minimizes communication conflicts among concurrent processes. For our purposes, the image architecture consists of a set of synchronous, static processes with communication requirements known prior to run time. The host architecture describes a point-to-point multiprocessor network with a fixed interconnection topology. To evaluate the quality of an assignment, an objective function is used to quantify the communication cost. The behavior of the mapping algorithm and the quality of generated assignments depend on the objective function chosen.

A. Host Architecture

The host architecture is represented by a **host graph** that describes the interconnection of processors in a multiprocessor network. The host graph is denoted by the undirected graph $G_H = <V_H, E_H>$ where V_H is a set of processors, and E_H is a set of edges describing the communication paths between processors. Every vertex in V_H corresponds to a distinct processing element, referred to as a **node**. Every edge $(n_1, n_2) \in E_H$ corresponds to a bidirectional data **link** between nodes n_1 and n_2. This implies that a single physical network link exists between each pair of directly connected processors. The host graph is assumed to be a connected graph, disallowing the possibility of isolated processor nodes. The following terminology is used when referring to the host architecture.

N The number of nodes in the host network, $N = |V_H|$.

This work was supported in part by a Digital Faculty Program/Incentives for Excellence grant from the Digital Equipment Corporation.

n_j A node j in the network, $1 \leq j \leq N$.

l_{jk} A link permitting communication from node n_j to node n_k. Note that l_{jk} and l_{kj} are equivalent.

t_{jk} The amount of communication traffic, in packets or units of traffic, flowing from node n_j to n_k.

$d(n_j, n_k)$ The distance between two nodes n_j and n_k, or the minimum number of links forming a path between the nodes. Several paths of length $d(n_j, n_k)$ may exist between the processes.

B. Image Architecture

The image architecture to be mapped onto the host network is represented by an **image graph** that describes the communication dependencies between concurrent processes. The image graph is a directed graph $G_I = < V_I, E_I, W_I >$. Every vertex in V_I corresponds to an individual **process**. Every edge $(p_1, p_2) \in E_I$ corresponds to a one-way data **connection** between processes p_1 and p_2. This does not impose any limitations on process communication, as a mutual data dependency may be represented as two opposing directed edges. The weight of an image edge $w_{12} \in W_I$ represents the expected traffic from p_1 to p_2. The following terminology is used when referring to the image architecture.

P The number of processes to be mapped, $P = |V_I|$.

p_j A process j in the image architecture, $1 \leq j \leq P$.

w_{jk} The communication requirement in packets or units of traffic between processes p_j and p_k.

c_{jk} The effective communication cost of a connection between processes p_j and p_k.

f_j $f_j \in V_H$ such that the mapping function $f: V_I \rightarrow V_H$ assigns process p_j to node f_j.

$d(p_j, p_k)$ The minimum number of links forming a path between the nodes which execute processes p_j and p_k, i.e. $d(f_j, f_k)$ Several paths of length $d(p_j, p_k)$ may exist between the processes.

An abbreviated form of the distance function d_{jk} is used where the meaning is apparent from context. The term d_{jk} may be interpreted as either $d(n_j, n_k)$ or $d(p_j, p_k)$ when appropriate. Communication weights are integer numbers, and traffic between two processes is indivisible; the traffic may not be split and routed along different network paths. Ordinarily, the number of processes P is equal to the number of available nodes N to maximize the use of processor resources. If P is less than N, however, $N - P$ dummy processes may be added to the image graph. To accommodate the possibility of isolated processes, the image graph G_I may be unconnected. The case $P > N$ poses load sharing problems which are not considered in this paper.

C. Prior Work

One evaluation criteria commonly implemented in optimization algorithms is the objective function found in the quadratic assignment problem [1]. Cast in terms of the mapping problem, the problem may be stated as follows. A set of P processes has associated with it a communication traffic intensity w_{jk} between each pair of processes p_j and p_k. A set of N processor nodes are configured with a distance or delay $d(n_p, n_q)$ between nodes n_p and n_q. Then the communication overhead between two processes j and k is the product of w_{jk} and $d(f_j, f_k)$, and the optimal mapping f minimizes

$$\sum_{j,k} w_{jk} \cdot d(f_j, f_k).$$

This objective function treats the communication traffic between every pair of processes as if it is independent of all other processes, which is only true if nodes communicate along dedicated network links. Thus it does not accurately characterize the high local traffic densities and communication bottlenecks that may arise among concurrent processes.

A mapping strategy using the cardinality of the mapping for the objective function was investigated by Bokhari [2]. Using cardinality as a measure of assignment quality, the objective is to maximize the number of pairs of communicating processes that fall on pairs of directly connected processors, thereby maximizing the number of image edges that map to host edges. This strategy fails to account for the significant effect that unmatched pairs of edges can have on the total communication overhead. Also, the algorithm assigns a uniform traffic intensity to all pairs of communicating processes, which limits its application. Bokhari states that a mapping algorithm using cardinality as an objective function exhibits behavior very similar to the quadratic assignment problem.

Bianchini and Shen [3]-[4] describe a method to automatically assign interprocessor communication in special purpose multiple processor systems, e.g. digital signal processing systems. The objective function used in the algorithm determines a communication cost based on the utility of network links, where utility is defined as the fraction of link capacity utilized by traffic. They do not consider the issue of process assignment; the traffic scheduler accepts a fixed placement of processes in the host architecture and then generates an optimal communication schedule for that particular assignment. This is acceptable when considering only dedicated heterogeneous architectures, where there may be little opportunity for optimizing the assignment of the image architecture to the processor nodes. For general-purpose homogeneous architectures, however, the assignment of processes has a substantial impact on the quality of the final traffic schedule and the overall system throughput.

To overcome the inadequacies of traditional objective functions, Lee and Aggarwal [5] formulate a set of new objective functions that accurately quantify communication overhead. Their functions measure the optimality of a mapping for general applications by considering the communication cost of all image edges along with the overall mode of communication, synchronous or asynchronous. This allows realistic evaluation of the network contention that occurs when concurrent processes compete for communication resources. Lee and Aggarwal also describe an efficient mapping strategy developed for the objective functions. While the mapping strategy addresses the problem of optimal process assignment, it utilizes a fixed routing scheme for traffic scheduling. Such a mapping scheme does not consider the possibility of exploiting the routing rules of a network to optimize the assignment of the image connections to network data paths.

D. Objective and Cost Functions

The objective function determines the performance characteristics of the mapping algorithm by specifying an appropriate optimization goal. To provide a realistic evaluation of the total communication overhead, the traffic intensity of each weighted image connection must be considered.

Cost Functions: The communication cost c_{jk} of an image connection between p_j and p_k is a function of the weight or traffic intensity of the corresponding edge in the image graph. If the connection is routed along dedicated network links, the communication cost $C1$ may represented as

$$C1 = c_{jk} = w_{jk} \cdot d(f_j, f_k).$$

In this case the cost of the connection is determined by the distance separating the nodes which p_j and p_k are mapped onto.

In general, a connection is established along network links that are shared by a number of different processes. Some links may be used by several processes, and communication along the connection will experience delays due to link sharing. The delay encountered at a network link is proportional to the total traffic intensity supported by that link. To quantify the effect of the overall delay on the cost of the connection, the delay at every link must be taken into account. Some additional definitions are needed.

L_i Link number i $(1 \le i \le d_{jk})$ in the connection between p_j and p_k under consideration.

D_i The amount of delay at link L_i.

$U_{st}(L_i)$ $U_{st}(L_i) = 1$ if the connection between processes p_s and p_t is routed along link L_i; $U_{st}(L_i) = 0$ otherwise.

Then the delay at each link in the connection is represented by

$$D_i = \sum_{s,\,t} w_{st} \cdot U_{st}(L_i),$$

and the cost of a connection between p_j and p_k by

$$C2 = c_{jk} = \sum_{i=1}^{d_{jk}} D_i.$$

If subscripts j and k are interpreted to mean the nodes n_j and n_k connected by link L_i, then the above expression for D_i is equivalent to t_{jk}, the traffic intensity of link l_{jk}. Therefore the delay or communication cost of a network link is proportional to the total traffic routed along the link.

Given a means to calculate the communication cost of each image connection, an objective function can be defined to determine the overall cost of a mapping. The following functions are adaptations of two of the four objective functions investigated in [5].

Objective Functions: A simple optimization criterion used in VLSI placement problems involves summing the costs associated with pairs of components to obtain an overall system cost. In the mapping problem, this corresponds to summing the communication cost between every pair of processes in the network. The total communication cost $F1$ can be written as

$$F1 = \sum_{j,\,k} c_{jk}.$$

Using this function with cost function $C1$ does give some measure of the quality of an assignment, but it ignores the conflicts due to link sharing by different image connections. Thus $F1$ should be combined with cost function $C2$ to form an objective function suitable for the mapping problem.

To more accurately describe the quantity being optimized in multiprocess communication, a second objective function $F2$ can be defined. When all processes in the network are synchronized, the image connection with the largest communication cost determines the overall performance. To characterize this behavior, $F2$ is defined as

$$F2 = \max_{j,\,k}(c_{jk}).$$

$F2$ may be used with either cost function $C1$ or $C2$. Minimizing either $F1$ or $F2$ does not necessarily minimize the other, so the objective function used must be chosen with care. The choice depends on the application under consideration as well as the mapping algorithm used.

III. ASSIGNMENT AND SCHEDULING USING SIMULATED ANNEALING

For large scale mapping problems, obtaining an exact optimal solution is not practical. Iterative improvement algorithms have been employed in the mapping problem with some success, however, they tend to produce solutions that are locally but not globally optimal. The simulated annealing method supplements iterative improvement by providing a mechanism to escape local optima and has been found to exhibit desirable solutions in combinatorial optimization problems similar to the mapping problem [6]. Existing mapping algorithms utilizing iterative improvement provide a basis for a new approach to the mapping problem that uses simulated annealing.

A. Partitioning the Problem for Simulated Annealing

In the mapping problem, both the assignment of processes to the host network and the scheduling of communication paths are critical to the overall system performance. To optimize the mapping of the image graph to the host graph, a two phase mapping strategy is required. The first phase is essentially a placement problem; it attempts to determine the best mapping of processes onto nodes without considering the details of traffic routing. The second phase is analogous to the wiring problem; it optimizes the decomposition of traffic connections onto network links, operating within the constraints imposed by the routing rules of the network. Both optimization phases may be implemented using simulated annealing.

The design of a good simulated annealing algorithm requires the specification of four elements: system configuration, annealing schedule, move set, and objective function [6]. By varying the elements, a single annealing algorithm can be extended to work with both optimization phases. In the following sections, we concentrate on the aspects of simulated annealing unique to the mapping problem. A general objective function suitable for annealing is formulated, and algorithm modifications specific to the assignment and scheduling phases are described.

B. Objective Function for Annealing

For effective annealing, the objective function should exhibit a wide range of values corresponding to the factors being optimized. Optimal configurations should have minimum cost, and inferior or physically unrealizable configurations should be penalized by high costs. Of the two objective functions previously defined, $F1$ produces a greater variation in cost, making it more desirable as a cost metric for simulated annealing. However, objective function $F2$ more accurately characterizes the quantity being optimized in the mapping problem. To satisfy these conflicting requirements, both $F1$ and $F2$ should be considered. Examining the form of $F1$ and $F2$ shows that there is negligible overhead incurred by keeping track of $F2$ as $F1$ is being calculated for a configuration. An assignment that minimizes $F1$ but increases $F2$ is not desirable, as $F2$ describes the actual limiting factor in synchronous multiprocess communication.

A new objective function is formulated to provide a single evaluation criterion by including both $F1$ and $F2$ as terms. Introducing a constant weight factor W, one possibility for such a function is

$$F = F1 + W \cdot F2,$$

where W penalizes any configuration that increases $F2$. The magnitude of W should be large enough so that the minimum variation in $W \cdot F2$ for a single move is greater than the maximum variation of $F1$. This ensures that a move increasing (decreasing) $F2$ will produce an increase (decrease) in the overall cost of a configuration. To achieve a similar effect, we consider both $F1$ and $F2$ during annealing by: 1) ignoring $F2$ during high temperature annealing when temporary increases in $F2$ and the objective function are to be expected, and 2) rejecting all moves generated during low temperature annealing that increase $F2$. This objective function used for annealing and defined as $F1$ combined with $F2$ will be referred to as the **standard objective function.**

C. Processor Assignment

The first mapping phase assigns image processes to host network processing nodes. The basic strategy of this phase is to assign processes with large mutual communication requirements to neighboring nodes in the host network. This phase does not consider detailed traffic routing. However, the spatial locality and communication requirements of processes must be considered simultaneously to generate an optimal mapping. Mapping phase one will be referred to as **process annealing**, and is characterized as follows:

1. **Move Set:** Moves are generated by pairwise exchanges of processes, Monte Carlo style. To evaluate the effect of a move on the objective function, the only process connections that need to be considered are those associated with the two swapped processes.

2. **Objective Function:** Since traffic routing is not considered during process assignment, cost function $C1$ must be used. Thus the cost of a connection is the product of the communication intensity and the distance between the nodes hosting the processes. The overall assignment cost is determined by $C1$ used with the standard objective function.

For process assignment the spatial location of nodes to which the process will be assigned are fixed by the physical structure of the host network. The only possible move is the pairwise exchange of two processes. However, one of the processes may be a dummy process inserted in the host network to account for excess processing nodes. This form of move corresponds to a process translation. For the special case $N > P$, this move gives the mapping algorithm an additional degree of freedom, enabling it to move processes among surplus nodes to determine the best distribution of processes. In the final stages of process annealing, the exchange of distant processes is unlikely to result in an improvement in the objective function, and only processes separated by small distances are considered for exchange. Limiting the range of attempted moves in this fashion maximizes the number of feasible moves attempted at each temperature stage.

D. Link Assignment

The second phase of mapping, referred to as **connection annealing**, schedules the interprocess communication onto a physical network topology. Given the fixed process mapping generated by process annealing, connection annealing determines an optimal assignment of image connections to host network data paths. This phase routes the connection between every pair of communicating processes onto a path of one or more network data links between the source and destination nodes. When communicating processes are assigned to directly connected nodes, the corresponding data path for the connection will consist of a single link. Otherwise, the connection must be routed along a series of data links. In general, a connection should be routed along the least possible number of links to minimize network path delay. However, an indirect route may be necessary to avoid heavily utilized data links if adding traffic to that link would exceed its capacity.

Unless all processes are assigned to directly connected nodes, corresponding to a perfect mapping, the possibility exists for link sharing among image connections. To avoid communication delays caused by the resulting link contention, the link assignment phase should route traffic along links supporting minimum traffic intensity whenever possible. By evenly distributing the traffic load among network links, the total network throughput can be maximized. Thus connection annealing must consider both path length and traffic intensity to determine an optimal link assignment. The annealing algorithm for link assignment incorporates the following elements:

1. **Move Set:** Path moves are more difficult to generate than the simple random pairwise exchanges in process annealing. A path is formed by starting at the source node, and then choosing links according to criteria based on both traffic intensity and remaining path distance. The link may be selected by fixed or adaptive means, depending on network routing rules. To evaluate the effect of a move on the communication cost of an assignment, the only data links that need to be considered are those affected by the connection being altered.

2. **Objective Function:** To characterize the interaction and contention between image connections that arise during traffic routing, cost function $C2$ must be used. Thus the cost of a connection is the sum of the traffic intensities supported by the network links assigned to the connection. The overall assignment cost is determined by $C2$ used with the standard objective function.

In process annealing the method used to generate moves is basically limited to the pairwise exchange of processes. For connection annealing, however, several possibilities exist for move generation. The method selected to rearrange a path depends on the routing flexibility allowed by the host network and the objective function used for scheduling. Given a source and destination node in the network and a criteria for selecting links, the path generator must establish a path if none is present, or produce a permutation of an existing path.

The scheme used by the host network for traffic scheduling limits the routing strategy used for link assignment. When the assignment of specific data links to a network path is predetermined by the routing rules of the network, path generation is limited to fixed path selection. If network routing rules allow for several possible paths between source and destination nodes, link assignments may be based on an adaptive selection criterion.

During path generation, the adaptive assignment approach considers the existing traffic conditions produced by previously routed image connections. Starting with the source node, there may be several feasible choices for an initial path link that reduce the remaining path distance. The adaptive criterion specifies that the link l_{jk} supporting the minimum traffic t_{jk} should be selected. If multiple links support the same minimum t_{jk}, then the next path link is selected from them randomly. This process is repeated, selecting minimum cost links until the path is completed. Due to its greedy nature, adaptive selection will not always produce a least cost path. In addition, there is no guarantee that a rearranged path will actually be different from the original path. Adaptive path selection is more efficient than random path generation, and is useful for both the initial link assignment and the connection annealing optimization phase.

E. Initial Assignment

For process and connection annealing, assignment of the initial system configuration can have a profound effect on both required run time and final mapping quality. Instead of relying on a random initial configuration, a procedure can be used to achieve a good initial assignment, and annealing can begin at a lower starting temperature to reduce run time. Using an initial assignment algorithm produces a system configuration that contains partially ordered domains. If the initial mapping is prepared carefully, the structure of these partially ordered regions corresponds to those existing in an optimal system configuration. Thus the amount of annealing required to locate a globally optimal mapping is greatly reduced. To preserve the advantages of a good initial mapping, the starting temperature must be chosen low enough to search the immediate state space without destroying the desirable features of the mapping. Starting too low, however, may cause the annealing algorithm to become trapped in an inferior local minimum.

Our experience indicates that using an initial assignment algorithm to generate process assignments for large image architectures can produce suboptimal configurations that are difficult to escape by low temperature annealing. Consequently we do not utilize the initial assignment algorithm for process annealing, and rely instead upon a random initial assignment with a complete temperature annealing run. The additional computation time is justified by the higher quality of final solutions obtained.

Unlike the case of initial process assignment, we found that the quality of traffic configurations produced by an initial link assignment algorithm coupled with a low temperature connection annealing run were comparable to those generated by a random initial assignment and full annealing. Connection assignments produced by such an algorithm are greatly superior to the unbalanced, chaotic network paths generated by a random initial connection assignment. The selection of the initial annealing temperature is crucial for an efficient interface between the initial link assignment and connection annealing algorithms.

IV. APPLICATIONS USING HYPERCUBE

In this section the performance of simulated annealing is demonstrated using two image architectures. The host network is implemented as a binary hypercube topology, a popular architecture for large scale multicomputers [7]. In the following discussion, D is the dimension of the hypercube, and L is the number of communication links in the hypercube, where $L = D \cdot 2^{D-1}$.

A. Hypercube as host architecture

In a sizable hypercube network incorporating hundreds or thousands of processor nodes, node assignment and communication scheduling are difficult problems. Process assignment is complicated by the ability to map a number of different, complex image topologies onto the hypercube. In addition, for every pair of communicating nodes separated by d_{jk} links, there are $d_{jk}!$ possible paths along which a connection can be routed. Determining a suitable mapping for a large hypercube network exercises the full power of the simulated annealing assignment algorithms.

For a general image graph, the communication overhead of the optimal mapping is unknown prior to assignment and scheduling. Due to the nondeterministic and heuristic nature of simulated annealing algorithms, there is no guarantee that the best mapping will be found. To accurately measure the performance of the mapping algorithm, an image graph with a known global minimum is used as a test case. The image graph chosen is similar in form to that of the hypercube host graph.

B. Hypercube Traffic Problem

The hypercube image graph is denoted by

$$G_{HI} = <V, E, W>,$$

$$(p_j, p_k) \in E \ \forall \ p_j, p_k \in V \mid d(p_j, p_k) = 1,$$

$$|E| = D \cdot 2^D,$$

$$w_{jk} = 1 \ \forall \ (p_j, p_k) \in E.$$

Here the distance function $d()$ is defined as for the hypercube, but is used instead with process indices. The connection structure of the graph corresponds to the hypercube graph, with weight 1 on each edge. For every communication link in the hypercube, there are two opposing directed edges in the hypercube image graph.

The performance of the mapping algorithm is evaluated by mapping random permutations of the hypercube image graph onto the hypercube host. The optimal solution is known in this case, so the relative quality of assignments generated by the algorithm can be determined. In the ideal mapping, every network link supports two connections with weight 1, so

$$F1_{IDEAL} = \sum_{j,k} C2_{jk},$$

$$= D \cdot 2^{D+1},$$

$$F2_{IDEAL} = 2.$$

If the algorithm succeeds in finding the optimal assignment for this graph, all pairs of communicating processes fall

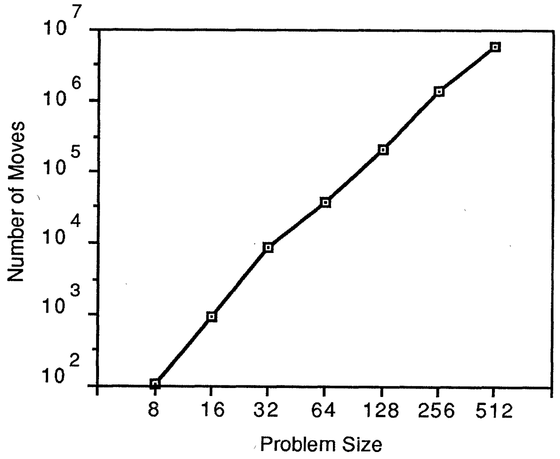


Fig. 1. Total Generated Moves vs. Problem Size

on nearest neighbor connected nodes in the hypercube, and no traffic scheduling is needed. Thus the first phase of annealing, processor assignment, is critical for this image architecture. For an optimal assignment, the actual communication overhead equals the ideal communication overhead. The mapping algorithm was run on hypercube traffic graphs containing from 8 to 512 processes. The results are tabulated in Table 1, and represent average values obtained using random initial assignments. The total moves column gives the total number of moves generated during all stages of annealing. Figure 1 demonstrates the relationship between computational effort and problem size.

By adjusting values for the initial temperature and rate of annealing according to the problem size, we were ɐble to converge into optimal solutions consistently for $N \leq 128$. The initial temperature is taken high enough so that the ratio of accepted moves to total proposed moves exceeds 0.9, ensuring that the majority of generated moves are accepted. Large problem sizes require a slower cooling rate to investigate a greater portion of the problem search space. This increases the probability of finding an optimum solution, at the expense of increased execution time.

C. Tree Traffic Problem

Tree image graphs are frequently encountered in parallel processing applications. The graph considered here has the form of a binary tree, described by

$$G_{TI} = <V, E, W>,$$

$$|E| = 2 \cdot (N-1),$$

$$w_{jk} = 1 \ \forall \ p_j, p_k \in V \mid (p_j, p_k) \in G_{TREE}.$$

An extra process is added to the graph and connected to the root of a standard binary tree so that N is a power of 2. There is no known closed form solution for the minimum communication overhead for a mapping of the tree graph onto a hypercube, so $F1_{IDEAL}$ and $F2_{IDEAL}$ are unknown.

To map the tree graph onto a hypercube, both process and connection annealing were used. Each phase of annealing is effective in reducing the overall communication cost. In all cases a random initial configuration was used, and the parameters for the annealing schedule were chosen to provide a good balance between mapping optimality and algorithm computation requirements. Table 2 shows the results for tree sizes containing 8 to 512 nodes. The tabulated values for total moves reflect the sum of process moves generated during process annealing, and path moves generated during connection annealing. Mapping results show that the hypercube network provides excellent support for the communication requirements of the tree image graph.

V. CONCLUSIONS

A graph-based scheme utilizing the simulated annealing optimization heuristic has been developed for the automated mapping of an arbitrary image graph onto a general-purpose multiprocessor architecture. The complete procedure employs two annealing-based optimization phases. Process annealing attempts to assign processes that exhibit high mutual communication requirements to neighboring nodes in the host network. Connection annealing incorporates an initial assignment procedure, and further reduces communication costs by performing traffic routing of data paths. A communication cost function is formulated that captures the effect of transmission delays and bottlenecks arising as processes compete for communication resources.

The simulated annealing technique is easily extended to generate mappings for a large class of host and image architectures. The underlying annealing procedure is completely general, and makes no assumptions about the interconnection structure of the host or image architectures. Depending on the application, varying parameters such as the class of moves generated, the cost functions, and the annealing schedule enable the behavior of the mapping algorithm to be modified for maximum performance. As currently implemented, the procedure uses a binary hypercube topology as the host architecture. The mapping scheme has been evaluated using a variety of image graphs. We were able to anneal into optimal solutions for $N \leq 128$, and near-optimal solutions for larger image architectures. Our results show that the strategy scales well for large problem sizes, obtaining good results with computational effort proportional to small powers of N.

REFERENCES

[1] M. Hanan and J. M. Kurtzberg, "A review of the placement and quadratic assignment problems," *SIAM Rev.*, vol. 14, pp. 324-342, Apr. 1972.

[2] S. H. Bokhari, "On the mapping problem," *IEEE Trans. Comput.*, vol. C-30, pp. 207-214, Mar. 1981.

[3] R. P. Bianchini, Jr. and J. P Shen, "Interprocessor traffic scheduling algorithm for multiple-processor networks," *IEEE Trans. Comput.*, vol. C-36, pp. 396-409, Apr. 1987.

[4] R. P. Bianchini, Jr. and J. P. Shen, "Automated compilation of interprocessor communication for multiple processor systems," in *Proc. IEEE Int. Conf. Comput. Des.*, Oct. 1986, pp. 262-268.

[5] S.-Y. Lee and J. K. Aggarwal, "A mapping strategy for parallel processing," *IEEE Trans. Comput.*, vol. C-36, pp. 433-442, Apr. 1987.

[6] S. Kirkpatrick, C. D. Gelatt, Jr. and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, May 1983, pp. 671-680.

[7] C. L. Seitz, "The Cosmic Cube," *CACM*, vol. 28, no. 1, pp. 22-33, Jan. 1985.

Table 1. Results for Hypercube Traffic Problem

N	Total Moves	Initial Comm. Overhead		Final Comm. Overhead		Ideal Comm. Overhead	
		F1	F2	F1	F2	$F1_I$	$F2_I$
8	110	272	14	48	2	48	2
16	940	756	21	128	2	128	2
32	9,000	2,384	28	320	2	320	2
64	37,500	8,400	49	768	2	768	2
128	212,200	26,060	64	1,792	2	1,792	2
256	1,415,600	72,780	82	5,004	12	4,096	2
512	6,536,100	203,350	101	13,912	20	9,216	2

Table 2. Results for Tree Traffic Problem

N	Total Moves	Initial Comm. Overhead		Final Comm. Overhead		Ideal Comm.* Overhead	
		F1	F2	F1	F2	$F1_I$	$F2_I$
8	35	92	10	32	4	32	4
16	250	248	15	72	4	64	4
32	7,600	558	20	144	4	128	4
64	49,800	1,340	22	272	4	256	4
128	280,300	2,752	24	546	4	512	4
256	1,552,400	6,182	30	1100	4	1024	4
512	6,802,000	12,974	32	2144	4	2048	4

*Lower bound.

COMPARING THE PERFORMANCE OF

TWO DYNAMIC LOAD DISTRIBUTION METHODS

L.V. Kalé[1]

Department of Computer Science

University of Illinois at Urbana–Champaign

1304 W. Springfield Ave., Urbana, IL–61801

Abstract — Parallel processing of symbolic computations on a message–passing multi–processor presents one challenge: To effectively utilize the available processors, the load must be distributed uniformly to all the processors. However, the structure of these computations cannot be predicted in advance. So, static scheduling methods are not applicable. In this paper, we compare the performance of two dynamic, distributed load balancing methods for small–grained tasks on large parallel machines.

1. Introduction

Processor utilization is a key factor that decides the speedup provided by a parallel system. A thousand processor system can provide a speedup of 1000 only *if all the processors can be kept busy all the time*. Ideally, the computation should be divided in P equal parts (where P is the number of processors), one for each processor. But, it is usually impossible to identify 'P equal parts' except for highly structured computations. An alternative is to divide the computation into many small granules. Then, even if these granules are of unequal sizes, their large number would allow us to distribute them equally. Many parallel evaluation schemes for functional programs, logic programs, problem–solving, searching etc., offer such a small grain of parallelism.

The large pool of tasks may lead to a increased speedup only if there is an effective load distribution scheme, one that ensures that no processors remain idle while there is work available in the system. This is particularly true on a message–passing multiprocessor.

What sort of load balancing system is needed for a message passing system? The unpredictability of computation structures implies that it must be a *dynamic* or run–time strategy, as opposed to a static or compile–time strategy. For scalability, it must not be centralized at a few PEs, but *distributed* on all of them. Also, it should not depend on global information. Each PE should only use the information provided by its neighbors.

There has been a substantial amount of research on the problem of load balancing and load distribution [1, 2, 8]. However, most of it has been in the context of either large–grain tasks, or a relatively small number of processors, or in the context of real–time tasks. Much work has been done for static load balancing[2], where the task–to–processor mapping is decided ahead of run–time. There has been very little work on dynamic load balancing for fine–grained parallel tasks running on a large number of (100s to tens of thousands) parallel processors.

In this paper, we compare the performance of two such load balancing schemes. One of them is '*contracting within a neighborhood*' (CWN), a relatively simple strategy proposed by us [3]. The other is the *Gradient Model* (GM) proposed by Lin and Keller [6].

2. The Competitors

The small grain tasks found in most application domains have some interesting features in common. When activated, they execute for a short time, and then either complete, or start some sub–tasks and awaits response from them. The same cycle is repeated on receiving a response. Usually, it is prohibitively expensive to move a task from a PE to another after it has spawned sub–tasks. Both the strategies we describe avoid that. They do differ as to when a task is distributed: CWN schedules a task on some PE as soon as it is created; the GM keeps the newly created tasks on the source PE, and distributes them when required.

2.1. Contracting Within Neighborhood

This scheme is based on the fact that allowing communication between arbitrary pairs of PEs is not scalable. In a system with global communication, as the number of PEs is increased, a point is reached beyond which the system is always communication bound. This is true for any interconnection scheme which uses a fixed number of connections per PE [4]. It is possible to avoid global communication in tree structured computations as the communication is almost exclusively between parent and child tasks. So CWN restricts a child task to be within a fixed radius – *neighborhood* – from its parent. Also, in the interest of agility, CWN sends every subgoal out to another PE as soon as it is created.

This research was supported in part by the National Science Foundation under grant number CCR–87–00988.

[2] In some of the large–grain load balancing literature, a distinction is made between the terms *load balancing* and *load distribution*. There the former term refers to initial distribution of work, whereas the latter refers to what we call redistribution of work. On fine–grained systems, the tasks are being created throughout the life cycle of the a computation with almost equal rate. We use the term load balancing to refer to the general problem of maintaining adequate levels of load on all processors.

Each PE maintains the load information about its immediate neighbors. This information can be a combination of various factors that gauge the current and future 'load' on that PE. A simple measure is the number of messages waiting to be processed by that PE. This information is maintained by broadcasting a very short message to all the neighbors periodically, or as an optimization, piggy–backing the load information 'word' with regular messages. Any time a subgoal is created on a PE it sends a new goal message to its least loaded neighbor. The message also includes a count field that says how many hops the message has traveled from the source. A PE that receives such a message keeps the goal for processing if the hop count is equal to the allowed *radius.* Otherwise it sends the goal to its least loaded neighbor after incrementing the count. If a PE finds its own load is less than its least loaded neighbors, it keeps the goal provided the message has already traveled a stipulated *minimum hops.* Thus, a new subgoal travels along the steepest load gradient to a local minimum. A goal, once it is accepted by a PE, remains there, and is finally executed by that PE.

As it follows the local load gradients, this scheme may not send a given subgoal to the least loaded PE in the neighborhood, because of the horizon affect. However, looking for the least loaded PE in the neighborhood would be expensive. The minimum hops are stipulated to alleviate this problem to some extent. A source PE cannot keep a piece of work even if it is the least loaded among its neighbors. It must send it some distance to 'look over the horizon', and then possibly get it back.

The scheme is naive on several counts. First, requiring every piece of work to be contracted out to another PE seems excessive. Also, once a goal reaches its 'destination' it remains stuck there, which removes opportunities for a correction as time goes on. However, the strategy is meant as a starting point. The simulation studies should suggest specific ways of improvement.

The scheme has two parameters: the *radius*, i.e. the maximum distance a goal message is allowed to travel, and the *horizon*, i.e. the minimum distance a goal message is required to travel.

2.2. The Gradient Model

The gradient model is a more elaborate scheme than CWN. A newly generated subgoal is simply entered in the local queue. A separate, asynchronous process handles the load–balancing functions. This process wakes up periodically, and computes the load on the PE as in CWN. Using two parameters, the *low–water–mark* and *high–water–mark*, it decides the *state* of the node as follows. If the load is below the *low–water–mark*, the state is *idle.* If the load is above the *high–water–mark*, the state is *abundant*; Otherwise, it is *neutral.* It then computes its *proximity*: The proximity of an idle node is 0. For others, it is one more than the smallest proximity of their immediate neighbors. All the PEs initially assume that the proximities of their neighbors are 0. If the calculated proximity is more than network diameter, then it is set to (network diameter +1), to avoid unbounded increase in proximity values. If the proximity is different from its previous value, it is broadcast to all the neighbors. If the state were *idle* or *neutral*, the process sleeps until the next interval. If the state were *abundant*, it sends a goal message from the local queue to the neighbor with least proximity. The neighbor just adds the message to its queue. This may change its state which is noticed when the gradient process on that PE wakes up.

The proximity of a PE represents a guess at the shortest distance to an idle PE. It is a 'guess' because by the time the information about an idle PE reaches another PE via the update–and–broadcast–proximity sequence, the state of some PEs may have changed.

The rationale behind the GM is to keep work locally as far as possible, and to send work out towards a PE that is in danger of being idle. This strategy is parameterized by: the *low–water–mark*, the *high–water–mark*, and the sleeping interval between two execution cycles of the gradient process.

3. The simulation set–up

The simulations were carried out on ORACLE, a multi–processor simulation system we are developing. ORACLE is written in SIMSCRIPT, which supports *process* abstraction. ORACLE has one process for each user process running on a PE, and one process for each communication channel. Thus it models contention for the basic resources of a parallel system.

ORACLE accepts input specifications such as the number of PEs and their interconnection scheme, the load balancing strategy to be used (from its repertoire of strategies), control strategy options, form and kind of output information required, a program to execute and times to be charged for primitive operations. ORACLE can provide statistics on a variety of performance aspects such as the overall average PE utilization, average utilization of individual PEs, average and individual utilizations of communication channels, and the time to completion.

A point worth noting is that when we run a program on ORACLE, we get the result of the program, in addition to the performance statistics. In contrast, a trace driven simulation approach would be to carry out the computation in advance, producing a trace, which will then be used by the simulation system to get the performance figures. We found such an approach would not save much in terms of simulation time. Another approach could be to use a statistical model of computation. In absence of any uniform model of parallel computations, it was thought to be too unreliable and adhoc an option. So we opted for executing specific computations with well-understood structures.

The sample points at which to compare the two schemes vary on many dimensions: the interconnection topologies, the number of PEs, the computation structure and size, and the communication to computation ratio.

We selected 2 interconnection topologies: the 2-dimensional grid (nearest neighbor grid) with wrap-around connections and the double-lattice-mesh (DLM) topologies. The grid was used in simulations of the gradient model by Lin [7]. The DLM is a bus-based topology proposed by us [4]. We also decided to simulate systems with 25 to 400 PEs. Beyond 400 PEs, the time required for simulations was prohibitive. This range should be sufficient to understand how the schemes will behave when the size of the system changes.

To be able to interpret the simulation results, and get an understanding of how the load balancing schemes behave, we needed a predictable computation, whose structure is easy to grasp. Then, there won't be ambiguities about whether a certain feature that is seen in the simulation data is due to the nature of the computation or due to the load-balancing scheme. We chose to use *divide-and-conquer*, and *naive-fibonacci* programs for these reasons. The *divide-and-conquer* (abbreviated *dc*) program was used by Lin, and may be written as:

```
dc(M,N) ← if M = N then M
                   else dc(M,(M+N)/2) + dc(1 + (M+N)/2, N)
```

The *naive-fibonacci* is the doubly recursive function to compute fibonacci numbers.

```
fib(M) ← if M < 2 then M else fib(M-1) + fib(M-2)
```

It must be pointed out that we are not really interested in how to compute this functions in parallel. There are much more efficient methods for computing them.

We used 6 different computation sizes for each program. Fibonacci of 7, 9, 11, 13, 15 and 18, and the *dc* computations of the same sizes, namely: dc(1,n) for n=21, 55, 144, 377, 987 and 4181. As we wanted to focus on effectiveness of load distribution, we decided to isolate the factor of communication load. We chose the ratio of communication to computation to be such that communication stagnation does not occur.

3.1. The optimization experiments

Each scheme has a few parameters that have to be selected. In the interest of fairness, the parameters must be chosen in such a way each scheme is working at its best. We chose a few sample points in the space of planned experiments, and ran the simulations for various combination of parameters. The winning combinations were used for the comparison experiments. The parameters so chosen are shown in the table below.

It is worth noting that the 20 units interval is fairly low, as the total execution time for simulations ranged from 1000 to 23000 units. That means the gradient process is running very frequently, which should be an asset to its performance. Also, we assume a communication co-processor to handle the routing and load-balancing functions (for both strategies). Without such a co-processor, the gradient model will suffer more, because it executes a more complex code and more frequently.

parameter	grids	lattice-meshes
CWN: *radius/horizon*	9/2	5/1
GM: high/low-water-mark	2/1	1/1
GM: sleeping interval:	20 units	20 units

Table 1: Selected Parameters

4. Simulation Results, and Interpretation

The choices of sample points mentioned above lead to 240 simulation runs (2 problem types * 6 problem sizes * 2 topology types * 5 topology sizes * 2 strategies). The simulations were run on a VAX Each run took between 15 minutes to 3 hours of time on a Vax-750.

Plots 1 through 6 show the performance of the two schemes on the divide-and-conquer computations. (See [5] for the complete set of plots, including simulations for hypercubes). Each plot depicts experiments done on a specific topology, for one problem type. Thus Plot 1 shows the results of 6 *dc* computations of varying sizes, running on a double-lattice-mesh with 400 (20x20) PEs. The Y-axis shows the average PE utilization in percents. The X-axis is the problem-size in total number of goals generated during the computation. The speedup can be computed by multiplying the number of PEs by (average utilization percentage/100).

On the grid topologies, the CWN is a clear winner by substantial margins. On the double lattice-meshes also CWN consistently performs better than the GM. The only one case seen in these plots where CWN is outperformed by the GM occurs in plot 2, while running *dc(1,4181)* on a DLM with 100 PEs.

The comparative figures from all the runs are shown in table 2. For each run, we show the ratio of speed-ups obtained using CWN to that obtained using

GM. In 118 out of 120 cases, the CWN is seen to be better. In 110 of those cases, the difference is significant, i.e. more than 10%. On grids at times the CWN leads to thrice as much speed as (i.e. the response time) GM.

The DLM topologies have smaller diameters (4–5) compared to the grids (ranges from 8 to 38). The superior performance of CWN on the grids leads us to conjecture that it performs better than the GM on large systems, which of course tend to have larger diameters.

To understand the operation of each method, we plot the utilizations during short sampling intervals throughout the course of computation, for a few selected computations. Plots 7 through 9 show the utilization as time varies for 3 Fibonacci computations on both topologies with 100 PE. The CWN has much faster 'rise–time' than GM: it spreads work quickly to all the PEs at beginning. The pitfalls of CWN are also seen, e.g. in Plot 7 and 8. Although it takes the system close to 100% utilization quickly, it cannot maintain the performance at that level. The Gradient model manages to maintain 100% when it reaches that level (plot 7). This is because of GM's ability to re–distribute work. For CWN, once a goal is sent to a PE, it must be executed there, although the load conditions may change after that. It can correct such imbalances only by using newly created goals, which limits its ability to supply work to idle processors.

The main problem with GM is that it is not agile enough. PEs hoard work until they are sure they are 'abundant'. On the grids, a stronger flattening is seen (plot 9). When about 40% of the PEs have received work, most PEs think there is not sufficient work to distribute it to others, and so keep the new goals they generate, which leads to loss of parallelism, and as a result not enough work gets generated. This 'vicious cycle' is responsible for the flattening of the plot.

Examination of the detailed simulation output, not shown here, reveals another potential problem with CWN. Typically, it requires thrice as much communication as the GM. In GM, the average distance traveled by a goal message is typically less than 1. A significant number of goals just stay at the PE they were created on. On the grids, with CWN the distance traveled is about 3. For example, in computing fib(18) on a 10x10 grid, the average distance was 3.15 for CWN and 0.92 for GM.

5. Conclusions and Future Work

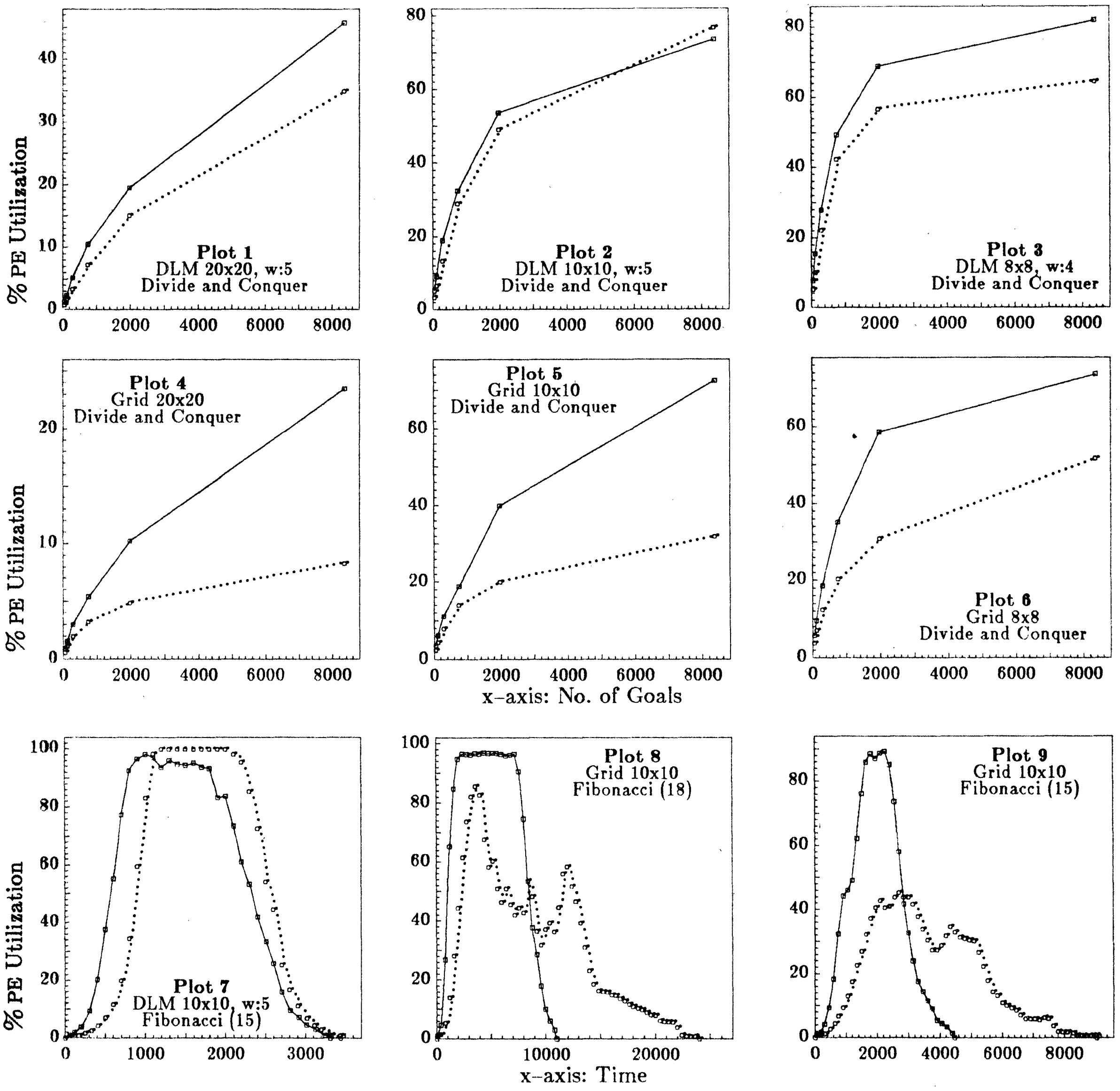
Although CWN performs better than GM in most experiments reported here, it still has a large room for improvement. First, CWN does not allow a goal to be re–distributed once it has been sent to another PE. As seen in Plots 7 and 8, the available work is just sufficient to keep every PE busy, but as the CWN cannot re–shuffle work, some PEs remain idle. However, re–shuffling is not useful when the work is more than sufficient or when it is too little. So, a small, well–controlled (i.e. responsive to run–time conditions) *re–distribution* component should be added to CWN. Also, the larger communication distances indicate that CWN needs ***saturation control***: When the system is running at 100% utilization, there is no need to send every goal out to other PEs. Detecting such a situation and then keeping goals locally until the situation changes would be worth investigating. Both of these amount to incorporating the good features of GM in CWN. Care must be taken not to lose the agility of CWN while modifying it.

A note of caution is in order. We chose a low communication to computation ratio to ensure that communication stagnation does not interfere with the property we were trying to measure: namely, the ability to distribute computation load effectively. When the ratio is higher, CWN, as it is, may lose some of its edge. Techniques of the last paragraph will then be *necessary.*

Acknowledgement: I am grateful to Michael Carroll, Valerie Rasmussen, and Wennie Shu for their help with the simulations.

6. References

1. D. L. Eager, E. D. Lazowska and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Transactions on Software Eng.*, **SE–12**, 5 (May 1986), 662–674.
2. D. L. Eager, E. D. Lazowska and J. Zahorjan, "A Comparison of Reciever–Initiated and Sender–Initiated Adaptive Load Sharing", *Performance Evaluation*, **6**, 1 (March 1986), 53–68.
3. L. V. Kale, "Parallel Architectures for Problem Solving", Doctoral Thesis, Dept. of Computer Science, SUNY, Stony Brook, NY–11794., December 1985.
4. L. V. Kale, "Optimal Communication neighborhoods", Proc. of ICPP, St. Charles, Illinois, August 1986.
5. L. V. Kale, "Comparing the Performance of Two Dynamic Load Distribution Methods", Tech. Report No. UIUCDCS–R–87–1387, September 1987.
6. R. Keller and F. C. H. Lin, "Simulated Performance of a Reduction Based Multiprocessor", *Computer*, **17**, 7 (July 1984), .
7. F. C. H. Lin, "Load Balancing and fault tolerance in applicative systems", Doctoral Thesis, Dept. of Computer Science, Univ. of Utah, August 1985.
8. J. A. Stankovic, "Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms", *Computer Networks*, **8**, 3 (June 1984), 199–217.



-□- Nbrhood Contracting
·○·· Gradient Model

	Grids					Double Lattice Meshes				
PEs	25	64	100	256	400	25	64	100	256	400
fib(7)	1.56	1.57	1.44	1.57	1.57	1.30	1.18	1.24	1.18	1.23
fib(9)	1.56	1.53	1.30	1.56	1.56	1.06	1.14	1.33	1.14	1.21
fib(11)	1.56	1.56	1.79	1.92	1.92	1.09	1.12	1.06	1.11	1.16
fib(13)	1.60	1.92	1.83	1.71	1.71	1.09	1.08	1.09	1.04	1.10
fib(15)	1.58	2.14	2.03	2.56	2.56	1.21	1.14	1.04	1.05	1.04
fib(18)	1.74	1.72	2.18	3.03	3.09	1.24	1.20	0.87	1.09	1.08
dc(1,21)	1.46	1.47	1.44	1.47	1.47	1.41	1.46	1.51	1.46	1.51
dc(1,55)	1.37	1.33	1.37	1.33	1.33	1.17	1.51	1.35	1.51	1.38
dc(1,144)	1.39	1.48	1.38	1.48	1.48	1.25	1.25	1.40	1.32	1.52
dc(1,377)	1.28	1.72	1.34	1.65	1.65	1.17	1.16	1.11	1.12	1.44
dc(1,987)	1.38	1.89	1.98	2.09	2.09	1.17	1.21	1.09	1.06	1.29
dc(1,4181)	1.36	1.42	2.27	2.91	2.82	1.30	1.27	0.96	1.18	1.31

Speedup of CWN over GM
Table II

AN APPROXIMATE LOAD BALANCING MODEL WITH RESOURCE MIGRATION IN DISTRIBUTED SYSTEMS

Ravi Varadarajan
Computer and Information Sciences Department
University of Florida, Gainesville, FL 32611

Eva Ma
Department of Computer and Information Science
University of Pennyslvania,Philadelphia, PA 19104

Abstract — Resource migration in a distributed computer system can be performed for performance enhancement as well as for reliability or availability improvement. The intractability of the general load balancing model with both job and resource migration suggests obtaining approximate solutions. The existing approach is to use heuristic rules to find approximate solutions. In this paper, we adopt an alternative approach of separating the job and resource migration problems and propose an approximate model (commodity distribution) for resource migration which can be solved by a polynomial-time algorithm. We demonstrate the application of this model to two load balancing problems: file migration in distributed databases and host migration in mobile computer networks. We also outline our efficient algorithm for solving a special case of this model.

Introduction

A resource in a computer system is defined as any hardware or software entity required for the execution of a user job. Examples of resources include processors, memories, interconnection networks, system processes, data files, database relations and file servers. In a distributed computer system, some of these resources are distributed among the various nodes in the system. If the distribution of a resource among the nodes can vary with time, then we call this resource a 'migratable' resource. Examples of such resources include datafiles, processes and mobile hosts. Traditionally, the term 'load balancing' refers to the operation of distributing or redistributing the user tasks among the different nodes in a distributed system, to achieve a desirable performance level; typical performance measures include job response time, throughput and processor utilization. We extend this definition of load balancing to include the operation of distributing or redistributing the *migratable* resources of a computer system to achieve a desirable performance level. Redistribution of the user jobs among the nodes in the system is known as *job migration*. We call the redistribution of migratable resources as *resource migration*.

Load balancing models without resource migration have been extensively studied in the literature (e.g. [3], [5]). We give a few examples of applications where resource migration is also used for load balancing. In a distributed database system, file migration is performed in order to maintain at all times, a desirable relation between the file access rates and the distribution of file copies among the nodes. Another example of resource migration in a distributed computer system can occur when a job in one host needs the services of a system process such as a file server, query processing program and editor process, running on a remote host. Here, instead of sending the request to the remote host and transferring the results back, the required process itself can be migrated from the remote host. An example in which a processor itself can migrate is a mobile computer network which consists of mobile hosts and in which the topology can change from time to time.

An important issue in load balancing with both job and resource migration is the problem of deciding which jobs or resource units to migrate. We refer to this problem the general load balancing problem with resource migration. In this problem, it is necessary to find a proper distribution of resources and jobs among the various nodes of the system so that the *desired* trade-off occurs between the *job migration cost* and the *resource migration cost*. In one formulation of this problem, the total migration cost (of jobs and resources) is minimized. This optimization problem had been shown to be NP-hard ([2]). The total cost criterion is useful when the resources and the jobs have to be migrated one at a time as for example, when a single broadcast bus such as Ethernet is used for migrating the resources and the jobs among the nodes. The bottleneck cost criterion is more appropriate when the resources and the jobs can be migrated in parallel. The load balancing problem with the bottleneck cost criterion can also be shown to be NP-hard. We omit the problem formulation and the proof of its intractability here (see [7]).

The exact solutions for the load balancing problem need either exhaustive or heuristic search procedures, all of which are prohibitively expensive to be executed in real time. As a result, approximate solutions are usually used instead for the problem. One approach which is common to all the existing techniques is to use heuristic rules for guiding the search to an approximate solution. However, the heuristic rules for obtaining approximate solutions are generally difficult to derive and the accuracy of the solutions are hard to verify. In this paper, we propose a new approximation approach to solve the general load balancing problem.

In the new approach, first we focus on resource migration only but with a view to reducing the job migration costs. For this resource migration problem, we propose an approximate model which can be solved in polynomial time. This approximate model partitions the given system into regions such that all the jobs as well as the resources in a region have similar characteristics. The partitioning helps to achieve local approximations for the job and resource characteristics (the smaller the regions, the better the approximations) as well as to separate job and resource migration problems. By suitable partitioning, good approximate solutions to resource migration problem can be obtained. The approximate model for resource migration reduces to a bottleneck transportation problem and hence can be solved by a polynomial-time algorithm.

In the next section, we introduce our approximate model for load balancing and also give a brief outline of an effcient algorithm we have developed for solving a special case of this problem. In Section 3, we discuss in detail the application of this model to file migration problem in distributed databases and the simulation results on two small examples. In Section 4, we discuss briefly the application of the model to host migration in mobile computer networks. Finally, we give conclusions and future directions.

Approximate Model for Resource Migration

First, we partition the given distributed system into a certain number of regions (say m) $W_1, W_2, \ldots, W_m$ such that all the jobs within a region have similar characteristics such as resource requirements. In our model, we only consider the migration of the resources but not the jobs among the regions. It may be possible that even after resource migration, a job at a node may need a resource which may not be available at the same node. In this case, the request for the resource can be processed remotely at some node within the same region or the job itself can be migrated to the node containing the resource. In either case, after resource migration, all the resource requirements within a region must be met by the resources that exist in the same region; this restriction is specified as a constraint in our problem.

A word on notation. We denote the set of non-negative integers by $\mathcal{N}$. Now we define the following parameters:

f_R — fixed cost of migrating one unit of resource
c_R — unit distance resource migration cost
b_i — average resource requirements for a job in region W_i
M_i — Number of jobs in region W_i that need the resource
N_i — Number of resource units in region W_i before migration
h_i — *average* time for a node in region W_i to communicate a resource request to a remote node within W_i and send the results back
a_i — *average* time to process a resource request in region W_i
d_{ij} — *average* distance between regions W_i and W_j
T — desired average job response time

The quantities a_i and b_i are averages over all the nodes and the jobs respectively while h_i is an average over all pairs of nodes in the region W_i. The quantity d_{ij} is an average over all pairs of nodes (v_1, v_2) such that $v_1 \in W_i$ and $v_2 \in W_j$. We assume that the total number of resource units remains the same after resource migration.

With the bottleneck migration cost criterion, the resource migration problem is formulated as follows:

$$\begin{aligned}
\text{Minimize} \quad & \max_{\{(j,k)|z_{jk}>0\}} (f_R + c_R.d_{jk}) \\
\text{s.t.}\ h_j + \frac{M_j.b_j.a_j}{\sum_{k=1}^{m} z_{kj}} &\leq T, \text{ for all } 1 \leq j \leq m \\
\sum_{k=1}^{m} z_{jk} &= N_j, \text{ for all } 1 \leq j \leq m \\
z_{jk} &\in \mathcal{N}, \text{ for all } 1 \leq j, k \leq m.
\end{aligned}$$

The variable z_{jk} denotes the number of resource units that need to migrate from the region W_j to the region W_k. The quantity $R_j = \lceil \frac{M_j.b_j.a_j}{(T-h_j)} \rceil$ is the minimum resource capacity (expressed in number of resource units) needed to meet the requirements of jobs in region W_j. The quantity $t_{jk} = (f_R + c_R.d_{jk})$ is the cost of migrating a resource unit from the region W_j to the region W_k.

With these notations, we reformulate the load balancing problem as follows :

$$\begin{aligned}
\text{Minimize} \quad & \max_{\{(j,k)|z_{jk}>0\}} t_{jk} \\
\text{s.t.}\ \sum_{k=1}^{m} z_{kj} &\geq R_j, \text{ for all } 1 \leq j \leq m \\
\sum_{k=1}^{m} z_{jk} &= N_j, \text{ for all } 1 \leq j \leq m \\
z_{jk} &\in \mathcal{N}, \text{ for all } 1 \leq j, k \leq m.
\end{aligned}$$

The above formulation is known as the *bottleneck transportation problem* in the Operations Research literature.

It is also possible to define two sets of regions, one for the resource distribution before migration and the other for the distribution after migration. We call these regions "supply" regions and "destination" regions respectively. The supply regions can be defined depending on the basis of job characteristics and also on the network characteristics to a certain extent. The destination regions can be defined on the basis of resource migration costs. Note that the number of supply regions need not be the same as the number of destination regions. This variation in the model provides a close approximation to the exact model we have defined and also it can be used to reduce the dimension of the problem with very little additional approximation.

Several efficient algorithms for the bottleneck transportation problems have been proposed in the literature (e.g. [4]). One special case of the problem is the $2 \times n$ ($n \times 2$) bottleneck transportation problem in which there are two suppliers (destinations) and n destinations (suppliers). We have developed an $O(n^2)$ algorithm to solve this special case. We will give a brief outline of our algorithm here (refer to [6] for details). The algorithm for the $2 \times n$ problem uses the fact that there is an optimal solution in which at most one destination needs to be supplied by both the suppliers. Hence we restrict our attention to only those solutions which satisfy this property. At each iteration, a new feasible solution is obtained which has a bottleneck value less than or equal to that of the previous solution. An *upper bound* for the optimal value is implicitly defined by each new feasible solution. In addition, we determine at each iteration, the optimal shipments and the corresponding suppliers for one or more destinations. Hence after each iteration, a new bottleneck problem is solved with these destinations eliminated. The bottleneck value among the eliminated destinations is used to define a *lower bound* for the optimal value. The algorithm stops when the lower bound is greater than or equal to the upper bound or when the optimal shipments can be determined for *all* the destinations.

To summarize our approach to solve the load balancing problem, we use an approximate model to obtain an approximate solution for the exact model of the load balancing problem with resource migration. This approach avoids using heuristic rules to obtain the approximate solutions directly from the exact model. The heuristic rules are difficult to derive in many instances and do not take into account the specific nature of the applications. In the approximate model we propose, how the system is partitioned into regions affects the accuracy of the solution and this partitioning can exploit the characteristics of the specific application domain. For example, in the case of local area networks gatewayed together, each local area network with its resources can be considered a region in the approximate model.

In the next section, we discuss in detail the application of this load balancing model to the file migration problem in distributed systems.

Database File Migration

In a distributed relational database, relations are partitioned either vertically (across attributes) or horizontally (across instances of the relations) into possibly overlapping fragments which are referred to as database files. These files are distributed (also replicated) across the nodes in the network. A transaction submitted by the user at a node can be either a query or an update on the database. The update transaction translates into requests to all the relevant sites or nodes for updating the appropriate files. We do not address the query decomposition problem here. If the file required by a subquery or a query does not exist locally (i.e. on the node at which the query is generated) then the subquery is sent to a remote node containing the file and processed there. We will explain later how a remote node is chosen for processing a subquery.

All the database transactions thus generate over a period of time an access pattern for the various database files; these access requests are classified into query and update requests. When the locality of file access patterns changes with time, files (along with the programs to process the subqueries) need to migrate among the nodes. For simplicity, we only consider the problem of single file migration (that is, the problem of migrating the multiple copies of a file).

The following costs are incurred during file migration: (1) costs of file and program storage , (2) costs of updating all file copies, (3) costs of file migration and (4) query costs. In determining optimal file migration, we want to minimize the first three costs while maximizing the average query throughput or minimizing the average query response time. The query response time consists of the following two components: (1) query communication time which includes the time for sending the request and receiving the results back and (2) query processing time which includes the time for processing the file access request. In formulation of the file migration problem, we make the following assumptions:

1. A constant number of file copies is maintained at all times.

2. Due to the first assumption, whenever a new file copy is generated at a node, some other existing copy at another node needs to be deleted.

3. All file copies can migrate in parallel.

4. Query communication delay is independent of the query traffic and is dependent only on the communication distance. We assume that the nodes have a limited processing capacity (expressed in file access requests per unit time) and the processing capacity is the same for all nodes. Hence the query processing delay is directly proportional to the query traffic directed to the node containing the file copy.

These assumptions, justifiable in many instances, are made primarily to simplify the presentation of our model, and to illustrate that even under these simplifying assumptions, the problem is already NP-hard. We use the following notations:

V — set of all nodes in the system and $P(V)$ its power set
I,I' — set of nodes containing file copies after and before migration
$f : (I - I') \rightarrow I'$ — migration function specifying how the files migrate
$g : V \rightarrow I$ — query assignment function specifying where a query request from a node needs to be processed
$h : I \rightarrow P(V)$ — indicates for a file copy node, the set of nodes whose queries need to be processed by that node (i.e. inverse function of g)
n — number of nodes in the system
q — query processing capacity of a node (file access requests per unit time)
u_x,b_x — update and query request rates from node x
$m_{x,y}$,$s_{x,y}$ — unit update and query communication costs from node x to node y
F_y — cost per unit time of storing a file copy at node y
$E_{x,y}$ — cost of migrating a file copy from node x to node y
$\delta : I \rightarrow \mathcal{N}$ — query delay function indicating the average delay in processing a query at a node
T — desired maximum average query response time

There are three cost components, namely the *file copy* *overhead cost* (denoted by $U(I)$), the *query cost* (denoted by $Q(I,g)$) and the *migration cost* (denoted by $R(I,f)$). These costs are defined as follows:

$$\begin{aligned} R(I,f) &= \max_{y \in I-I'} E_{f(y),y} \\ U(I) &= \sum_{y \in I} [\sum_{x \in V} u_x m_{x,y} + F_y] \\ Q(I,g) &= \max_{\{x \in V | b_x > 0\}} [s_{x,g(x)} + \delta(g(x))] \end{aligned}$$

Here $\delta(y) = \sum_{x \in h(y)} b_x/q$. The general file migration problem is posed as follows: Find I, g and f (injective) such that $R(I,f)$ is minimized with the constraint that $Q(I,g) \leq T$, $U(I) \leq C$ (T and C are positive constants) and $|I| = |I'|$.

A different formulation of the file migration problem is given in [8]. In this model, the *total cost criterion* is used 'for all the above three costs and the sum of all these three costs is minimized while fixing g and f as follows: $g(x) = \min_{y \in I} s_{x,y}$ and $f(x) = \min_{y \in I'} E_{y,x}$. Also in this model, $Q(I,g) = \sum_{x \in V} b_x \; g(x)$, that is, the query processing time is assumed to be dependent only on the communication cost but not on the query traffic. In our formulation (we call it "bottleneck file migration problem"), we separate the query cost into a constraint on query response time. This formulation is useful to guarantee a maximum query response time.

We can easily show that the bottleneck file migration problem is NP-hard even if there is no constraint on the file copy overhead cost ($U(I)$); for the proof see [7]. We will illustrate how our approximate model can be used to solve this problem. In our approximate model, we simplify the problem by first eliminating the constraint on the file copy overhead cost. In most applications, when a constant number of file copies is maintained, this overhead cost would not differ significantly among different allocations of these copies to the nodes. Next we separate the query access problem (i.e., determining g) from the file migration problem (i.e., determining f and I). For this, we partition the set of nodes into regions $W_1, W_2, \ldots, W_m$. Let $I'_1, I'_2, \ldots, I'_m$ be the corresponding sets of nodes in the regions containing file copies before file migration; that is, $I'_j = \{x \in W_j | x \in I'\}$, for $1 \leq j \leq m$. One of the possible rules for partitioning is discussed below. First we introduce the following additional notations:

N_j = number of file copies in W_j before migration
= $|\{x \in W_j | x \in I'\}|$
L_j = number of nodes in region W_j (= $|W_j|$)
S'_j = average query communication cost in region W_j
= $\sum_{x,y \in W_j} s_{x,y}/(L_j^2 - L_j)$
$E'_{j,k}$ = average cost of file copy migration from W_j to W_k
= $\sum_{x \in I'_j} \sum_{y \in (W_k - I'_k)} E_{x,y}/(N_j \times (L_k - N_k))$
Q'_j = total query traffic in region W_j (= $\sum_{x \in W_j} b_x$)
$z_{j,k}$ = number of copies to be migrated from W_j to W_k

One possible rule to use in the partitioning is to require that the nodes in each region have similar migration cost characteristics and communication cost characteristics. More formally, for $1 \leq j \leq m$ and $x, y \in W_j$ with $x \neq y$, $|s_{x,y} - S'_j| < \epsilon_1$, for ϵ_1 very small. Also for $1 \leq j, k \leq m$, $x \in W_j$ and $y \in W_k$, $|E_{x,y} - E'_{j,k}| < \epsilon_2$ for ϵ_2 very small. Larger the values of ϵ_1 and ϵ_2, larger will be the number of regions and greater will be the time to solve the problem. On the other hand, smaller these values, smaller the number of regions but further from optimality the solution from this model will be. Thus a "good" partitioning rule makes a suitable trade-off between accuracy and time and this trade-off depends on the specific distributed system under consideration.

Now we formulate the file migration problem as follows:

$$\begin{aligned} \text{Minimize} \quad & \max_{\{(j,k) | z_{jk} > 0\}} E'_{j,k} \\ \text{s.t. } S'_j + \frac{Q'_j}{\sum_{k=1}^{m} z_{kj} \cdot q} &\leq T, \text{ for all } 1 \leq j \leq m \\ \sum_{k=1}^{m} z_{jk} &= N_j, \text{ for all } 1 \leq j \leq m \\ z_{jk} &\in \mathcal{N}, \text{ for all } 1 \leq j, k \leq m. \end{aligned}$$

As in Section 2, if we introduce for each $j = 1, 2, \ldots, m$, the quantities $R_j = \lceil \frac{Q'_j}{(T - S'_j).q} \rceil$, then the problem becomes a bottleneck transportation problem. R_j represents the minimum number of file copies needed in region W_j to satisfy all the query requests within region W_j. We require here that any query request within a region be directed to a location within the same region.

In our preliminary investigation of the performance of this model, we considered the five-node example given in [1] and an eight node example. We approximated the file migration problems in these examples by the $2 \times n$ bottleneck transportation models and used the algorithm we have developed to solve these problems. For each example, we considered different partitioning of the system into regions. Since the diameters of the graphs (with respect to query communication cost) in the examples were small compared to the response time, only the second partitioning rule (that is, the one requiring the regions to have similar communication characteristics) could be tested in this experiment. For each partition, we had 500 runs and in each run, we varied the file access (both the query and the update request) pattern randomly according to an uniform distribution. For each run, we compared the optimal solution of the exact model with the optimal solution of the approximate model based on whether the number of file copies in each destination region is the same in the two solutions. When the nodes within a region have similar communication cost characteristics, about 50% of the runs gave solutions that agreed with the optimal solutions. For the arbitrary partitions, this figure varied from 2% to 25%. Thus the partitioning rule we have mentioned before has a significant impact on the "goodness" of the solutions.

Though the file copy overhead cost was not considered in the approximate model, we also compared these costs for the exact and the approximate models; for the approximate model, these costs can only be estimated since the exact locations of the file copies are unknown. For a "good" partitioning such as the one in which nodes within a region have small costs to communicate within themselves, the difference in these costs averaged to within 20% over all the runs. Thus the file copy overhead does not appreciably change due to elimination of this cost from the approximate model. These results are encouraging considering that only a simple partitioning rule has been used in these examples. We plan to perform a more extensive analysis of the performance of the proposed model and of different partitioning rules in particular, by applying it to larger scale examples.

Host Migration in Mobile Computer Networks

A mobile computer network consists of mobile hosts which communicate with each other, using wireless radio channels. Thus the topology of a mobile computer network changes from time to time. The mobile computer networks are becoming a commercial reality due to the rapid advances that are being made in mobile communication technology. Another practical example of a mobile computer network arises in robotics applications, where a team of robots is employed to perform certain tasks in a cooperative manner. The robots in this case, in addition to having the processing power, also must have the ability to communicate with each other within an operational area like mining fields and other harsh environments.

In a mobile computer network, the mobility of the hosts can be used to advantage in balancing the workload among the hosts. In the load balancing problem in a mobile computer network with homogeneous hosts, the objective is to minimize the host migration cost with a constraint on the job response time. There may also be an additional constraint in that the hosts must be able to communicate with each other at all times either directly or using many hops. Further we may also require that the results of running a job must be available at a host which can at most be at a distance r from the location of the host to which the job is submitted.

Since this problem is NP-hard (see [6]), we can use the approximate model proposed in Section 2 to solve this problem. For this, we partition the network into geographical regions. One partitioning rule to use is as follows: the distance between any two nodes in a region is less than r; smaller the value of r, more the number of regions and vice versa. The value or r also affects the "goodness" of the solutions obtained through the approximate model. The network can also be partitioned on the basis of the structure of the backbone network that is usually defined for communication among the hosts. In a mobile computer network, hosts are divided into clusters with a clusterhead for each cluster. A backbone network connects the cluster heads in some configuration. The hosts within a cluster communicate with each other directly while the intercluster communication takes place using the backbone network. A natural choice of regions here will be the clusters themselves. We can then formulate the approximate load balancing model in the same fashion as defined in Section 2. The bottleneck transportation problem here determines the number of host units that should migrate from one region to another such that the bottleneck host migration cost is minimized while guaranteeing the maximum average response time.

Conclusions

We have proposed a commodity distribution model as a tractable approximate model for load balancing with resource migration. If the given distributed system is partitioned on the basis of job and network characteristics, then the approximate solution is reasonably close to the optimal solution of the exact model. This has been demonstrated to a certain extent in the case of file migration in distributed data bases. A special case of this problem can be solved by an efficient algorithm which we have developed. This resource migration model must be supplemented with appropriate job migration models for load balancing within each region. Though we have demonstrated that partitioning rules have an impact on the accuracy of the solutions, a formal analysis supported by experimental results is necessary. This is the subject of our future investigation.

References

[1] R. G. Casey, "Allocation of Copies of a File in an Information Network," *SJCC*, 1972, pp.617-625.

[2] K. P. Eswaran, "Placement of Records in a File and File Allocation in a Computer Network," *Information Processing 74*, IFIPS, 1974, pp.304-307.

[3] D. J. Farber, et. al.,"The Distributed Computer System," *Proc. Seventh Annual IEEE Computer Society International Conference*,February 1973.

[4] R. S. Garfinkel and M. R. Rao, "The Bottleneck Transportation Problem," *Nav. Res. Logistics Quarterly*, 18, 1971, pp.465-472.

[5] J. A. Stankovic and I. Sidhu, "An Adaptive Bidding algorithm for Processes,Clusters and Distributed groups," *Proc. Int. Conf. Distributed Comp. Systems*, May 1984.

[6] Ravi Varadarajan, "Reliability and Performance Models for Reconfigurable Computer Systems," Department of Computer and Information Sciences, University of Pennsylvania, PA, Technical report MS-CIS-87-65.

[7] Ravi Varadarajan and Eva Ma, "Load Balancing Models with Resource Migration in Distributed Systems," Computer and Information Sciences Department, University of Florida, Gainesville, Technical Report in preparation.

[8] Benjamin W. Wah, "A systematic Approach to the Management of Data on Distributed Databases," *Ph.D. thesis*, University of California, Berkeley, 1979.

An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems

Kazuaki Rokusawa Nobuyuki Ichiyoshi Takashi Chikayama
Institute for New Generation Computer Technology *

Hiroshi Nakashima
Mitsubishi Electric Corporation †

Abstract

This paper describes an algorithm for termination detection and abortion in distributed processing systems, where processes may exist not only in processing elements but also in transit. The algorithm works correctly whether the communication channels are first-in-first-out or not, and no acknowledgement message is required. Assigning weights to all processes and maintaining the invariant that the sum of the weights is zero are the main features of the algorithm.

1 Introduction

Termination detection and abortion of all processes in a system are major functions in parallel processing. They are easy in closely-coupled systems, such as shared memory multiprocessors, but difficult in distributed systems, particularly when there are processes in transit.

We have devised an algorithm for termination detection and abortion in distributed processing systems, where processes may exist not only in processing elements but also in transit. This algorithm is called the *weighted throw counting* scheme, which is an application of the weighted reference counting scheme [1] [5], a garbage collection scheme for parallel processing systems.

The algorithm will be applied to parallel implementation of KL1, a parallel logic programming language based on GHC [4], on the Multi-PSI [3], a collection of Personal Sequential Inference Machines [6] (PSI's) interconnected by a fast communication network.

This paper is organized as follows. Section 2 defines the computation model employed. Section 3 shows the problems of termination detection and abortion in distributed systems. A naive solution is presented in section 4. Section 5 describes the algorithm for termination detection and abortion where the communication channels are first-in-first-out. The algorithm for the system with non-first-in-first-out communication is presented in section 6. Finally the comparison of the algorithm with the naive one is given in section 7.

2 Computation Model

The following process model is assumed:

- A process pool consists of one controlling process and a finite number of child processes;
- There are a finite number of process pools in the system;
- Each process pool is assigned a unique process pool identifier (PID);
- A child process can terminate at any time;
- A child process can generate another child process having the same PID and a new process pool having a new PID as well.

In this paper, "process" means "child process" unless otherwise indicated. A process pool *terminates* if all the children terminate. *Aborting* a process pool is forcing all the children to terminate. A process pool described above is distributed over the following machine:

- A finite number of processing elements (PEs) interconnected by a communication network;
- No global storage; PEs may communicate by passing messages;
- Asynchronous communication, in which messages are delivered with arbitrary finite delay.

It is assumed that a PE can detect the termination of all processes in it having the same PID and can force them to terminate. The controlling process and PEs can communicate in both directions. A PE may send a message to the controlling process informing it of the termination of all processes, and the controlling process may send a message to abort processes.

Although there exist a finite number of process pools in the system at a given time, there is no limitation of total

*Fourth Research Laboratory, Institute for New Generation Computer Technology, 4-28, Mita 1-chome, Minato-ku, Tokyo 108 JAPAN

†Computer Systems Development Department, Information Systems and Electronics Development Laboratory, Mitsubishi Electric Corporation, 1-1, Ohfuna 5-chome, Kamakura-shi, Kanagawa 247 JAPAN

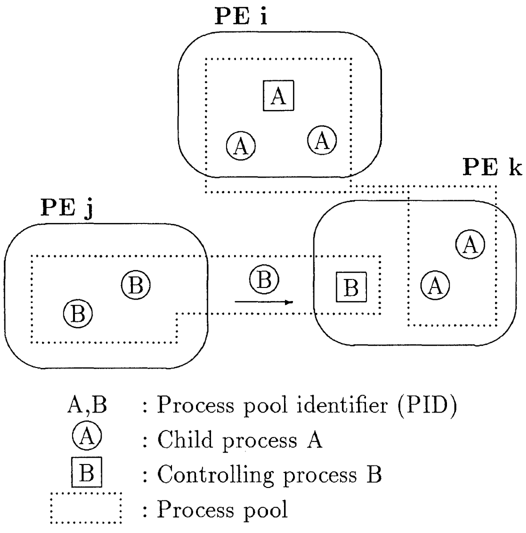

PE i
A
A
A
PE k
PE j
B
B
B
B
A
A
A,B : Process pool identifier (PID)
A : Child process A
B : Controlling process B
: Process pool

Figure 1: Computation Model

number of process pools, since any process can generate a new process pool at any time.

Processes may migrate among PEs for load balancing. To achieve this, a PE may throw a process in the PE to another PE and the thrown process is delivered with arbitrary finite delay. Therefore, at a given time, processes may be in transit in the communication network but not in any PEs.

3 Problems

This section describes why termination detection and abortion of processes distributed over several processors are difficult, particularly when there are processes in transit.

3.1 Termination Detection

The controlling process must detect the termination of all processes having the same PID as the controlling process.

Each PE can detect the termination of all processes with the same PID in the PE locally and can send a message indicating termination (*terminated* message) to the corresponding controlling process.

However, even if the controlling process receives *terminated* messages from all PEs, it is not sure that all processes have terminated. There may be processes in transit, which will be received by a PE after the PE has sent a *terminated* message.

3.2 Abortion

The controlling process must force to terminate all processes having the same PID as the controlling process.

If the controlling process broadcasts a message causing a process to terminate (*abort* message), it is possible to abort all the processes in the PE, but impossible to abort the processes in transit. After receiving an *abort* message and aborting the processes, the PE may receive a thrown process.

If a PE memorizes the PID carried by the *abort* message, and, ignores received processes with the same PID as that memorized, the *abortion by broadcast* scheme described above may work. However, this scheme has disadvantages. First, if only a few PEs have the process to be aborted, most of *abort* messages are useless. Second, it is impossible to reuse a PID, because the controlling process cannot detect the termination of the abortion; this is a major disadvantage.

4 The Naive Scheme

Ichiyoshi et al. [2] describe a termination detection scheme using *acknowledge* messages. It effectively does the following, although different terminology is used. A non-empty set of processes in one PE having the same PID forms a subpool of processes, which is called a "process subpool", or a "subpool" in short. Processes in a PE are under the control of a subpool. On receiving a thrown process, the PE decides whether there is already a subpool having the same PID as the thrown process. If there is, the PE adds the process received to the subpool and sends back an *acknowledge* message; otherwise, creates a new subpool and memorizes the sender PE of the process in it. Each subpool has a counter which is incremented on throwing a process, and is decremented on receiving the *acknowledge* message or *terminated* message. When all processes in it are terminated and the value of the counter reaches zero, the subpool terminates and sends a *terminated* message to the PE memorized.

This scheme is simple and termination can be detected correctly; if the value of the counter reaches *zero*, there is neither process thrown from the corresponding subpool in transit nor subpool created by the thrown process from the corresponding subpool. However, it has a serious disadvantage; termination of a subpool depends on terminations of other subpools. Since subpools form a tree structure, a root cannot terminate unless all its leaves terminate. In the worst case, a chain of subpools is created, where each subpool terminates sequentially.

5 The WTC Scheme

We have devised a new scheme which requires no acknowledge message and makes it possible to reuse the PID. This new scheme is the *weighted throw counting* (WTC) scheme which is an application of the weighted reference counting scheme [1] [5], a garbage collection scheme for parallel processing systems.

5.1 Termination Detection

We associate *weight* with the controlling process, each process and each subpool. The weight of a process in transit and that of a subpool are positive integers, while the weight of the controlling process is a negative integer. The WTC scheme maintains the invariant that:

> The sum of the weights is *zero*.

This ensures that the weight of the controlling process reaches *zero* if and only if all processes terminate; there is no processes neither in a PE nor in transit (see figure 2).

When a PE throws a process from a subpool, the PE assigns a weight to the thrown process and subtracts the same amount from the weight of the subpool. The new weight of the subpool and that assigned to the thrown process should both be positive, and the sum of the two weights is equal to the original weight of the subpool. For example, if a subpool originally weighs 1000, the weight of a thrown process and the new weight of the subpool can be set to 50 and 950. When a PE receives a thrown process, it adds the weight assigned to the received process to the weight of the subpool having the same PID. If there is no subpool with the same PID, a PE creates a new subpool containing the received process and sets its initial weight at the weight of the received process.

When the weight of a subpool becomes *one*, the PE cannot throw a process, because non-zero weight must be assigned to the thrown process and non-zero weight must remain also in the subpool after throwing. The operation when this situation occurs is described in section 5.3.

When all processes in it are terminated, the subpool *terminates* and sends a *terminated* message to the corresponding controlling process. This *terminated* message gives notification of the termination of the subpool and carries the weight of the terminated subpool. On receiving a *terminated* message, the controlling process adds the weight carried by the *terminated* message to its (negative) weight. If the weight of the controlling process reaches *zero*, the termination of all processes is detected.

5.2 Abortion

This section describes an abortion scheme for the computation model with first-in-first-out communication; messages are delivered in the order sent. A scheme without this assumption is described in section 6.

The controlling process should be able to force all processes with the same PID as the controlling process to terminate, and detect the termination of all processes to reuse the PID. Termination is detected using the WTC scheme described above. Thus, only delivery of the *abort* message to each PE containing the subpool is required. To achieve this, the controlling process needs to detect the creation of a subpool and to send an *abort* message to a PE containing a subpool.

We introduce here a new message, named the *ready* message which gives notification of the creation of a subpool. On creation of a subpool, a PE sends a *ready* message to the corresponding controlling process. On receiving a *ready* message, the controlling process memorizes the sender PE, which is deleted on receiving a *terminated* message.

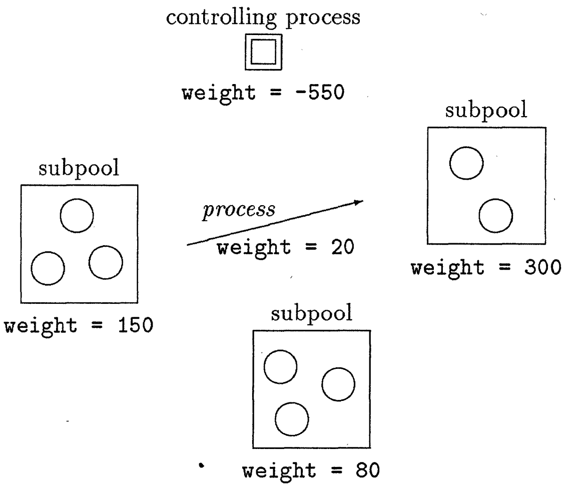


Figure 2: The WTC Scheme

The controlling process performs the following operations to achieve the abortion:

(1) Sending an *abort* message to each PE memorized;

(2) Sending an *abort* message to the sender PE of a *ready* message received after operation (1).

Once the controlling process receives a *ready* message, a subpool may exist in the sender PE until a *terminated* message is received from the same PE. The controlling process therefore performs operation (1), which aborts all subpools already detected by the controlling process. Operation (2) aborts such subpools that were not recognized by the controlling process when operation (1) was carried out: a subpool that is created after operation (1), or created before operation (1) but whose *ready* message is still in transit.

It is necessary to assign a weight to an *abort* message like the thrown process, while not necessary to a *ready* message, because once the controlling process receives a *ready* message, it will receive a *terminated* message later from the sender PE of the *ready* message (the FIFO assumption).

On receiving an *abort* message, a PE performs either of the following operations:

(3a) Forcing the subpool with the specified PID to terminate, and sending back a *terminated* message which carries the sum of the weight of the terminated subpool and the *abort* message;

(3b) If there is no subpool having the specified PID, sending back a *return* message which carries back the weight assigned to the *abort* message.

Figure 3 shows the abortion operations described above.

When a subpool terminates before receiving an *abort* message, an *abort* message may reach a PE having no subpool with the same PID as the *abort* message. In this case,

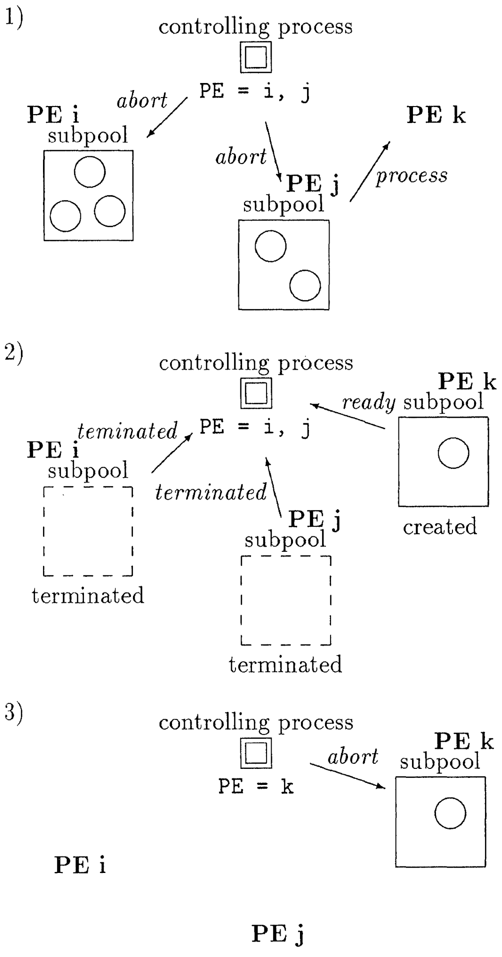


Figure 3: Abortion Operations

operation (3b) is performed and the *return* message is sent as the response to the *abort* message. On receiving a *return* message, the controlling process adds the weight of the message to its own weight. If the weight of the controlling process reaches zero by this operation, the termination of all processes is guaranteed.

During the operations of abortion, the following cyclic situation may occur. The controlling process sends an *abort* message to abort a subpool. A process is thrown from the subpool before the *abort* message arrives. The thrown process is delivered to a PE where there is no subpool having the same PID as the thrown process. Then a new subpool is created and a *ready* message is sent. On receiving the *ready* message, the controlling process sends *again* an *abort* message to abort this newly created subpool.

On receiving one *abort* message, one subpool is aborted and the non-zero weight of the subpool is sent back to the controlling process. Since the sum of the weights of subpools and processes in transit is finite, all processes can be aborted by sending a finite number of *abort* messages, even if the above situation occurs.

5.3 When the Weight becomes One

As mentioned in the section 5.1, when the weight of a subpool becomes *one*, the PE cannot throw a process.

In this case, the PE sends a message requesting more weight (*request* message) to the controlling process. Process throwing is suspended until the weight of the subpool becomes more than one. On receiving a *request* message, the controlling process sends back a message which carries some weight to the sender PE (*supply* message) and reduces the same amount from its own weight. When a PE receives a *supply* message, it adds the weight carried by the *supply* message to the weight of the subpool, which enables it to throw any suspended processes. Since receiving of a thrown process also increases the weight of the subpool, a subpool may terminate before receiving a *supply* message, and a *supply* message may reach a PE that contains no subpool. In this case, a *return* message is sent back to the controlling process. This is similar to the action when a PE without a subpool receives an *abort* message.

It is not necessary to assign any weight to the *request* message, because a *terminated* message is delivered to the controlling process only after this *request* message (the channel is FIFO), and the weight of the controlling process never reaches zero, leaving *request* messages in transit.

5.4 How to Assign a Weight

This section describes the strategy to assign a weight which decreases the number of additional messages (*request* and *supply* messages).

In the worst case, that is, to assign a weight of *one* in any case, the same number of additional messages as the thrown processes are required, while no additional messages are required in the best case. If the weight carried by a *supply* message is large enough compared with the weight assigned to a thrown process, the weight of the subpool will not reach easily one after receiving a *supply* message. The weight assigned to the thrown process must be less than the weight of the subpool, while the weight carried by a *supply* message does not have this limitation. Using the following strategy, one subpool almost always needs only to send a *request* message *once*.

- Assign a fixed weight (say 2^{10}) to a thrown process if the weight of the subpool is more than twice of that; otherwise assign half of the weight of the subpool.
- A *supply* message carries a very large weight (say 2^{20}).

On receiving a *supply* message, the weight of the subpool becomes more than 2^{20} and it can throw a process at least 2^{10} times without receiving any weight.

If a subpool receives a *supply* message before its weight becomes one, it need not to send a *request* message. A subpool which is created by receiving a process assigned a weight of 2^{10} can throw a process at least 10 times until its weight becomes one. Therefore, if the controlling process sends back a *supply* message on receiving a *ready* message, a *request* message is expected to be needless.

6 Non-FIFO Communication

In the computation model with non-first-in-first-out communication, the following situations may occur:

- A *terminated* message may be delivered before a *ready* message and a *request* message.
- The controlling process may receive several *ready* messages (or *terminated* messages) before receiving a *terminated* message (or a *ready* message).

The former may cause the weight of the controlling process to reach zero, leaving *ready* messages or *request* messages in transit. On account of the latter, simply memorizing or deleting the sender PE of a *ready* message or a *terminated* message will not work. To cope with the situations mentioned above, we modify the scheme as follows:

- Assign a weight to a *ready* message and a *request* message (a *request* message will be sent when the weight reaches *two*).
- The controlling process has a set of counters corresponding to each PE, which is incremented on receiving a *ready* message and is decremented on receiving a *terminated* message.

The former change assures that the weight of the controlling process never reaches zero leaving any messages or processes in transit. By the latter change, if a subpool may exist in a PE, the value of the corresponding counter becomes positive. The controlling process thus performs the following operations to achieve the abortion:

(1) Sending an *abort* message to each PE whose corresponding count is positive;

(2) Sending an *abort* message to the sender PE of a *ready* message received after operation (1) if the count corresponding to the sender PE, after increment, is positive.

Since no more than one subpool can exist in one PE at a time, it is enough to send one *abort* message to one PE.

7 Comparison

The WTC scheme is much superior to the naive scheme using acknowledgement in two points.

First, the WTC scheme requires fewer additional messages than in the naive scheme. The number of subpools created is expected to be small enough compared with the number of thrown processes. The WTC scheme requires about the same number of *request* messages and *supply* messages as the number of the creations of subpools, while the naive scheme requires almost the same number of *acknowledge* messages as the number of thrown processes.

Second, in the WTC scheme, each subpool can terminate independently, while in the naive scheme, termination of a subpool depends on terminations of other subpools.

8 Summary

We have devised an efficient algorithm for termination detection and abortion. Its major advantages are as follows.

- Only a few additional messages are required.
- Each subpool can terminate independently.
- Reuse of the process pool identifier is possible.

The techniques described in this paper are applicable to many kinds of distributed processing systems.

Acknowledgements

We thank the members of the Multi-PSI group in the ICOT Research Center and cooperating companies, the Director of ICOT, Dr. Kazuhiro Fuchi, and Dr. Shunichi Uchida for valuable discussions and encouragement.

References

[1] D. I. Bevan. Distributed garbage collection using reference counting. In *Proceedings of Parallel Architectures and Languages Europe*, pages 176–187, June 1987.

[2] N. Ichiyoshi, T. Miyazaki, and K. Taki. *A Distributed Implementation of Flat GHC on the Multi-PSI.* Technical Report TR-230, ICOT, 1987. Also in Proceedings of the Fourth International Conference on Logic Programming, 1987.

[3] K. Taki. The parallel software research and development tool: Multi-PSI system. In *Proceedings of France-Japan Artificial Intelligence and Computer Science Symposium 1986*, pages 365–381, 1986.

[4] K. Ueda. *Guarded Horn Clauses.* Technical Report TR-103, ICOT, 1985.

[5] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *Proceedings of Parallel Architectures and Languages Europe*, pages 432–443, June 1987.

[6] M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida. *The Design and Implementation of a Personal Sequential Inference Machine: PSI.* ICOT Technical Report TR-045, ICOT, 1984. Also in New Generation Computing, Vol.1 No.2, 1984.

DISTRIBUTED SYNCHRONIZERS

Doddaballapur N. Jayasimha
Center for Supercomputing Research and Development
University of Illinois, Urbana, IL 61801.

Abstract

In this paper we introduce a new synchronization primitive, the *distributed synchronizer*. This primitive, based on the notion of partially shared variables, suits the synchronization requirements of parallel algorithms executing on large, shared memory multiprocessors. We consider the commonly required forms of synchronization in a multiprocessor: barrier, reporting, and mutual exclusion. We introduce the *synchronization tree* through an algorithm to implement barrier synchronization. An efficient implementation of the distributed synchronizer primitive requires a) the embedding of the synchronization tree in the processor–memory multistage interconnection network, and b) simple hardware enhancements at the switching elements of the network. For n processors, this primitive implements reporting with zero synchronization overhead and the barrier with a log n cycle overhead. We show that the implementation of the semaphore operations using the distributed synchronizer is *bounded fair*. Finally, we discuss some implementation issues and a few limitations of our synchronization scheme.

1. Introduction

It is well known that the synchronization overheads have a deleterious effect on the speedup of parallel algorithms. It has been observed that for some applications with extensive synchronization requirements, the speedup reaches a maximum for a small number of processors, and thereafter decreases [Axel86]. In this paper we introduce a new synchronization primitive, the *distributed synchronizer*. An implementation of this primitive is shown be efficient and bounded fair. The primitive is based on the notion of partially shared variables, and suits the synchronization requirements of parallel algorithms executing on large, shared memory multiprocessors. Examples of such architectures are the Cedar system [Kuck84], the Ultracomputer [GGKM83], and the RP3 [Pfis85]. Typically such a multiprocessor consists of n homogenous and autonomous processing elements (PEs). An interconnection network connects the PEs to a set of main memory modules such that each PE can access any memory module.

2. Synchronization

2.1. Classification

Multiprocessors commonly require the following forms of synchronization: a) barrier, b) reporting, and c) mutual exclusion. (Note: Mutual exclusion is necessary in the first two cases also.) Following Axelrod [Axel86], we define a synchronization barrier to be a logical point in the control flow of an algorithm at which *all* processes must arrive before *any* of them are allowed to proceed further. Reporting requires that *all* processes must arrive at a control point before another specified process continues.

We illustrate the need for these forms of synchronization through the following example: suppose the maximum of r numbers is to be computed on an n–PE shared memory multiprocessor. Let M be a shared variable initialized to $-\infty$. Each PE computes the (local) maximum l of the numbers that it is assigned, and updates M to l if $l > M$. To detect the completion of finding the maximum, each PE decrements a shared variable S (initialized to n) after it updates M. The decrementing of S to zero implies that the maximum has been computed. Note that each PE requires exclusive access to update M and S. Suppose a specific PE is required to compute the maximum. It continually checks S until S becomes zero, and then reads M. The form of synchronization that we have described is an example of reporting. Alternatively, assume that every PE must obtain the maximum for its subsequent computations. Every PE checks S until S becomes zero and then reads M. This form of synchronization where values are reported (the value of S is updated by all the PEs) and then communicated to all the PEs is an example of barrier synchronization. The barrier is said to be *complete* when every PE knows that S is zero. The synchronization overheads (owing to the continual checking of S and the exclusive accesses to M and S) cause serious performance degradation in multiprocessors, especially as the number of PEs increases [Axel86, Jaya87a, PfNo85]. To reduce these overheads, various schemes have been proposed [GGKM83, GLee86, Jaya87a, PfNo85, YeTL86]. A particularly elegant method, called *combining*, has been proposed for the Ultracomputer [GGKM83], and is being considered for implementation on the RP3 [PfNo85]. Combining, which detects and combines memory requests to the same location, requires expensive hardware. Furthermore, simulations show that combining may not be required, or effective in many cases [GLee86]. In the Cedar system, synchronization overheads are reduced by providing additional hardware at each memory module to perform simple synchronization related computations [ZhYe84]. Both combining and the Cedar scheme require the locking of shared variables in order to access them with mutual exclusion. Consequently, algorithms using these schemes generate wasted memory traffic due to busy waiting. Our synchronization technique does *not* require the locking of globally shared variables.

2.2. Walk–in Walk–out Scheme

Rather than force all PEs to access a single variable, we could allow a fixed number of PEs, m $(2 \leq m \ll n)$ to access each variable. (We assume n to be a power of m throughout the paper). Such a scheme increases the number of synchronization variables required but might reduce memory contention. In particular, we could arrange the synchonization variables in the form of a *synchronization tree* as shown in Figure 2.1. All the variables in the tree are initialized to m. If a PE decrements a variable and finds the resulting value to be zero, then it proceeds to the next higher level of the tree. Otherwise the PE waits for the variable to assume a special value, say, -1. The last arriving PE decrements the root to zero and sets it to -1. At this time the walk-in ends. When a PE finds the variable on which it is waiting to be -1, it communicates this information to the next lower level. This procedure is repeated recursively. The completion time would be the time at which the last PE at the lowest level finds its variable to be -1. At this time the walk-out ends. This algorithm is similar to the software combining algorithm proposed in [YeTL86].

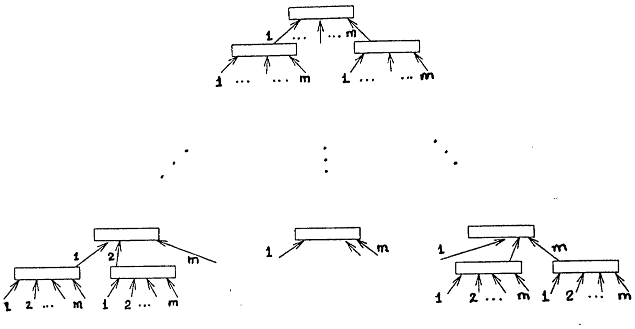


Figure 2.1 Synchronization Tree for the Walk–in Walk–out Scheme.

This work was supported in part by the **National Science Foundation** under Grant No. NSF MIP-8410110, by the **US Department of Energy** under Grant No. DOE DEG02-85ER25001, and by the **IBM** donation.

Notation: In the figures, the PEs and the nodes of the synchronization tree are numbered from left to right. We number the PEs from 1 to n. The PE with number i is denoted by PE_i. Upper and lower case names represent shared and local variables respectively. Let $h = \log_m n$, the number of stages in the interconnection network.

3. The New Primitive

3.1. Partially Shared Variables

The Walk-in Walk-out scheme has the advantage of requiring only a fixed number of PEs m to access a node of the synchronization tree. Furthermore, a node at level i need be shared only by m^i PEs (*note*: some m out of these m^i PEs access the node). This observation suggests that the nodes of the synchronization tree may be placed in a memory hierarchy according to the degree to which the nodes are shared. By a memory hierarchy we mean a set of partially shared memories such that a variable at the level i is shared by more number of PEs than a variable at the level j ($j < i$). Variables could be placed at an appropriate level in the memory hierarchy, thus eliminating expensive global memory trips to access shared variables that need to be only partially shared. The multi-memory hierarchy effectively distributes the "von Neumann bottleneck" and consequently achieves a better performance. The partial sharing of variables would lead to an increased complexity in the hardware and memory management.

3.2. Distributed Synchronizer

Consider any multistage interconnection network with the *full access capability* (which means that any input terminal of the network can reach any output terminal in one pass through the network) and the *unique path property* (which means that each input terminal has exactly one path through the network to reach any particular output terminal). Feng [Feng81] surveys such interconnection networks. Examples of these networks include the Omega, the baseline, the banyan, and the indirect binary n-cube. They are typically designed using $\log_m n$ stages of $m \times m$ switching elements. We embed the synchronization tree into the interconnection network with the leaves placed in the switching elements at the first stage and the root placed in a switching element at the last stage of the network. The connections between stages that lead to the switching element containing the root correspond to the branches of the tree. Figure 3.1 shows an embedding of the binary synchronization tree (with eight leaves) into an 8-input 2×2 Omega network. The heavy lines in the figure represent branches of the tree. Each stage of the interconnection network corresponds to an intermediate level in the memory hierarchy mentioned earlier, and each node in a switching element to a partially shared variable. Mutual exclusion must be guaranteed among the m children accessing their parent node. We explain how this is achieved for each synchronization operation in the relevant sections. The name *distributed synchronizer* refers to the embedded synchronization tree together with the operations defined on the tree.

4. Reporting and Barrier

A switching element in the network is a $m \times m$ bidirectional router. On its PE side each switching element has m input ports PI_1, ..., PI_m and m output ports PO_1, ..., PO_m. On its memory side each switching element has m input ports MI_1, ..., MI_m and m output ports MO_1, ..., MO_m. An input port PI_i on the PE side gets connected to an output port MO_j on the main memory side during a message transfer (message originating at a PE). Similarly, an input port MI_i gets connected to an output port PO_j during a message transfer (message originating at the main memory). Additionally, each switching element has a modulo m counter, a decoder, and some combinational logic.

For simplicity, we make the following assumptions:

(A1) All PEs participate in the synchronization operation.

(A2) At any instant at most one concurrent set of synchronization operations is being executed.

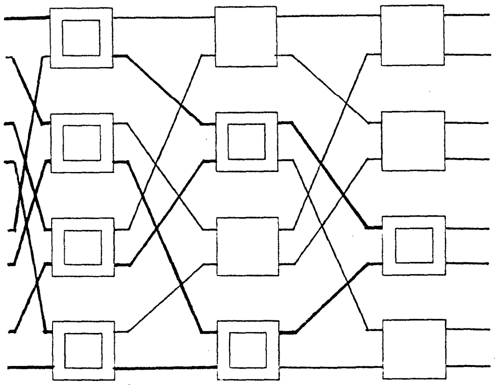


Fig. 3.1 Embedding a Synchronization Tree into an 8X8 Omega Network.

4.1. Reporting

To perform the reporting operation, each PE executes the *Rep*(S) instruction. S is a flag variable in global memory which is initially reset. S is set after *all* PEs have reported. The semantics of Rep(S) are shown in Figure 4.1, and are informally described below: A PE executing Rep(S) enters the synchronization tree at its appropriate leaf node. It decrements the counter at the switching element. If it finds the value of counter to be zero, then it is the last arriving PE for this synchronization operation at the node. It reinitializes the counter to m and proceeds to the node in the next higher level of the tree. Otherwise the instruction completes. This procedure is repeated recursively. The PE that decrements the root to zero then sets S, at which time the synchronization operation completes. This scheme has the following advantages: a) *The shared variable S need not be locked.* To ensure mutual exclusion, traditional multiprocessors use locking, which incurs an unnecessary global memory access whenever a PE tries to lock a variable that has been already locked by some other PE. The decrementing of the counter at a node has to be performed atomically. But this is easily achieved in the hardware. b) There are *no* synchronization overheads except for the constant time to execute the Rep(S) instruction.

Note that the reinitialization of the synchronization tree is achieved in a distributed manner since the last arriving PE at each node sets the counter to its initial value.

```
Algorithm Rep (S, PE#, level#);
/*The counter variables $C_{ij}$ are initialized to $m$, and $S$ is reset */
begin
    $j := \left\lceil \frac{PE\#}{m^{level\#}} \right\rceil$;
    $C_{level\#,j} := C_{level\#,j} - 1$;
    if $C_{level\#,j} = 0$ then /* last arriving PE at some node */
        if $level\# = h$ then
        begin
            $C_{level\#,j} := m$; /* reinitialize */
            set $S$; /* reporting is done */
        end
        else /* last arriving PE at a non-root node */
        begin
            $C_{level\#,j} := m$; /* reinitialize */
            $Rep(S, PE\#, (level\# + 1))$; /* to next higher level */
        end
end Rep;
```

PE_i executes $Rep(S, i, 1)$.

Figure 4.1 Semantics of the *Rep*(S) Instruction.

The following cases: a) requirement of d ($d > 1$) concurrent synchronization operations, and b) requirement of reporting among only n_1 ($n_1 < n$) PEs, are discussed in [Jaya87b].

4.2. Barrier

We will assume that (A1) and (A2) hold. Each PE performs the barrier synchronization by executing the instruction *Barrier*(v); v refers to a flag, initially reset, that is present in each PE. The semantics of the instruction, shown in Figure 4.2, are essentially an implementation of the Walk-in Walk-out scheme in hardware. The hardware requirements are the same as for reporting, except for the extra bit flag in each PE. If a request finds the counter at a stage to be zero, it proceeds to the next stage. Otherwise the requesting PE waits for its flag v to be set. The request that decrements the counter at the root to zero is the last process to arrive at the barrier. It signals the completion of the barrier to the $(m-1)$ requests waiting at the root by placing a return request containing the address of the instruction, at each of the m *PO* ports of the root's switching element. The procedure of "walking out" is recursively repeated till the leaves are reached. The final step consists in setting the v flag of each PE, at which time *every* PE comes to know that the barrier has been completed.

```
Algorithm Barrier (v, PE#, level#);
/*The counter variables $C_{ij}$ are initialized to $m$, and $v$ is reset */
begin
1.  $j := \left\lceil \frac{PE\#}{m^{level\#}} \right\rceil$;
2.  $C_{level\#,j} := C_{level\#,j} - 1$;
3.  if $C_{level\#,j} = 0$ then/* last arriving PE at some node */
4.     if level# = h then/* last arriving PE at barrier */
5.        Walkout(v, level#, j) /* begin walk-out */
       else /* last arriving PE at a node which is not the root */
6.        Barrier(v, PE#, (level# +1)); /* to next higher level */
end Barrier;

procedure Walkout(v, level#, k);
begin
1.  if (level# = 1)then
2.  set the flag in each PE
    else
3.     for $l := (k-1) \times m + 1$ to $k \times m$ do
       begin
4.        place return request in each $PO_l$ port;
5.        Walkout(v, (level# -1), l);
       end;
end Walkout;

$PE_i$ executes Barrier(v, i, 1).
```

Figure 4.2 Semantics of the *Barrier*(v) Instruction.

The delay between the time at which the last process finishes (i.e., the time at which the root is decremented to zero), and the time at which every PE knows that the barrier is complete is just $\log_m n$ cycles (assuming no network conflicts). The synchronization is accomplished with no wasted memory accesses, and without the need for any shared globally variables.

Each PE could "busy wait" on the flag v till it is set, or could context switch to a different process after executing the *Barrier*(v) instruction. In the latter case, the PE may be interrupted to signal the completion of the barrier. Observe that, unlike normal busy waiting, PEs in the former case do *not* generate wasteful memory traffic, which can cause a degradation in performance.

Assumptions (A1) and (A2) may be relaxed by ptoviding extra hardware. See [Jaya87b] for details.

5. Semaphore Operations

A semaphore is a shared variable S together with the atomic operations $P(S)$ and $V(S)$ defined as follows: *P(S)*: <**while** $S \le 0$ **do** **skip**; $S := S-1$>; $V(S)$: <$S := S+1$>; (Instructions within the angle brackets are executed atomically). The variable S is initialized to one.

Most implementations of P and V require the continual checking of S, as implied by the *while* loop, before the P operation is successful. An alternative to this busy waiting on S on an unsuccessful P operation is to context switch to a different process. Context switching, however, requires intervention by the operating system, and consequently large overheads. Further, self scheduling and guided self scheduling algorithms [PoKu87, TaYe86], which are used in a number of application programs in a multiprocessor system, require some form of busy waiting.

We next describe P and V implemented with the new primitive. For simplicity, we deal with binary ($m = 2$) synchronization trees. We will also assume that assumption (A2) holds. Not all PEs need participate in the synchronization, however. Each node of the tree is a bit variable which is initially zero (reset).

5.1. The DSP Instruction

To implement the P operation, each PE executes the *D*istributed *S*ynchronizer *P* (*DSP*) instruction, the semantics of which are shown in Figure 5.1. The P operation for a process belonging to a PE is complete if it can set the node at the root of the synchronization tree. If a process finds that the root is already set, then another process is in the critical region.

In the following discussion, when we talk of a node at level i, it refers to the node at level i that lies between the PE of interest and the root. Note that by the unique path property, there is only one path from a PE to the root, and hence there is a single node on the path at any level. A request contains the address of the instruction and the address of the PE where it originates. Consider a request which has reached level i successfully by setting the node at level i. If the node at level $i+1$ has not been already set (by an earlier process), then the node at level $i+1$ is set and the node at level i is reset atomically. Otherwise, the process waits at level i. The atomic operation is accomplished easily, without starvation, in hardware [Jaya87b]. Nodes with their bit variables set represent processes waiting to enter their critical regions. We next describe a schedule to wake up waiting processes.

5.2. The DSV Instruction

To describe the *D*istributed *S*ynchronizer *V* (*DSV*) operation, we use another property that the class of interconnection networks described in Section 3.2 share: they can employ simple and distributed routing algorithms. Recall that $h = \log_2 n$, and let $a_h a_{h-1} \cdots a_1 a_0$ be the binary expansion of an integer j. Further, let PE_j have a request that is waiting at the node at level i to enter

```
Algorithm DSP (j, k);
/* Let $FL_{j1}$, $FL_{j2}$, ..., $FL_{jh}$ be the partially shared
nodes on the path from $PE_j$ to the root. $FL_{j0}$ is a bit variable
in $PE_j$ */
begin
1.  if k = h then enter critical region;
    else
2.     if $FL_{j,k+1} = 0$ then
       begin
3.        <$FL_{j,k+1}$ := 1;  if $k \ne 0$ then $FL_{jk}$ := 0;> /* atomically
done in the hardware */
4.        DSP(j, k+1);
       end;
end DSP;

$PE_j$ executes DSP(j, 0) to enter its critical region.
```

Figure 5.1 Semantics of the *DSP* Instfuction.

the critical region. The request stores the values a_i and a_{i+1} as part of its information at the node. The semantics of *DSV* are shown in Figure 5.2, and informally explained below. Let a process belonging to PE_k finish, and let $k_h k_{h-1} \cdots k_1 k_0$ be the binary expansion of k. By virtue of the semantics of *DSP*, if there are processes waiting at level i, then there are processes waiting at level $(i+1)$ also. Hence PE_k has to check only the children at the root of the synchronization tree. If PE_k is at a leaf of the left (right) subtree, then the switching element logic at the root enables a waiting process, if any, at the right (left) subtree (i.e., a PE whose address in the $(h+1)$th position is $\bar{k}_h$, the complement of k_h). If there is no such process waiting, then a process, if any, belonging to its own subtree is enabled. The awakened process repeats this procedure recursively using the appropriate position of the binary expansion of its PE number. Thus the process belonging to PE_j, when awakened at a node at level i, decides which new process should occupy the vacant node by examining j_i and the ith positions of the PE numbers of the processes at the children of the node.

```
Algorithm DSV (j, k);
/* Let $FL_{j1}$, $FL_{j2}$, ..., $FL_{jh}$ be the partially shared
nodes on the path from $PE_j$ to the root. $FL_{j0}$ is a bit variable
in $PE_j$. Let $a_h \cdots a_k \cdots a_0$ be the
binary expansion of $j$ */
begin
1.  if $FL_{a_h \cdots \bar{a}_k \ldots a_0, k-1} = 1$ then
    begin
2.      $FL_{a_h \cdots a_k \cdots a_0, k} := 1$;
3.      transfer process from $FL_{a_h \cdots \bar{a}_k \cdots a_0, k-1}$ to next higher level;
4.      $FL_{a_h \cdots \bar{a}_k \cdots a_0, k-1} := 0$;
5.      $DSV(newPE, k-1)$; /* newPE is the address of the PE on
which the new process runs */
  end
  else
6.  if $FL_{a_h \cdots a_k \cdots a_0, k-1} = 1$ then
    begin
7.      $FL_{a_h \cdots a_k \cdots a_0, k} := 1$;
8.      transfer process from $FL_{a_h \cdots a_k \cdots a_0, k-1}$ to next higher level;
9.      $FL_{a_h \cdots a_k \cdots a_0, k-1} := 0$;
10.     $DSV(newPE, k-1)$; /* newPE is the address of the PE on
which the new process runs */
    end
end DSV;
```

PE_j executes $DSV(j, h)$ after leaving its critical region.

Figure 5.2 Semantics of the *DSV* Instruction.

Example: Consider an 8–PE system with the synchronization tree as shown in Figure 5.3a. All the nodes in the tree are initially reset. The timing figure of Figure 5.3b shows a particular set of requests to the critical section and the order in which they are honored. In that figure, r stands for a request to enter the critical region, h for a request honored by the scheduler, and c for the completion of the use of the critical region. The prefix digit denotes the PE number. Observe that, for this example, PEs 5 and 7 get serviced out of turn. The order in which the requests are honored is readily seen by following the informal explanations given for *DSP* and *DSV*.

The hardware requirements at a switching element are a 3–bit register and simple logic (to perform the logical OR and the reset operations atomically). If the first bit of the 3–bit register at level i is set, then some process belonging to PE_k is waiting at this node. The other two bits then store the ith and the $(i+1)$st position of the binary expansion of k.

The following cases: a) relaxing assumption (A2), and b) the case of $m = 2^l$ $(l > 1)$, are considered in [Jaya87b].

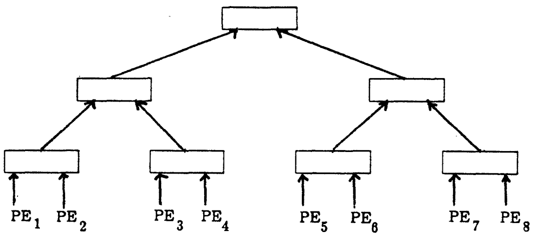


Figure 5.3a Illustration of *DSP* and *DSV*.

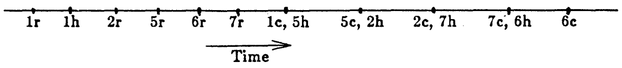


Figure 5.3b Illustration of *DSP* and *DSV* (contd).

5.3. Bounded Fairness

Many notions of fairness for concurrent systems have been proposed [Fran86]. In the context of (mutually exclusive) access to a shared resource (such as a critical region), fairness is synonymous with "starvation freedom". The P and V operations are used to ensure mutual exclusion. Many implementations of P and V require busy waiting on the semaphore variable, which, in turn, may lead to starvation. We show that *DSP* and *DSV* not only are starvation free, but satisfy the stronger notion of *bounded fairness* [JaDe86]. For our purposes, we define bounded fairness within the context of a scheduler (which may be centralized or distributed). A scheduler q is *k–bounded fair* if a process p wishing to enter its critical region is *guaranteed* to do so at one of the next k times that q schedules either p or a a process arriving after p (k is referred to as the *bounded fairness number*). For example, a FIFO scheduler is *1-bounded fair.*

We first make the following observations before we prove the bounded fairness of *DSP* and *DSV*.

Observation 5.1: The semantics of *DSP* ensure that a process wishing to enter its critical region either does so, or traverses to a level of the synchronization tree until it can no longer proceed.

Observation 5.2: From the above observation, we can infer that it is sufficient for a process leaving the critical region to activate another process to enter its critical region by examining the nodes at the children of the root of the synchronization tree.

Observation 5.3: Further, by virtue of the semantics of *DSV* and the above observations, a process that "vacates" an internal node (to travel to the next higher level of the synchronization tree) can also decide which new process should occupy the vacant node by examining the children of the node.

Theorem 5.1: For an n–PE multiprocessor, *DSV* is $(2n-(1+\log_2 n))$–bounded fair.

Proof: Let us append the n PEs as leaves of the synchronization tree and call the result the *extended synchronization tree.* Consider such a tree shown in Figure 5.4. Suppose a process belonging to PE_j sends a request r to enter the critical region. In the case of a conflict among the processes in the m $(m=2)$ subtrees at a node, the scheduling policy (i.e., the semantics of *DSV*) chooses the subtree which has *not* been most recently chosen. From observation 5.3, this selection can indeed be done by examining the children at the root of the subtree of interest. Hence, in the worst case, the request r accesses the root (i.e., enters its critical region) in at most as many tries (each entry into a critical region by *any* process is a try) as the number of nodes in the subtrees T_L and T_R, i.e., $2(n-1)$. The following cases arise:

Case 1: If r accesses the root in at most $(2n-(1+\log_2 n))$ tries, then the condition for bounded fairness is trivially satisfied.

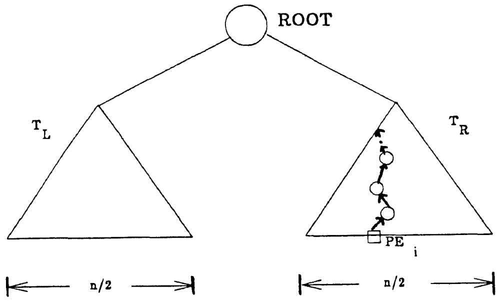


Figure 5.4 Extended Synchronization Tree for Proving Bounded Fairness.

Case 2: If r accesses the root in exactly $2(n-1)$ tries, then at *every* node on the path from PE_j to the root, *a PE from the subtree not belonging to the subtree containing PE_j was chosen.* By induction it is clearly seen that at every node on the path from PE_j to the root, PE_j should have a request sent *before* r. Hence the number of requests in the tree before r is at least $\log_2\frac{n}{2}$ (i.e., the height of the subtree whose leaves are one level higher than the leaves of T_R). Since the request r can be satisfied in $2(n-1)$ tries, and there were at least $\log_2\frac{n}{2}$ requests before r, k, the bounded fairness number is $2(n-1) - \log_2\frac{n}{2} = (2n-(1+\log_2 n))$.

Case 3 (Sketch): If r accesses the root in $(2n-(1+\log_2 n)) < i < 2(n-1)$ tries, then by an argument similar to that used in case 2, it can be shown that there are at least $(i-(2n-(1+\log_2 n)))$ nodes on the path between PE_j and the root that contain requests sent from PE_j before r. The bounded fairness result immediately follows. ■

Note: The bounded fairness result may be extended to m-ary synchronization trees. In [Jaya87b] it is shown in such a case that *DSV* is $(\frac{nm-1}{m-1} - \log_m n)$-bounded fair.

6. Further Remarks

Our implementation of the synchronization operations has a few limitations. They are the following:

(1) *Number of Synchronization Operations:* The present implementation of the distributed synchronizer allows only a limited number of concurrent synchronization operations.

(2) *Restricted Schedules*: In the case that barrier and reporting are required only among $n_1(n_1 < n)$ PEs, the root of the synchronization tree may be placed at an intermediate stage of the interconnection network. In such a case, not every subset of PEs could use the distributed synchronizer to perform these operations. This observation translates to a restriction on processor scheduling.

(3) *Process Migration*: In our barrier and reporting schemes, processes may not migrate across PEs. This is not a serious limitation (except when fault tolerance is to be provided), since, for efficiency reasons, most multiprocessor operating systems do not allow processes to migrate.

These limitations may be partially overcome. See [Jaya87b] for details.

7. Conclusion

Synchronization and communication overheads become important performance criteria in large multiprocessors. A number of researchers have recognized that synchronization requirements lead to serious performance degradation in such systems. In this paper we have introduced a new synchronization primitive based on the notion of partially shared variables. Using this primitive, we have shown that the commonly required synchronization operations may be efficiently performed with practically no overheads. Though this implementation of the distributed synchronizer does not wholly remove the need for conventional synchronization operations based on the locking of shared variables, our paper shows a promising way to distribute synchronization operations and to exploit the power of partially shared variables. An interesting extension of this work is to make the distributed synchronizer fault tolerant. An important research area is the feasibility of architectures based on the notion of hierarchical memories.

Acknowledgements: The author is thankful to Professor Duncan Lawrie for initial discussions on the subject. He is indebted to Professor Michael Loui for constructive comments on the form and content of this paper. The author is also thankful to one of the anonymous referees whose comments have helped improve the quality of the presentation of this paper.

REFERENCES

[Axel86] T. S. Axelrod, "Effects of Synchronizations Barriers on Multiprocessor Performance," Parallel Computing, Vol. 3, pp. 129–140, May 1986.

[Fran86] N. Francez, "Bounded Fairness," Springer–Verlag New York Inc., 1986.

[GLee86] G. Lee, "Some Issues in General Purpose Shared Memory Multiprocessing: ParallelismExploitation and Memory Access Combining," Ph.D. thesis, Center for Supercomputing Research and Development Report No. 589, June 1986.

[GGKM83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. M. McAuliffe, L. Rudolph, M. Snir, "The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer," IEEE Trans. on Computers, Vol. C–32, No. 2, 1983.

[JaDe86] D. N. Jayasimha, N. Dershowitz, "Bounded Fairness," Rpt. No. 615, Center for Supercomputing Research and Development, University of Illinois, Dec. 1986.

[Jaya87a] D. N. Jayasimha, "Parallel Access to Synchronization Variables," Proc. International Conference on Parallel Processing, pp. 97–100, Aug. 1987.

[Jaya87b] D. N. Jayasimha, "Distributed Synchronizers," Rpt. No. 713, Center for Supercomputing Research and Development, University of Illinois, Nov. 1987.

[Kuck84] D. J. Kuck, *et al.*, "Cedar," Proc. of COMPCON, Spring 1984.

[Pfis85] G. F. Pfister, *et al.*, "The IBM Research Parallel Processor Prototype (RP3): Introductionand Architecture," Proc. International Conference on Parallel Processing, pp. 764–771, 1985.

[PfNo85] G. F. Pfister, V. A. Norton, "'Hot Spot' Contention and Combining in Multistage InterconnectionNetworks," Proc. International Conference on Parallel Processing, pp. 790–797, 1985.

[PoKu87] C. D. Polychronopoulous, D. J. Kuck, "Guided Self–Scheduling: A Practical Scheduling Scheme for ParallelSupercomputers," IEEE Trans. on Comput., vol. C–36, no. 12, pp. 1425–1439, Dec. 1987.

[TaYe86] P. Tang, P. C. Yew, "Processor Self–Scheduling for Multiple–Nested Parallel Loops," Proc. International Conference on Parallel Processing, pp. 528–535, 1986.

[YeTL86] P. C. Yew, N. F. Tzeng, D. H. Lawrie, "Distributing Hot Spot Addresssing in Large Scale Multiprocessors," Proc. International Conference on Parallel Processing, pp. 51–58, Aug. 1986.

[ZhYe84] C. Q. Zhu, P. C. Yew, "A Synchronization Scheme and its Application for Large MultiprocessorSystems," Proc. International Conference on Distributed Computing Systems, pp. 486–493, 1984.

Graph–based partitioning of matrix algorithms for systolic arrays: application to transitive closure

Jaime H. Moreno and Tomás Lang *
Computer Science Department
University of California, Los Angeles
3680 Boelter Hall
Los Angeles, Calif. 90024

Abstract. We propose a technique to partition algorithms for execution in systolic arrays, based on transformations to the dependency graph of algorithms. We illustrate this method through its application to the computation of transitive closure of a directed graph. We derive linear and two–dimensional structures for such algorithm that exhibit maximal utilization, no overhead due to partitioning and simple control. In the process, we obtain a graph suitable for an array for fixed–size problems that exhibits better characteristics than arrays previously proposed for this algorithm. Our method also allows evaluating trade–offs among implementations.

Introduction

The implementation of matrix algorithms as collections of regularly connected processing elements (arrays of PEs) has been extensively studied lately. Many applications require processing large matrices for which it is not feasible to build an array of the required size, while others require solving problems of variable size using the same array. In such cases, it becomes necessary to decompose the problem into sub–problems so that the sub–problems fit into a target array. This is known as *partitioning* the algorithm and has been studied by many researchers [1]–[5].

In this paper, we summarize a partitioning technique based on the dependency graph of algorithms. A complete description of the technique can be found in [6]. This is a transformational approach, that uses a fully–parallel dependency graph as the description of the algorithm. Such a graph is first transformed to remove properties not desirable for an implementation (i.e., data broadcasting, bi–directional data flow) and converted into a graph suitable for partitioning (i.e., with simple communication requirements). The resulting graph is mapped onto the target array. The transformations are performed taking into account issues such as I/O bandwidth, throughput, delay, and utilization of PEs. We illustrate the technique through its application to the design of arrays for partitioned computation of the transitive closure of a directed graph. We derive and evaluate linear and two–dimensional structures to compute such algorithm. These arrays exhibit maximal utilization, no overhead and simple control. In addition, we show that an intermediate graph used by the methodology is suitable for implementation of fixed–size arrays for transitive closure, with better characteristics than arrays previously proposed for such computation.

We have applied our partitioning technique to several algorithms for matrix computations, among them LU–decomposition, QR–decomposition, and Faddeev algorithm [7]. Our results show that the graphical nature of our approach makes it easier to use than methodologies based on mathematical expressions proposed in the literature. Moreover, the method allows evaluating trade–offs between linear and two–dimensional arrays for partitioned execution of algorithms. This technique is an extension to one for the design of arrays for fixed–size problems that we have previously proposed [8,9].

Partitioning the computation of transitive closure of a directed graph has been recently addressed by Núñez and Torralba [10]. They propose an algorithm and partition it through decomposition into a block–algorithm. Although they do not address the details of an implementation, their algorithm requires rather complex control to chain the different sub–problems.

Graph–based partitioning

Partitioning consists of mapping the computation of an algorithm with large–size data onto an array smaller than the size of the data. Three basic approaches have been proposed to achieve such mapping:

- coalescing [1,5]
- cut–and–pile [1]
- decomposition into subalgorithms [1]

The relative merits of these approaches are discussed in [6].

We summarize here a partitioning technique based on the dependency graph of algorithms that uses the cut–and–pile approach due to its generality and smaller memory requirements. A complete description of such technique is given in [6]. This partitioning procedure is as follows:

1. Transform the dependency graph to remove properties undesirable for an implementation, such as data broadcasting or bi–directional data flow. Procedures for these purposes have been presented in [8,9].
2. Transform the graph obtained in (1) into a new graph, which we call the *G–graph*, by collapsing groups of nodes into new nodes (*G–nodes*). The objective of this transformation is to obtain a graph more suitable for partitioning, that is, with simple communication requirements.

   Criteria to perform the selection of primitive nodes composing a G–node are reported in [6].
3. Map G–nodes to a target array with m cells by scheduling sets of m neighbor G–nodes (a *G–set*) for concurrent computation. G–sets scheduled successively are executed in overlapped (pipelined) manner in the array. The selection of G–sets depends on the structure of the target

*J. Moreno has been supported by an IBM Computer Sciences Fellowship. This research has also been supported in part by the Office of Naval Research, Contract N00014–83–K–0493 "Specifications and Design Methodologies for High–Speed Fault–Tolerant Algorithms and Structures for VLSI"

array. In addition, for maximal utilization, all nodes in a G-set should have the same computation time.

The G-graph obtained with our procedure can be directly used to implement an array for a fixed-size problem. However, since the G-graph might be composed of nodes with different computation time, its direct implementation could lead to low utilization of cells.

Partitioned Computation of Transitive Closure

We present now the application of the proposed partitioning technique to the design of arrays to compute transitive closure of a directed graph. We first describe briefly the algorithm and then apply the three-step procedure indicated above.

The transitive closure problem

A directed graph G is a tuple $G(V, E)$, where V is the set of nodes and E is the set of edges in the graph. G can be described by the *adjacency matrix* A, where element $a_{ij} = 1$ if there is an edge from node i to node j or if $i = j$, otherwise $a_{ij} = 0$. A directed graph $G^+(V, E^+)$ is called the *transitive closure of* G if it has the same vertex set as G and has an edge from node v to node w if and only if there is a path of length zero or more from v to w in G. G^+ can be described by the *adjacency matrix* A^+.

The computation of the transitive closure of a graph is usually performed by Warshall's algorithm [11]. Given the adjacency matrix A, then A^+ is obtained through the application of the following recurrence:

```
For $k$ from 1 to $n$
For $i$ from 1 to $n$
For $j$ from 1 to $n$
    $x_{ij}^{k} \leftarrow x_{ij}^{k-1} \oplus (x_{ik}^{k-1} \otimes x_{kj}^{k-1})$
```

In this expression, $X^0 = A$, $A^+ = X^n$ and the operators $\oplus$ and $\otimes$ stand for binary-OR and binary-AND, respectively.

The fully-parallel dependency graph [8] of the transitive closure algorithm is shown in Figure 1, for a problem of size $n = 4$ (i.e., to compute the transitive closure of a directed graph with four nodes). The graph has four levels, where each level corresponds to one iteration of the outermost index in the algorithm above.

Some evaluations of the expression in the algorithm above do not change the corresponding x_{ij}^{k}. In particular, the value of a diagonal element in the adjacency matrix is always 1, because a node in the directed graph is always adjacent to itself. In addition, for $k = i$ or $k = j$ one of the two operands in $x_{ik}^{k-1} \otimes x_{kj}^{k-1}$ becomes x_{kk}^{k-1} which is a diagonal element and thus always equal to 1. Consequently, the result from the $\otimes$ operation is equal to the second operand, the $\oplus$ operator gets two identical operands (x_{ik}^{k-1} or x_{kj}^{k-1}) and the result is that same operand. These properties can be utilized to simplify the design of arrays to compute the transitive closure and to reduce the complexity of the algorithm since fewer operations need to be performed. Nodes surrounded by dashed areas in Figure 1 correspond to superfluous nodes (i.e., they do not need to be computed).

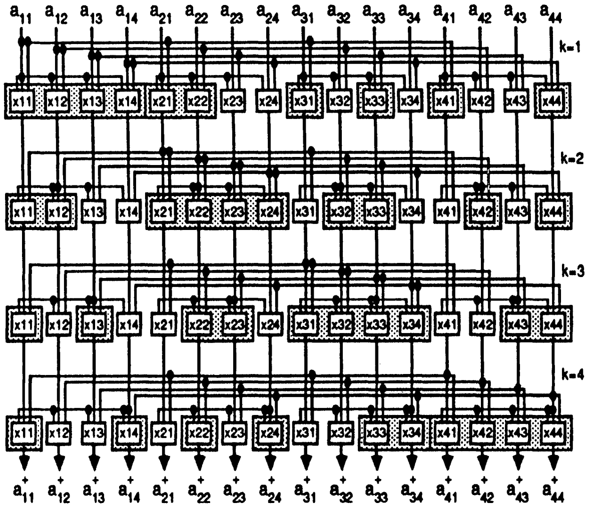

a11 a12 a13 a14 a21 a22 a23 a24 a31 a32 a33 a34 a41 a42 a43 a44
k=1
x11 x12 x13 x14 x21 x22 x23 x24 x31 x32 x33 x34 x41 x42 x43 x44
k=2
x11 x12 x13 x14 x21 x22 x23 x24 x31 x32 x33 x34 x41 x42 x43 x44
k=3
x11 x12 x13 x14 x21 x22 x23 x24 x31 x32 x33 x34 x41 x42 x43 x44
k=4
x11 x12 x13 x14 x21 x22 x23 x24 x31 x32 x33 x34 x41 x42 x43 x44
a+11 a+12 a+13 a+14 a+21 a+22 a+23 a+24 a+31 a+32 a+33 a+34 a+41 a+42 a+43 a+44

Figure 1: Fully-parallel dependence graph of transitive closure

Arrays for partitioned computation of transitive closure

We apply now our partitioning procedure to the computation of the transitive closure of a directed graph. The fully-parallel dependency graph shown in Figure 1 is not suitable for implementation, because it exhibits broadcasting of data and complex communication requirements. We address these issues first, according to the procedure described previously.

There are two types of data broadcasting in Figure 1. At the *k-th* level of the graph, data elements from row k of the matrix are broadcasted to all other rows. Moreover, the *k-th* element of each row of the matrix is broadcasted to all other elements within each row. Because of the varying pattern of broadcasting, other researchers have considered transitive closure an *irregular* algorithm [13].

We transform the graph replacing broadcasting by pipelining, as suggested in [8]. Given that the *k-th* row of the adjacency matrix remains unchanged at the *k-th* level of the graph, we remove the nodes corresponding to updating those values and draw the flow of data for such row horizontally and intersecting with the flow of data of the other rows of the matrix. Such modification is shown in Figure 2. Data which is evaluated at each level of the graph flows vertically, while the data element broadcasted within each row of the matrix flows diagonally through the row.

Because of the varying source of broadcasting, the transformed graph in Figure 2 exhibits bi-directional flow of data. However, this bi-directional flow can be eliminated by moving nodes dependent on the broadcasted data to one side of the source of broadcasting. We have described such transformation as an approach to solve this problem in dependence graphs [9]. In this case, the transformation is applied in two steps: first, nodes to the left of sources of horizontal broadcasting are flipped to the right end of each row of the graph. Then, nodes above sources of diagonal broadcasting are flipped to the bottom end of such diagonals. In addition, delay nodes are placed at the boundaries of the graph with the same depen-

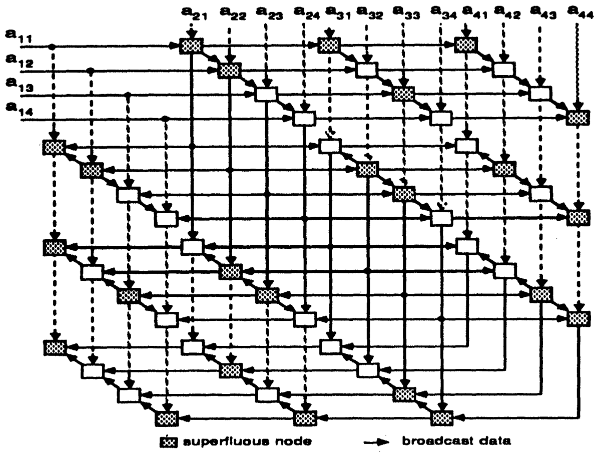


Figure 2: Replacing broadcasting by pipelining

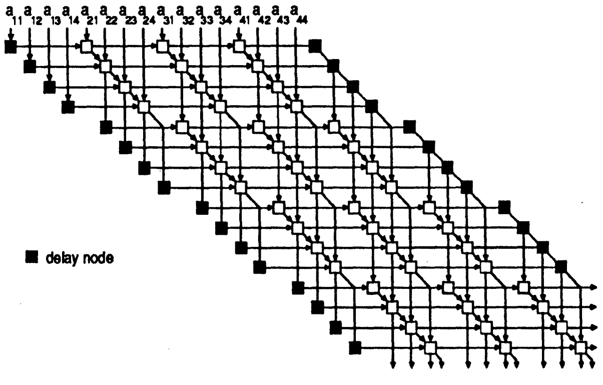

a11 a12 a13 a14 a21 a22 a23 a24 a31 a32 a33 a34 a41 a42 a43 a44
delay node

Figure 3: Transformed transitive closure dependence graph

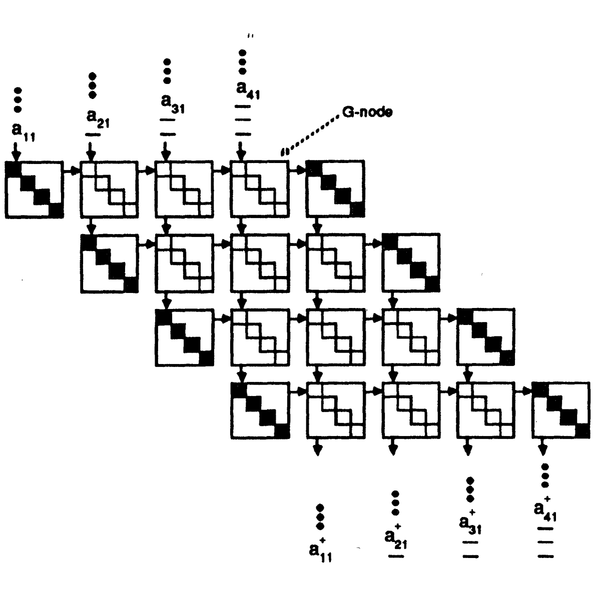


Figure 4: Transitive closure G-graph

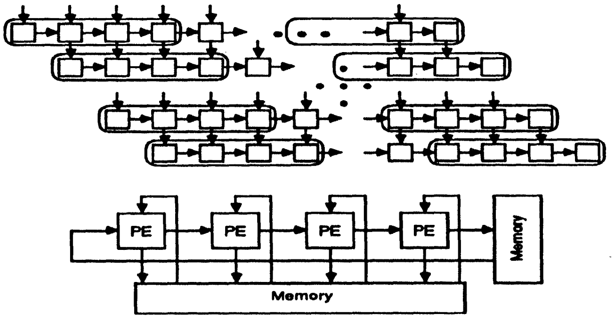


Figure 5: Mapping G-graph onto a linear array

dency structure that dominates the graph, as proposed in [9]. The resulting graph is shown in Figure 3.

Once properties of the original dependency graph not suitable for partitioning have been eliminated, we apply the remaining of our partitioning procedure. We first transform the graph into a G-graph by selecting sets of primitive nodes in such a way as to reduce communication requirements and obtain G-nodes with the same computation time. In this case, diagonal paths are a good alternative for grouping, because nodes in such paths communicate among themselves in a repetitive manner and all paths have the same number of primitive nodes. The result of performing such grouping is the G-graph shown in Figure 4.

As an aside, Figure 4 is suitable for direct implementation as an array for fixed-size problems. Such an array achieves maximum utilization because all G-nodes have the same computation time and the algorithm is computed in pipelined manner in the array. Throughput is $1/n$ because the computation time of G-nodes is n cycles. Successive instances of the algorithm can be chained without restrictions. This array is simpler than the one proposed in [13] because it has a single communication path between cells and no control complexity. Furthermore, data transfers and computations are overlapped while the array proposed in [13] requires that "data be first loaded in the nodes and then reused for a period of n cycles" so that "certain control is required in the systolic array."

Another advantage of the array for fixed-size problems described above relies on the simplicity of its derivation through the graph-based methodology. This is in contrast with the scheme in [13], which uses a rather complex mathematical approach. Furthermore, the G-graph in Figure 4 can be collapsed into a linear structure by grouping each horizontal path into a single node. The resulting graph can be directly mapped onto a linear array with throughput $[n(n+1)]^{-1}$ and all cells fully utilized.

Arrays to compute the transitive closure in partitioned mode can be derived directly from the G-graph in Figure 4, as we describe next.

Linear array

Let's assume that we want to partition the computation of transitive closure of a directed graph with n nodes so that it fits in a linear structure with m cells, where $m \ll n$. We map G-sets from the transformed graph onto a linear array by selecting G-sets of m G-nodes from horizontal paths, as shown in Figure 5 for $m = 4$. Intermediate results from G-sets are saved in external memories. Such data is available at the boundary of the set, so that saving it in external memories is straight-forward.

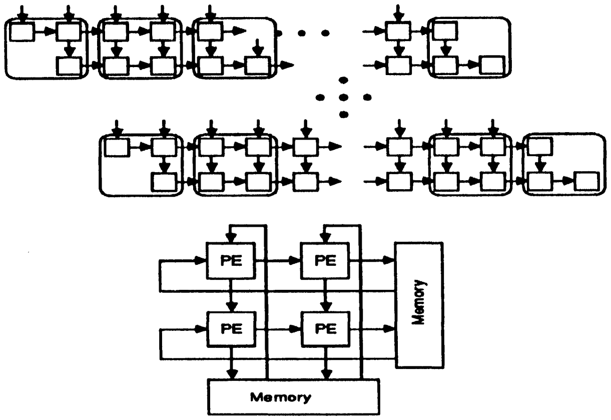


Figure 6: Mapping G–graph onto two–dimensional array

The structure resulting from this approach enjoys maximal utilization because all G–nodes executed concurrently have the same computation time, except when executing boundary sets in some horizontal paths that might not use all cells in the array. The number of connections to external memories is $m+1$.

Two–dimensional array

Mapping the G–graph for execution in a two–dimensional structure with m cells requires to simulate a triangular array and a square array, because those are the major components of the G–graph. Both requirements can be fulfilled in a square array. G–sets are selected as square blocks of $\sqrt{m}$ by $\sqrt{m}$ nodes, excepting sets at the boundaries of the G–graph which are composed of triangular blocks of G–nodes. As in the linear case, intermediate results are saved in external memories. The structure resulting from this approach is shown in Figure 6 for $m = 4$. Utilization of this array is maximal, except when executing boundary sets because such sets do not use all cells in the array. The number of connections to external memories is $2\sqrt{m}$.

To use the arrays obtained above it is necessary to schedule the execution of G–sets. Such scheduling is discuss in detail in [6] and it is shown that *linear and two–dimensional arrays require the same I/O bandwidth from the host.*

Conclusions

We have proposed a technique to partition algorithms for execution in arrays, based on dependency graphs of algorithms. We described the application of such technique to the computation of transitive closure of a directed graph. Through this example, we have shown that the approach is general and powerful. This technique is suitable for a class of important matrix algorithms, produces implementations with maximal utilization of cells and no overhead due to partitioning, and allows evaluating trade–offs between linear and two–dimensional structures. Moreover, this graph–based approach is simpler to use than schemes based on mathematical expressions.

We derived linear and two–dimensional arrays for partitioned computation of transitive closure. In the process, we have obtained a dependence graph which is suitable for implementation of a fixed–size array for transitive closure, with better characteristics than structures previously proposed for this algorithm.

In [6], we describe other issues in partitioning algorithms, in particular trade–offs between linear and two–dimensional structures. We show there that, with the same number of cells, linear arrays are simpler, have the same throughput and require the same I/O bandwidth from the host than two–dimensional ones, and might exhibit better utilization. Moreover, linear arrays are more advantageous than two–dimensional ones because they are better suited to incorporate fault–tolerant capabilities. Consequently, we conclude that *linear arrays offer better performance and implementation than two–dimensional arrays for partitioned execution of algorithms.*

References

[1] J. Navarro, J. Llaberia, and M. Valero, "Partitioning: an essential step in mapping algorithms into systolic array processors," *IEEE Computer*, vol. 20, pp. 77–89, July 1987.

[2] D. Moldovan and J. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. C-35, pp. 1–12, Jan. 1986.

[3] S. Kung, *VLSI Array Processors*, pp. 374–382. Prentice Hall, 1988.

[4] K. Hwang and Y. Cheng, "Partitioned matrix algorithms for VLSI arithmetic systems," *IEEE Transactions on Computers*, vol. C–31, pp. 1215–1224, Dec. 1982.

[5] J. Nash, S. Hansen, and K. Przytula, "Systolic partitioned and banded linear algebraic computations," in *SPIE Real-Time Signal Processing IX*, pp. 10–16, 1986.

[6] J. Moreno and T. Lang, "Graph-based partitioning of matrix algorithms for systolic arrays," Technical Report CSD-880015, Computer Science Department, University of California Los Angeles, March 1988.

[7] J. Moreno and T. Lang, "On partitioning the Faddeev algorithm," in *International Conference on Systolic Arrays*, May 1988.

[8] J. Moreno and T. Lang, "Design of special–purpose arrays for matrix computations: preliminary results," in *SPIE Real-Time Signal Processing X*, pp. 53–65, 1987.

[9] J. Moreno and T. Lang, "Reducing the number of cells in arrays for matrix computations," Technical Report CSD-880014, Computer Science Department, University of California Los Angeles, March 1988.

[10] F. Núñez and N. Torralba, "Transitive closure partitioning and its mapping to a systolic array," in *International Conference on Parallel Processing*, pp. 564–566, 1987.

[11] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[12] J. Moreno, "A proposal for the systematic design of arrays for matrix computations," Technical Report CSD–870019, Computer Science Department, University of California Los Angeles, May 1987.

[13] S. Kung, *VLSI Array Processors*, pp. 248–266. Prentice Hall, 1988.

Iterative Sparse Linear System Solvers on Warp

P. S. Tseng

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Warp is a systolic computer developed by CMU and manufactured by GE. The machine has 10 or more linearly connected cells. Each cell in the array is capable of performing 10 million floating point operations per second (10 MFLOPS). The 10-cell array can achieve a peak performance of 100 MFLOPS. This paper describes parallel iterative sparse linear system solvers developed for the Warp systolic computer. For general sparse linear systems, Warp achieves 12.5 MFLOPS in sparse matrix vector multiplication, which competes with supercomputers such as Cray-1S and Cyber-205. We implemented the general sparse linear system solver IC-PCCG (Incomplete Choleski Pre Conditioned Conjugate Gradient method) using the sparse matrix vector multiplication kernel. The solver was exercised on sparse linear systems derived from production finite element applications. Speedups of more than 100 over the VAX/780 with floating point accelerator are achieved. For solving regular sparse linear systems, domain partitioning is used to speedup solving finite difference equations on a regular mesh. For a model problem of Laplace's equation on a square mesh of 500 by 500 unknowns, Warp is able to achieve 14.6 MFLOPS using the generic SOR relaxation scheme and 49.4 MFLOPS using the 2-color SOR relaxation scheme.

1. Introduction

Large sparse linear systems of order 10^4 to 10^5 frequently arise in large scale scientific and engineering analysis such as computational fluid dynamics, structural mechanics, electronic device simulation and electric magnet field analysis. Direct methods designed for solving dense linear systems such as LU and Choleski decomposition are impractical for solving very large sparse systems, because very large storage is required. Driven by demands from applications, extensive efforts have been invested in the search for practical solvers for large sparse linear systems. There are two approaches. One is to pick an appropriate direct method and adapt it to exploit the sparsity of linear systems. Typical adaptation strategies involve the intelligent use of data structure and special pivoting strategies that minimize fill-in of the coefficient matrix [5, 6]. In contrast to the direct methods are the iterative methods. These methods start with an initial guess to the solution and generate a sequence of successively improved solutions until it converges to the desired solution within the accepted tolerance. Iterative methods are much more efficient for very large sparse systems because the coefficient matrix is not decomposed and remains unchanged throughout iterations, therefore no fill-in is created.

In this paper, several iterative sparse linear system solvers on the CMU Warp machine are described. We first give a brief review of the Warp machine and its architectural strength in supporting sparse matrix computations. Secondly, we consider the crucial kernels used in solving general sparse linear systems, sparse matrix vector multiplication and sparse triangular system solving. Implementation of these kernels on Warp are described and compared with vector supercomputers such as Cray-1S and Cyber-205. These kernels were integrated into the general sparse linear system solver IC-PCCG and exercised on sparse linear systems derived from production finite element applications. Speedups of more than 100 over the VAX/780 with floating point accelerator are achieved. Finally, we consider the problem of solving finite difference equations on a square mesh. For the model problem of Laplace's equation on a square mesh of 500 by 500 unknowns, Warp is able to achieve 14.6 MFLOPS using the generic SOR relaxation scheme and 49.4 MFLOPS using the 2-color SOR relaxation scheme.

2. The Warp Machine

A brief overview of the Warp machine is given below. (for architectural details, programming tools, and its other applications see [1, 2, 3]) The Warp machine has three components - the Warp processor array, or simply Warp array, the interface unit, and the host, as depicted in Figure 2-1. The Warp processor array performs the bulk of the computation. The interface unit handles the input/output between the array and the host. The host has two functions: carrying out high-level application routines and supplying data to the Warp processor array.

The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533.

The Warp processor array is a programmable, linear systolic array, in which all processing elements (Warp cells) are identical. Data flow through the array on two data paths (X and Y) (as shown in the Figure 2-1). Each Warp cell contains one floating-point multiplier, one floating point ALU and one integer ALU. The floating-point units can deliver up to 5 MFLOPS each. This performance translates to a peak processing rate of 10 MFLOPS per cell or 100 MFLOPS for a 10-cell processor array. A 32K-word memory is provided for resident and temporary data storage. The datapath of a Warp processor cell is shown in Figure 2-2. The host is a general purpose computer (currently a Sun workstation, with added MC68020 cluster processors for I/O and control of the Warp array). It is responsible for executing high-level application routines as well as coordinating all the peripherals.

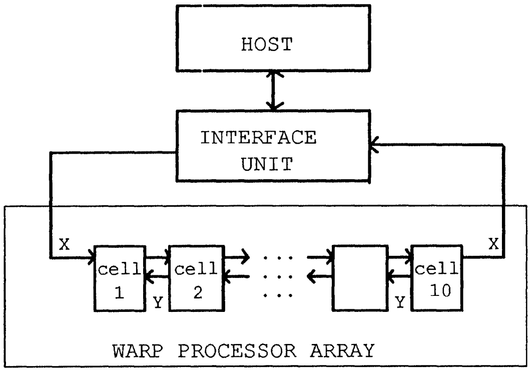


Figure 2-1: The Warp systolic computer

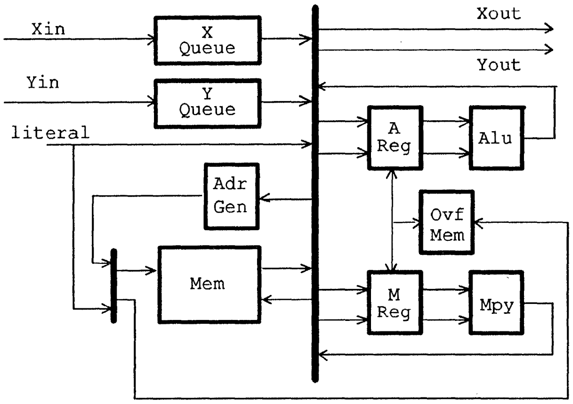


Figure 2-2: Warp cell datapath

A feature that distinguishes the Warp cell from many other processors of similar computation power is its high intercell communication bandwidth - an important characteristic for systolic arrays. Each Warp cell can transfer up to 20 million words (80 Mbytes) to and from its neighboring cells per second. We have been able to implement this high bandwidth communication link with only modest engineering efforts, because of the simplicity of the linear interconnection structure and clocked synchronous communication between cells. This high inter-cell communication bandwidth makes it possible to transfer large volumes of intermediate data between neighboring cells and thus supports fine grain problem decomposition. For communicating with the outside world, the Warp array can sustain a 20 Mwords/sec peak transfer rate. In the current setup, the host can only support up to 2 Mwords/sec transfer rates.

3. General Sparse Linear Systems

The compact matrix storage structure makes sparse matrix computations different from those for dense matrices. Figure 3-1 shows a widely used storage structure for general sparse matrices. The A array stores the non-zero elements of the matrix, the JA array stores the column index of each non-zero elements, and the IA array is an index array which points the starting element of each row in the A and JA arrays. This compact matrix format is used in general sparse matrix packages such as the Itpack [8].

We identify two sparse matrix kernels which dominate the computation of the iterative solvers under our consideration. They are

- sparse matrix vector multiplication;
- sparse triangle system solving.

$$\begin{pmatrix} 1 & 2 & 0 & 4 & 0 & 0 \\ 0 & 1 & 13 & 0 & 0 & 16 \\ 21 & 0 & 1 & 0 & 0 & 0 \\ 0 & 32 & 0 & 1 & 0 & 36 \\ 0 & 42 & 0 & 0 & 1 & 0 \\ 51 & 0 & 53 & 0 & 0 & 1 \end{pmatrix}$$

```
A  = [ 1, 2, 4; 1,13,16;21, 1;
      32, 1,36;42, 1;51,53, 1]
JA = [ 0, 1, 3; 1, 2, 5; 0, 2;
       1, 3, 5; 1, 4; 0, 2, 5]
IA = [ 0, 3, 6, 8,11,13]
```

Figure 3-1: Sparse matrix storage

3.1. Sparse Matrix Vector Multiplication

Consider the algorithm for multiplying a sparse matrix with a dense vector $\vec{y}=A\vec{x}$:

```
for i := 0 to n-1 do begin
  jbgn := IA[i] ; jend := IA[i+1] - 1;
  sum := 0.0;
  for j := jbgn to jend do begin
    sum := sum + A[j] * x[JA[j]];
  end
  y[i] := sum;
end
```

The algorithm steps through the sparse matrix row by row and does an inner product of the sparse row vector with the dense vector $\vec{x}$. The inner product computation is optimized by collecting elements from the dense vector indexed by the sparse vector to avoid multiplication and addition with zeros. This process of randomly collecting elements from a dense vector to match a sparse vector is known as the gather operation. The sparse matrix vector multiplication algorithm is an example where the innermost loop is sequential while the outer loop is completely parallel. To parallelize the computation, we simply distribute the rows of the matrix to the 10-cell array by interleaving, that is, cell i has rows $10k+i$, for $0 \geq k \geq \lfloor n/10 \rfloor$. The dense vector is duplicated on all the cells. Because of the 7-stage pipelined floating point adder, a Warp cell can only do the sparse dot product step at the rate of one every 8 cycles, that is, 1.25 MFLOPS out of its 10MFLOPS peak performance. The 10-cell Warp array can achieve 12.5 MFLOPS in sparse matrix vector multiplication. This performance figure is as good as or better than supercomputers such as Cray-1S (Gather: 5 to 11 MFLOPS, Peak 210 MFLOPS) and Cyber-205(Gather: 4 to 17 MFLOPS, Peak 800 MFLOPS) [4]. Note that, Warp is a single precision machine while the performance on CRAY-1S and Cyber-205 are double precision results. The relatively bad performance on vector computers is due to their heavily pipelined memory system and vector oriented processor units, which do not perform well in random indirect addressing and short vector computations.

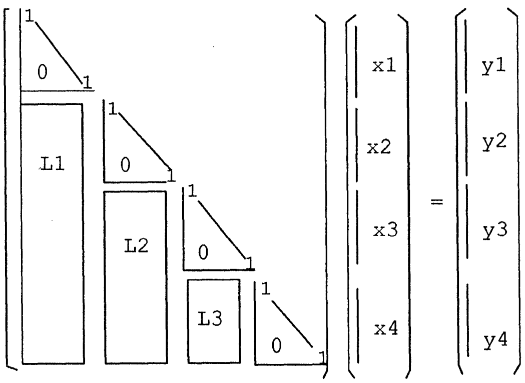
3.2. Sparse Triangular System Solving

Sparse triangular system solving is an inherently sequential process. Consider the algorithm for solving sparse lower triangular system $A\vec{x}=\vec{y}$:

```
for i := 0 to n-1 do begin
  j := IA[i];
  sum := y[i];
  while ( JA[j] < i) do begin
    sum := sum - A[j] * x[JA[j]];
    j := j+1;
  end
  x[i] := sum;
end
```

and similarly the algorithm for solving sparse upper triangular system $A\vec{x}=\vec{y}$:

```
for i := n-1 to 0 do begin
  j := IA[i+1]-1;
  sum := y[i];
  while (JA[j] > i) do begin
    x[i] := x[i] - A[j] * x[JA[j]]
    j := j -1;
  end
  x[i] := sum;
end
```



L1,L2,L3are general sparse matrices.

Figure 3-2: p-color ordered sparse triangular system, p=4

The innermost loops of these algorithms are gather operation and its outer loops are strictly sequential, which is not the case in matrix vector multiplication. The technique of multicolor reordering suggested in [11, 10] is used to restructure the sparse matrix and parallelize the computation of sparse triangular system solving. As shown in Figure 3-2, a p-color sparse triangular matrix has p identity blocks along the diagonal. Note that they are blocks of identity matrices, not blocks of identical size. A triangular system with such a structure can be solved in $p-1$ steps instead of $n-1$ steps, where n is the degree of the system. Each step of the solving is a sparse matrix vector multiplication, for example, Figure 3-3 shows the 3 steps for solving a 4-color triangular system. Since sparse matrix vector multiplication can be done in parallel, the sparse triangular system solving is thus parallelized.

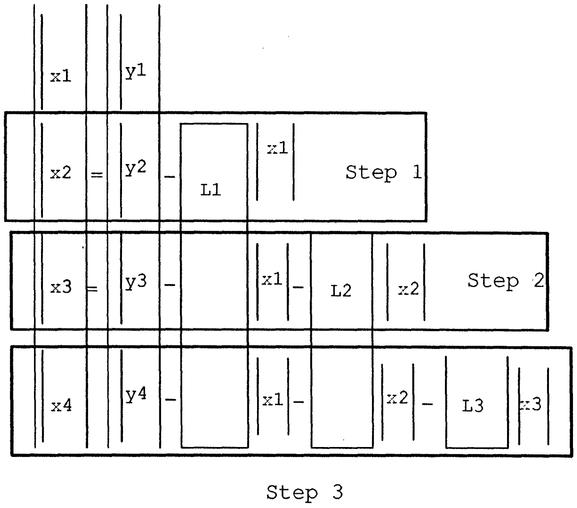


Figure 3-3: Forward solving : a 4 color ordered system

3.3. The IC-PCCG Solver

The Conjugate Gradient (CG) method was developed by Hestenes and Stiefel in 1952 [7] and subsequently widely used for solving minimization problems. Only since the 70's has the CG method been used for solving linear systems of equations with the symmetric positive definite (SPD) property. Its success is connected with the development of the Pre Conditioned CG (PCCG) iterations. The Incomplete Choleski precondion (IC-PCCG) [9] is one of the most successful general purpose precondition strategies popularly used in practice.

Let A be a symmetric positive definite (SPD) n by n sparse matrix. We want to solve a linear system with $A\vec{x} = \vec{b}$. If we define the error functional as

$$F(\vec{z}) = 1/2(\vec{x}-\vec{z})^TA(\vec{x}-\vec{z}) = 1/2(\vec{r})^TA^{-1}(\vec{r})$$

where $\vec{x}-\vec{z}$ is the error vector and the residual vector $\vec{r}$ is defined as $\vec{r} = \vec{b}-A\vec{z}$. This functional is minimized by the exact solution of $A\vec{z} = \vec{b}$. The CG method prescribes how to choose a sequence of approximations $\vec{x}_k$ such that the functional $F(\vec{x}_k)$ is minimized in an optimal way. In the steepest descent method the new approximation $\vec{x}_{k+1}$ is found in the direction of the gradient, which is the residual vector $\vec{r}_k$.

$$F(\vec{x}_{k+1}) = min_{\alpha} F(\vec{x}_k + \alpha\vec{r}_k)$$

The CG method converges in at most n iterations in the absence of round-off errors, the convergence rate is strongly determined by the clustering of the eigen values of the A matrix. The generic CG method is not practical for applications because of its slow convergence rate. The PCCG method is thus introduced, which instead of solving the system $A\vec{x} = \vec{b}$, solves the preconditioned system

$$(M^{-1}A)\vec{x} = (M^{-1}\vec{b})$$

where M is the precondition matrix with the properties that

- M is positive definite
- $M^{-1}A$ has better spectral properties than those of A, that is, a smaller spectral radius and more clustered eigen values.
- It is relatively cheap to solve a system with M, $M\vec{z} = \vec{d}$.

Although the detailed theory of PCCG may be complicated, it turns out that the approximation of $\vec{x}_k$ can be simply computed by the following iterative algorithm with ζ as the stopping criterion.

```
Initialization:
$\vec{x}_0 = \vec{0}$
$\vec{p}_0 = \vec{b}$
Solve: $M\vec{z}_0 = \vec{b}$
$k = 0$

Iteration:
While $\sqrt{\vec{z}_k \bullet \vec{r}_k} > \zeta$ do
    $\alpha_k = \frac{\vec{z}_k \bullet \vec{r}_k}{\vec{p}_k \bullet A\vec{p}_k}$
    $\vec{x}_{k+1} = \vec{x}_k + \alpha_k\vec{p}_k$
    $\vec{r}_{k+1} = \vec{r}_k - \alpha_k A\vec{p}_k$.
    Solve: $M\vec{z}_{k+1} = \vec{r}_{k+1}$
    $\beta_k = \frac{\vec{z}_{k+1} \bullet \vec{r}_{k+1}}{\vec{z}_k \bullet \vec{r}_k}$
    $\vec{p}_{k+1} = \vec{z}_{k+1} + \beta_k\vec{p}_k$
    $k = k + 1$
```

For IC-PCCG algorithm, the precondition matrix $M = LDL^T$ is derived from the incomplete Choleski decomposition of A, where D is a diagonal matrix and L is a lower triangular matrix with the same sparse pattern as A. A detailed discussion of IC-PCCG algorithm can be found in [9]. In addition to the vector additions ($\vec{a}+s\vec{b}$) and inner products ($\vec{a}\bullet\vec{b}$), a sparse matrix vector multiplication and two sparse triangular system solving are kernels used inside an IC-PCCG iteration.

In our implementation of the IC-PCCG algorithm, matrix A is first multi-color renumbered and then M is derived from the renumbered matrix by incomplete Choleski decomposition. Matrices A and M are ditributed to the 10 processor cells by row interleaving. Vectors $\vec{p}$, $\vec{r}$, $\vec{z}$ and $\vec{x}$ are also distributed to the 10 processor cells by interleaving. Scalars are duplicated in all cells. One working vector of length n is allocated in each cell for sparse matrix vector multiplication and triangular system solving. The $\vec{p}$ vector is copied to the working vector before $A\vec{p}$ is performed. A segement of the $\vec{z}$ vector is generated after a parallel forward (backward) substitution step completes. The segement is then copied to the working vector before the next parallel forward (backward) substitution step starts. The inner products are done by computing the partial result in each cell. All the partial results are summed together from the first cell to the last cell then broadcast backwards from the last cell to all the cells. Vector additions are naturally parallelized because the vectors are distributed. Scalar computations and convergence test are performed by all cells, that is, sequential computations are duplicated. The host is not involved in the iterative process at all, thus its limited host I/O bandwidth does not affect the computation.

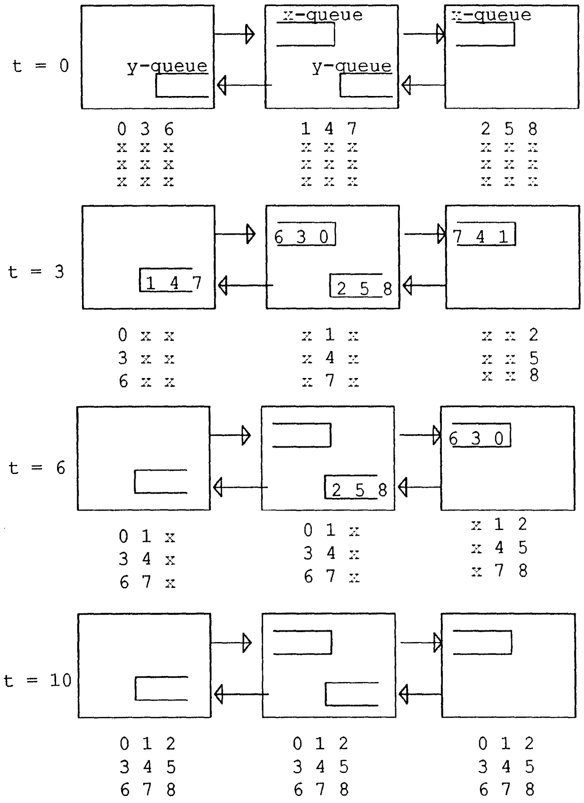


Figure 3-4: Copy a distributed vector to all the cells

Copying a distributed vector to all the cells is the major communication overhead in this mapping. With the support of the systolic communication pathway, we are able to reduce this overhead significantly. Figure 3-4 shows a method to achieve fast communication. The method can copy a distributed vector of length n to all the cells in $n+c-2$ cycles, where c is the number of cells. Limited by the local memory bandwidth, this result is only $c-2$ cycles away from the best possible achievable result of n cycles. We have exercised our IC-PCCG implementation on sparse matrices arising in the finite element analysis applications from GE Corporate Research and Development, GE-CRD. Limited by the small cell memory, these matrices have no more than 4000 unknowns. The performance of GE production IC-PCCG code run on a VAX-780 (with floating point accelerator) VMS system is compared with Warp. Warp is more than 100 times faster, depending on the sparsity of a matrix.

4. Regular Sparse Linear Systems

In this section, we describe the mapping methods for solving finite difference equations on a regular mesh, or equivalently a sparse linear system with multiple nonzero diagonals in the coefficient matrix. Problems of this type frequently arise in numerical solution of partial differential equations by finite difference approximation. The regular structure of the sparse matrix makes index vectors obsolete and more effective mapping schemes can be used to improve the computation speed.

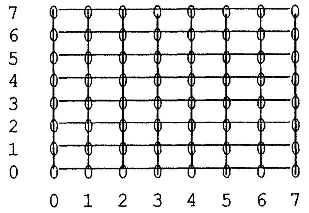


Figure 4-1: Descretized square domain (5-point stencil)

For simplicity and ease of presentation, we illustrate the example of solving Laplace's equation on a square domain with a Dirchlet boundary condition. The domain is descretized using the five point difference scheme , as shown in Figure 4-1. Linear equations of the form

$$4u_{i,j}-(u_{i-1,j}+u_{i,j-1}+u_{i,j+1}+u_{i+1,j}) = 0$$

are obtained for the interior grid points. The solution of $u_{i,j}$ can be derived by the method of Successive Over Relaxations (SOR) [13]. Of course, the problem can be solved using other fast methods [12]. But, it makes a convenient example with which we are able to illustrate a mapping scheme for general problems of this type. The SOR iteration can be formulated by the recurrence equation:

$$u^{k}_{i,j}=(1-\omega)u^{k-1}_{i,j} + 0.25\omega(u^{k}_{i-1,j}+u^{k}_{i,j-1}+u^{k-1}_{i,j+1}+u^{k-1}_{i+1,j})$$

where the super script denotes the iteration number and ω is the relaxation parameter. The generic SOR iteration is considered sequential on vector computers. It can not be vectorized in either the column or the row dimension because of the recurrence definition in the algorithm. For Warp, even the nested recurrence computation can naturally be parallelized using the systolic pathway.

Our mapping is based on a simple domain partition, no preprocessing is needed to achieve the splitting. Consider the mapping on a linear array of two processor cells. The mesh of unknowns is evenly partitioned into 2 cells on the column dimension with one column overlapped, as shown in Figure 4-2. In one relaxation step, the computations between two cells are scheduled as follow. When cell 0 completes the computation for the first half of row i, it sends the overlapped element to cell 1 and continues to compute row i+1 in its domain. In the mean time, cell 1 receives the value generated by cell 0 and continues the computation on the second half of row i. Cell 1 sends its overlapped element back to cell 0 when it is generated. The datum is stored in the queue and will not be retrieved by cell 0 until it finishes the computation of the first half of row i+1. This process repeats until the last row. Figure 4-3 illustrates the communication between two cells.

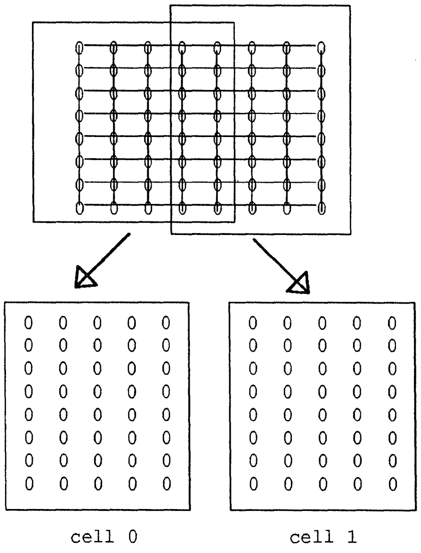


Figure 4-2: Domain partition

In this example, the queue between processors is used both for communication and synchronization, a unique feature of systolic arrays. With the combined communication and synchronization scheme, the mapping is free of overhead. The zero cost of synchronization remains as the number of processor increases and the granularity between synchronizations decreases. This nice property can not be achieved in many shared memory multiprocessors, where the synchronization is done sequentially. A complete SOR algorithm needs to compute the norm of error vector between successive iterations to determine the convergence of the solution. The norm of error vector is computed locally inside each cell. After one relaxation step is completed, the partial norms of all cells are combined together from the first cell to the last cell then broadcast backward from the last cell to all the cells. The convergence test is done by all the cells. As in the IC-PCCG algorithm, the host is not involved in the iterative process. For a mesh of 500 by 500, the Warp computer can finish one SOR iteration with convergence test in 187 ms, which is 754 ns per point per iteration or 14.6 MFLOPS. The bad absolute performance is caused by the 7-stage pipelined processing unit inside each cell. A simple fix to avoid the cell pipeline problem is to use the 2-color relaxation scheme. In the first half of an iteration, we update the unknowns u_{ij}, where $i+j$ is even. In the second half of an iteration we update u_{ij}, where $i+j$ is odd. Each half iteration is completely parallel, thus the cell pipeline unit can be utilized more effectively. One complete 2-color SOR relaxation for the 500 by 500 mesh can be done in 54 ms, which is 224 ns per point per iteration or 49.4 MFLOPS.

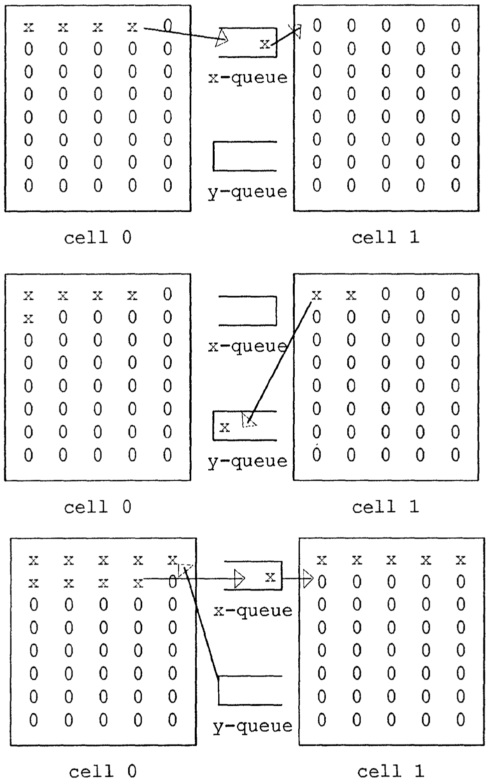


Figure 4-3: Parallel SOR relaxation on a 2-cell array

5. Conclusions

We have demonstrated that a linear systolic array of powerful processors like Warp can be used effectively in solving sparse linear systems. The high bandwidth systolic intercell pathway is very powerful for fast communication and synchronization. It is used to reduce the communication overhead in the IC-PCCG algorithm and to parallelize the nested recurrence computation in the generic SOR relaxation. The MIMD array is useful because multiple gather operations in the sparse matrix vector multiplication can not be sequenced by a single instruction stream across the array. It is the heavily pipelined processor cell, not the linear array, limits the achieved performance. The cell's single precision floating point arithmetic and the small local memory capacity also limit the Warp computer's use for large scale sparse matrix applications.

References

1. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. and Webb, J. A. "The Warp Computer: Architecture, Implementation and Performance". *IEEE Transactions on Computers C-36*, 12 (December 1987), 1523-1538.

2. Annaratone, M., Bitz, F., Deutch, J., Hamey, L., Kung, H. T., Maulik, P., Ribas, H., Tseng, P. and Webb, J. Applications Experience on Warp. Proceedings of the 1987 National Computer Conference, AFIPS, 1987, pp. 149-158.

3. Bruegge, B., Chang, C., Cohn, R., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D. The Warp Programming Environment. Proceedings of the 1987 National Computer Conference, AFIPS, 1987, pp. 141-148.

4. Bucher, I. The Computational Speed of Supercomputers. The Proceeding of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, ACM, August, 1983, pp. 151-165.

5. Duff, I. The Solution of Sparse Linear Equation on the CRAY-1. Proceedings of the NATO Workshop on High Speed Computations, NATO, 1984, pp. 293-309.

6. George, J. and Liu, J.. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, N. J., 1981.

7. Hestenes, M. and Stiefel, E. "Methods of Conjugate Gradients for Solving Linear Systems". *Journal of Research Naional Bureau of Standard. 49* (1952), 409-439.

8. Kincaid, D., Respess, J., Young, D. and Grimes, R. "Algorithm 586 ITPACK 2C: A FOTRAN Package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods". *ACM Transactions on Mathematical Software 8*, 3 (September 1982), 1302-1322.

9. Manteuffel, T. Shifted Incomplete Cholesky Factorization. Sparse Matrix Proceedings, SIAM, Nov, 1978, pp. 41-61.

10. Poole, E. and Ortega, J. Incomplete Choleski Conjugate Gradient on CYBER 203/205. In *Supercomputer Applications*, Plenum Press, New York, 1984, pp. 19-28.

11. Schreiber, R and Tang, W. Vectorizing the Conjugate Gradient Method. Proceedings Symposium Cyber205 Applications, CDC, 1982.

12. Swarztrauber, P. and Sweet, R. "Algorithm 541: FORTRAN Subprograms for the Solution of Seperable Elliptic Partial Differential Equations". *ACM Transactions on Mathematical Software 5*, 3 (September 1979), 325-364.

13. Young, D.. *Iterative Solution of Large Linear Systems.* Academic Press, New York, 1971.

Mapping Two Dimensional Systolic Arrays to One Dimensional Arrays and Applications [1]

V. K. Prasanna Kumar and Yu-Chen Tsai

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-0781

Abstract – A general methodology to map the computations of two dimensional systolic arrays onto one dimensional arrays is developed. Since two dimensional arrays have been developed for a large class of problems, using our technique they can be translated into one dimensional arrays with bounded I/O bandwidth requirement. As applications of our methodology we show a) improved linear systolic arrays for several matrix oriented computations such as matrix multiplication, transitive closure and dynamic programming, b) systolic arrays with tradeoff between number of PEs, local storage and I/O bandwidth and c) fault tolerant systolic designs which can be implemented in Wafer Scale Integration. Compared to known designs in the literature our methodology leads to modular systolic arrays with constant hardware in each PE, few control lines, lexicographic data input/output format and improved delay time.

1. Introduction

VLSI arrays have been designed to implement cost effective and efficient parallel solutions in hardware. Using this methodology, parallel solutions to a large class of numerical, signal and image processing problems have been implemented in hardware [7][15]. Most of these designs consist of two dimensional array of PEs which solve problems involving $O(n^3)$ computations in $O(n)$ time using $O(n^2)$ PEs. In general, such two dimensional arrays have $O(n)$ I/O bandwidth and hence data can be easily aligned for the desired operations to be performed as the data flows through the array.

Recently design of parallel algorithms for linear arrays has become increasingly important [23][12][3]. Linear arrays offer several advantages compared to two dimensional arrays. They require constant I/O bandwidth. As the problem size becomes larger, linear arrays become attractive to implement because only a fixed number of I/O pins are needed on the chip. Also, in Wafer Scale Integration (WSI), it has been shown [8] that unidirectional linear array structure leads to 100% utilization of good PEs. No technique is known to result in high PE utilization on a wafer in case of two dimensional arrays (in the worst case) [29] [4]. However, due to the limited I/O access in the one dimensional arrays, the mapping techniques in the literature cannot be directly utilized to design such arrays or result in complex systolic designs. Some of the known linear systolic array design methodologies lead to bidirectional data flow which is not desirable in wafer scale integration [29] [13]. Other methodologies lead to complicated control and nonuniform I/O which makes it difficult to interface with the host [23] [24] [28] [29].

In this paper, we develop a general mapping technique to map the computations of two dimensional arrays onto one dimensional arrays. Using our method, "clean" linear systolic arrays can be designed for a general class of problems (including Matrix Multiplication, Transitive Closure, Dynamic Programming etc.) for which two dimensional arrays have been designed in the past. The resulting linear arrays have continuous I/O sequence and modular extensibility property. Using our methodology, *family of linear systolic arrays* for matrix multiplication and related problems in signal and image processing can be designed, exhibiting tradeoff between I/O bandwidth, local storage, processor complexity and number of PEs. In addition, our technique also leads to designs with unidirectional flow of data and control which makes our designs easily implementable in well known reconfiguration schemes proposed for WSI.

The rest of the paper is organized as follows. In section 2, we present our technique to map algorithms onto linear systolic arrays which results in simple designs for many problems. In section 3, we apply our mapping methodology to design 1) linear systolic arrays for some matrix problems, 2) family of linear arrays, and 3) fault tolerant linear arrays. Finally, some comparisons and conclusions are made.

2. Mapping from 2-D to 1-D arrays

In this section, we discuss the basic idea of our mapping technique and its limitations. Timing analysis and details of implementation are discussed in the following sections.

Our technique starts with a two dimensional systolic array. For clarity of presentation of our ideas, we will consider a two dimensional array to compute C=A × B (where A, B and C are matrices) as an example. A 4×4 array is shown in figure 1 where the PEs are numbered in row major order. The input data in each row (and each column) flow through the row (and column) of the array. For example, a_{11} passes through PE_1, PE_2, PE_3, PE_4 and b_{11} passes through PE_1, PE_5, PE_9, PE_{13}. All the computations to compute C_{ij} are performed in $PE_{(i-1)*n+j}$. As a data item passes through a PE it performs computation with the data arriving at its other input and updates C_{ij}. For example, in figure 1, if the computations begin at t=1, then a_{23} will meet b_{34} at time t=7 in PE_8 to calculate C_{24}.

One way of mapping the above two dimensional array into a linear array is by *partitioning* each row and *stretching* it with their links in row major order as shown in figure 2. Thus, the resulting linear array will have n^2 PEs. The PEs in a row in the 2-D array correspond to a *block* (of n PE's) in the 1-D array. However, the array in figure 2 is not the desired linear array. The data has to be fed to internal PEs in the array, which is not allowed in a one dimensional array. The desired structure is shown in figure 3. It features local connections with I/O performed at the leftmost and rightmost PE. The input matrices A and B are partitioned into A and B bands as shown in figure 1. The input data is fed at the leftmost PE of the array as shown in figure 3.

In order to enforce the linear array in figure 3 to simulate the 1-D array in figure 2, we have to address the following problems.

1. *Activation of PEs*
   This problem is concerned with activating the PEs to perform operations when the desired operands arrive at a PE. In the above example, in order to simulate the operations of the 2-D array with linear array, we use some control signals to let a_{11} be 'activated' from PE_1 to PE_4 and be 'deactivated' in other PEs. When a_{11} is activated in a PE, it performs a computation (of the form $c_{11} \longleftarrow c_{11} + a_{11} * b_{1j}$) with b_{1j}, in PE_j, $1 \leq j \leq 4$. Similarly, b_{11} is activated only in PE_1, PE_5, PE_9, PE_{13} to compute with a_{i1}, $1 \leq i \leq 4$. In other words, in the linear array, a_{11} is transported from PE_5 to PE_{16} without doing any operation and b_{11} is transported through PE_2, PE_3, PE_4, PE_6, PE_7, PE_8, PE_{10}, PE_{11}, PE_{12}, PE_{14}, PE_{15}, PE_{16} while being deactivated. In the above example, the element a_{ij} is to be activated at $PE_{(i-1)*n+m}$ where $1 \leq m \leq n$, and b_{ij} be activated at $PE_{(k-1)*n+j}$ with $1 \leq k \leq n$.

[1] This research was supported in part by the National Science Foundation under grant IRI-8710836 and by a grant from USC Faculty Research and Innovation Fund.

2. *Operand alignment*
This problem is concerned with ensuring the right operands meet in a PE to perform an operation. In order to satisfy the alignment of operands in the linear array, we use two types of channels: *fast* and *slow* channels. Suppose the data in the fast channel takes α time units to pass through a PE while the data in the slow channel takes γ time units where $\alpha < \gamma$. If the elements of A (B) matrix are fed into a slow (fast) channel in a column (row) major order, then, for $\gamma = 2$ and $\alpha = 1$, if a_{ik} and b_{kj} reach PE_p at time t_0, then a_{ik} will meet $b_{k,j+1}$ at time $t_0 + 1$ at PE_{p+1}.

3. *Transportation of data from row to row*
This problem is concerned with simulating the movement of data from one row to another row in the 2-D array (those data crossing the dashed lines in figure 1) on the one dimensional array. We use an extra channel (in the above example, BS) to transport this data within each block, the data to be used by the PEs in the next block is stored in this channel.

In summary, a general design methodology is as follows:

1. Start with a 2-D systolic array with data flow along the positive coordinate direction. Without loss of generality let the array size be $m \times n$ without diagonal connections.
2. Partition the array and compress it into a linear array. A general partition rule will be given in section 3.3.
3. Assume there are x channels along the X axis and y channels along the Y axis connecting adjacent PEs in the 2-D array. Each horizontal channel in the 2-D array corresponds to one slow channels connecting adjacent PEs in the resulting 1-D array and each vertical channel corresponds to a slow and a fast channel. Thus, we have $x + 2y$ channels connecting PEs in the 1-D array.
4. Feed the A and B bands into the leftmost PE.
5. Design a scheme to solve
   (a) The activation of operations (using control signals).
   (b) The alignment of operands (using fast and slow channels).
   (c) The transportation of operands (using transportation channel and mechanism to switch data channels).
6. The delays in each PE can be determined by the following procedure:
   (a) Let the amount of delay within each PE of each channel be a parameter.
   (b) Using the design scheme in step 5 above, obtain timing equations of channels involving these parameters.
   (c) Using the alignment and activation requirements, obtain constraint equations to assure that the desired data meet in the active PEs.
   (d) Using the timing equations and constraint equations choose the optimal set of parameters to minimize delay.

An important requirement for this methodology to be applicable is the data flow in the original two dimensional array is unidirectional along coordinate axes. The following proposition states that our methodology can be applied to arbitrary two dimensional arrays [9].

Proposition 1 *If a computation can be performed in $O(n)$ time on a two dimensional $n \times n$ systolic array, then it can be transformed such that the resulting array has $O(n^2)$ PEs and the data flow is* unidirectional along *X* and *Y* axis *with no asymptotic loss in time. This array can be further transformed into a* unidirectional *linear array using the proposed mapping technique.*

3. Applications

In this section, we illustrate our mapping technique by designing linear systolic arrays for several applications.

3.1 A linear array for Matrix Multiplication

Consider the 2-D array for matrix multiplication as shown in figure 1. The linear systolic array shown in figure 3 consists of n^2 PEs numbered 1,...,n^2 from left to right. The PEs are connected by three data channels which carry the input data, i.e. a *fast* Channel BF for elements of B, and two *slow* channels AS and BS for the elements of A and B respectively. One bit wide control lines *ACT*, *I*, *J* connect adjacent PEs. All the control signals and data move from left to right only. We will use the above connections to solve the following problems (step 5 in our method).

Activation of PEs

When the AS and BF channels have data which commute in a PE, then the PE must be ***activated*** to perform a computation of the form $C_{ij} \Leftarrow C_{ij} + a_{ik} * b_{kj}$. We implement this by inputing a control signal denoted *ACT* at the left end of the array to set a flag *ACTIVE* inside each *PE*. A PE is said to be *active* if it has *ACTIVE* set to 1. It will then perform a partial product computation during that clock period. In our design, when *ACTIVE*=1, a_{ik} in AS channel is multiplied with b_{kj} in BF channel.

Alignment of Operands

To get the correct operands together to perform an operation in each PE data channels with different speeds are used to align the operands. There are two types of alignment in our matrix multiplication design. The first type concerns the alignment of operands within a row. The second type concerns with the alignment of operands from block to block. For example, an activated a_{ik} (in the i^{th} block) has finished all its operations with b_{kj}s ($1 \leq j \leq n$) when it reaches PE_{n*i}, $a_{i+1,k}$ (which immediately follows a_{ik}) is activated and performs operations with $b_{k;}$s in the next block (These b_{kj}s in BS are also copied to BF at the beginning of $block_{i+1}$ to supply the operands that are needed in that block). To implement this type of alignment we use a multiplexer M_A to make the data in the AS channel to gain one time unit at the last PE of each block. This multiplexer is controlled by a flag ψ which is set at the last PE of each block. Flag ψ is set by control signals *I* and *J* whose operations are described in the appendix. Notice the signal in ACT channel does not gain one unit of time at the end of each block. Thus, in $block_{i+1}$, $a_{i+1,k}$ is active if a_{ik} was active in $block_i$, for some k.

Transportation of Data from Row to Row

To simulate the data flow from one row to another row in the 2-D array on the 1-D array, an extra slow channel BS is used to transport the data. The data from a block to its adjacent block is saved in this slow channel. This data will be used by the PEs in the next block (by copying the data in slow channel BS into the fast channel BF at the beginning of that block) as operands of that block. *Switching of data* from BS to BF is implemented by multiplexer M_B which is also controlled by the flag ψ.

The overall system structure is shown in figure 4. The structure of PE is shown in figure 5. The operation of the *PE*s is as follows:

```
Read data into registers from input ports.
If ($ACTIVE$=1) then C $\Leftarrow$ C + AS.LR*BF.R
If ($\psi$=1) then
        begin
            $M_A$ selects data from AS.LR.
            $M_B$ selects data from BS.RR.
        end
```

```
else
      begin
            $M_A$ selects data from AS.RR.
            $M_B$ selects data from BF.R.
      end
```

The algorithm uses a simple data input sequence in which the data is input in every clock continuously without any delay. At t_0, a_{11} is fed into AS channel, and b_{11} is fed into BS and BF channels of PE_1 (leftmost PE) in the array. Matrix A is fed in column major order, i.e., a_{11}, a_{21}, $a_{31}, \cdots, a_{n1}$, $a_{12}, \cdots, a_{nn}$. Matrix B is fed in row major order, i.e., b_{11}, b_{12}, $b_{13}, \cdots, b_{1n}$, $b_{21}, \cdots, b_{nn}$. Also, the control input ACT is set to 1 every time a_{1k}, $1 \leq k \leq n$, is inserted into the array.

Timing Analysis

By assuming the delay of each channel within each PE as a parameter, the transportation of data in each channel can be described by timing equations and the alignment and activation requirements can be described by constraint equations. Using these timing and constraint equations, optimal parameters can be chosen to minimize the delay. In the following design we assume that the computations begin at t=1.

Timing equations

Let

$I(a, u, v)$
$= \text{time at which } a_{uv} \text{ is input to } PE_1, \ 1 \leq u, v \leq n.$
$= (v-1) * n + u$ (1)

$I(b, r, s)$
$= \text{time at which } b_{rs} \text{ is input to } PE_1, \ 1 \leq r, s \leq n$
$= (r-1) * n + s$ (2)

$I(ACT, k)$
$= \text{time at which } k^{th} \ ACT = 1 \text{ is input to } PE_1, \ 1 \leq k \leq n$
$= (k-1) * n + 1$ (3)

The following are the timings of the data a_{uv}, b_{rs} and control signal ACT that appear at processor p, $1 \leq p \leq n^2$. PE_p computes C_{ij} where p is given by,

$$p = (i-1) * n + j \tag{4}$$

1. For a_{uv}, $1 \leq u, v \leq n$,

$t(a, u, v, p)$
$= \text{time at which } a_{uv} \text{ appears at } PE_p$
$= I(a, u, v) + \alpha(p-1) - \beta\lfloor (p-1)/n \rfloor$ (5)

In the above equation, the first term is the time at which a_{uv} was input to PE_1. The second term is the delay experienced in AS channel of $(p-1)$ PEs. α is the delay of the AS channel within a PE. The last term corresponds to the time gained at the end PE of those $\lfloor (p-1)/n \rfloor$ blocks in front of PE_p. β is the time gained by a_{uv} at the end of each block.

2. For b_{rs}, $1 \leq r, s \leq n$,

$t(b, r, s, p)$
$= \text{time at which } b_{rs} \text{ appears at } PE_p$
$= I(b, r, s) + \gamma * \lfloor (p-1)/n \rfloor * n + \delta((p-1) \ mod \ n)$ (6)

In the above equation, the first term corresponds to the time at which b_{rs} is input to PE_1. The second term corresponds to the delay experienced by the data as it travels in the BS channel in all the blocks before the block to which p belongs. γ is the delay in the BS channel. The last term is the delay experienced within the block to which PE_p belongs. δ is the delay in the fast channel BF.

3. For ACT signal,

$t(ACT, k, p)$
$= \text{time at which } k^{th} \ ACT = 1 \text{ (denoted as } ACT_k\text{) appears in } PE_p$
$= I(ACT, k) + \omega(p-1)$ (7)

In the above equation, the first term corresponds to the time at which ACT signal is input to PE_1. The second term corresponds to the delay experienced by the data as it travels in the control channel. ω is the delay of the control channel within each PE.

Constraint equations

In order to correctly perform matrix multiplication, we need to implement the following operation: activate PE_p to perform operations of the type $C_{ij} \Longleftarrow C_{ij} + a_{ik} * b_{kj}$ during the activation period, where $p = (i-1) * n + j$. That is, when ACT_k arrives at PE_p, the specific a_{ik} and b_{kj} should also be in that PE. Thus, the data a_{uv}, b_{rs} and ACT arriving at a PE must satisfying the following conditions:

1. $u = i$
2. $s = j$
3. $v = r = k$
4. $t(a, \ u, \ v, \ p) = t(ACT, \ k, \ p) = t(b, \ r, \ s, \ p)$

Using the timing and constraint equations we obtain the following equations:

1. $\omega - \delta = 1$
2. $\beta = 1$
3. $\omega = \alpha = \gamma$

A set of values satisfying the above are:

$$\alpha = 2, \beta = 1, \gamma = 2, \delta = 1, \omega = 2.$$

The above parameters mean that in each PE there are 2 time units delay in AS channel, a_{uv} gain 1 unit time at the end of each block. The rest of the delays of channels are: [BS 2], [BF 1], [ACT 2] where [x, y] denotes there is y units of delay in the x channel. □

The above analysis leads to:

Theorem 1 *The 1-D array correctly performs all the computations of the 2-D array for matrix multiplication at the end of time $t = 3n^2 - n - 1$, assuming the computation begins at time $t = 1$.*

3.2 Linear Arrays for Transitive Closure

As another illustration of the mapping technique, we design linear arrays for the transitive closure problem by mapping the computations of a well known 2-D array.

A 2-D systolic array for the transitive closure problem has been derived in [5]. Its structure is shown in figure 6 which is the same as figure 1 except for the end around connections.

The input is two copies of $n \times n$ adjacency matrix, with 1's on the diagonal, read into an $n \times n$ array of processors. The output is found in the processor array and is read out of the right and bottom edges. Three passes are needed for the computations [26].

Unlike the systolic array for matrix multiplication the data in the array is updated at certain times by the PEs in each row (column) before moving to the next row (column). Two control signals $DIAGA$ and $DIAGB$ can be used to implement this update. $DIAGA$ ($DIAGB$) is associated with a_{ij} (b_{ij}) such that if $i = j$ then $DIAGA$ ($DIAGB$)=1 else $DIAGA$ ($DIAGB$)=0.

When PE_{ij} receives $DIAGA$ it updates b_{ij} (i.e., $b_{ij} \Longleftarrow C_{ij}$) if $DIAGA$=1. Similarly when $DIAGB$ is equal to 1, a_{ij} is updated (i.e., $a_{ij} \Longleftarrow C_{ij}$). We call this the *update mechanism.*

To map the 2-D array in figure 6 into 1-D array, we use the same partitioning and stretching method as in matrix multiplication. Therefore solution to the alignment, transportation and activation of PEs problems are the same in this design. Hence, we will only address two major differences compared to matrix multiplication: (1) Simulation of the end around connections in the 2-D array and (2) implementation of the update mechanism.

The activation design of PEs in matrix multiplication makes the $O(n)$ end around connections in the 2-D array easy to implement in the 1-D array. For example, in the 2-D array a_{11} goes from PE_1 through PE_4 and back to PE_1. This operation can be simulated in 1-D array by letting a_{11} go through PE_1 to PE_4 activated and from PE_5 to PE_{16} deactivated and then back to PE_1 again. The same design can be used for the elements in B matrix. In this way, $2n$ end around connections in the 2-D array can be simulated by 2 connections in the 1-D array.

We now consider implementing the update mechanism in the 1-D array. The control signals for update mechanism can be associated with the data as in the 2-D case. The data to be moved from row to row is the B matrix. However, in the 1-D case the b_{ij} that does the operation with a_{ii} is in the BF channel while b_{ij} to be updated and transported to the next block is in the BS channel. This does not lead to any timing problems, since the b_{ij} in BF is moving faster than the corresponding b_{ij} in BS. Thus, the updated data inside a PE must be placed onto the slow channel corresponding to the B matrix.

The linear array is shown in figure 7. There are n^2 PEs numbered 1 to n^2. The PEs are connected by the following data paths (figure 8).

1. Two *slow* channels corresponding to A and $DIAGA$ inputs.
2. Two *slow* channels corresponding to B and $DIAGB$ inputs.
3. Two fast channels corresponding to B and $DIAGB$ inputs.
4. One bit control signal which passes through the array is used to indicate whether the b_{ij} in the BS channel has been updated or not. The control signal, UP_B, together with flag NEW_B, is used to update the b_{ij} in the BS channel. It is initialized to 0 when it is fed at the leftmost PE.

The detailed operation of the PE during each clock period and the operation of the array can be found in [11].

Using the timing analysis as in matrix multiplication, it is easy to show:

Theorem 2 *The 1-D systolic array of figure 10 computes the the transitive closure of a $n \times n$ adjacency matrix in time $7n^2 - 3n + 1$.*

3.3. Family of Arrays for Matrix Computations

A general methodology to design a family of arrays for matrix computations is as follows.

1. Partition the 2-D array into collection of disjoint rows, $CROW_1$, $CROW_2$, ..., $CROW_r$, $r = \lceil n/m \rceil$. The number of rows in a collection is equal to memory size m available in each PE, except $CROW_r$ which may have less than m rows.
2. The linear array consists of $\lceil n/m \rceil$ blocks each block having n PEs. The computations performed by the PEs in $block_i$ is the same as the computations of PEs in $CROW_i$, $1 \leq i \leq r$.
3. Feed the A and B bands of the input matrices at the leftmost PE.
4. Selectively activate the PEs to perform a step of the matrix multiplication algorithm.
5. Within each block save the elements of B matrix in a slow channel which will be used by the PEs in the next block. At the end of each block, switch the B matrix data from slow to fast channel so that they can commute with the elements of A matrix within the next block.

We will use the above technique to map the two dimensional array for matrix multiplication onto two linear array models. In each model we will show different partitioning schemes to result in an optimal family of arrays for matrix multiplication.

Variable Memory Family (VMF) Model

In this model, the number of I/O channels is fixed. Thus, when designing a special purpose chip, the number of pins per chip is fixed for all members in this family.

Suppose we can build chips with $O(s)$ storage and an ALU. In this scheme, the 2-D array having n row is partitioned into collection of disjoint rows as follows: $CROW_i$ has s consecutive rows starting at $(i-1)s + 1^{th}$ row of the 2-D array. Thus, there are $\lceil n/s \rceil$ $CROW$s. The resulting linear array will have $n\lceil n/s \rceil$ PEs grouped into $\lceil n/s \rceil$ blocks of n PEs. The PEs in the i^{th} block perform the computations of the PEs of the i^{th} $CROW$. The computations of PE_{ij} $1 \leq i,j \leq n$ in the 2-D array is performed by $PE_{(m-1)n+j}$ in the 1-D array, where $m = \lceil i/s \rceil$. The resulting linear array consists of $n\lceil n/s \rceil$ PEs.

As an illustration consider 4 × 4 matrix multiplication. A partitioning of 2-D array for 4×4 matrix multiplication and its mapping to linear array are shown in figure 9 and figure 10 for s=2. In this example, $CROW_1$ has rows 1 and 2 and $CROW_2$ has rows 3 and 4 of the 2-D array for matrix multiplication. Since $\lceil n/s \rceil$=2 there are two blocks of PEs each block having n=4 PEs. Thus, the resulting linear array has 8 PEs as shown in figure 10. The PEs are connected by three channels which carry the input data: *fast* channel (BF), and two *slow* channels AS and BS which are used by the elements of A and B respectively. In addition, one bit wide control lines ACT, OC, I, J connect adjacent PEs. The detailed design can be found in [10].

The performace of the design can be summarized as follows:

Theorem 3 *The above method performs the multiplication of two $n \times n$ matrices using $n\lceil n/s \rceil$ PEs having $O(s)$ memory per PE in time $t = n^2 + 2n\lceil n/s \rceil - \lceil n/s \rceil + 1$.*

Variable Channel Family (VCF) Model

In this model we assume we can build k I/O channels, $1 \leq k \leq n$, per PE. In this scheme, the 2-D array having n row is partitioned into collection of disjoint rows as follows: $CROW_i$ has $i + (r-1)(\lceil n/s \rceil)^{th}$ row of the 2-D array, where $1 \leq r \leq s$. Thus, there are $\lceil n/s \rceil$ $CROW$s. The resulting linear array will have $n\lceil n/s \rceil$ PEs grouped into $\lceil n/s \rceil$ blocks of n PEs. The PEs in the i^{th} block perform the computations of the PEs of the i^{th} $CROW$. The computations of PE_{ij} $1 \leq i,j \leq n$ in the 2-D array is performed by $PE_{(m-1)n+j}$ in the 1-D array, where m= (i *mod* s). The partition graph and its mapping to linear array are shown in figure 11 and figure 12 for a 4 × 4 matrix multiplication with k=2.

The systolic array in general consists of $n\lceil n/k \rceil$ PEs where k is the number of channels. There are k *slow* channels AS[i], $1 \leq$

$i \leq k$, which are used by the elements of A matrix. The elements of B matrix are fed in row major order, which correspond to B bands in figure 1. However, the elements of A matrix are input in the following way. Channel AS[l] carries $\lceil n/k \rceil$ elements of each column of A starting at the $\lceil n/k \rceil(l-1)+1^{st}$ element. Append $n-\lceil n/k \rceil$ dummy data denoted "Λ" at the end of each column data. Thus, for $n=4$, $k=2$ the input sequence is a_{11}, a_{21}, Λ , Λ , a_{12}, a_{22}, Λ , Λ , a_{13},... to AS[1]. Similarly a_{31}, a_{41}, Λ , Λ , a_{32}, a_{42}, Λ , Λ , a_{33},... is the input sequence to AS[2].

By performing a timing analysis [10], we can show:

Theorem 4 *: The above method performs the multiplication of $n \times n$ matrices using $n\lceil n/k \rceil$ PEs, each PE having O(k) storage and O(k) I/O channels in time $t = n^2 + 2n\lceil n/k \rceil - \lceil n/k \rceil + 1$.*

The time complexity of matrix multiplication on both VMF and VCF models is the same. The VMF model has fixed number of I/O channels. The time available for the execution of a scalar multiplication is one clock cycle. Thus, high speed multipliers are needed in this design. The VCF model uses more I/O channels but if k channels are used, then k scalar multiplications need to be performed over n cycles. Thus, if k is small compared to n then multipliers with low hardware complexity is sufficient to implement this design.

3.4 Designing Fault Tolerant Systolic Arrays for Wafer Scale Integration

The advantages of a special class of linear systolic arrays suitable for WSI technology have been reported in [8]. The most important property of this type of linear array is that *all its data flows is in one direction.* By modeling a systolic array as a directed graph, the following result has been shown in [8]:

Proposition 2 *For any design, if all the edges in a cut set are unidirectional, adding the same delay (bypass) registers (which simulate faulty PEs) to all the edges in the cut will result in an equivalent design.*

As a result, the faulty PEs can be replaced by bypass registers. The above discussion can be captured in the following fault model which will be used in this paper [29]:

1. The PEs are arranged in a straight line with a system of buses running parallel to them. Each bus can has a constant number of buffer registers (per PE) embedded in it. The buffer registers correspond to the delay when a signal passes through a PE. Also, a switch mechanism is used to select the data route of each bus. The route depends on the fault pattern.
2. Propagation delay is assumed to be proportional to the wire length. We incorporate this into our design by introducing a constant unit of delay whenever a signal bypasses a PE.
3. As in other models, the buses and switches are assumed to be reliable, while the PEs may be faulty. Fault tolerance is achieved by hooking working PEs into a desired logical structure, in our case, a linearly connected array.

The complexity of matrix multiplication on this model has been studied in [29]. They establish the following lower bound:

Proposition 3 *Any systolic algorithm computing the product of $n \times n$ matrices using n^3 scalar multiplications on the above model must take $\Omega(n\sqrt{n})$ time.*

In [29], a matrix multiplication algorithm is designed on the above model which has $O(n\sqrt{n})$ delay. Our technique in section 2 also leads to a simple optimal fault tolerant array for matrix multiplication with improved performance.

The intended partition rules are similar to that in VMF model. However, the scheme to solve the alignment and transportation problems is similar to the VCF model. These are summarized as belows:

1. Partition the 2-D array for matrix multiplication into Collection of disjoint ROWs, $CROW_1, CROW_2, ..., CROW_{\sqrt{n}}$, The number of rows in a collection is equal to $\sqrt{n}$.
2. The linear array consists of $\sqrt{n}$ blocks, each block having n PEs. The computations performed by the PEs in $block_i$ is same as the computations of PEs in $CROW_i$, $1 \leq i \leq \sqrt{n}$.
3. Divide each columns of A matrix (each rows of B matrix) into $\sqrt{n}$ parts and feed them into AB_i (BB_i) buses $1 \leq i \leq \sqrt{n}$, in column major order (row major order).

This leads to [9]:

Theorem 5 *The above systolic array computes the elements of $C = A \times B$ in time $3n\sqrt{n} - 2 + 2r$ where r is the number of faulty PEs. Further all the data flows are unidirectional and the distance covered by every signal is one unit in each clock period.*

4. Conclusion

In this paper, we presented a new technique to design linear systolic arrays with limited I/O bandwidth. All our designs have simple control, lexicographic I/O and require a minimum number of processors. These designs can be shown to be optimal with respect to area and time [6]. Table 1 compares the performance of several designs in the literature with the proposed design for matrix multiplication on linear arrays. Table 2 compares the designs for transitive closure on linear arrays. Table 3 compares the designs for family of linear arrays. In addition, our designs result in unidirectional data flow. Therefore, they can be easily implemented in WSI with fault tolerance capability. Table 4 compares our matrix multiplication design on the fault model with known results in the literature.

	This Paper	Method in [20]	Method in [23]
1. Number of Processors	n^2	n^2	$\approx 3/2n^2$
2. Delay Time	$3n^2 - n - 1$	$4n^2 - 3$	$\geq 9/2n^2$
3. Data input	simple	need to insert zeros	complex

Table 1: Comparison of Matrix Multiplication on Linear Array

	This Paper	Method in [27]
1. Number of processors	n^2	$2n - 1$
2. Delay Time	$7n^2 - 3n + 1$	$9n^2 + n - 2$
3. Area of PE	$O(1)$	$O(n)$

Table 2: Comparison of Transitive Closure on Linear Array

	This Paper	Method in [24]
1. Number of processors	$n\lceil n/s \rceil$	$> n\lceil n/s \rceil$
2. Memory Space	$O(s)$	$O(s)$
3. Delay Time	$n^2 + 2\lceil n/s \rceil n - \lceil n/s \rceil + 1$	$> n^2 + 2p$
4. Data input sequence	continuous	complex

Table 3: Comparison of family of Linear Array for Matrix Multiplication

	This Paper	Method in [29]
1.Number of processors	$n\sqrt{n}$	$n\sqrt{n}$
2.Delay Time	$3n\sqrt{n}-2$	$4n\sqrt{n}-n-3\sqrt{n}$
3.Total number of buses	$2\sqrt{n}+2$	$4\sqrt{n}$

Table 4: Comparison of optimal matrix multiplication on the fault model with known result

References

[1] M. C. Chen. A design methodology for synthesizing parallel algorithm and architectures. *Journal of Parallel and Distributed Computing*, 1986.

[2] P. R. Cappello and K. Seiglitz. Unifying VLSI array designs with geometric transformation. In *International Conference on Parallel Processing*, 1983.

[3] K. A. Doshi and P. J. Varman. Optimal graph algorithms on a fixed-size linear array. *IEEE transactions on Computers*, C-36(4), 1987.

[4] J. A. B. Fortes and C. S. Raghavendra. Gracefully degradable processor arrays. *IEEE transactions on Computers*, C-34(11), 1986.

[5] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Caltech Conference on VLSI*, 1979.

[6] J. Ja'Ja' and V. K. Prasanna Kumar. Information transfer in distributed computing with applications to VLSI. *JACM*, January 1984.

[7] H. T. Kung and C.E. Leiserson. Systolic arrays (for VLSI). In *SIAM Conference on Applied Mathematics*, pages 256–282, 1979.

[8] H. T. Kung and Monica S. Lam. Wafer-scale integration and two-level pipelined implementations. *Journal of Parallel and Distributed Computing*, 1(1), 1984.

[9] Yu-Chen Tsai. Linear Systolic Array Design for Wafer Scale Integration. Ph.D Thesis, Department of EE-systems, USC, in preparation.

[10] V. K. Prasanna Kumar and Yu-Chen Tsai. On designing an optimal family of linear systolic arrays for matrix multiplication. Technical Report CRI-87-43, USC, June 1987.

[11] V. K. Prasanna Kumar and Yu-Chen Tsai. Designing Linear Systolic Arrays. In *Journal of Parallel and Distributed Computing*, 1988.

[12] H. T. Kung. Systolic algorithms for the CMU WARP processor. In *Seventh International Conference on Computer Vision and Pattern Recognition*, July 1984.

[13] S. Y. Kung. On supercomputing with systolic/wavefront array processors. In *proceedings of the IEEE*, July 1984.

[14] F. T. Leighton and C. E. Leiserson. Wafer-Scale Integration of systolic arrays. In *23rd Annual Symposium on Foundations Computer Science*, November 1982.

[15] C. Mead and L. Conway. *Introduction to VLSI system.* Addison-Wesley Publishing Company, 1980.

[16] W. L. Miranker and A. Winkler. Space time representations of computational structures. *Computing*, 1984.

[17] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE transactions on Computers*, C-35(1), 1986.

[18] R. G. Melhem and W. C. Rheinbold. A mathematical model for the verification of systolic networks. *SIAM Journal on Computing*, 13(3), August 1984.

[19] S. Purushothaman. Reasoning about modular systolic algorithms. In *International Conference on Parallel Processing*, 1987.

[20] C. S. Raghavendra, V. K. Prasanna Kumar, and A. Varma. On systolic processing with bounded I/O bandwidth. In *ICCD*, 1985.

[21] A. Rosenberg. The Diogenes approach to testable fault-tolerant networks of processors. *IEEE transactions on Computers*, C-32(10), 1983.

[22] I. V. Ramakrishnan and P. J. Varman. Modular matrix multiplication on a linear array. *IEEE transactions Computers*, C-33(11), 1984.

[23] I. V. Ramakrishnan and P. J. Varman. *Synthesis of an Optimal Family of Matrix Multiplication Algorithms on Linear Arrays.* Technical Report, University of Maryland, Computer Science Department, 1985.

[24] I. V. Ramakrishnan and P. J. Varman. Synthesis of an optimal family of matrix multiplication algorithms on linear arrays. *IEEE transactions on Computers*, C-35(11), 1986.

[25] C. D. Thompson. *A complexity theory for VLSI.* PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa., 1979.

[26] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[27] P. J. Varman and I. V. Ramakrishnan. Dynamic programming and transitive closure on linear pipelines. In *International Conference on Parallel Processing*, 1984.

[28] P. J. Varman and I.V. Ramakrishnan. Optimal matrix multiplication on fault-tolerant VLSI arrays. In *ICALP*, 1985.

[29] P. J. Varman and I. V. Ramakrishnan. A fault-tolerant VLSI matrix multiplier. In *International Conference on Parallel Processing*, 1986.

Appendix

Setting of ψ

I and J are used to set ψ inside each PE. I is set to 1 every n clock periods, and J is set to 1 at the start of the operation of the array. Thus, for $t_0 \le t \le t_0 + 3n^2 - n - 1$.

$$I = \begin{cases} 1 & \text{if } t = t_0 + n * (i-1),\ 1 \le i \le (n-1). \\ 0 & \text{otherwise.} \end{cases}$$

$$J = \begin{cases} 1 & \text{if } t = t_0. \\ 0 & \text{otherwise.} \end{cases}$$

The signals I, J are fed at the leftmost PE and are propagated with delay of one and two units respectively in each PE. Let I_k and J_k denote I, J that enter PE_k respectively $(1 \le k \le n^2)$. Then,

$$I_k = 1 \text{ at } t = t_0 + (k-1) + n * i,\ 1 \le k \le n^2.$$
$$J_k = 1 \text{ at } t = t_0 + 2(k-1) + 1,\ 1 \le k \le n^2.$$

It is easy to verify that $I_k = 1$ and $J_k = 1$ only when $k = n * i$. This occurs at time $t = t_0 + 2(n * i) - 1$.

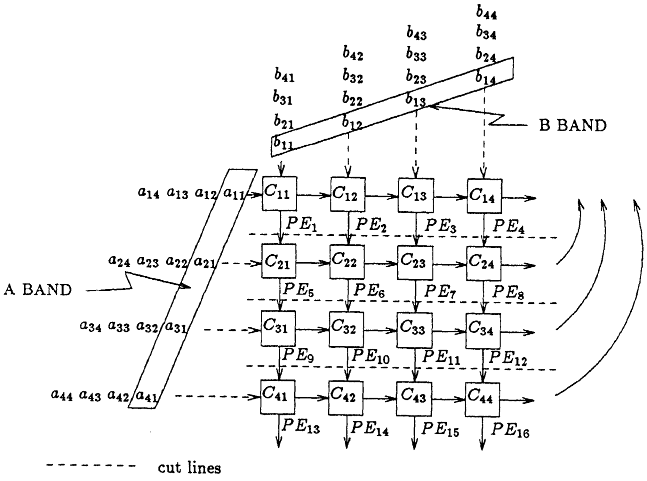

b_{44} b_{34} b_{24} b_{14}
b_{43} b_{33} b_{23} b_{13}
b_{42} b_{32} b_{22} b_{12}
b_{41} b_{31} b_{21} b_{11}
B BAND
A BAND
a_{14} a_{13} a_{12} a_{11}
a_{24} a_{23} a_{22} a_{21}
a_{34} a_{33} a_{32} a_{31}
a_{44} a_{43} a_{42} a_{41}
C_{11} C_{12} C_{13} C_{14}
C_{21} C_{22} C_{23} C_{24}
C_{31} C_{32} C_{33} C_{34}
C_{41} C_{42} C_{43} C_{44}
PE_1 PE_2 PE_3 PE_4
PE_5 PE_6 PE_7 PE_8
PE_9 PE_{10} PE_{11} PE_{12}
PE_{13} PE_{14} PE_{15} PE_{16}
cut lines

Figure 1: Partitioning a 2-D array

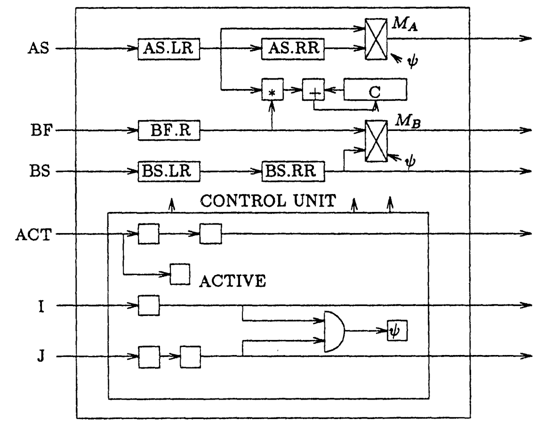


Figure 5: The internal structure of PE

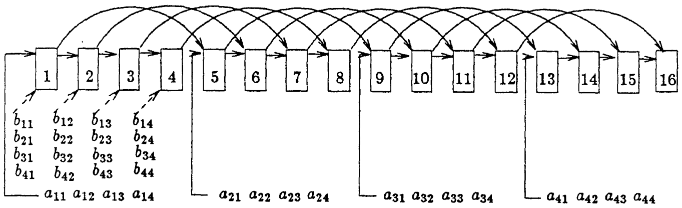

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b_{11} b_{21} b_{31} b_{41}
b_{12} b_{22} b_{32} b_{42}
b_{13} b_{23} b_{33} b_{43}
b_{14} b_{24} b_{34} b_{44}
a_{11} a_{12} a_{13} a_{14}
a_{21} a_{22} a_{23} a_{24}
a_{31} a_{32} a_{33} a_{34}
a_{41} a_{42} a_{43} a_{44}

Figure 2: Stretching to form a linear array

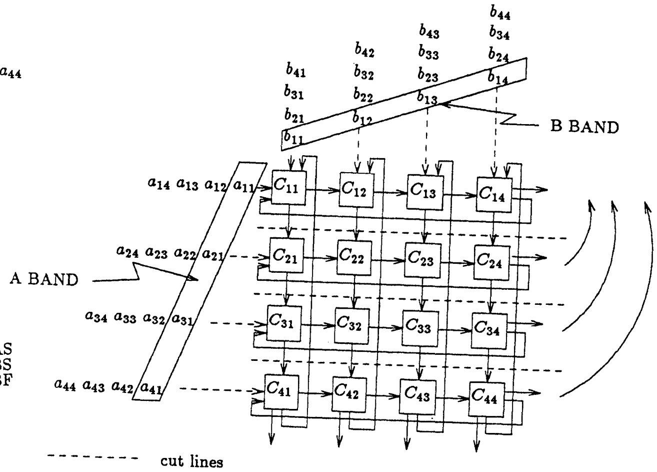

b_{44} b_{34} b_{24} b_{14}
b_{43} b_{33} b_{23} b_{13}
b_{42} b_{32} b_{22} b_{12}
b_{41} b_{31} b_{21} b_{11}
B BAND
A BAND
a_{14} a_{13} a_{12} a_{11}
a_{24} a_{23} a_{22} a_{21}
a_{34} a_{33} a_{32} a_{31}
a_{44} a_{43} a_{42} a_{41}
C_{11} C_{12} C_{13} C_{14}
C_{21} C_{22} C_{23} C_{24}
C_{31} C_{32} C_{33} C_{34}
C_{41} C_{42} C_{43} C_{44}
cut lines

Figure 6: Transitive Closure on 2-D Array

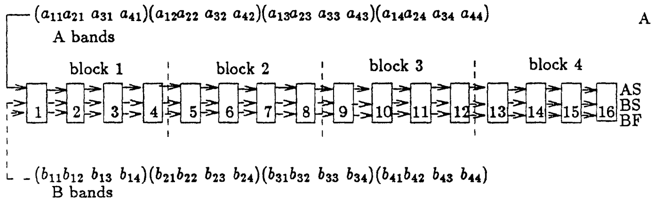

$(a_{11}a_{21}\ a_{31}\ a_{41})(a_{12}a_{22}\ a_{32}\ a_{42})(a_{13}a_{23}\ a_{33}\ a_{43})(a_{14}a_{24}\ a_{34}\ a_{44})$
A bands
block 1
block 2
block 3
block 4
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
AS
BS
BF
$(b_{11}b_{12}\ b_{13}\ b_{14})(b_{21}b_{22}\ b_{23}\ b_{24})(b_{31}b_{32}\ b_{33}\ b_{34})(b_{41}b_{42}\ b_{43}\ b_{44})$
B bands

Figure 3: Linear array fed with continuous data

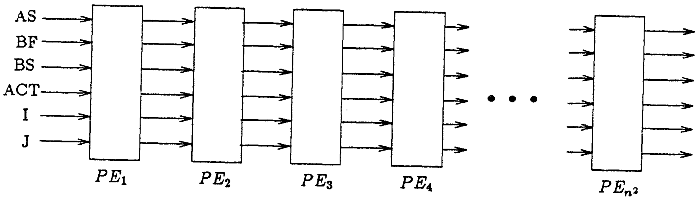


Figure 4: System structure of linear array for matrix multiplication

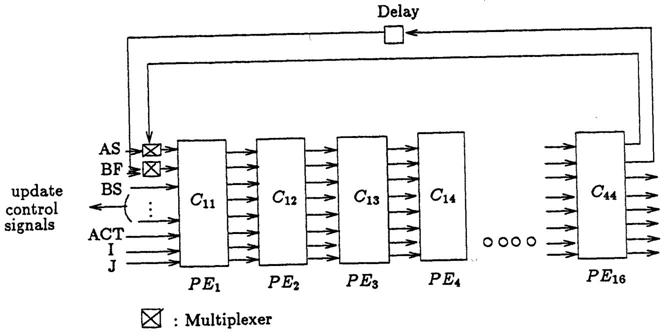


Figure 7: System structure for Transitive Closure on Linear Array for n=4

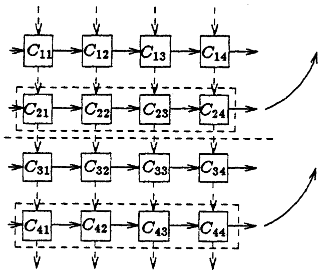


Figure 11: Partitioning 2-D array for n=4 and k=2

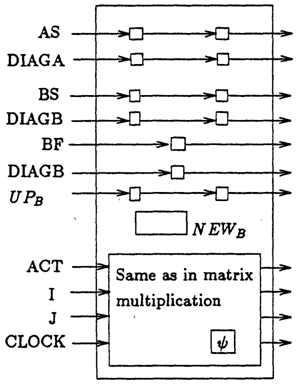


Figure 8: The PE structure for transitive closure

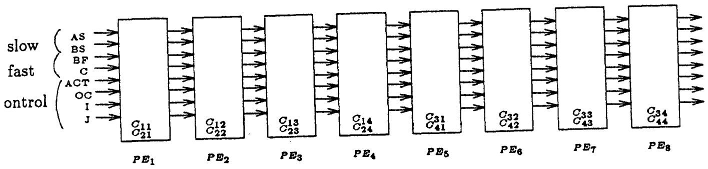


Figure 10: Mapping onto VMF model for n = 4 and s = 2

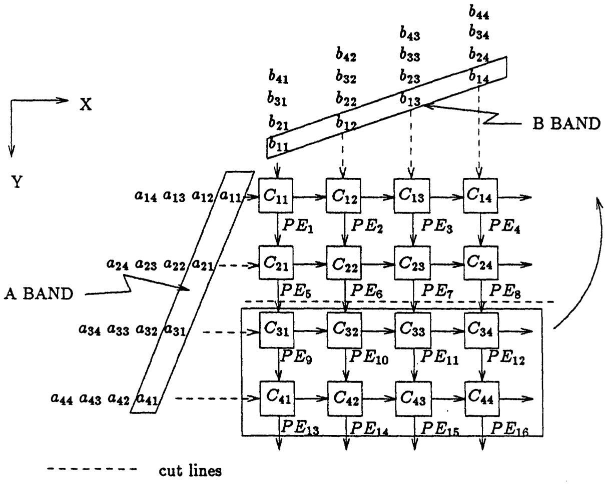


Figure 9: A Partition of a 2-D Array

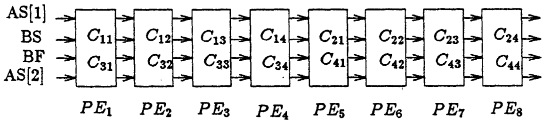


Figure 12: Mapping onto VCF model for n=2 k=2

CESAR – THE ARCHITECTURE AND IMPLEMENTATION OF A HIGH PERFORMANCE SYSTOLIC ARRAY PROCESSOR

Bard Tokerud, Vidar S. Andersen, Morten Toverud
Division for Electronics
Norwegian Defence Research Establishment
N-2007 Kjeller, Norway

Abstract

This paper describes the architecture and implementation of the CESAR computer system. The computing unit in CESAR has from one to four programmable systolic arrays working strictly in parallel, representing a SIMD (Single Instruction Multiple Data) structure. Each array consists of 128 custom designed processing elements capable of performing bit-serial operations on 32-bit data. Including control logic and memory units, a complete CESAR system with four systolic arrays is implemented on 13 circuit boards. Originally developed for processing of images from Synthetic Aperture Radar, CESAR is also suitable for other applications demanding extensive vector processing.

Introduction

Parallelism and pipelining are two classical concepts which have proven to be the keys to exploitation of the huge resources offered by today's VLSI technology.[1] In the CESAR computer system,[2-4] parallelism and pipelining are combined on different levels to achieve the necessary throughput for computationally intensive problems. Focusing on processing of images from Synthetic Aperture Radar (SAR), the CESAR computer is a result of comprehensive research and development activities at the Norwegian Defence Research Establishment over the past decade.

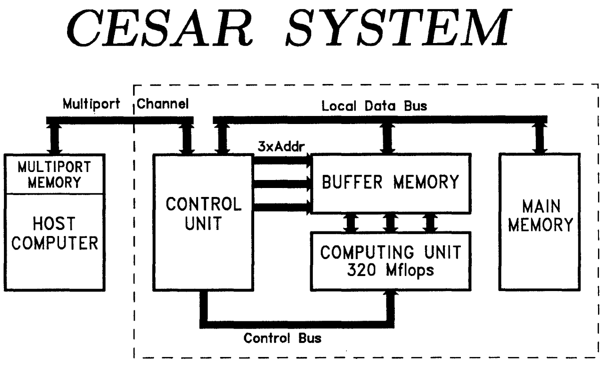


Figure 1 The CESAR Computer System

CESAR Architecture

The fundamental structure of the computing unit in CESAR resembles that of a systolic array architecture.[5,6] As shown in Fig.2 a), an 8x16 array of bit-serial processing elements operates on strings of data that flow regularly through the network and interact where they meet. Each serial element (S-element) is a custom designed 2μ CMOS chip, capable of performing 32-bit floating point or integer arithmetic and logic operations. In parallel with performing mathematical operations, an S-element allows data to be routed through. By adding programmable time delays for synchronization, computed results and bypassed data can be merged in neighbouring S-elements for new computations as shown in Fig.2 b).

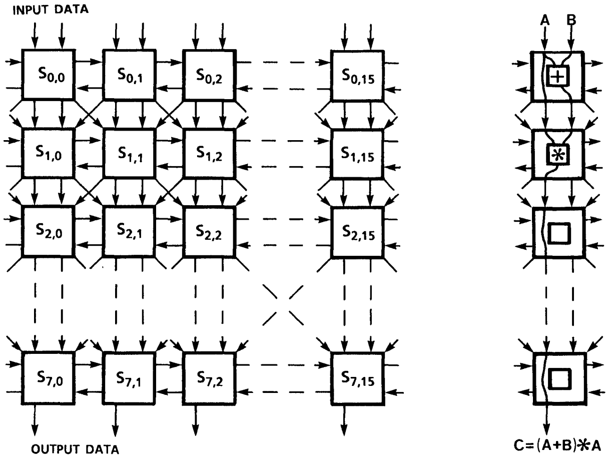


Figure 2 a)

Figure 2 b)

Figure 2 The Systolic Array of S-elements.

As shown in Fig.3, the two-dimensional array is configured as a cylinder, where pairs of input data are fed from the top, and the outputs are tapped at the bottom. When a pair of 32-bit input data have been fetched from memory, a serial conversion starts, whereby data is shifted into the selected column of the cylinder. With an internal cycle time of 50 ns, the total shift-in time for 32 bits becomes 32*50ns = 1600ns.

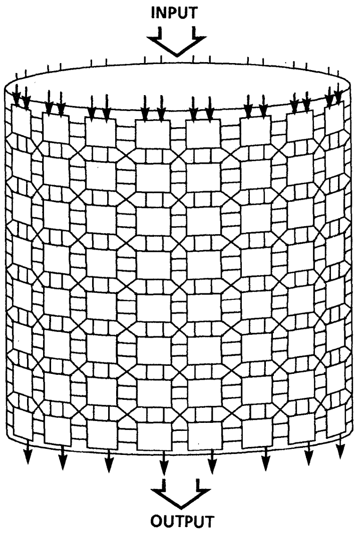


Figure 3 MALU - Microprogrammable Arithmetic Logic Unit

Since the buffer memory is capable of delivering a pair of 32-bit data every 100ns and also receiving a 32-bit result at the same speed, 16 columns can be run in parallel. Enabling each column of the cylinder successively, data flows in and out of the S-elements as continous bit streams consisting of 32-bit words lying head to tail. The parallel to serial conversion in MALU is accomplished by having three distributed shiftregisters in each column of the array.

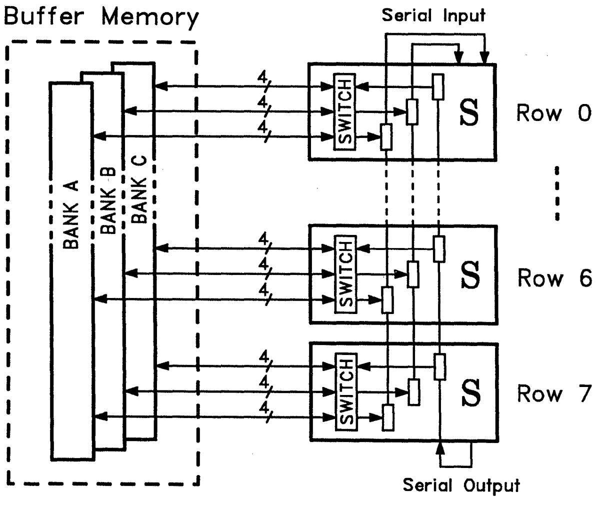


Figure 4 Parallel to Serial Converision in MALU

The S-element has a four bit parallel input/output register for each of the three data channels to the buffer memory. The eight S-elements in a column together form a 32 bit shiftregister whose output is shifted in at the top. Similarly, the results are serially output at the bottom and shifted upwards in their respective columns.

For many algorithms, the cylinder can be divided into strips, each strip performing the same pipeline of computations as its neighbouring strips. Representing a SIMD structure, this level of parallelism provides a high utilization of the computing power available in CESAR. Since each S-element is producing a result every 32 clock cycles, i.e every 1600ns, the theoretical maximum capacity of one complete MALU is ;

$$\frac{128}{1.6 * 10^{-6}} \text{ flops } = 80 \text{ Mflops}$$

This capacity is obtained when, for a certain algorithm, every S-element is doing a (floating point) computation. As inherent in the CESAR architecture, a complex algorithm utilizing many S-elements yields a higher performance than a simple algorithm occupying few elements.

MALU is fully programmable; that is, a combination of instruction words in the S-elements constitutes a MALU program. The S-elements are fitted with on-chip RAM with a capacity of 32 programs (instructions, routing and delay) and 32 constants for use in the computations. Changing MALU programs between two bursts of data is done by merely switching the global program address to the arrays. In the applications studied so far, the program memory has proven to be large enough to cover the entire algorithm without having to perform a program reload. Thus, all setup can be done in an initialization phase to avoid a degradation of the computational performance. In addition, each of the three data paths between the buffer memory (BUF) and MALU are easily configurable as either inputs or outputs, allowing consecutive refinements of the results without having to move data between BUF banks.

In the current version of CESAR, four identical pairs of MALUs and buffer memories are working strictly in parallel. During a computation, all the four MALUs execute the same program, but on different sets of data. These data (vectors) are located at the exact same addresses relative to the start of their buffer memories. The system has been designed to facilitate the distribution of input data to the four buffer memories and the collection of results without any extra overhead compared to a single MALU version. Listed in Table 1 is a selection of existing MALU algorithms and their actual capacity in a four-MALU version:

Name of Algorithm	Actual cap (Mflops)
FFT Radix 4 Butterfly	170
Convolution with 4 pt. filter	280
Addition of complex numbers	40
Complex Multiplication	120
Folding with 4 pt. filter	280

Table 1 Actual Capacity for Different Algorithms

Another commonly used way of measuring system performance is in terms of time required for a specific computation, e.g. a 1024 point complex FFT. On a four-MALU CESAR this typical signal processing application executes in 0.257 milliseconds (average) compared to 0.4037 milliseconds[7] on an 8.5 nanosecond Cray X-MP.

The execution time is specific for each instruction, which affects the time it takes from data enters the MALU array until the first results are ready at the outputs. This is often referred to as the tail of the computation pipeline and differs in length depending on the algorithm. For most signal processing algorithms the tail varies from 10-40 µsec, which for a 32k vector contributes 0.2%-0.6% of the total processing time. It should also be noted that once the array is filled with data, operands are presented and results are produced at the same rate independent of the program executed.

Compared to what we often see in other systolic arrays, MALU has several striking characteristics:

A. MALU.

- Each array element is capable of performing relatively complex operations.
- The array elements have rich connections to their neighbours (6 inputs, 6 outputs).
- The elements are individually programmable, and grouped together they form variable pipelines of computations.

B. Other known systolic arrays.

- Array cells are usually limited to simple bit-serial operations.
- Hardwired interconnections are often used between the elements.
- Each element is only capable of doing one, dedicated operation.

Hardware Realization

In Fig.5, a block diagram shows the different hardware modules in the prototype version of CESAR, which is currently in its final stage of debugging and testing. A full system with four MALUs is implemented on 13 PCBs, each of size 11' by 16'. Compared to other systems with approximately the same performance, the hardware is compact, and, due to the use of CMOS and TTL logic, small sized fans is the only cooling necessary. A brief description of the modules is given below:

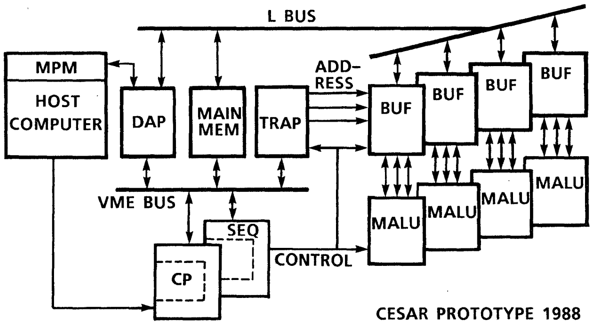


Figure 5 Hardware Modules in CESAR

- MALU (Microprogrammable Arithmetic Logic Unit)
  A complete array of 8x16 S-elements is fitted on one circuit board. The S-elements are packaged in 68 pin PLCCs which are surface mounted on both sides of the board. This rather complex hardware solution did, however, put some restrictions om the design of the S-element in terms of power consumption, and the 100K transistor chip only dissipates 0.25W at 20 MHz.

- BUF (BUFfer Memory)
  Each BUF contains three separate two-port 2 Mbyte static RAM banks for intermediate storage of MALU data.

- MAINMEM (MAIN MEMory)
  CESAR has 32Mbytes of main memory for storage of intermediate data when BUF space is inadequate.

- TRAP (TRiple Address Processor)
  The three bit-slice address processors on TRAP are necessary for selecting the correct data to be sent into the MALU and addressing the storage area for the results. Each address processor is programmable for different addressing algorithms, e.g. data stored with fixed increments or FFT bitreversing.

- CP (Control Processor)
  CP is based on the Motorola 68020 microprocessor and is responsible for the overall control in CESAR. The application programs written in the high level language CESAR Pascal[8] as well as system software are executed in CP. A local VMEbus[9] is used to interchange control information between CP and the other hardware modules in CESAR.

- SEQ (SEQuencer)
  The Sequencer provides the detailed control signals for the CESAR computations. It synchronizes the address generation in TRAP with the internal computations in MALU to ensure correct dataflow between BUF and MALU.

- DAP (DAta Port)
  The Dataport controls all DMA transfers between separate memory modules, i.e the buffer memories, main memory and the multiport memory residing in the host computer. DAP also enables CP to access any location in the different memories.

As can be seen in Fig.5, the system is flexible with respect to memory access. Controlled and addressed by the Dataport (DAP), the physical data transfers take place on the Local Data Bus called LBUS. LBUS is a 40 Mbyte/s data channel capable of serving all four BUFs with altogether 12 connections to the MALUs.

The complexity of the MALU circuit board makes it hard to debug in production and in the field. To help solving this problem, the S-element has a built-in selftest option that enables the system or user to run a parallel diagnostic in all

512 S-elements in CESAR. The selftest, which is based on signature analysis, tests the entire chip with exception of the on-chip static RAM. The RAM is verified by the control processor before the selftest is initiated. The S-element can also be set in a special mode to enhance the testability during production testing, reducing the number of testpatterns significantly.

Programming the System

In parallel with designing the hardware, a substantial effort has been put into the development of software tools for programming and debugging of the CESAR system. At the application level, a high order language called CESAR Pascal has been developed.[8] In addition to standard Pascal, it includes special features for describing and synchronizing concurrent processes as well as data transfers between the memory modules inside and outside CESAR.

Typically, a library of the most commonly used vector-/signal processing algorithms will be supported. If, however, the user wants to write his own MALU or TRAP programs, several tools are available. A graphic editor for MALU programs allows the user to interactively choose instructions and create data paths between the S-elements in the array. An assembler automatically adds routing delays for synchronization, and a simulator verifies the correctness of the algorithm. Similarily, the address processors are programmable in a "C"-like language with constructs for generating complex address sequences. A TRAP simulator is developed to check the address programs before downloading to the hardware.

Conclusion

A major goal in the research and development of the CESAR computer system has been to create a powerful number cruncher for processing of SAR images, while retaining a low cost/performance ratio. Preliminary studies have also shown that the CESAR architecture provides the necessary flexibility to solve other computationally intensive vector problems, such as the ones in seismic and metheorological processing.[4] Also in a variety of other applications, the ever increasing demand for extensive computing capacity clearly manifests the need for unconventional, high performance designs like CESAR.

Commonly Used Terms

BUF	Buffer memory
CESAR	Computer for Experimental Synthetic Aperture Radar
FFT	Fast Fourier Transform
LBUS	Local Data Bus
MALU	Microprogrammable Arithmetic Logic Unit
Mflops	Million Floating Point Operations per Second
PCB	Printed Circuit Board
PLCC	Plastic Leadless Chip Carrier
SAR	Synthetic Aperture Radar
TRAP	Triple Address Processor
VLSI	Very Large Scale Integration

References

1. L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," *IEEE Computer,* Volume 15, 1, (Jan. 1982), pp. 47-56.

2. V. Andersen, T. Haugland, and O. Sorasen, "CESAR - A Programmable Systolic Array Multiprocessor System", *Proc. IEEE First International Conference on Supercomputers,* (Dec. 1985), pp. 8-15.

3. V. Andersen, and T. Haugland, "CESAR - A Programmable Systolic Array Multiprocessor System", *NDRE Report-86/7020,* (Aug. 1986), 34 pp.

4. O. Sorasen, "CESAR-maskiner med utvidete muligheter", *NDRE Report-87/7068,* (June 1987), 56 pp.

5. H.T. Kung, "Why Systolic Architectures?", *IEEE Computer,* Volume 15, 1, (Jan 1982), pp. 37-46.

6. J.A.B. Fortes, and B.W. Wah, "Systolic Arrays - From Concept to Implementation", *IEEE Computer,* Volume 20, 7, (July 1987), pp. 12-17.

7. G.R. Lang et al. "An Optimum Parallel Architecture for High-Speed Digital Signal Processing", *IEEE Computer,* Volume 21, 2, (Feb. 1988), pp. 47-57.

8. D. Belsnes, O. Hanseth, S. Meldal, "The HOLM Language, a Proposal", *Norwegian Computing Center Report no. 728, ISBN 82-5390209-3* (Dec. 1982), 144 pp.

9. Micrology pbt, Inc. "VMEbus Specification Manual, Revision C.1", (Oct. 1985), 263 pp.

SIGNAL GRAPHS: A MODEL FOR DESIGNING CONCURRENT LOGIC

A.Yu.Kondratyev, L.Ya.Rosenblum, A.V.Yakovlev

Computing Science Department
Leningrad Electrical Engineering Institute
Leningrad 197022 USSR

Abstract -- Asynchronous digital circuits exhibit a high degree of concurrency. Self-timed implementation is the most appropriate design discipline for them. We examine the signal graphs that are subject to formal treatment and mechanical translation to delay-insensitive circuits. An example of designing a piece of logic for typical interface adapter effectively illustrates the approach and sheds light on future work.

1. Introduction

Modern technologies allow to build VLSI circuits whose internal behavior exhibits a high degree of parallelism. To operate correctly under the presence of such undesired phenomena as electronic metastability, signal skews due to higher values of wire vs gate delay ratios, parametric instabilities of gates etc. these circuits are designed using self-timed, or delay-insensitive fashion [1,2]. The most widely cited examples of concurrent hardware are regular structures like pipeline and wavefront arrays which are easily decomposed in sequential,parallel or recursive way. On the other hand such objects as asynchronous interface adapters which are a lot less regular but can be equally concurrent are far from being attempted at a formal treatment as they have been the privelege of engineers using normally timing diagrams or flow charts.

The ultimate goal of our research is to mechanize the design process to such a degree when it is comfortably fitted in a CAD environment for developing distributed systems, e.g. for translating a physical layer protocol specification into a collection of self-timed modules. This paper demonstrates the technique of using a formal model of concurrency for constructing basic units of interfacing logic. This technique accomodates a stepwise design procedure involving such steps like architectural decomposition, functional specification of components, their behavioral signalling expansion, and its validation with respect to correctness and completeness notions, and finally Boolean function derivation.

2. Modelling concurrency in logic

A self-timed system is often regarded as a collection of self-timed modules that communicate via asynchronous protocols [1]. It does not require a global clock. All system level events are ordered in time by the causal relations between the modules actions. The order as it has been established by the designer must further be preserved in a final circuit thereby guaranteeing the correct operation independently of element and wire delays.

The evolution of logic design methods shows that the Huffman state machine model is no longer an adequate model for asynchronous logic since it can not deal with "granulated" concurrency in VLSI. The existing formal models for self-timed VLSI systems can be split into four groups:
(i) graphical notations, state or event oriented, like Petri nets, transition diagrams, parallel flow charts etc.;
(ii) symbolic notations, like traces or path expressions;
(iii) models based on high level programming languages, e.g. Ada-like notation;
(iv) combined models.

The study of these formalisms shows that the usefulness of a model for the self-timed circuit design depends on a large number of various issues. For example, it is affected by the structure type (regular vs non-regular, or data-flow vs control-flow), the degree or granularity of parallelism and data dependence, the necessity of abstract data typing, the depth of delay-independence (with respect to transistor, gate or component level).

Our formalism, a signal graph based on a subclass of Petri nets, is an effective substitute for widely used timing diagrams because it can be analyzed in a mathematically sound manner and mechanically translated to Boolean functions implementation.

3. Signal graphs: properties and analysis

Signal graphs are very attractive formal model for analyzing behavioral specifications of both signalling protocols and corresponding interface logic. They represent a more narrow class of processes than that that can be generally defined by, say, Petri nets. This is concerned with their inability to define alternatives in processes. However, when we need to define a highly concurrent behavior they provide the succinct description and what is more important, the polynomially complex analysis.

We presume some knowledge of Petri nets and their subclasses, particularly marked graphs. Marked graph (MG) generates distributive marking diagram (MD) 3 . MD is an oriented graph whose vertices are reachable markings and arcs are labeled with firing transitions. The term "distributivity" is related to the lattice which can be defined on a set of vectors of transition firing numbers with respect to a given initial marking.

In order to define a signal graph a set of binary variables (signals) $Z = \{z_1, z_2, ..., z_n\}$ is introduced. We denote transitions of signal z_i: from 0 to 1 by $+z_i$ and from 1 to 0 by $-z_i$.

Signal graph (SG) is defined as an MG in which vertices are labeled with signal transitions (changes) of the form dz_i where $d \in \{+,-\}$.

We call a labeling function conflict-free if for each reachable marking and variable z_i there is at most one enabled vertex labeled with dz_i . SG with a conflict-free labeling is called coherent. The coherence is not sufficient for the specification to be correct because despite all the changes for each z_i are linear-ordered they may be unmatched with respect to their signs.

We call a labeling function sign-balanced if for each sequence of signal transitions with respect to initial marking between any two transitions of the same sign there exists at least one transition of the other sign. SG with a sign-balanced labeling is called consistent. The consistency implies the necessary level of correctness of a specification given by SG that is expressed in the following statement.

Statement 1. A consistent SG generates a state transition diagram.

A state transition diagram (STD) is an oriented graph whose vertices are labeled with full states of a specified circuit, i.e. they are binary n-tuples of values of z_i , and arcs are labeled with corresponding changes dz_i. The values that can change between a given state and another one connected to each other by an arc are marked with *-token. A variable whose value in n-tuple is marked with * is called excited in a given state. In this paper we omit the description of algorithms of converting a consistent SG to STD and vice versa. We only hint that such a conversion may use the ordinary procedure of building an MD by the depth-first search where each marking in MD relates to a corresponding state in STD.

A consistent SG may however generate a STD with multiple states, i.e. the states which are labeled with equal n-tuples of signal values. Such an STD is called contradictory. Informally, the contradiction of this kind means that the system is under-specified and some components are still hidden from the designer's eye. For example, when SG defines an interface protocol these components may be interpreted as an internal memory of controller.

We further incorporate a higher level of correctness into the hierarchy of SG classes by the notion of a normal SG which guarantees the completeness of a specification. An SG is called normal if it is consistent and for each allowed sequence of markings it has no proper subset of variables $Z' \subset Z$ which can proceed through the full cycle of their values while other variables (from $Z \setminus Z'$) stay unchanged. An STD of a normal SG is non-contradictory and distributive [3].

It is suitable to check the consistency and normalicy using the relations of precedence and concurrency built on the set of signal transitions. The formalization of these relations requires the introduction of a concept of a history, or so-called unfolding, which is an infinite and acyclic object generated by an SG. Each occurrence of a transition in an SG yields a unique vertex in the unfolding. This technique due to the lack of space can not be fully described here though we mention that the unfolding can be floored to its first two periods and the above relations can thus be computed on a finite object. The algorithm of checking consistency has the complexity of $O(n^3)$ where n is the number of vertices in the original SG.

In order to establish whether a consistent SG is normal we use a special formal concept - operational coupledness. We define a coupled relation on a set of variables Z. This relation has the following hierarchy: directly strongly coupled, strongly coupled, weakly coupled of rank r , $r \geq 0$, and coupled. The coupled relation partitions the set Z into the disjoint classes. Omiting here formal definitions and proofs which can be found elsewhere [4] we only state the following.

Statement 2. A consistent SG is normal iff all its variables belong to single coupledness class.

The complexity of an algorithm for normalicy check is of $O(n^4)$.

The main advantage of our checking techniques stems from the fact that they do not require to convert an SG to

MD or STD - a step having exponential complexity with respect to the power of Z.

4. An example of self-timed logic design

In the above section we have sketched how we can check the normalicy of an SG which is a sufficient condition for the existence of a distributive STD and hence of a delay-insensitive circuit [2] . The circuit can be derived from the normal SG by means of obtaining the Boolean functions (BFs) for variables z_i of set Z using a truth table (TT) which can be built from the STD corresponding to the SG. However the chain SG-STD-TT-BFs involves exponentially complex steps. Therefore we look for an alternative technique for the direct (but semantics preserving) conversion of the SG to the system of BFs. Such a bridling of the design complexity is concerned, first of all, with laying out some restrictions upon the complexity of the coupledness hierarchy.

In this paper we are far from being ambitious to show how the problem of obtaining the general way of deriving functions directly from an SG can be solved. We rather illustrate our design approach with an instructive example of designing a piece of interface logic.

FIFO buffers are typically incorporated in interfacing adapters as they help to keep the performance of the whole distributed system at its highest communication rate. The original specification of a one-value FIFO cell was inspired by [5] .

Let the FIFO cell consist of two subcells: the data cell (DC) and the control cell (CC) as shown in Fig.1.

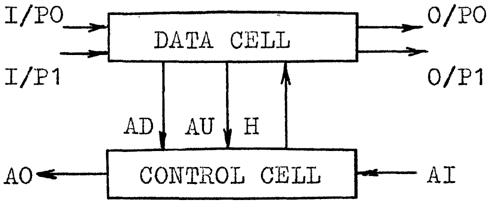


Figure 1. The structure of FIFO cell

The meaning of the signals is as follows. I/P0 and I/P1 are data inputs, and O/P0 and O/P1 are data outputs. Both use the two-rail coding discipline [1] where for "zero" and "one" values the combinations 10 and 01 are respectively used on the above pairs, and the all-zero spacer (00) is used for representing the "data undefined" value. AO and AI are the acknowledgement signals: AO is generated by the cell and AI is produced by the environment. AD is "All defined" indication signal, AU is "All undefined" indication signal, and H is "Hold" command signal. AD and AU are both used to detect the state of the inputs (if I/P0 = I/P1 = 0 then AD = 0, AU = 1, and if I/P0 ≠ I/P1 then AD = 1, AU = 0). H directs the DC to latch the incoming value. All AD, AU, and H wires run the width of the buffer.

Fig.2 shows the SG specification of the CC operation. Analyzing this SG we can establish that it is consistent: each variable has all its transitions ordered within one synchrocycle (a cycle containing exactly one token). However the SG is not normal. The coupled relation partitions the set Z = {AI,AO,AD,H,AU} into two disjoint classes: K1 = {AI,H} and K2 = {AD,AU, AO} . It can be shown that adding only one extra variable to the specification while preserving the established order of signal changes for variables in Z will not suffice for making all variables coupled. After adding two variables d1 and d2 we obtain the resulting SG shown in Fig.3 which is normal.

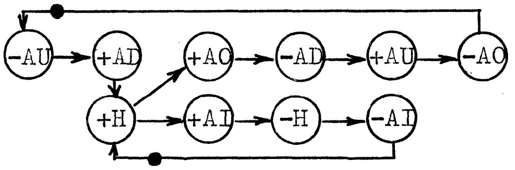


Figure 2. An original SG specification of the control cell operation

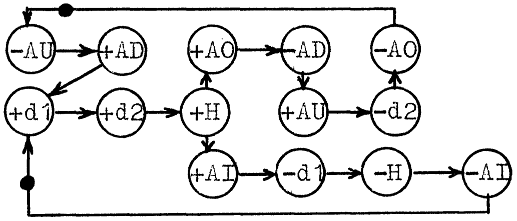


Figure 3. A normal SG obtained after adding extra variables

From this SG we derive BFs in the following form:

$$z = Sz + \overline{Rz} \cdot z,$$

where Sz is the set function and Rz is the reset function. Both Sz and Rz are independent of z. We also demand that the invariant $Sz \cdot Rz = 0$ holds in order to avoid conflicts between transitions which may lead to undesired races in a circuit.

In order to derive Sz and Rz we search through the SG for immediate predecessors of the transition +z for including them into the essential Sz-term, and those of transition -z for Rz-term. If these predecessors correspond to the variables that are strongly coupled with z we proceed further to the orthogonalization step. If some of the variables whose transition is a predecessor for a given transition dz is weakly coupled (of any rank $r \geqslant 0$) with z then there is a so-called overtaking of the essential term by some other term which must be added to corresponding set or reset function. For example, when deriving function S_{AO} the essential term is $H \cdot d2$ but before AO changes from 0 to 1 H may begin to change from 1 to 0 (in parallel with AO changing), and hence we must cure the overtaking by an additional term which will involve a variable that is strongly coupled with H, i.e. the term $\overline{d1} \cdot d2$. Using d2 in both terms for S_{AO} helps us also to eliminate the inclusion of the term for R_{AO} which is simply $\overline{d2}$ because d2 is immediately strongly coupled with AO. Thus we obtain a BF for AO which is non-selfdependent, i.e. free of feedback

$$AO = (H + \overline{d1}) \cdot d2 + d2 \cdot AO = (H + \overline{d1})d2.$$

One of the important issues in deriving Sz and Rz is their mutual orthogonalization, i.e. providing that $Sz \cdot Rz = 0$ is satisfied. This can be done by strengthening their terms with common variables. For example, when we obtained S_{AO} we had d2 as such a variable. Another example is the function for H whose $S_H = d1 \cdot d2$ is strengthened by d1 because $R_H = \overline{d1}$.

Finally, the above technique yields the following system of BFs:

$$\begin{aligned}
d1 &= AD \cdot \overline{AI} \cdot \overline{d2} + \overline{AI} \cdot d1 \\
d2 &= \overline{AU} \cdot \overline{H} \cdot d1 + \overline{AU} \cdot d2 \\
H &= d1 \cdot d2 + d1 \cdot H \\
AO &= \overline{d1} \cdot d2 + d2 \cdot H
\end{aligned}$$

This system is easily implemented with four AND-OR-NOT gates and six inverters (four of them produce d1,d2,H,AO, and the other two complement AI and AU, however the latter can obviously be eliminated at the transistor level by using the inhibit inputs of the first two gates).

The circuit is delay-insensitive with respect to delays in gates and inverters as well as in those wires which are not the feedback connections. The feedback delays are presumed negligible as the corresponding elements are accommodated within equichronic regions.

5. Conclusion

The main characteristic of the above approach comparing it with those given elsewhere[6,7,8]is that it provides the technique for effective managing with concurrency at the logic level using the formal model which is quite simple for comprehension for a wide audience of hardware designers used to timing diagrams, and at the same time powerful enough to be formally analyzed with respect to correctness and completeness by means of such key concepts as normalicy and coupledness. This facilitates some constructive ways to the correction of specifications while preserving the original semantics of signal change ordering. The method has been tested on a large number of difficult examples including designing asynchronous control logic for interfaces (Unibus, Futurebus, token ring etc.) and FIFO buffers of various architectures.

The proposed technique obviously needs further research efforts both in theory as, for example, in establishing restrictions on coupledness classes to find out how they affect the BF derivation rules outlined above, and in practical aspects through developing the software for such a mechanized translation to be a versatile interactive design environment. Some pieces of such an environment are in progress now.

References

[1] C.L. Seitz, System Timing, Chapter 7 in: Introduction to VLSI Systems, C.Mead and L.Conway, Addison-Wesley, (1980), 400 pp.

[2] V.I.Varshavsky, Hardware Support of Parallel Asynchronous Processes, Digital Syst. Lab., Helsinki University of Technology, Series A, No.2, (Sept. 1987), 236 pp.

[3] L.Ya. Rosenblum, A.V. Yakovlev, Signal Graphs: from Self-Timed to Timed Ones, Intern. Workshop on Timed Petri Nets, Torino, Italy,(1985), pp.199-207.

[4] A.V. Yakovlev, Design and Implementation of Asynchronous Interface Protocols, PhD Thesis, (1982).

[5] E.E. Barton, Non-metric Design Methodology for VLSI, in: VLSI-81, Academic Press, London, (1981), pp. 25-34.

[6] A.J. Martin, Compiling Communicating Processes into Delay-Insensitive VLSI Circuits, Distributed Computing, Vol. 1, No. 4 (1986), pp. 205-225.

[7] T.-A. Chu, On the Models for Designing VLSI Asynchronous Digital Systems, Integration, the VLSI journal, Vol.4 (1986), pp. 99-113.

[8] P.F. Lister, A.M. Alhelvani, Design Methodology for Self-Timed Systems, Proc. IEE, Pt. E, Vol. 132, No.1 (1985), pp 25-32.

Optical Arithmetic Using Signed-Digit Symbolic Substitution

Kai Hwang and Ahmed Louri
Department of EE-Systems
University of Southern California
Los Angeles, California, 90098-0781

Abstract A new class of digital arithmetic algorithms is presented in this paper for supporting massively parallel computing with state-of-the-art optical technology. We use a two-dimensional symbolic substitution approach. Signed-digit (SD) representation is used to enable carry-free addition/subtraction. Based on SD addition, parallel algorithms for SD multiplication and division are developed. The potential advantages of performing digital arithmetic with optics include the significant increase in speed, full exploitation of massive parallelism, higher communication bandwidth, and higher system throughput; as compared with existing electronic arithmetic computers. We concentrate on optical computing using the signed digit set $\{\bar{1}, 0, 1\}$. The parallel algorithms being presented can be easily extended to perform optical arithmetic with higher radices.

1 Introduction

The *signed-digit* (SD) representation was originally proposed by Avizienis[1], and recently introduced to the optical community by Drake et al.[2]. The binary SD system uses the digit set $\{\bar{1}, 0, 1\}$, where $\bar{1}$ stands for -1. The introduction of redundancy (three values for a binary system) provides a much weaker interdigit dependency as opposed to the strong dependency manifested by carry propagation in a nonredundant number system using the digit set $\{0, 1\}$. As a consequence of weak dependency, carry generated at any stage is confined within two adjacent digital positions in the SD code. This makes it possible to perform the addition/subtraction of any two SD numbers of arbitrary length in constant time[1,3].

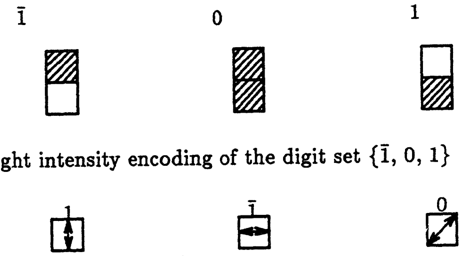
Based on the SD *addition*, we have developed new algorithms for SD *multiplication* and SD *division*. The multiplication of two n-digit SD numbers is done in $O(\log_2 n)$ time by first generating all the n partial products simultaneously and then adding them in a tree-like fashion. The parallel generation of all partial products is done in constant time, independent of the word length n. It is the adder tree that requires $\log_2 n$ time. The SD division algorithm is generalized from the quadratic convergence division method[4]. With the provision of high-speed multiplication and parallel addition, the number of required iterations for SD division is reduced to $O(l\sigma g_2 n)$, where n is the fraction length. The advantages of optics have been expounded upon on numerous occasions[5,6]. These include high space-bandwidth and time-bandwidth produts, and inherent parallelism.

2 Symbolic Substitution Technique

In order to exploit the massive parallelism and ultrahigh speed in optics, Huang[7] introduced a technique called *symbolic substitution* (SS) for performing digital arithmetic optically. In his method, information is represented by optical patterns within a two-dimensional image. An optical pattern is a spatial arrangement of dark and bright spots corresponding to binary values 0 and 1. Computation proceeds in transforming these patterns into other patterns according to predefined SS rules. Symbolic substitution logic is sensitive not only to the values of *pixels* (picture elements) carrying information, but also to their spatial locations in the binary image (image of bright and dark spots).

In order to implement SD arithmetic optically, we need an optical encoding for the digit set $\{\bar{1}, 0, 1\}$. There are several properties of light that can be used. These include *light intensity* and *light polarization* as illustrated in Fig.1a-b. Using light intensity, two pixels of different light intensity are needed to encode the three digits. A possible encoding scheme is to represent the digit 1 by a bright pixel above a dark one, the digit $\bar{1}$ by a reversed pixel pattern, and the digit 0 by two dark pixels as shown in Fig.1a. Note that, the extra pattern consisting of two bright pixels can be used as a delimiter to denote the fraction point. Using light polarization we need three states of polarization. A possible encoding scheme would be to represent 1 by vertically polarized light, $\bar{1}$ by horizontally polarized light, and 0 by light polarized at 45° as shown in Fig.1b. In this paper, we have chosen to represent the digit set with light intensity exclusively.

Symbolic substitution consists of two phases: a *recognition phase* where the presence of a specific pattern is detected within a binary image and a *substitution phase*, where the present pattern is replaced by another pattern according to a predefined SS rule. Optical implementation of the two SS phases have been investigated by several researchers[8,9,10].



(a) Light intensity encoding of the digit set {$\bar{1}$, 0, 1}

(b) Light polarization encoding of the digit set {$\bar{1}$, 0, 1}

Fig.1 Optical encoding of the signed-digit set $\{\bar{1},0,1\}$

3 Optical SD Addition/Subtraction

Given an SD number $Y = y_{n-1}y_{n-2}\cdots y_0.y_{-1}\cdots y_{-m}$, the algebraic value of Y is evaluated as :

$$Y_v = \sum_{i=-m}^{i=n-1} y_i \times 2^i, \quad \text{where } y_i \in \{\bar{1},0,1\} \qquad (1)$$

In this number system, there is no need for an explicit sign digit. In fact, the polarity of the most significant digit y_{n-1} determines the sign of Y. Although the representation of an SD number is not unique, the zero (0) is uniquely represented with all zero digits.

The addition of two SD numbers represented as $X = x_{n-1}\cdots x_0.x_{-1}x_{-2}\ldots x_{-m}$ and $Y = y_{n-1}\cdots y_0.y_{-1}y_{-2}\ldots y_{-m}$ results in an SD number $S = s_n s_{n-1}\cdots s_0.s_{-1}s_{-2}\ldots s_{-m}$. Avizenis has defined three pipelined steps to perform the SD addition[3]. At the first step, $x_i + y_i = 2t_{i+1} + w_i$ is performed at the i-th digit position, for $i = -m,\ldots,n-1$, where w_i and t_{i+1} are called the ***interim sum digit*** and the ***transfer digit*** respectively. These digits assume the following values:

$$w_i = \begin{cases} 1 & \text{if } x_i + y_i = \bar{1} \\ 0 & \text{if } |x_i + y_i| \neq 1 \\ \bar{1} & \text{if } x_i + y_i = 1 \end{cases} \quad t_{i+1} = \begin{cases} 1 & \text{if } x_i + y_i \geq 1 \\ 0 & \text{if } x_i + y_i = 0 \\ \bar{1} & \text{if } x_i + y_i \leq \bar{1} \end{cases} \quad (2)$$

At the second step, $w_i + t_i = 2t'_{i+1} + w'_i$ is performed to produce another pair of digits, w'_i and t'_{i+1}:

$$w'_i = \begin{cases} 1 & \text{if } w_i + t_i = 1 \\ 0 & \text{if } |w_i + t_i| \neq 1 \\ \bar{1} & \text{if } w_i + t_i = \bar{1} \end{cases} \quad t'_{i+1} = \begin{cases} 1 & \text{if } w_i + t_i = 2 \\ 0 & \text{if } |w_i + t_i| \neq 2 \\ \bar{1} & \text{if } w_i + t_i = -2 \end{cases} \quad (3)$$

The third step generates the final *sum digit*, s_i, as specified below:

$$s_i = w'_i + t'_i = \begin{cases} 1 & \text{if } w'_i + t'_i \geq 1 \\ 0 & \text{if } w'_i + t'_i = 0 \\ \bar{1} & \text{if } w'_i + t'_i \leq \bar{1} \end{cases} \qquad (4)$$

Figure 2 shows a totally parallel adder constructed by three types of optically implemented logic Cells (I, II, III), whose truth-table specifications are given in Table 1. There is no carry propagation beyond any two adjacent digits in the adder. Each sum digit s_i depends on only six digits $(x_i, y_i), (x_{i-1}, y_{i-1})$, and (x_{i-2}, y_{i-2}). Zeros are padded at the second and the third stages to preserve the same input/output format at each stage.

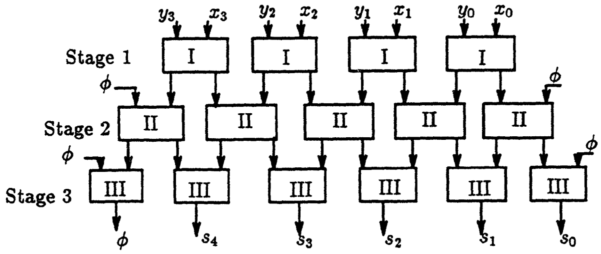


Fig2. A totally parallel optical adder with 3 pipeline stages

Example 1 below illustrates the addition of two SD numbers, $X = (-0.125)_{10} = (\bar{1}.111)_{SD}$, and $Y = (0.375)_{10} = (0.10\bar{1})_{SD}$ using the same 3-stage adder shown in Fig.2. The result is an SD number $S = (0.25)_{10} = (000.1\bar{1}0)_{SD}$. In this example, ϕ represents a padded zero.

Signed-digit *subtraction* is performed by first negating the nonzero digits of the subtrahend and then performing the addition of the two operands. Since the negation operation can be done in parallel for all digits, subtracting two SD numbers can also be done in parallel across all digits.

Example 1(SD Addition)

$X = (\bar{1}01\bar{1})_{sd}$			$= \bar{1}$	0	1	$\bar{1} = (-7)_{10}$	
+							
$Y = (010\bar{1})_{sd}$			$= 0$	1	0	$\bar{1} = (3)_{10}$	
Stage 1		ϕ	1	$\bar{1}$	$\bar{1}$	0	w_i
		$\bar{1}$	1	1	$\bar{1}$	ϕ	t_{i+1}
Stage 2	0	1	0	0	0	0	w'_i
	ϕ	$\bar{1}$	0	$\bar{1}$	0	ϕ	t'_{i+1}
Stage 3	0	0	0	$\bar{1}$	0	0	s_i

$Z = (00\bar{1}00)_{sd} = (-4)_{10}$

Using the truth tables in Table 1, we derive below a set of SS rules required for optical implementation of the SD addition. The search patterns of these rules correspond to the input combinations and the replacement patterns are the truth table entries as shown in Fig.3. Note that for Cell Type I and II, the replacement patterns are spatially displaced by one digit position, which accounts for the fact that the transfer digits (t_i and t'_{i+1} respectively) are to be combined with the next higher-order digit in the addition process.

On the surface, it seems that we need $3^3 = 27$ SS rules corresponding to the nine entries of each of the 3 truth tables. However, a closer look at Table 1, reveals that the logic for the first and the second stages are very similar. Furthermore, if we pad the third stage output with 0, five of the nine entries become similar to stages 2 and 3. Therefore, the total SS rules needed for SD addition becomes 17. In fact, when the search pattern is all dark (both operands digits are 0) the replacement pattern is also all dark, which does not need any optical processing. Consequently, the actual number of useful rules for the SD addition becomes 16. The subtraction needs one extra stage to perform the digit-wise negation. This stage requires two additional SS rules to negate the nonzero digits as shown in Fig.3d.

Table 1 Truth-table of three Cell Types used in designing the optical adder in Fig.2

Type I Cell

$x_i \backslash y_i$	1	0	$\bar{1}$
1	t 1 w 0	1 $\bar{1}$	0 0
0	1 $\bar{1}$	0 0	$\bar{1}$ 1
$\bar{1}$	0 0	$\bar{1}$ 1	$\bar{1}$ 0

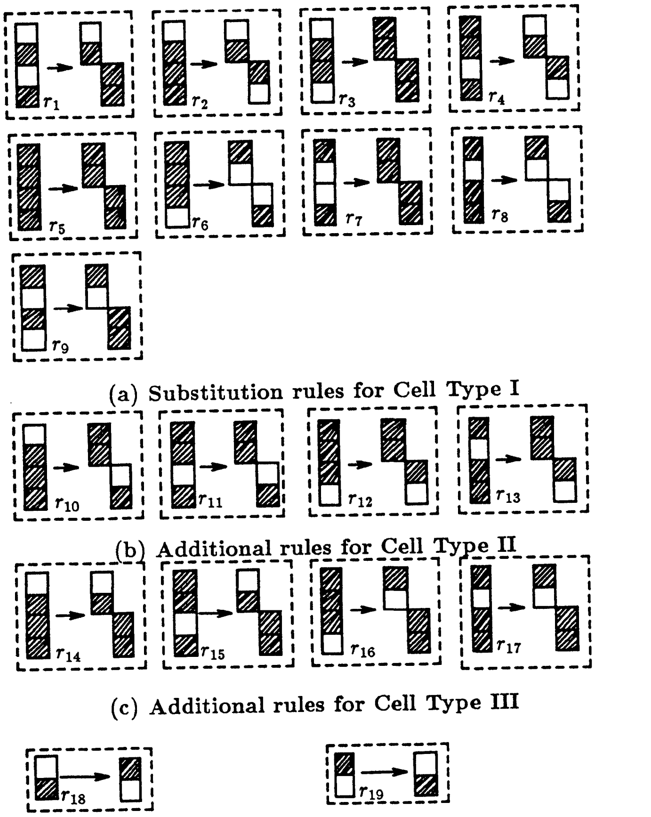
Type II Cell

$t_i \backslash w_i$	1	0	$\bar{1}$
1	t' 1 w' 0	0 1	0 0
0	0 1	0 0	0 $\bar{1}$
$\bar{1}$	0 0	0 $\bar{1}$	$\bar{1}$ 0

Type III Cell

$t'_i \backslash w'_i$	1	0	$\bar{1}$
1	s 1	1	0
0	1	0	$\bar{1}$
$\bar{1}$	0	$\bar{1}$	$\bar{1}$

To illustrate the use of these SS rules, let us consider Example 1 in light of 2-D symbolic substitution. The input operands are optically encoded and stacked on each other as illustrated in Fig.4a. Next, the SS rules for Cell type I are applied to the input image. All nine input combinations are searched and then replaced in parallel. This results in 3 successive new images as shown in Fig.4b-d, corresponding to the outputs of the 3 adder stages.



(a) Substitution rules for Cell Type I

(b) Additional rules for Cell Type II

(c) Additional rules for Cell Type III

(d) substitution rules for signed-digit negation

Fig.3 Optical symbolic substitution rules for signed-digit addition and negation

In Fig.5 we show a schematic block diagram for an optical digital adder using the signed-digit symbolic substitution technique. Note that 17 rules are used. The optical implementation of each substitution rule is detailed in [10]. there are other methods that have been reported to implement the SD addition optically[11,12].

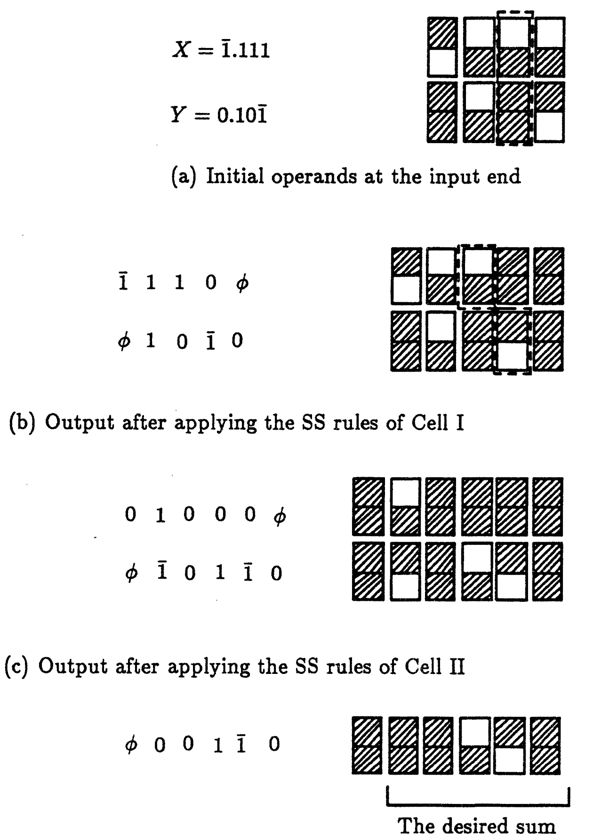
4 Optical SD Multiplication

The optical multiplication of two SD numbers $X = x_{n-1}\cdots x_0.x_{-1}x_{-2}\cdots x_{-m}$ and $Y = y_{n-1}\cdots y_0.y_{-1}y_{-2}\cdots y_{-m}$ produces an SD product $P = p_{2n-1}p_{2n-2}\cdots p_0.p_{-1}p_{-2}\cdots p_{-2m+1}p_{-2m}$, expressed as:

$$P = (y_{n-1} * X) \times 2^{n+m-1} + \cdots + (y_{-m} * X) \times 2^0 (5)$$

where y_i is the i-th multiplier digit, and $*$ is the signed AND operation defined as follows for any $x, y \in \{\bar{1}, 0, 1\}$:

$$x*y = \begin{cases} 1 & \text{if } x = y = 1 \\ 0 & \text{if } (x = 0) \vee (y = 0) \\ \bar{1} & \text{if } (x = 1 \wedge y = \bar{1}) \vee (x = \bar{1} \wedge y = 1) \end{cases} \tag{6}$$



(a) Initial operands at the input end

(b) Output after applying the SS rules of Cell I

(c) Output after applying the SS rules of Cell II

(d) Output after applying the SS rules of Cell III

Fig.4 An SD addition example showing the use of symbolic substitution rules

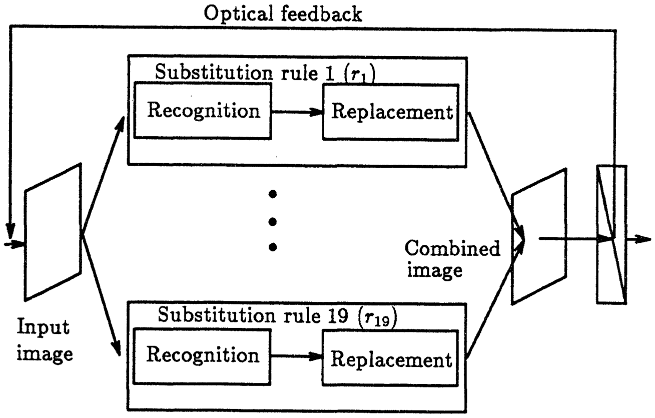


Fig.5 An optical adder(subtracter) symbolic substitution

The notations, $\vee$ and $\wedge$ are used to represent the conventional logical OR and the logical AND operations. The notation $y_j * X$ defines the following digit-wise operations:

$$y_j * X = y_j * x_{n-1}, y_j * x_{n-2}, \ldots, y_j * x_{-m} \qquad (7)$$

We have previously developed a sequential algorithm for computing the product P in $n+m$ iterations using SD additions and right shifts[13]. In what follows, we present a parallel algorithm that computes the product of two SD numbers in $\log_2(n+m)$ iterations, where $(n+m)$ is the word length including n integer digits and m fraction digits. For clarity, we use integer numbers where the fractional length $m = 0$. The algorithm is composed of three steps :

Step 1: Given two signed n-digit numbers, generate all n partial products concurrently, each having length n as follows:

$$P_{0,j} = y_j * X \qquad \text{for } j = 0, \ldots, n-1 \qquad (8)$$

where the term $P_{0,j}$ is an n-digit SD number representing the j-th partial product.

Step 2: Introduce the necessary shifts for each partial product. Each initial partial product $P_{0,j}$ will be shifted j digits to the left, corresponding to the weight factor 2^j shown in Eq.5:

$$P_{0,j} = y_j * X \times 2^j \quad \text{for } j = 0, \ldots, n-1 \qquad (9)$$

Step 3: Pairwise add all the partial products by means of an adder tree. With a total of n partial products at the leaves of the tree, the summation process takes $\log_2 n$ levels in the tree. At each level i, we perform $n/2^i$ SD additions in parallel :

$$P_{i,j} = P_{i-1,2j-2} + P_{i-1,2j-1} \quad \text{for } j = 1, 2, \ldots, n/2^i (10)$$

The final product is produced at the root of the tree after $\log_2 n$ iterations. Step 1 and Step 2 are carried out in constant time. For a multiplier of length n, Step 3 requires $\log_2 n$ iterations. Since each SD addition takes constant time, then the multiplication of two n-digit SD numbers can be carried out in $O(\log_2 n)$ time.

Example 2 below shows the parallel multiplication of two 4-digit SD numbers, $X = (1.0\bar{1}\bar{1})_{SD} = (0.375)_{10}$ and $Y = (1.0\bar{1}1)_{SD} = (0.875)_{10}$. In Step 1, we generate all the partial products using Eq.8. In Step 2, we introduce the necessary shifts. Finally, we add all the shifted partial products according to Eq.10), using a tree of SD adders to produce the final product $P = 000.10010\bar{1} = (0.546875)_{10}$.

Example 2:

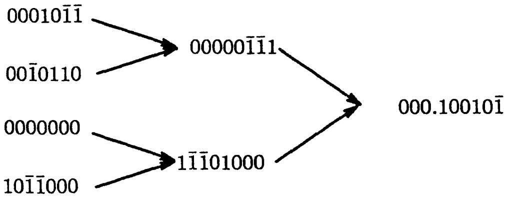
Step One : Generation of the partial products

$$P_{0,0} = y_{-3} * X = 1.0\bar{1}\bar{1}$$
$$P_{0,1} = y_{-2} * X = \bar{1}.011$$
$$P_{0,2} = y_{-1} * X = 0.000$$
$$P_{0,3} = y_0 * X = 1.0\bar{1}\bar{1}$$

Step Two: Shift the partial products

$$(y_{-3} * X) \times 2^0 = 00010\bar{1}\bar{1}$$
$$(y_{-2} * X) \times 2^1 = 00\bar{1}0110$$
$$(y_{-1} * X) \times 2^2 = 0000000$$
$$(y_0 * X) \times 2^3 = 10\bar{1}\bar{1}000$$

Step Three: Summation of all the shifted partial products



$X \times Y = Z = (000.10010\bar{1})_{SD} = (0.546875)_{10}$

The SD multiplication algorithm uses the signed AND operation (*) in generating all partial products simultaneously, and a tree of SD adders to sum them up. Using Eq.6, we derive the SS rules needed for implementing the * operation as shown in Fig.6. Let us consider the optical implementation of the computations in Example 2. The multiplicand and multiplier are arranged in 1-D arrays as shown at the left of Fig.7. The multiplicand is shown horizontally and the multiplier is shown vertically. The generation of all partial products $P_{0,j}$ for $j = 0, \ldots, 3$ is carried out in three stages. First, the multiplicand is spread out vertically by the astigmatic optics (represented by the cylindrical lens L1) to fill the 4 × 4 data plane M1. Similarly, the multiplier is spread out horizontally using the cylindrical lens L2, so that each digit of the multiplier is duplicated vertically 4 times to fill the 4 × 4 plane M2. Next, planes M1 and M2 are 2-D ***perfect shuffled*** [10] and then stored in an 8 × 4 plane R. For clarity, the optics required for the 2-D perfect shuffle permutations is omitted from Fig.7. The 2-D shuffle permutations intended here affect only the row position, leaving the column position of the data unchanged.

The resulting image, R, has alternating rows from M1 and M2 such that odd rows contain the multiplicand and even rows contain a replicated digit of the multiplier. Therefore, row 1, row 3, row 5, ..., row $n-1$ contain the multiplicand X; and row 2, row 4, row 6, ..., row n, contain the replicated digits $y_1, y_2, y_3, \ldots, y_{n-1}$ of the multiplier respectively. In the third stage, plane R is replicated 9 times, each copy is used for applying one SS rule of the * operation. Therefore, every combination of the input operands is searched and is replaced in parallel. Finally, the output planes of all the SS rules applied are optically superimposed. To this end, all the partial products have been generated in parallel as shown in plane P of Fig.8. Step 2 of the SD multiplication algorithm involves spatial shifts. There are a variety of ways one can perform spatial shifts in optics[14].

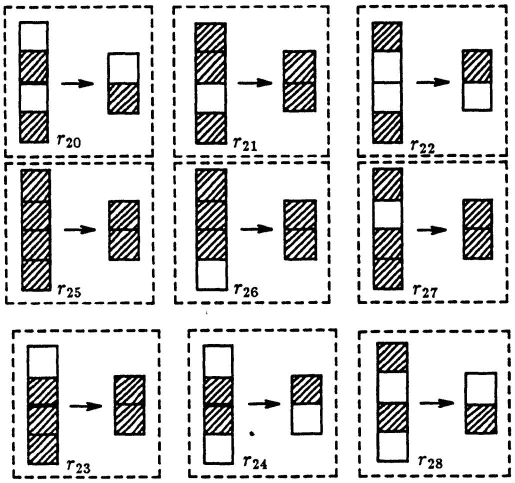


Fig.6 Symbolic substitution rules for the SD AND

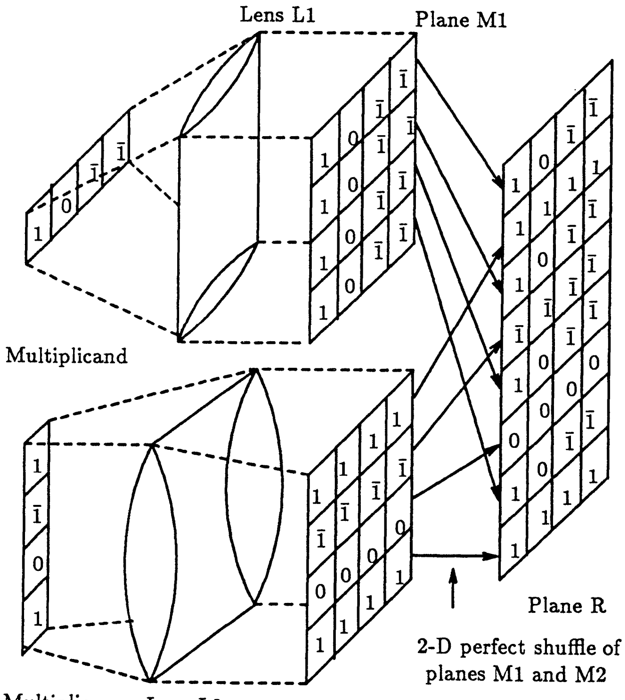


Fig.7 The spreading of the operands in Example 2 for parallel SD multiplication

The plane P, consisting of all partial products with appropriate shifts, is then fed to the adder described in the previous section in order to perform the last step of the multiplication algorithm. This is accomplished by applying the SD addition rules for $\log_2 4$ iterations.

In general , with a multiplicand of length n and a multiplier of length m, the planes M1, M2, and P in Fig.7 are all $m \times n$ arrays, R is a $2m \times n$ array, and the shifted P is a $m \times (m+n)$ array. It should be noted that, if the 1-D arrays which are used to input the operands are replaced by 2-D arrays and associated optics for spreading and shuffling, many operand pairs can be multiplied in parallel using the same set of SS rules.

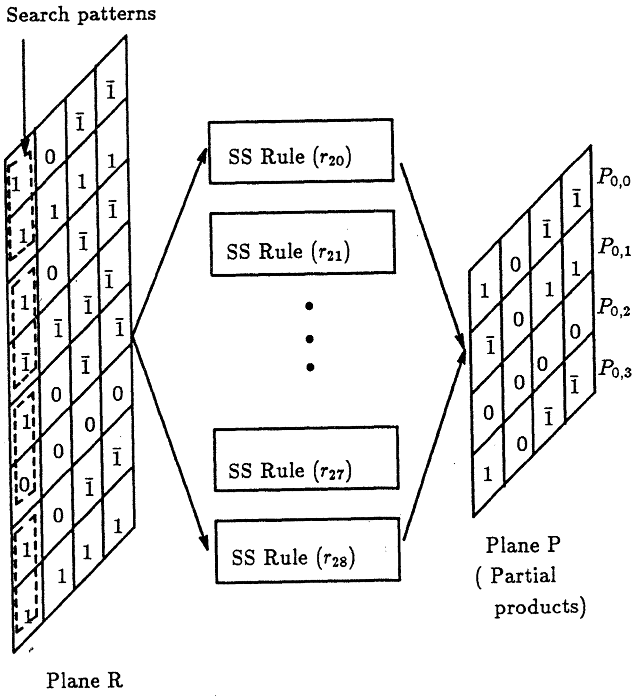


Fig.8 Parallel generation of partial products using the SS rules for the SD AND operation

5 Optical SD Division

The conventional restoring and nonrestoring division methods require knowledge of the sign of the partial remainder for exact selection of the quotient digits. However, in SD representation, the sign of a partial remainder is not readily available if several most significant digits are zero. This difficulty prevents the use of conventional methods for SD division. Robertson division method[15] was applied in [1] for SD number systems with radix $r \geq 3$. In that method, the quotient is represented in redundant form and the value of the next quotient digit is selected by comparing approximated values of both the divisor and the partial remainder. In searching for an effective division algorithm for SD numbers with radix $r = 2$, we have to achieve the following two goals:

(*1*) The algorithm should overcome the difficulty of testing the polarity of the remainder after each iteration.

(*2*) The algorithm should make effective use of the 2-D parallel SD addition and multiplication schemes described in previous sections.

An SD division algorithm satisfying the above goals is developed below based on the convergence approach[4,16,17]. Let us consider a dividend X and a divisor Y both SD fractions in normalized from, that is:

$$1/2 \leq |X| < Y < 1 \qquad (11)$$

We want to compute the quotient $Q = X/Y$ without a remainder. The algorithm consists of finding a sequence of multiply factors $m_0, m_1, m_2, \ldots, m_n$ such that $Y \times (\prod_{i=0}^{i=n} m_i)$ converges to 1 (within an acceptable error criterion). Initially, we set $X_0 = X$, and $Y_0 = Y$. The algorithm repeats the following recursions:

$$X_{i+1} = X_i \times m_i, \qquad Y_{i+1} = Y_i \times m_i \qquad (12)$$

such that for a small n:

$$Y \times (\prod_{i=0}^{i=n} m_i) \rightarrow 1, \qquad Q = X \times (\prod_{i=0}^{i=n} m_i) \qquad (13)$$

The effectiveness of this method relies on the ease of computing the multiply factors m_i's, using only SD addition and SD multiplication operations. The recursive formula of Eq.12 can be rewritten as:

$$Y_{i+1} = Y_i \times m_i = f(Y_i) \qquad (14)$$

We desire the function $f(Y_i)$ to converge to 1, starting from an initial value $Y_0 = Y$. Equation 14 can be rewritten in a polynomial form:

$$f(Y_i) - Y_i = 0 \qquad (15)$$

Flynn has described several iterative methods [16] to enable such a polynomial to converge to a given value say k. We are interested only in the quadratic convergence as this appears more convenient for optical realization. To achieve this, let us rewrite Eq.15 in a more general quadratic form:

$$f(Y_i) - Y_i = (Y_i - k_1)(Y_i - k_2) = 0 \qquad (16)$$

One of the roots of Eq.16 should be equal to the convergence limit 1. Krishnamurthy[17] has found that in order for $Y_i \times m_i$ to converge quadratically to 1, the factors m_i's should be selected as:

$$m_i = 2 - Y_i \quad \text{provided that} \quad 0 < Y_i < 2. \tag{17}$$

Equation 17 implies that the multiply factor for each iteration can be easily obtained as the two's complement of the denominator Y_i. In SD code, the arithmetic expression $2 - Y_i$ can be computed in constant time using SS rules for SD negation and addition. Since the convergence is quadratic, the accumulated denominator length is doubled after each iteration. Hence for a desired quotient of length n, the maximum number of iterations needed is $\log_2 n$. The convergence division of two SD numbers is formally specified below:

SD Division Algorithm

```
begin
for $i := 0$ to $\log_2 n - 1$ do
    $m_i := 2 - Y_i$;
    $X_{i+1} := X_i \times m_i$ ;
    $Y_{i+1} := Y_i \times m_i$;
endfor;
$Q := X_{\log_2 n-1}$;
end.
```

Example 3 illustrates the SD convergence division of $X = (0.\bar{1}0)_{SD} = (-0.5)_{10}$ by $Y = (0.11)_{SD} = (0.75)_{10}$. For a 16-digit precision, the algorithm generates the quotient after 3 iterations, $Q = (\bar{1}.1\bar{1}10\bar{1}\bar{1}10\bar{1}\bar{1}1\bar{1}1\bar{1}1)_{SD} = (-0.66664)_{10}$. As for the optical implementation of the algorithm, each iteration of the SD division consists of three major operations : a pair of two SD multiplications, $Y_{i+1} = Y_i \times m_i$ and $X_{i+1} = X_i \times m_i$, and a two's complement operation $m_i = 2 - Y_i$. Each SD multiplication can be optically carried out as described in Section 4. The two's complement is carried out by an SD negation followed by an SD addition. The subtrahend Y_i is negated using the SS rules in Fig.3d. All nonzero digits of Y_i are negated in parallel. The expression $2 - Y_i$ then becomes $2 + \bar{Y}_i$, which is computed using the SD addition rules in Fig.3a-c. The two SD multiplications required to generate X_{i+1} and Y_{i+1} can be computed concurrently by replicating the SD multiplication hardware into two channels, one for the numerator and one for the denominator as shown in Fig.9.

6 Performance Analysis

We estimate the potential speed of the optical arithmetic algorithms introduced in this paper. The analysis is based on the optical implementation models presented in previous sections. These estimates should reflect the state-of-the-art in optical computing technology. Our estimates cover both conservative and optimistic sides of the expected performance.

The SD addition is performed in three stages. The total time to perform each stage is attributed to the time needed : (1) to replicate the input image; (2) to propagate the image through the first hologram to provide the shifts; (3) to activate the optical NOR-gate array for inverting the superimposed image; (4) to propagate light through the second hologram for substitution; (5) to superimpose the output of all the rules; and (6) to feed back the intermediate result. Therefore the total SD addition time is expressed as:

$$T_{add} = 3(\overbrace{T_p}^{(1)} + \overbrace{T_p}^{(2)} + \overbrace{T_{activ}}^{(3)} + \overbrace{T_p}^{(4)} + \overbrace{T_p}^{(5)}) + 2\,\overbrace{T_f}^{(6)} \tag{18}$$

where:

T_p = Propagation time of a light beam through passive optical devices such as lenses, beam splitters, holograms, etc.

T_f = Feedback time (light propagation through the feedback interconnect)

T_{activ} = Response time of an optical NOR-gate array used for inversion and thresholding.

Example 3 : SD division steps based on repeated multiplications

Iteration step	**Multiply factor**	**Accumulated denominator**	**Accumulated numerator**
$i = 0$	$m_0 = 2 - Y_0$ $(1.01)_{SD}$	$Y_1 = Y_0 \times m_0$ $(1.000\bar{1})_{SD}$	$X_1 = X_0 \times m_0$ $(0.\bar{1}\bar{1}10)_{SD}$
	$= (1.25)_{10}$	$= (0.9375)_{10}$	$= (-0.625)_{10}$
$i = 1$	$m_1 = 2 - Y_0 \times m_0$ $(1.001\bar{1})_{SD}$	$Y_2 = Y_0 \times m_0 \times m_1$ $(1.000000\bar{1})_{SD}$	$X_2 = X_0 \times m_0 \times m_1$ $(\bar{1}.1\bar{1}10\bar{1}\bar{1}10)_{SD}$
	$= (1.0625)_{10}$	$(0.99509)_{10}$	$(-0.66406)_{10}$
$i = 2$	$m_2 = 2 - Y_0 \times m_0 \times m_1$ $(1.0000001)_{SD}$	$Y_3 = Y_0 \times m_0 \times m_1 \times m_2$ $(1.00000000000000\bar{1})_{SD}$	$X_3 = X_0 \times m_0 \times m_1 \times m_2$ $Q = (\bar{1}.1\bar{1}10\bar{1}\bar{1}10\bar{1}\bar{1}1\bar{1}1\bar{1}1)_{SD}$
	$= (1.00390625)_{10}$	$Y_3 \rightarrow 1$	$= (-0.6666..)_{10}$

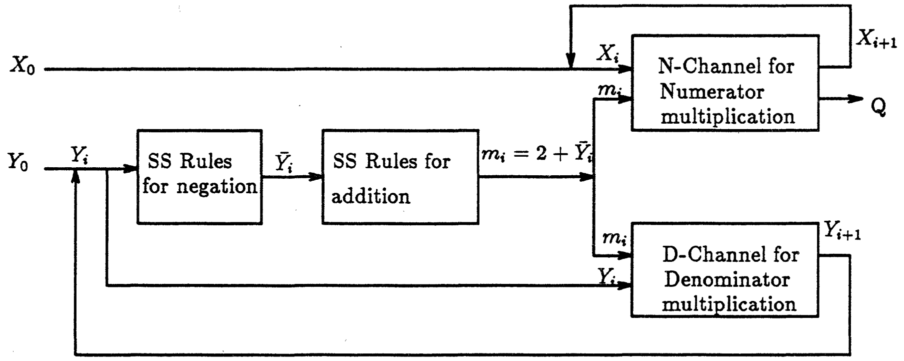


Fig.9 An optical convergence divider using two channels of optical multipliers

The numbers over the braces in Eq.(18) indicate the times needed to accomplish each subtask. T_p and T_f can be approximated by 0.1 nsec[14] (light propagates at 1 ft/nsec in free space). The dominant limitation to speed is the switching time of the optical NOR-gate array, representing the only active element in the addition path. Therefore, the total SD addition time would be $T_{add} \approx 3T_{activ}$. An n-digit SD addition requires $(n+1) \times 4$ pixels, where the factor 4 is introduced by the encoding scheme used (2 light pixels for each digit). Therefore, for an optical gate array of size $l \times l$ pixels and a switching time τ, the optical SD adder is able to perform Θ_a n-digit additions per second, where:

$$\Theta_a = \frac{l \times l}{3\tau \times ((n+1) \times 4)} \quad \text{(SD additions/sec)} \quad (19)$$

Optical gate arrays of very small sizes (say 2×2 to 5×5) have been recently demonstrated [18]. These arrays offer the possibility of achieving a 10^{-12} sec switching time. However, these optical gate arrays can not be used in a practical system due to their small size and high power consumption. If we were to use a commercial *spatial light modulator* (SLM) such as the *liquid crystal light valve* (LCLV) with a 500×500 pixel resolution and 20 ms switching time, we can perform about 63×10^3 32-digit SD additions per second. This yields to an average of $1/\Theta_a$ $= 15 \times 10^{-6}$ sec per SD addition. This speed is not much faster than today's fast adders. However, faster SLMs are being produced in research laboratories[18]. If the response time of the SLM were reduced to 0.01 μsec, a 500×500 resolution will bring the 32-digit SD addition time down to 1.5×10^{-13} sec (0.15 ps), which will represent 10^4 times improvement over electronic adders of the same size.

Referring to the optical implementation model in Sec.4, the SD multiplication time is attributed to the time needed : (1) to generate the partial products; (2) to shift them; and (3) to add up the shifted partial products. This time is expressed as:

$$T_{mult} = \overbrace{T_s + 4T_p + T_{activ}}^{(1)} + \overbrace{T_p}^{(2)} + \overbrace{T_{add} \times \log_2 n}^{(3)} \quad (20)$$

where T_s represents the time needed to spread and to shuffle the operands. This time corresponds to light propagation through passive devices which can be estimated by 0.1 nsec. Since $T_s \approx T_p << T_{activ}$ and $T_{add} \approx 3T_{activ}$, hence $T_m \approx T_{activ}(1 + 3\log_2 n)$, where n is the precision of the multiplier. An n-digit SD multiplication requires $4 \times (n \times 2n)$ pixels, where the factor 4 is related to the light encoding of the digit set $\{\bar{1}, 0, 1\}$. Using an SLM with $l \times l$ pixel resolution and τ switching time, we obtain the number Θ_m of n-digit multiplications performable per second:

$$\Theta_m = \frac{l \times l}{4 \times (n \times 2n) \times \tau \times (1 + 3\log_2 n)} \quad (21)$$

If we were to use standard off-the-shelf SLM (LCLV), there could be 96 SD multiplications per second. This corresponds to a speed of $1/\Theta_m = 10$ msec per one 32-digit SD multiplication. This looks very slow. However, if the switching time of the SLM were reduced to 0.01 μsec, the 32-digit SD multiplication time would be reduced to 5 nsec, which is 100 times faster than today's fastest electronic multipliers of the same word length.

Consider the optical implementation shown in Fig.9, the time required to perform one iteration of the SD division consists of the time needed : (1) to generate the multiplicative factor m_i ; and (2) to produce the next numerator and denominator X_{i+1}, Y_{i+1}. This time is then multiplied by the logarithm of the fraction length to obtain the total SD division time T_{div}:

$$T_{div} = (\overbrace{4T_p + T_{activ} + T_{add} + T_f}^{(1)} + \overbrace{T_{mult} + T_f}^{(2)}) \times \log_2 n (22)$$

Substituting T_{add} and T_{mult} in Eq.22 with Eq.18 and Eq.20 respectively, we obtain $T_{div} \approx T_{activ} \log_2 n(5 + 3\log_2 n)$. An important feature of the SD division algorithm is that several dividends can be divided simultaneously by the same divisor. This is due to the fact that the multiply factors and the convergence rate depend only on the magnitude of the divisor. An n-digit SD division requires $4 \times (n \times 2n)$ pixels to hold the accumulated numerators or denominators (assuming that we are truncating the intermediate products by n digits after each iteration). Therefore, for an optical gate array of $l \times l$ resolution and τ switching time, we estimate the number of SD division per second as:

$$\Theta_d = \frac{l \times l}{4 \times (n \times 2n) \times \tau \times \log_2 n(5 + 3\log_2 n)} \qquad (23)$$

For a resolution $l \times l = 500 \times 500$ and a switching time $\tau = 0.01\mu$sec, the time needed for a 32-digit SD division would be $1/\Theta_d$ which is around 30 nsec, a rather impressive figure that no existing electronic divider can achieve.

In Fig.10a, we plotted the optical addition, multiplication and division times against a wide range of the optical clock rate (or the inverse of the optical switching time τ). The speedup of the optical arithmetic operations over their electronic counterparts is plotted in Fig.10b. We fixed the resolution of the optical gate arrays to $l \times l = 500 \times 500$, and the precision $n = 32$ SD digits. For the speedup curves, we used 20 nsec, 500 nsec, and 2 μsec for 32-bit electronic addition, multiplication and division, based on current electronic technology [19,3]. Both scales are in logarithm with base 10.

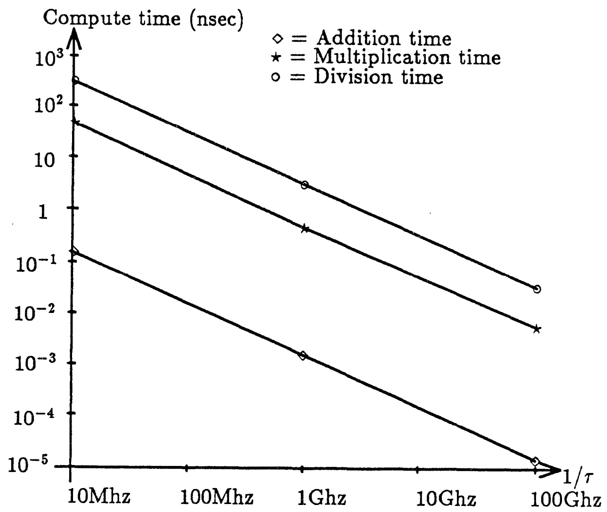


Fig.10a Optical compute time as a function of the clock rate ($1/\tau$)

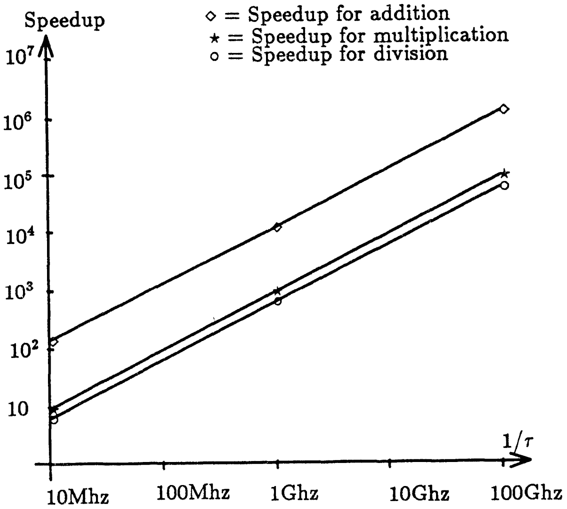


Fig.10b Potential speedup of optical over electronic arithmetic computations.

7 Conclusions

The SD representation allows parallel addition to be performed in constant time. The execution times of the proposed SD multiplication and SD division algorithms are both proportional to $\log_2 n$, where n is the length of the multiplier and of the divisor. We have presented the optical setups to achieve 2-D optical symbolic substitution. The carry-free nature of SD arithmetic matches well with the space-invariant property of optical symbolic substitution.

We have introduced two new sets of SS rules for implementing SD arithmetic in optics. The optical implementations are based on available optical hardware. We have assessed the performance of optical arithmetic based on the state-of-the-art optical and electro-optical technologies. We conclude that the speedup over electronic counterparts is rather limited due to the slow switching time of today's 2-D spatial light modulators.

If the switching time of the optical gate arrays were reduced to nanosecond range, we could perform 32-digit optical addition, multiplication and division with a speedup ranging from $O(10^2)$ to $O(10^5)$ over existing electronic counterparts as shown in Fig.10b. Therefore, the potential of building future supercomputers with optical arithmetic units looks very promising and encouraging. The algorithms developed in this paper are meant to prepare computer designers for the new challenges brought over by optical technology.

Acknowledgments

This research was supported by an ONR Contract No. N14-86-k-559 and in part by a grant AFOSR-86-8

References

[1] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *Trans. Elect. Computers*, vol. EC-10, pp. 389–398, 1961.

[2] B. L. Drake, R. P. Bocker, M. E. Lasher, R. H. Patterson, and W. J. Miceli, "Photonic computing using the modified signed-digit number representation," *Optical Engineering*, vol. 25, pp. 038 – 043, Jan. 1986.

[3] K. Hwang, *Computer Arithmetic : Principles, Architecture, and Design.* John Wiley & Sons, New York, 1979.

[4] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM system/360 model 91, floating point execution unit," *IBM J. Res. and Develop.*, vol. 11, pp. 34 – 53, Jan. 1967.

[5] A. Huang, "Architectural considerations involved in the design of an optical digital computer," *Proceedings of the IEEE*, vol. 72, pp. 780 – 787, July 1984.

[6] A. A. Sawchuk and T. C. Stand, "Digital optical computing," *Proceedings of the IEEE*, vol. 72, pp. 758–779, July 1984.

[7] A. Huang, "Parallel algorithms for optical digital computers," in *Proceedings IEEE Tenth Int'l. Optical Computing Conf.*, 1983.

[8] K. H. Brenner, A. Huang, and N. Streibl, "Digital optical computing with symbolic substitution," *Applied Optics*, vol. 25, 15 Sept 1986.

[9] Y. Li, G. Eichmann, R. Dorsinville, and R. R. Alfano, "An AND operation-based optical symbolic substitution," *Optics Communications*, vol. 63, pp. 375–379, 15 September 1987.

[10] A. Louri and K. Hwang, "A bit-plane architecture for optical computing with two-dimensional symbolic substitution," *In Proc. of the 15th Int'l. Symp. on Computer Architecture*, Honalulu, Hawaii, May 30 - June 2, 1988. (An extended version has been submitted for Journal publication).

[11] P. A. Ramamoorthy and S. Anthony, "Optical modified signed-digit adder using polarization-coded symbolic substitution," *Optical Engineering*, vol. 26, no. 18, Aug. 1987.

[12] Y. Li and G. Eichmann, "Conditional symbolic modified signed-digit arithmetic using optical content-addressable memory logic elements," *Applied Optics*, vol. 26, no. 12, 15 June 1987.

[13] K. Hwang and A. Louri, "New symbolic substitution algorithms for optical arithmetic using signed-digit representation," *Proc. Soc. Photo-Opt. Instr. Eng. (SPIE)*, vol. 880, January 1988, (A significantly extended version has been submitted for Journal publication under the title "Parallel Optical Arithmetic Using 2-D Symbolic substitution).

[14] A. Huang, Y. Tsunoda, J. W. Goodman, and S. Ishihara, "Optical computation using residue arithmetic," *Applied Optics*, vol. 18, no. 2, 15 Jan. 1979.

[15] J. E. Robertson, "A new class of division methods," *IRE Trans. Electronic Computers*, vol. EC-7, pp. 218 – 222, Sept. 1958.

[16] M. J. Flynn, "On division by functional iteration," *IEEE Transactions on Computers*, vol. C-19, no. 8, pp. 702 – 706, Aug. 1970.

[17] E. V. Krishnamurthy, "On optimal iterative schemes for high-speed division," *IEEE Transactions on Computers*, vol. C-19, no. 3, pp. 227 – 231, March 1970.

[18] A. D. Fisher and J. N. Lee, "Current status of two-dimensional spatial light modulators," *In Proc. SPIE, optical and Hybrid Computing*, vol. 634, pp. 352 – 372, 1986.

[19] K. Hwang, "Advanced parallel processing with supercomputer architectures," *Proceedings of the IEEE*, vol. OC- 75, pp. 1348 – 1379, Oct. 1987.

AN ANALYSIS OF PARALLEL LOGIC SIMULATION ON SEVERAL ARCHITECTURES

Steven P. Smith, Bill Underwood & Joe Newman

Microelectronics and Computer Technology Corporation
Computer Aided Design Program
3500 West Balcones Center Drive
Austin, Texas 78759

ABSTRACT

Due to its tendency towards large and unpredictable amounts of interprocessor communication, parallel logic simulation places enormous demands on the performance of both individual processor elements and interprocessor communications. To explore the relative importance of processor and communications speed and to compare the merits of different architectures for this application, results are offered from the simulation of a number of test circuits on models of five parallel architectures. Three different schemes are used to partition the circuit representations across processors, and both 4 and 16 processor configurations are considered for each architecture. The relative cost of device evaluation and signal communication is also varied. The five architectures examined are: a parallel processor with a single interprocessor communications bus, a ring of processors, a simple processor array with nearest-neighbor connections, a hypercube, and a processor array with crossbar communications. The results are compared both to the single processor case and to the ideal parallel case, and they indicate that the performance potential of parallel event driven logic simulation at this level is questionable.

INTRODUCTION

The march towards ever larger and faster computer systems has continually outpaced the rate of advances in the computer aided design (CAD) tools used to develop them. Nevertheless, CAD tool developers struggle to keep up by employing combinations of three basic tactics. The first tactic seeks to reduce the size of the problem, either through hierarchical modeling of computer systems or by considering only small portions of the design at a time. The second tactic centers on the discovery of more efficient algorithms. And the third tactic involves the exploitation of parallel architectures.

Approaches to parallel logic simulation can generally be divided into two categories. Those in the first category pursue speed gains by breaking the algorithm into pieces that are then executed on separate processors. Although parallel logic simulators of this variety have been successfully implemented in hardware,[1] their performance potential would appear to be too severely restricted by the limited parallelism inherent in traditional simulation algorithms to be of lasting interest. Approaches in the second category attempt to leverage the parallelism evident in the behavior of real circuits by partitioning the circuit representation being simulated among several processors.

At first blush, it seems reasonable to speculate that the performance of parallel logic simulators based on circuit partitioning need not degrade markedly as the size of circuit representations increases. Two factors invalidate this speculation. First is the fact that in traditional event driven logic simulators, it is necessary to maintain the same simulation time across all processors, requiring all processors to complete their work at a given time unit before any may proceed to the next time at which there is activity. The second factor is the time expense incurred by communicating signal values for devices modeled on one processor needed as input to devices on another processor.

The goal was to explore the potential of parallel logic simulation based on circuit partitioning in light of these considerations. To aid in comparative analysis, an instrumented event driven simulator was employed to model performance for five different parallel architectures using a set of 11 test circuits. The five architectures examined were: a parallel processor with a single interprocessor communications bus, a ring of processors, a simple processor array with nearest-neighbor connections, a hypercube, and a processor array with crossbar communications. Three circuit partitioning algorithms were tried in each case to examine the sensitivity of simulator performance to this task. And finally, four different sets of device evaluation time to signal transmission time ratios were used to determine the relative performance criticality of these parameters.

PARALLEL LOGIC SIMULATION

In this section, we define the parallel logic simulation algorithm to be used in the analysis below. As mentioned earlier, parallel techniques can generally be divided into those that distribute the algorithm among several processors and those that distribute the data. The limitations on useful decomposition of the basic logic simulation algorithm renders approaches in the first category unworkable for large scale parallelism. Therefore, the discussion at hand will be restricted to approaches that distribute the circuit representation among processors.

Circuit models used in logic simulators are typically composed of structure instances that represent individual device occurrences with electrical connectivity indicated via pointers from structure to structure. This representation is split among processors for parallel logic simulation using one of the partitioning algorithms presented in the following section. Once every device structure (and hence every device model) is assigned to a processor, the circuit representation is loaded onto the appropriate element. Devices that drive inputs not found on the same processor are tagged so that when their outputs change, sink devices on other processors can be notified of the change via a signal change message.

Assuming a standard one-pass simulation algorithm, it is possible to develop simple equations for determining performance. For simulation on a single processor, the time for a simulation run is given by

$$t_{sim} = \sum_{i=1}^{sim\_passes} evals_{pass_i} * t_{eval}. \qquad (1)$$

Ignoring the negligible synchronization overhead, the general equation for simulation on a machine with multiple processor elements (PEs) is given by

$$t_{psim} = \sum_{i=1}^{sim\_passes} \underset{PEs}{MAX}\left[MAX\left\{t_{com_i}+1*t_{eval}\ ,\ t_{evals_i}\right\}\right], \tag{2}$$

where the maximum communications time is architecture dependent and will be discussed in due course, and the maximum evaluations time at each pass is equal to the product of the largest evaluation count found on the processors and the time per evaluation. In qualitative terms, Equation 2 states that the time per simulation pass is equal to the maximum time required by any of the individual processors and that this time is determined by the maximum of the communication and evaluation times. And, of course, the total simulation time is equal to the sum of the times spent on all of the passes. The single evaluation time added to the communications time simply indicates that, if the time consumed by a given processor is dominated by communication then after the last signal for the current pass has been received, a final evaluation must be performed to complete the pass. Note that for our analysis, all evaluations are assumed to take the same amount of time. This assumption is valid for most hardware implementations using simple look-up tables for evaluations,[4] and is a reasonable approximation in general. With these equations, we can calculate the speed-up of parallel logic simulation directly from

$$speed\text{-}up = \frac{t_{sim}}{t_{psim}}. \tag{3}$$

Equation 2 will be elaborated for each of the five architecture models presented below to account for the effect of the different architectures on communications performance.

CIRCUIT PARTITIONING

The goal of circuit partitioning is to assign devices to processors in a manner which maximizes the resultant simulation speed-up. As shown in Equation 2, this speed-up is dependent upon message transmission time and gate evaluation time. The optimal partition will produce the minimum number of messages as well as an even distribution of evaluations at each processor for each simulation pass, however finding such a partition is a dramatically more expensive process than the simulation task itself. Therefore, two heuristics are used that have been shown to produce satisfactory results for a variety of circuits:[5] input cone and output cone partitioning. These schemes involve placing occurrences into the circuit block to which they have the greatest attraction, that is, the block that contains the greatest number of occurrences in their input or output cone. In addition to the two heuristic schemes, random assignment partitioning is also included as a baseline. Since signal activity tends to be clustered for many circuits, this method offers the potential for relatively high degrees of processing concurrency. However, the advantage is frequently offset by large numbers of message transfers.

Clearly, the success of any given partitioning scheme is dependent on the relative weighting of communication and evaluation costs. Since partitioning by cones seeks to minimize interprocessor communications by grouping connected devices on the same block, the performance of these approaches hinges on the assumption that communications costs dominate simulation time.

THE PARALLEL ARCHITECTURES

In this section, we describe the five architectures modeled in our analysis. To aid in relative comparisons, the same performance assumptions are made in each case. To wit, it is assumed that direct point-to-point signal change messages require a constant amount of time, and that processors do not buffer these messages. This assumption implies that a message being sent through a single intermediary processor will require one message time unit on the originating processor, two units on the intermediary processor, and one unit on the destination processor.

Processor elements are assumed to be identical general purpose machines capable of performing both the device evaluation and local time-management tasks. The time consumed during a device evaluation is assumed to be constant and includes scheduling overhead so that in the absence of any message traffic, the time per simulation pass on a processor simply equals the product of the evaluation count during the pass and the time per evaluation.

Finally, it is assumed that none of the architectures possess any global memory. The local memory on each processor element contains both a unique portion of the circuit being simulated and the simulation time management structures required during simulation. Devices that drive signals on other processors are tagged with the data needed to route the information to its destination.

The data presented below were gathered from a heavily instrumented event driven logic simulator operating on test circuits partitioned during preprocessing. Partition blocks were assigned to processor elements randomly. Message traffic figures obtained from the simulations are exact and represent a complete picture of the expected interprocessor communications load for the 11 test cases used in our analysis. In taking this approach, it has proven remarkably easy to model new architectures and to focus on the considerations currently of greatest interest, namely, the effects of processor count and interconnection architectures.

Single Bus Architecture

A bus supports the transmission of only one message at a time, but arbitration is assumed to occur in parallel and is therefore not considered in overall simulation time. From Equation 2 it is clear that the time consumed by a processor per simulation pass is roughly equal to the greater of the communications time and the evaluation time. The evaluation time per processor is independent of the interconnection architecture in use, but the communications time per processor is highly dependent on this factor. For the single bus architecture, the communications time per pass is given by

$$com\ t_{pass_{single\ bus}} = \sum_{PEs} msgs_{PE_i} * t_{com} + 1*t_{eval}, \tag{4}$$

which states simply that, since all messages are transmitted via the same channel, the communications time is determined by the total count of interprocessor messages. As mentioned earlier, the single evaluation time is added to account for the work required after the last signal change message is received. For the single bus architecture, the maximum and average path for messages is equal to one, but this one channel is likely to be very busy.

Ring Architecture

For our ring model, we assumed unidirectional message flow. Assuming that each message flows in a clockwise direction until its destination processor is encountered, and assuming that messages are not buffered, the communications time per simulation pass is

$$com\ t_{pass_{ring}} = \underset{PEs}{MAX}\left\{src\ msgs_{PE_i} + dest\ msgs_{PE_i} + t_{eval} 2*via\ msgs_{PE_i} + \right\} \tag{5}$$

and the total time per pass is again the greater of this value and the largest of the individual evaluation times. The assumption that messages going through intermediary processors consume two message time units is rather pessimistic, but is made to maintain consistency with the other models. If there are η processor elements, then

$$message\ path_{\max} = \eta - 1 \tag{6}$$

and

$$message\ path_{avg} = \frac{\eta}{2}. \tag{7}$$

The average path assumes a random distribution of message traffic. For 4 processors, the longest path visits 3 processor elements and the average path visits 2, and for 16 processors, the longest path is 15 and the average path is 8.

Array Architecture

A simple array architecture has up to four nearest-neighbor connections per processor element. In the model, there are frequently multiple paths for a message from a given source element to its destination, so two approaches were used for selecting paths under these circumstances. In the first approach, the route was selected randomly; and in the second, the link at each step with the lowest cumulative message traffic was always chosen. The relative worth of these approaches will be discussed along with the rest of the results in the following section. The communications time per processor element for each simulation pass is the same as that for the ring and is given in Equation 5.

A square array of η elements will have a longest path for interprocessor communications of

$$message\ path_{\max} = 2^{*}\left[\sqrt{\eta}-1\right] \tag{8}$$

and an average path of

$$message\ path_{avg} = \sqrt{\eta}^{*}\frac{2}{3} \tag{9}$$

For the 4 processor case, the longest path is 2 and the average path is 1.33. For 16 processors, these values become 6 and 2.67, respectively.

Hypercube Architecture

A hypercube architecture with eight processor elements has dimensionality of $k=\log_2(\eta)=3$. The most common scheme for routing messages in hypercube architectures uses a fixed routing scheme based on the difference between the bit encoded destination processor identifier and the current location of the message.[3] However, to maintain comparable communications schemes, the approach for message routing used in the array model is also employed for hypercubes. The results are roughly equivalent to fixed routing for the random routing case, and better than fixed for the balanced case. Each processor element in a hypercube of dimensionality k has k interprocessor communications paths. A hypercube with η elements will have a longest communications path of

$$message\ path_{\max} = \log_2(\eta) \tag{10}$$

and, if k is the dimensionality, an average path of

$$message\ path_{avg} = \frac{k^{*}2^{k-1}}{2^{k}-1}. \tag{11}$$

For the 4 processor case, the cube collapses to an array. For the 16 processor case, the longest path is 4 and the average path is 2.13.

Crossbar Architecture

A crossbar switch architecture contains point to point connections from each processor to every other processor. In this case, the communications time consumed per processor in each simulation pass is given by

$$com\ t_{pass_{crossbar}} = \underset{PEs}{MAX}\left\{src\ msgs_{PE_i}+dest\ msgs_{PE_i}+1^{*}t_{eval}\right\}. \tag{12}$$

RESULTS AND ANALYSIS

This section presents results for parallel logic simulation on the five models described above. 11 test cases were run for each model using three partitioning schemes and four different assumptions about the relative cost of interprocessor message transfers and device evaluations. The first circuit was a simple ALU bit-slice; the rest were obtained from the *1985 International Symposium on Circuits and Systems*.[2]

The four evaluation to communication cost ratios used were 1 to 0, 3 to 1, 1 to 1, and 1 to 3. A ratio of 3 to 1 implies that a single evaluation is assumed to require three times the time required to complete a single message transmission between two processors. The 1 to 0 ratio is intended to model an ideal situation, i.e., a system that can transmit messages instantaneously.

Table 1 lists average ideal concurrency figures for 4 and 16 processor systems using all three partitioning schemes. Each entry indicates the attainable speed-up using the applicable partitioning approach and processor count if messages could be sent in zero time. As such, these data address how well the three partitioning schemes evenly distribute evaluation work among processors. A concurrency figure equal to the number of processors could occur only if the same number of evaluations are performed on each processor during every simulation pass. The average figure of 3.471 for random partitioning into 4 blocks represents a peak processor usage efficiency of 87%. For the 16 processor case, the efficiency drops to 66% for random partitioning.

Although the random partitioning exhibits clearly superior evaluation concurrency behavior, its appeal diminishes greatly when message transmissions are weighed into the figures. Tables 2 and 3 show the performance improvement of the other two partitioning schemes as the message transmission time is weighed in more heavily. In qualitative terms, it is not surprising that cone partitioning schemes generate fewer interprocessor messages; their entire goal is to group signals with the devices they drive.

The results also indicate that, even for as few as 4 processors, it is quite possible to actually slow down a logic simulator by implementing it in parallel. Regardless of how the circuit is partitioned, if an evaluation takes one third the time of a message transfer, parallel simulation on the bus or ring architectures will on average result in a speed-up of less than unity (i.e., an overall decrease in performance). For a "hardwired" simulation engine with evaluation routines based on high speed table lookups, ratios of this order are not at all unlikely. For output cone partitioning, if message transfers are three times as fast as evaluations, the resultant speed-up of approximately 3 implies that 75% of the ideal result is achieved.

The crossbar interconnection results shown in Table 2 illustrate the attraction of the architecture where feasible. However, if interconnect usage efficiency is considered, then the results are less impressive. A 50% increase in processor links over the array architecture yields average performance improvements of less than 20% in all cases.

Table 3 presents results for the test cases executed using 16 processor models. For the bus and ring architectures with a 1 to 3 ratio, concurrencies of less than zero are produced. The array and hypercube results show that for higher processor counts, the message routing scheme has a greater influence on performance than was the case for 4 processors.

It is interesting to note that, for the case in which message transfers are assumed to consume three evaluation times, only 71% of the test runs resulted in a performance increase over the single processor case. Also worthy of mention is the modest size of performance increases from the 4 to 16 processor systems for each of the architectures. Four times as many processors increased the average performance of the single bus architecture by at most 80%. For the ring, the larger system improved the speed-up by only 62%. A 400% increase in processor elements brought about up to a 162% increase in overall speed for the array. The hypercube achieved a 190% overall speed-up for the 16 processor system relative to the smaller configuration. Finally, the crossbar managed a 199% increase in going from 4 to 16 processors. Of course, these figures are all the more disappointing in light of the significant increase in interconnect hardware which accompanies the expansion of these systems (excepting the single bus system) from 4 to 16 processors.

CONCLUSIONS

We have compiled detailed modeling data for event driven parallel logic simulation on five architectures varying both circuit partitioning schemes and processor and interconnect performance. The results indicate that performance is extremely sensitive to both the partitioning scheme and the interprocessor communications speed. These two factors are obviously related: when the average device evaluation time dominates message transmission time, random partitioning produces the best results. But, as the relative cost of message·transfers rises, the two cone based partitioning schemes which seek to minimize message traffic surpass random partitioning. This seems to imply that simulation algorithms such as fault simulation and high level simulation that can exploit this relationship show significant potential for parallel applications.

At the same time, the results are not really very encouraging. In order for the 16 processor crossbar architecture to gain a factor of 10 performance advantage over a single processor, it was necessary to add 15 processors and 120 interconnection links. If a more likely cost ratio of unity is used, the speed-up is halved. Obviously, these results do not bode well for parallel logic simulation of gate level design representations.

This work has led us to focus on a search for simulation algorithms better suited for parallel processing. For example, if simulation were carried out at a higher level so that the time cost of evaluations could be substantially increased relative to communications costs, results much closer to the ideal concurrency figures of Table 2 might be possible. Other areas of interest in ongoing work include configuration heuristics for parallel simulation. Which partitioning scheme is most appropriate for a given circuit? What number of processor elements will yield the briefest simulation time?

REFERENCES

1. Barto, R. and S. Szygenda, "A Computer Architecture for Digital Logic Simulation," *Electronic Engineering*, September, 1980, pp. 35-66.
2. Brglez, F., P. Pownall, and R. Hum, "Accelerated ATPG and Fault Grading Via Testability Analysis," *Proceedings of the International Symposium on Circuits and Systems, 1985.*
3. Heath, M., "The Hypercube: A Tutorial Overview," *Hypercube Multiprocessors 1986*, Siam, Philadelphia, 1986, pp. 7-10.
4. Pfister, G., "The Yorktown Simulation Engine: Introduction," *Proceedings of the 19th Design Automation Conference*, June, 1982, pp. 51-54.
5. Smith, S. P., Underwood, B., and M. R. Mercer, "An Analysis of Several Approaches to Circuit Partitioning for Parallel Logic Simulation," *Proceedings of the 1987 International Conference on Computer Design*, October, 1987, pp. 664-667.
6. Smith, S. P., Wood, B., Little, J. and P. Hunter, "Proteus-1: A General Accelerator for CAD," *Proceedings of the 1987 Fall Joint Computer Conference*, November, 1987, pp. 512-519.

Table 1
Average Ideal Concurrency and Message Transmissions for Different Partitioning Schemes

DEPENDENT MEASURE	INPUT CONE		OUTPUT CONE		RANDOM	
	$\eta = 4$	$\eta = 16$	$\eta = 4$	$\eta = 16$	$\eta = 4$	$\eta = 16$
IDEAL CONCURRENCY	2.576	7.708	3.123	8.604	3.471	10.543
MESSAGE TRANSMISSIONS	27607	40056	25553	38403	116367	128704

Table 2
Average Concurrency Figures for 4 Node Architectures

ARCHITECTURE	INPUT CONE			OUTPUT CONE			RANDOM		
	3:1	1:1	1:3	3:1	1:1	1:3	3:1	1:1	1:3
SINGLE BUS	2.574	1.987	0.909	2.987	1.813	0.870	2.909	1.003	0.337
RING	2.560	1.832	0.744	2.784	1.568	0.718	2.570	0.876	0.294
ARRAY*	2.575	2.195	1.087	3.011	2.074	1.026	3.285	1.317	0.443
ARRAY**	2.576	2.313	1.191	3.035	2.201	1.116	3.363	1.424	0.480
CROSSBAR	2.576	2.462	1.310	3.040	2.337	1.312	3.465	1.764	0.588

* Random message routing
** Balanced message routing

Table 3
Average Concurrency Figures for 16 Node Architectures

ARCHITECTURE	INPUT CONE			OUTPUT CONE			RANDOM		
	3:1	1:1	1:3	3:1	1:1	1:3	3:1	1:1	1:3
SINGLE BUS	4.651	1.916	0.654	4.009	1.793	0.651	2.512	0.855	0.287
RING	4.151	1.534	0.518	3.285	1.406	0.501	2.265	0.775	0.257
ARRAY*	5.836	2.786	0.980	5.070	2.475	0.942	4.017	1.382	0.465
ARRAY**	6.807	3.473	1.256	6.117	3.092	1.224	5.173	1.793	0.606
HYPERCUBE*	7.113	3.847	1.431	6.571	3.408	1.401	6.471	2.261	0.766
HYPERCUBE**	7.492	4.563	1.858	7.436	4.108	1.799	7.588	3.066	0.916
CROSSBAR	7.695	5.798	2.372	7.978	4.813	2.128	10.106	4.700	1.566

* Random message routing
** Balanced message routing

Semantics of a Parallel Computation Model and its Applications in Digital Hardware Design

Zebo Peng

Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden

Abstract

This paper describes a parallel computation model based on a data/control flow notation which consists of separate but related sub-models of data path and control. The data path is formulated as a directed graph. The control structure, on the other hand, is modelled as a Petri net. This model is used for specification and synthesis of digital hardware with a high degree of concurrency and parallelism. The semantics of the proposed model is defined in terms of its interactions with the environment. That is, two pieces of hardware are considered to be semantically equivalent if they interact with an environment in the same way. This allows manipulation of the internal structure of the hardwares to improve performance as well as reduce cost. A set of transformations for the model which preserve its semantics is presented. A sequence of such transformations can be used to move a design from an abstract description to a final implementation.

1. Introduction

One approach to the design of complex digital hardware for VLSI implementation is to use top-down synthesis technique. A synthesis approach starts the design with an abstract specification and refines it step by step towards a physical implementation by adding details [6]. Automated synthesis of parallel systems requires a parallel computation model to support the description of the system being designed. Such a computation model must be able to express the existence of multiple hardware resources for data storage, computation, and communication. At the same time, it must be able to represent the existence of multiple control flows and synchronization schemes.

This paper describes a parallel computation model in which a data path is used to represent the available hardware resources for data manipulation. The organization of this set of hardware resources to perform the prescribed computation is defined as a control structure which specifies the *partial* ordering of the given operations. Those operations which are not ordered, i.e., do not dependent on each other, can be carried out in parallel by physically distributed hardware resources. The control structure is formulated as a Petri net in the proposed model.

One important task of a hardware synthesis process is to perform design optimization. As such, there must be as much freedom as possible to alter parts of the control as well as data path in ways that do not change the behavior of the given system. For this possible, we must be able to characterize the behaviors of a system and define precisely the concept of equivalent systems. The semantics of the proposed model is defined in terms of its interactions with the environment. That is, only the external events are relevant to the semantics of the system. In this way, the internal structure of the digital system can be change without changing its semantics. The system's interaction with its environment is in turn defined based on two factors. First the functional relationship between each output variable and its relevant input variables must be the same; secondly the temporal relationship between input/output operations should not be different. This definition differs from other approaches which consider only the input/output functional relation in terms of the values being exchange between a system and its environment.

Most of other parallel system models have concentrated only on the synchronization aspect, or the partial ordering of communications, of parallel systems [5], [2]. For example, a Petri nets could be used to represent event/condition system where a partial ordering of the occurrence of events is specified but the contain of the events are ignored [5]. CCS (a Calculus for Communicating Systems) defined by Milner [2], on the other hand, models the occurrence of potentially concurrent events as a shuffle (interleaving) of those events; i.e., the events can occur in either order. As such, it has the composition explosion problem. That is when several agents are composed together, the possible number of behaviors are of the exponential order of the number of agents. Consequently the complexity of the behavioral expressions is also increase exponentially. Further, the computational aspects are also abstracted away in CCS. Our model, on the other hand, model both the computations and their synchronizations, which are necessary for synthesis of hardware systems.

Another description model for hardware synthesis which also used external events to characterize semantics of a system has been proposed by McFarland [1]. However, it uses regular expression to formulate the event structures. Consequently it is difficult to deal with concurrent event structures. We are more interested in synthesis of algorithms (finally implemented as hardware) which are expressed as partially ordered events.

2. Definition of the Computation Model

The proposed computation model is based on the concepts of data flow and control flow. The data flow part is modelled as a data path, which represent the existence of multiple hardware resources to perform different operations. The control flow, on the other hand, dictates the partial ordering of these operations. In a parallel computation, there exist more than one control signal streams which move on with their own paces and synchronize with each other only when necessary. The partial ordering relationship between different set of operations is modelled by a Petri net notation.

Definition 2.1: A *data path*, $\mathbf{D}$, over an algebraic structure is a five-tuple, $\mathbf{D} = (\mathbf{V}, \mathbf{I}, \mathbf{O}, \mathbf{A}, \mathbf{B})$,

where $\mathbf{V} = \{V_1, V_2, \ldots, V_n\}$ is a finite set of vertices each of which represents a data manipulation node;

$\mathbf{I} = \mathbf{I}(V_1) \cup \mathbf{I}(V_2) \cup \ldots \cup \mathbf{I}(V_n)$ with $\mathbf{I}(V_j)$ = the set of input ports associated with vertex V_j;

$\mathbf{O} = \mathbf{O}(V_1) \cup \mathbf{O}(V_2) \cup \ldots \cup \mathbf{O}(V_n)$ with $\mathbf{O}(V_j)$ = the set of output ports associated with vertex V_j;
$\mathbf{P} = \mathbf{I} \cup \mathbf{O}$ is the set of ports; it is assumed that $\mathbf{I} \cap \mathbf{O} = \emptyset$.

$\mathbf{A} \subseteq \mathbf{O} \times \mathbf{I} = \{ \langle O, I \rangle \mid O \in \mathbf{O}(V_i), I \in \mathbf{I}(V_j), \text{i,j} = 1,2,\ldots,\text{n} \}$, is a finite set of arcs each of which represents a connection from an output port of a vertex to an input port of another vertex or the same vertex;

$\mathbf{B} : \mathbf{O} \rightarrow \mathbf{OP}$, is a mapping from output ports to operations. $\mathbf{OP} = \{OP_1, OP_2, \ldots, OP_m\}$ is a set of operations which define

This work was supported in part by the Swedish National Board for Technical Development (STU).

the functional relation between an output port of a vertex and its input ports. The set of operations are divided into the sequential set **SEQ** and the combinatorial set **COM**.

Intuitively, a data path is a directed graph with each node having possibly multiple input ports and output ports. The nodes are used to model data manipulation units, for example data storages, arithmetic operators, or communication channels. The arcs are used to model the connections of these data manipulation units.

Therefore, the above definition is concerned mainly with the structure rather than the function of the data path. How the data path is used to perform computation is not explicitly defined. We assume that there exists an implicit interpretation of the underlying algebraic structure which supports the computation rules. Such an algebraic structure should consist of a domain of values for constants and variables, an assignment of values to the constants and a function definition for each operator. This algebraic structure is not considered here as it does not directly affect the basic formulation of the model. Further, to define the semantics of the system independent of any particular interpretation makes it possible to cope with different implementation environments. However, we assume that some modules exist in a module library which can perform the defined operations of the data path.

The notion of ports here is used as a basic abstraction of the input/output behavior of a data manipulation unit and thus separates the implementation of the operation associated with the vertices from the specification. The operation of the vertices are defined only by the relation between the output ports and the input ports. It is assumed that the output port will present a value which has the given relationship with the values present in the input ports.

Definition 2.2: A *data/control flow system*, $\mathbf{\Gamma}$, is a seven-tuple, $\mathbf{\Gamma} = (\mathbf{D}, \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}, M_o)$

where $\mathbf{D} = (\mathbf{V}, \mathbf{I}, \mathbf{O}, \mathbf{A}, \mathbf{R})$ is a data path;

$\mathbf{S} = \{S_1, S_2, ..., S_n\}$ is a finite set of *S-elements*, or *control states (places)*;

$\mathbf{T} = \{T_1, T_2, ..., T_m\}$ is a finite set of *T-elements*, or *transitions*;

$\mathbf{F} \subseteq (\mathbf{S} \times \mathbf{T}) \cup (\mathbf{T} \times \mathbf{S})$ is a binary relation, the *flow relation*.

$\mathbf{C} : \mathbf{S} \rightarrow 2^{\mathbf{A}}$ is a mapping from control states to sets of arcs of the given data path; an arc A_i is *controlled* by a control state S_j if $A_i \in \mathbf{C}(S_j)$.

$\mathbf{G} : \mathbf{O} \rightarrow 2^{\mathbf{T}}$ is a mapping from output ports of data path vertices to sets of transitions; a transition T_i is *guarded* by output port O_j if $T_i \in \mathbf{G}(O_j)$.

$M_o : \mathbf{S} \rightarrow \{ 0, 1 \}$ is an *initial marking* function.

The definition of the data/control flow model is based on the marked Petri net notation. The Petri net S-elements are used to capture the control state concept. When a control state holds a token, a control signal will be generated to control the corresponding arcs in the data path specified by the control mapping function **C**. As there could be more than one control state which holds tokens, there exist multiple control signals in the systems. Further, the flow of these control signals (the temporal relation between signals) is defined by a partial ordering structure, which is captured by the flow relation **F**. To express the control flow being affected by the results of some internal computation, we must be able to use conditional signals (as results of some computation) to affect the control flow. For this purpose, the guarding condition concept is introduced into the Petri net notation; a transition may be guarded by a condition produced from the data path represented as the output port of some vertices.

Definition 2.3: For a data/control flow system $\mathbf{\Gamma} = (\mathbf{D}, \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}, M_o)$:

1. $\mathbf{X} = \mathbf{S} \cup \mathbf{T}$ is the set of *control structure elements.*
2. $\mathbf{F}^{+} = \{\mathbf{F}^{n} \mid n \in \mathbf{N}^{+} \}$, where $\mathbf{F}^{o} =$ *identity* and $\mathbf{F}^{n} = \mathbf{F} \circ \mathbf{F}^{n-1}$ for $n \in \mathbf{N}^{+}$, is the *transitive closure* of **F**.
3. $S_i \Rightarrow S_j$ iff $(S_i; S_j) \in \mathbf{F}^{+}$; $\Leftarrow = (\Rightarrow)^{-1}$
4. $\propto = \Rightarrow \cup \Leftarrow$. S_i and S_j are said to be in *sequential order* if $S_i \propto S_j$.
5. $|| = (\mathbf{S} \times \mathbf{S} \setminus \propto)$. S_i and S_j are said to be in *parallel order* if $S_i \,||\, S_j$.

The data path consists of two kinds of elements, the nodes together with their ports representing the data manipulation units and the arcs representing the connection between those units. Each arc is controlled by, or said to be associated with, some control signals coming from the control Petri net. We can also associate the data manipulation units with the control signals by the following definition.

Definition 2.4: V_k is said to be *associated* with S_j if
$\exists\langle o,i\rangle \in \mathbf{A} \; [\; (i \in \mathbf{I}(V_k)\;) \cap (\langle o,i\rangle \in \mathbf{C}(S_j)) \;]$.

By this definition, only the input ports of a vertex are significant for the *associative* relation. The output ports are irrelevant here because an output port can send data to more than one place at a time without resulting in conflicts. A single input port, on the other hand, cannot receive signals *simultaneously* from more than one resource.

The set of vertices and arcs associated with a control state S forms a subgraph of the data path graph. This graph is called the *associated graph* of S.

Definition 2.5: The arcs and vertices associated with control state S_i, denoted by $\mathbf{ASS}(S_i)$, are said to be *active* under S_i.

Intuitively, the arcs representing the data paths (e.g., a bus) are open, i.e., allow signal to pass, when their associated control signals are on; the associated data manipulation units, on the other hand, will perform predefined operations.

Before we go to the formal definition of the concepts of semantics and semantic equivalence, let us look at some simple examples. Under the above formulation, a simple adder with two input ports and one output port can be modelled as a vertex V_1 with $\mathbf{I}(V_1) = \{P_{i1}, P_{i2}\}$, $\mathbf{O}(V_1) = \{P_{o1}\}$. A register can be modelled as a vertex V_2 with $\mathbf{I}(V_2) = \{P_{i3}\}$ and $\mathbf{O}(V_2) = \{P_{o2}\}$. A data path which connects the output of the adder to the register can be modelled as an arc $A_1 = \langle P_{o1}, P_{i3}\rangle$, which states that the output port of the V_1 component is connected to the input port of V_2.

If the output of the adder is only fed into the register when control state S_1 is on, then $A_1 \in \mathbf{C}(S_1)$ and $\{V_2, A_1\} \subseteq \mathbf{ASS}(S_1)$. Note that V_1 need not necessarily be associated with S_1; if, for example, the adder has a local accumulator, a series of additions can be performed and finally the sum be fed into register V_2 when S_1 is on. When the sum is being sent to V_2, V_1 can continue with another addition associated with, e.g., S_2 without conflict.

3. Semantics of the Model

We now turn our attention to the definition of the semantics of the proposed computation model. The basic idea is that we can characterize the semantics of a system by the external events, i.e., its interactions with the outside world. An external event is either a *read* or *write* operation of the externally accessible ports. The semantics of a hardware system is defined as a set of events observed in its external ports.

Before formally giving the definition of semantics of the computation model, we have to define the behaviors of the system which is in turn based on the execution rules of the control Petri net and its interaction with the data path.

Definition 3.1: Given a data/control system $\mathbf{\Gamma} = (\mathbf{D}, \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}, M_o)$, its *behavior* is defined as below:

1. A function M: $\mathbf{S} \rightarrow \mathbf{N}$ is called a *marking* of Γ ($\mathbf{N} = \{0, 1, 2, \ldots\}$). A marking is an assignment of *tokens* to the S-elements.
2. Initially there is a token in each of the initial control states, or the set of S-elements S_i such that $M_0(S_i) = 1$ as defined by the initial marking M_0.
3. A transition T is *enabled* at a marking M iff for every S such that $(S, T) \in \mathbf{F}, M(S) \geq 1$; that is, all the T-elements' input control states have at least one token.
4. A transition T may be *fired* when it is enabled and the guard condition is true (i.e., the output port which guards T has a TRUE value). If a transition has more than one guard condition, an *OR* operation is applied to them; therefore, if any guard condition is true, the transition's guard condition as a whole is true.
5. Firing an enabled transition T removes a token from each of its input control states and deposits a token in each of its output control states.
6. If no token exists in any of the control states, the execution is terminated.
7. $\mathcal{V}(P)$ is the data value present at port P.
8. When a control state, S, holds a token, its associated arcs in the data path will open for data to flow; i.e., the data value presents at the input port, I, is equal to the corresponding output port, O, which is denoted as $\mathcal{V}(I) \dashv S \; \mathcal{V}(O)$.
9. For every vertex V, $\mathcal{V}(O) := OP(\mathcal{V}(\mathbf{I}(V)))$, where $OP \in \mathbf{B}(O)$. The assignment operator, :=, means that if OP is sequential it takes the last defined value of the expression; otherwise it takes the present value of the expression.
10. If all the pending arcs of an input port are not active, its value is *undefined.* If the operation of an output port is not a sequential one and the output port depends on an undefined input value, its value is also *undefined.*

The possible existence of an undefined value and the intrinsic non-deterministic properties of the Petri net firing sequence together result in difficulties in determining the behavior of a system. We would like to exclude the nondeterministic properties by the following definition.

Definition 3.2: A data/control flow system $\Gamma = (\mathbf{D}, \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}, M_0)$ is properly designed if:

1. $\mathbf{ASS}(S_i) \cap \mathbf{ASS}(S_j) = \emptyset$, if $S_i \parallel S_j$.
2. There should not be more than one token appearing at the same control state; that is, the Petri net must be safe.
3. If $(S, T_1) \in \mathbf{F}$, $(S, T_2) \in \mathbf{F}$, $T_1 \in \mathbf{G}(P_{o1})$, and $T_2 \in \mathbf{G}(P_{o2})$, then $\mathcal{V}(P_{o1})$ *AND* $\mathcal{V}(P_{o2})$ = FALSE. That is, the Petri net must be conflict-free.
4. The subgraph that belongs to a control state should not include a combinatorial loop.
5. $\forall S_i \in \mathbf{S}$ $\mathbf{ASS}(S_i)$ must include at least one sequential vertex.

This definition singles out those data/control flow systems which are safe, conflict-free and well-behaved. From now on we only consider properly designed systems.

Definition 3.3: For a data path $\mathbf{D} = (\mathbf{V}, \mathbf{I}, \mathbf{O}, \mathbf{A}, \mathbf{R})$, there is a set of *external vertices*, $\mathbf{V}_e$, which only have either one single input port (the set of *output vertices*, $\mathbf{V}_o$) or one single output port (the set of *input vertices*, $\mathbf{V}_i$). The set of ports of the external vertices $\mathbf{V}_e$ are called *external ports.* The set of arcs, $\mathbf{A}_e$, which connect to the external ports, are called *external arcs.*

Definition 3.4: A *external event* is a pair $\langle A_i, v_i \rangle$, with A_i being an external arc and v_i a value passed over the arc. A external event is controlled by, or labelled with, the Petri net control state that is associated with the arc. That is, the external event happens at the time when the associated control state has a token.

Definition 3.5: Given a data/control flow system $\Gamma = (\mathbf{D}, \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}, M_0)$, its *external event structure* is defined as $\mathcal{S}(\Gamma) = (\mathbf{E}, \prec, \asymp)$ where

$\mathbf{E} = \{E_1, E_2, \ldots, E_n\}$ is a set of external events;

$\prec \subseteq (\mathbf{E} \times \mathbf{E})$ is a binary relation, the *precedent* relation. $E_i \prec E_j$ with $E_i = \langle A_i, v_i \rangle$ and $E_j = \langle A_j, v_j \rangle$, iff E_i occurs before E_j and $S_i \Rightarrow S_j$, where $A_i \in \mathbf{C}(S_i)$ and $A_j \in \mathbf{C}(S_j)$;

$\asymp \subseteq (\mathbf{E} \times \mathbf{E})$ is a binary relation, the *concurrent* relation. $E_i \asymp E_j$ with $E_i = \langle A_i, v_i \rangle$ and $E_j = \langle A_j, v_j \rangle$, iff E_i and E_j occurs at the same time and $A_i \in \mathbf{C}(S)$, $A_j \in \mathbf{C}(S)$.

An external event structure specifies all the possible external events of a system as well as the temporal relationship between them. If two external events are in the precedent (concurrent) relation, they must always occur in the specified order (simultaneously). On the other hand, if two events are not in either of the two relations, they can occur in any order and are said to be in a *casual* relation. In a distributed system with a set of modules, for example, the temporal relations between some of the external events of two different modules can best be expressed as having a casual relation. Trying to force a total ordering on events of different modules will simply introduce unnecessary constraints and make it difficult to implement the system.

In the above discussion, we assume that when an external event occurs whose operation is to obtain a value from the outside world, the environment will supply a value of the appropriate type to the system. We also assume that a sequence of such values is implicitly predefined for each input vertex, when an external event structure is specified.

Definition 3.6: The semantics of a data/control flow system Γ, denoted also by $\mathcal{S}(\Gamma)$, is defined by its external event structure.

4. Semantics Equivalence

Two systems are considered to be semantically equivalent if they behave identically with respect to the corresponding external ports; their internal behavior does not matter.

Definition 4.1: Two data/control flow systems Γ and Γ' are *semantically equivalent*, denoted by $\Gamma \equiv \Gamma'$, if $\mathcal{S}(\Gamma) = \mathcal{S}(\Gamma')$.

For the purpose of synthesis, however, the above semantic equivalence relation is still too weak. In general, it is undecidable whether two systems are equivalent to each other by this definition. It is very difficult, or simply impossible in some cases, to analyze a data/control flow system and obtain the complete external event structure as specified by definition 3.5. We have thus to introduce a stronger equivalence relation which requires every data dependence operation to be carried out in exactly the same order. This latter requirement is stronger than necessary. For example, two addition operations can be carried out in reverse order without changing the outcome of the computation. This strong definition, however, greatly reduces the complexity of the synthesis process and still provides enough room for the optimization algorithm to make large changes in the described system.

Definition 4.2: The domain of a control state S, denoted as $\mathbf{dom}(S)$, is defined as the set of vertices that have some output port connected to an arc controlled by S. The codomain of S, denoted as $\mathbf{cod}(S)$, is defined as the set of vertices which have some input port connected to an arc controlled by S. The operations performed on a control state S are the set of operations defined on the output ports of its codomain. The subset of vertices of the codomain of S that consists of some sequential output ports is called the result set of S and denoted as $\mathbf{R}(S)$.

Definition 4.3: S_i and S_j are *directly data dependent*, denoted as $S_i \leftrightarrow S_j$, if one of the following is true:

(a) $\mathbf{R}(S_i) \cap \mathbf{dom}(S_j) \neq \emptyset$.

(b) $\mathbf{R}(S_j) \cap \mathbf{dom}(S_i) \neq \emptyset$.
(c) $\mathbf{R}(S_i) \cap \mathbf{R}(S_j) \neq \emptyset$.
(d) S_i and S_j are in a control dependence relation; i.e., $M(S_i)$ depends on a subset of $\mathbf{R}(S_j)$ or vice versa.
(e) $\mathbf{C}(S_i)$ and $\mathbf{C}(S_j)$ both contain some external arcs.

Definition 4.4: The transitive closure of $\leftrightarrow$, denoted by $\diamond$, i.e., $\diamond = \leftrightarrow^+$, is called a *data dependence* relation.

The data dependence relation is defined as the relationship between the operations which will contribute "data" to each other; in other words, two operations are data dependent if they must be executed in the predefined order in order to retain the semantic integrity of the prescribed computation. Those sets of control signals which are not in a data dependence relation, however, can be arranged in any order without changing the semantics of the system.

Definition 4.5: Given $\Gamma = (\mathbf{D}, \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}, M_o)$ and $\Gamma' = (\mathbf{D}, \mathbf{S}, \mathbf{T}', \mathbf{F}', \mathbf{C}, \mathbf{G}, M_o)$, Γ and Γ' are data-invariantly equivalent to each other, iff
for every $S_i \Rightarrow S_j$ and $S_i \diamond S_j$ in Γ $(S_i \in \mathbf{S}, S_j \in \mathbf{S})$,
we have $S_i \Rightarrow' S_j$ and $S_i \diamond' S_j$ in Γ';
and vice versa.

The above definition ensures that two operations are performed in parallel only if they are data independent and all of the data dependent operations in the two systems are performed exactly in the same order. Therefore the data-invariant equivalence relation satisfies the semantic equivalence relation. This means that we can reconstruct the control structure (without changing the data path) of a hardware system to improve system performance, for example, by carrying out as much operations in parallel as possible.

Theorem 4.1: The data-invariant equivalence relation satisfies the semantic equivalence relation.

Proof 4.1: *see the appendix.*

Definition 4.6: Given $\Gamma = (\mathbf{D}, \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}, M_o)$ with $\mathbf{D} = (\mathbf{V}, \mathbf{I}, \mathbf{O}, \mathbf{A}, \mathbf{B})$ and $\Gamma' = (\mathbf{D}', \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}', M_o)$ with $\mathbf{D}' = (\mathbf{V}', \mathbf{I}', \mathbf{O}', \mathbf{A}', \mathbf{B}')$, Γ and Γ' are control-invariantly equivalent to each other, iff Γ' is the result of a *vertex merger* of V_i into V_j of Γ, both V_i and V_j have the same operational definition and port structure, and their associated control states are in sequential order. The result of a vertex merger is defined as:

$\mathbf{V}' = \mathbf{V} - \{V_i\}$.
$\mathbf{I}' = \mathbf{I} - \{\mathbf{I}(V_i)\}$.
$\mathbf{O}' = \mathbf{I} - \{\mathbf{O}(V_i)\}$.
$\mathbf{A}'$ is the same as $\mathbf{A}$ except that each $\langle O_i, I\rangle$ with $O_i \in \mathbf{O}(V_i)$ is replaced by $\langle O_j, I\rangle$ with $O_j \in \mathbf{O}(V_j)$ and each $\langle O, I_i\rangle$ with $I_i \in \mathbf{I}(V_i)$ replaced by $\langle O, I_j\rangle$ with $I_j \in \mathbf{I}(V_j)$.
$\mathbf{G}'$ is the same as $\mathbf{G}$ except that each $T \in \mathbf{G}(O_i)$ is substituted by $T \in \mathbf{G}(O_j)$.

The intrinsic property of a merger operation is to share hardware resources by operations so as to improve the implementation in terms of cost. For example two addition operations can be implemented with the same adder by merging the two addition vertices together. By merging communication channels together we can also create structure components like buses in the implementation.

As a merger is only performed when the two vertices have their associated control states in sequential order, they will not attempt to use the vertex at the same time. As such the two sets of operations can share the same operator safely. Because the two vertices to be merged also have the same operational definition and port structure, the merger will not change the computational aspect of the given system.

Theorem 4.2: The control-invariant equivalence relation satisfies the semantic equivalence relation.

Proof 4.2: *see the appendix.*

5. Hardware Synthesis

This section discusses briefly the application of the proposed parallel computation model in a hardware synthesis environment. For a detailed description of the synthesis algorithms and comparisons to other related works, please see [3] and [4].

To synthesize hardware from some algorithmic description of its behavior, we first transform the description into the data/control flow notation. Based on such a formal description, some formal analysis techniques can first be used to check whether the systems are properly designed before the synthesis process starts [4].

The major part of the synthesis process is carried out by a sequence of control-invariant and data-invariant transformations as defined in the previous section. Since both transformations do not change the semantics of the system, they can freely be applied to transform a design to satisfy certain given criteria. For example, adding one more control flow path in the Petri net and possibly additional data manipulation units in the data path will allow more operation units to operate at the same time, thus increasing the parallelism of the computation.

The synthesis algorithm starts with a preliminary design and transforms it step by step towards an optimal one. As from each step there are usually several ways to go, it is necessary to have some strategy to guide the transformation process. A critical path analysis technique is used for this purpose. The set of transformation, analysis, and optimization algorithms has been designed and implemented in the CAMAD design aid system [3], [4].

6. Conclusions

We have given the formal definition of a data/control flow model for parallel computation and its semantic equivalence notation. The concept of semantic equivalence is defined based on two criteria. First the functional relationship between each output variable and its relevant input variables must be the same; secondly the temporal relationship between input/output operations should also be the same.

Unlike other computation models used mainly for descriptive and analysis purposes, the proposed model addresses issues of design directly and allows graphical representations of the structures as well as behaviors of hardware system. To apply this model for hardware synthesis, we have introduced two basic transformations which change the internal structure of the hardware but keep the data dependency operations in the predefined order. The requirement that all data dependency operations be carried out in the predefined order is actually stronger than necessary. For example, two addition operations can be carried out in a reversed order without changing the outcome of the computation. It, however, greatly reduces the complexity of the synthesis process. The use of such a formal computation model to represent the design of parallel hardware has led to the efficient use of CAD and automatic tools in the synthesis process.

Appendix

Proof 4.1: Let $\Gamma = (\mathbf{D}, \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}, \mathbf{M}_o)$, $\Gamma' = (\mathbf{D}, \mathbf{S}, \mathbf{T}', \mathbf{F}', \mathbf{C}, \mathbf{G}, \mathbf{M}_o)$, and Γ and Γ' are data-invariant equivalent to each other, i.e., for every $S_i \Rightarrow S_j$ and $S_i \diamond S_j$ in Γ $(S_i \in \mathbf{S}, S_j \in \mathbf{S})$, we have $S_i \Rightarrow' S_j$ and $S_i \diamond' S_j$ in Γ'; and vice versa. We will show that the external event structure of Γ and that of Γ' are the same.

Suppose that a sequence of external events, $\langle A_i, v_{i1}\rangle$, $\langle A_i, v_{i2}\rangle$, $\langle A_i, v_{i3}\rangle$, ..., are observed in arc A_i which is associated with control state S in system Γ. As the data path of system Γ' is the same as that of Γ, A_i should also be present in Γ' as an external arc and controlled by S in Γ'.

For the values exchanged over A_i, we have two situations:

(1) If A_i is connected to an input vertex, the function of the external events is to input data from the environment. The values passed over the arc are then provided by the environment. As we assume that the sequence of such values

provided for each input vertex is fixed when we check the semantic equivalence relation between different systems, the same sequence of external events will be observed in system $\mathbf{\Gamma}'$.

(2) If A_i is connected to an output vertex, the function of the external events is to output data to the environment. The values passed over A_i are, therefore, determined by the computation performed by the systems.

Let $A_i = \langle O, I \rangle$, and when $M(S) = 1$, an external event $\langle A_i, v_i \rangle$ occurs with $v_i = \mathcal{V}(O)$ (definition 3.4). If $O \in \mathbf{O}(V)$ and V is an input vertex (i.e., A_i connects an input vertex directly to an output vertex), $\mathcal{V}(O)$ depends again on the environment. Therefore, both systems exchange the same values at A_i.

If $O \in \mathbf{O}(V)$ and V is not an input vertex, we have $\mathcal{V}(O) := OP(\mathcal{V}(\mathbf{I}(V)))$, where $OP \in \mathbf{B}(O)$; and $\mathcal{V}(I_i) \dashv S_i\ V(O_i)$ for each $I_i \in \mathbf{I}(V)$ (definition 3.1). As $V \in \mathbf{dom}(S)$ and $V \in \mathbf{R}(S_i)$ (we have assumed that V is a sequential vertex, without loss of generality), we have $S \Diamond S_i$ and, therefore, $S_i \Rightarrow S$. Since both systems have the same data path and $S_i \Rightarrow S$ in both situations, the values exchanged at A_i should be the same provided that each $\mathcal{V}(O_i)$ for the corresponding systems is the same.

To show that $\mathcal{V}(O_i)$ is the same for $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$, we can use the same proof process as above. This recursive procedure will also converge to the situation where V is an input vertex. At that time the same argument as from (1) can be applied again. Therefore, the the same sequence of external events will be observed in A_i of both system $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$.

From (1) and (2), it is clear that the sequence of external events observed at A_i of $\mathbf{\Gamma}'$ is exactly the same as that of $\mathbf{\Gamma}$ in any situation. As A_i can be any arbitrary external arc, this means that the sequence of external events which occur at *every external arc* is the same for both systems.

As $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$ have the same number of corresponding external arcs, it follows from the above result that the complete sets of external events for both systems are the same.

Next let us look at the partial relation between the external events of the two systems. Suppose that $E_i \prec E_j$ with $E_i = \langle A_i, v_i \rangle$ and $E_j = \langle A_j, v_j \rangle$ in $\mathbf{\Gamma}$. That is, E_i occurs before E_j and $S_i \Rightarrow S_j$, where $A_i \in \mathbf{C}(S_i)$ and $A_j \in \mathbf{C}(S_j)$. By definition, we have $S_i \Rightarrow' S_j$ in $\mathbf{\Gamma}'$, where $A_i \in \mathbf{C}(S_i)$ and $A_j \in \mathbf{C}(S_j)$, because $S_i \Diamond S_j$ (thus $S_i \Diamond' S_j$).

Assuming E_j occurs before E_i in $\mathbf{\Gamma}'$, then we must have $S_j \Rightarrow S_i$ in both $\mathbf{\Gamma}'$ and $\mathbf{\Gamma}$. That is, S_i and S_j are in a loop situation. Consequently, there exists a total ordering between the external events associated with these two control states, and it should be the same in both $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$. Thus the assumption that E_j occurs before E_i in $\mathbf{\Gamma}'$ is a contradiction. Therefore we have also E_i occurs before E_j in $\mathbf{\Gamma}'$. That is, $E_i \prec E_j$ is also in $\mathbf{\Gamma}'$.

Finally, we show that the concurrent relations of both systems are also the same as follows.

If $E_i = \langle A_i, v_i \rangle$ and $E_j = \langle A_j, v_j \rangle$ occur at the same time and $A_i \in \mathbf{C}(S)$, $A_j \in \mathbf{C}(S)$ in $\mathbf{\Gamma}$, then we should have $A_i \in \mathbf{C}(S)$, $A_j \in \mathbf{C}(S)$ in $\mathbf{\Gamma}'$, because the control mapping $\mathbf{C}$ is the same for both systems. Consequently, E_i and E_j should also occur at the same time in $\mathbf{\Gamma}'$ as they are associated with the same control state. Therefore, both system have the same concurrent relation.

Since both system $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$ have the same external event set, the same precedent relation, and the same concurrent relation, $\mathcal{S}(\mathbf{\Gamma}) = \mathcal{S}(\mathbf{\Gamma}')$. That is, they are semantically equivalent to each other.

Proof 4.2: Let $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$ be control-invariant equivalent to each other. That is, (a) $\mathbf{\Gamma}'$ is resulted from a *vertex merger* of V_i into V_j of $\mathbf{\Gamma}$, (b) both V_i and V_j have the same operational definition and port structure, and (c) their associated control states are in sequential order.

Assume that the merger of V_i and V_j changes the semantics of the system. That is, $\mathcal{S}(\mathbf{\Gamma}) \neq \mathcal{S}(\mathbf{\Gamma}')$, or $(\mathbf{E}, \prec, \asymp) \neq (\mathbf{E}', \prec', \asymp')$. Because the control structures of both systems are the same, the temporal relationship between any two control states remain the same for both systems.

As the number of arcs also remains the same after the merger operation and they are controlled by the same control states, the number of external events and their temporal relation remain the same for both systems. That is, the precedent relation and concurrent relation of both systems are the same. Therefore, the only possible difference between the two external event structures is that some of the external events have different values.

For the external events that occur at an arc connected to an input vertex, the same argument of Proof 4.1(1) can be used to prove that both $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$ have the same values passed in these external events.

For the external arcs that are connected to an output port, let $A = \langle O, I \rangle$ with $I \in \mathbf{I}(V_e)$ and $V_e \in \mathbf{V}_o$; $\{ \langle A, v_{i1} \rangle, \langle A, v_{i2} \rangle, \langle A, v_{i3} \rangle, \dots \} \subseteq \mathbf{E}$ and occur in the listing order in $\mathbf{\Gamma}$; and $\{ \langle A, v_{j1} \rangle, \langle A, v_{j2} \rangle, \langle A, v_{j3} \rangle, \dots \} \subseteq \mathbf{E}'$ and occur in the listing order in $\mathbf{\Gamma}'$.

Let also $\langle A, v_{ik} \rangle$ and $\langle A, v_{jk} \rangle$ occur when $M(S) = 1$ in $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$ respectively. We have $v_{ik} = \mathcal{V}(O)$ in $\mathbf{\Gamma}$ and $v_{jk} = \mathcal{V}(O)$ in $\mathbf{\Gamma}'$.

If $O \in \mathbf{O}(V)$ and V is an input vertex in $\mathbf{\Gamma}$, we have also $O \in \mathbf{O}(V)$ and V as an input vertex in $\mathbf{\Gamma}'$. Since in both cases $\mathcal{V}(O)$ depends on the environment, $v_{ik} = v_{jk}$.

If $O \in \mathbf{O}(V)$, V is not an input vertex, and $V \neq V_i$, we have $\mathcal{V}(O) := OP(\mathcal{V}(\mathbf{I}(V)))$, where $OP \in \mathbf{B}(O)$ both in $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$. By definition 3.1, $\mathcal{V}(I_i) \dashv S_i\ V(O_i)$ for each $I_i \in \mathbf{I}(V)$. As both system have $S_i \Rightarrow S$ (see Proof 4.1), $v_{ik} = v_{jk}$, provided that each $\mathcal{V}(O_i)$ for both systems is the same.

If $O \in \mathbf{O}(V)$, V is not an input vertex, and $V = V_i$ in $\mathbf{\Gamma}$, we have $\mathcal{V}(O) := OP(\mathcal{V}(\mathbf{I}(V_i)))$, where $OP \in \mathbf{B}(O)$ in $\mathbf{\Gamma}$ and $\mathcal{V}(O) := OP(\mathcal{V}(\mathbf{I}(V_j)))$, where $OP \in \mathbf{B}(O)$ in $\mathbf{\Gamma}'$. Since (a) both V_i and V_j have the same operational definition and port structure; (b) $\mathcal{V}(I_i) \dashv S_i\ \mathcal{V}(O_i)$ for each $I_i \in \mathbf{I}(V_i)$ in $\mathbf{\Gamma}$ with $\mathcal{V}(I_i) \dashv S_i\ \mathcal{V}(O_i)$ for each $I_i \in \mathbf{I}(V_j)$ in $\mathbf{\Gamma}'$; and (c) both systems have $S_i \Rightarrow S$; we have $v_{ik} = v_{jk}$, provided that each $\mathcal{V}(O_i)$ for both systems is the same.

To show that $\mathcal{V}(O_i)$ is the same for $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$ in the above two cases, we can use the same proof process again. The recursive procedure will also converge to the situation where V is an input vertex; then the same argument as from Proof 4.1(1) can be applied. Therefore, the same sequence of external events will be observed in A of both $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$.

This result contradicts the assumption that some corresponding events of $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'$ are different. That is, the assumption must be false. Therefore, $\mathbf{\Gamma} \equiv \mathbf{\Gamma}'$.

References

[1] McFarland, Mickeal C. and Parker, Alice C., *An Abstract Model of Behavior for Hardware Descriptions*, IEEE Trans. on Computers, Vol.32, No.7, 1983, pp.621-637

[2] Milner, R., *A Calculus for Communicating Systems*, Lecture Note in Computer Science, No. 92, 1980

[3] Peng, Z., *Synthesis of VLSI Systems with the CAMAD Design Aid*, Proc. 23rd ACM/IEEE Design Automation Conf., 1986, pp.278-284

[4] Peng, Z., *A Formal Methodology for Automated Synthesis of VLSI Systems*, Ph.D. Dissertation, Dept. of Computer and Information Science, Linköping University, No. 170, 1987

[5] Peterson, J., *Petri Net Theory and the Modeling of Systems*, Prentice-hall, 1981

[6] Thomas, D., Hitchcock, C., Kowalski, T., Rajan, J., and Walker, R., *Automatic Data Path Synthesis*, Computer, IEEE, 1983, pp.59-70

An Asynchronous Distributed Approach for the Simulation of Behavior-Level Models on Parallel Processors

Sumit Ghosh　　Meng-Lin Yu

AT&T Bell Laboratories Research
Holmdel, NJ 07733.

Abstract

This paper presents an asynchronous distributed approach for the simulation of behavior-level models representing complex digital and VLSI components on a parallel processor. The underlying architecture is a set of concurrent processors that share data through explicit messages such as a hypercube [1]. The approach is implemented on the Bell Labs hypercube [2] that consists of 64 concurrent processors connected by a network of point-to-point communication channels in the plan of a binary 6-cube and provides a protocol-based operating system. A complex design is first partitioned and the behavior-level models corresponding to the components of each partition are assigned to a processor. A model determines, based on the input signal transitions at the input ports, whether it may be scheduled for execution and, consequently, scheduling is distributed in the models. However, within each processor, only one behavior model may execute at any time instant. During execution of a behavior description, the signal transitions at an output port may be determined based on the signal values at all input ports defined up to t = t_1 such that every input signal is is defined up to t = t_1. In addition, the assertion of a signal transition at an output port is deferred until the model description may determine with certainty that no future input signals may prove it inconsistent and require its deletion [3,4]. The behavior of digital and VLSI components including complex timing are expressed through the language constructs of C++ [5].

1 Introduction

The discipline of synchronous distributed simulation of digital designs at the logic level on parallel processors has been addressed by the Yorktown Simulation Engine [6], IBM Los Gatos Logic Simulation Machine [7], and ZYCAD [8]. The subject of asynchronous distributed simulation with a focus on queuing networks has been addressed in the recent past by Misra [9], Chandy [10], Lamport [11] and Peacock [12]. The Daisy Megalogician [13] and ULTIMATE [14] machines address the issues of parallelizing a simulation algorithm.

Behavior models of complex digital and VLSI devices are flexible and provide a competitive means of system simulation [15] and results of simulation are more comprehensive to the high-level architects as opposed to the gate-level simulation results. Consequently, the importance of distributed simulation of such models on parallel processors is obvious. The difference between this approach and the one proposed by Misra [9] may be expressed as follows.

Accurate representation of components' behavior including timing in the models require the representation of the unique high-to-low (t_{phl}) and low-to-high (t_{phl}) propagation delays for every component. For an input signal transition at t = t_1, the "predictability" condition [9] would imply the generation of an output transition at t = $t_1 + t_{phl}$ or t = $t_1 + t_{plh}$ depending on the nature of the transition and its assertion at the output port. The predictability condition is an important aspect of the approach proposed by Misra [9]. Such an assertion may cause incorrect simulation results as an input signal transition at a future time t = t_2 ($t_2 > t_1$) may, under certain circumstances, generate a new output transition that requires the previous output transition to be discarded [3,4]. The cause of such potentially unreliable simulation results may be attributed to the anticipatory semantics of the behavior description language and event driven simulation. In the approach presented in this paper, the behavior description first determines with certainty that an output transition may not be discarded and then asserts it at the output port. In contrast to Chandy's [10] proposal of simulating to a deadlock and then recovering from it, the approach presented here may be characterized by an absence of deadlocks.

2 Asynchronous Distributed Simulation on Parallel Processors

An asynchronous distributed approach for the simulation of behavior models on a special parallel processor architecture - hypercube, is presented in this section. The potential advantage of this approach over conventional sequential simulation on an uniprocessor is faster speed. Execution of digital or VLSI hardware may be characterized by exchange of signals between the component modules that is constituted by a sequence of signal transitions. A transition may be characterized by a logical value and assertion time. In conventional simulation, the ordering of the signal transitions or events for correct results is achieved through a global entity - time, and centralized control. In this approach, the ordering is guaranteed by a sequence of messages between the models and their proper interpretation and usage by each of the behavior models. In this paper a message represents a signal transition. The overall philosophy may be expressed as follows. Each and every behavior model correctly interprets messages at the input ports, determines the output signal transition based on the input signals, and asserts only correct output assignments at the output port through messages. Consequently, for a given set of external signals at the primary input ports, correct simulation results are guaranteed. In addition, explicit identification of the clock lines are not required and as transitions corresponding to every signal including clocks may be expressed through messages and the output determined by the behavior description in a model solely based on the input transitions, synchronous and asynchronous including self-timed designs may be simulated in this approach.

First a given digital or VLSI design is partitioned into 63 or less partitions corresponding to 63 processors and processor 0 is dedicated to the task of asserting the external signal transitions at the primary input ports of the design. For a modest-size design with less than 63 behavior models, each processor may be allocated a model for simulation.

The task of scheduling behavior models for execution is distributed in them and a model schedules itself when it determines that necessary conditions, described subsequently, have been satisfied at the input ports. Given n input ports I_1,...,I_n of a component C and signal transitions at the ports defined up to t = t_1, ..., t = t_n, respectively, the corresponding model may execute and determine the signal transition at the output port that is based on the input signals defined up to t = t_x where t_x is the minimum of $\{t_1, ..., t_n\}$. Assuming a value "d" for the propagation delay of the component, the output signal transition may be defined at t = t_x + d but its assertion is deferred because of the possibility that a future input transition defined at t > t_x may cause an output transition that is inconsistent with the previously generated transition at t = t_x + d and require its deletion. An output transition that is defined at t $\leq$ t_x and was generated corresponding to a previous execution of the model may not be affected by any future input transition defined at t > t_x and the behavior model may, with certainty, assert the transition at the output port. This principle is referred to as the deferred assertion of output assignments. The issue of generation, detection, and deletion of inconsistent output assignments is detailed in [3,4] and is not presented here.

Consider the simulation of a circuit shown in Figure 1. Although a simple circuit is chosen for simplicity of explanation, the distributed approach applies equally to complex behavior models. The output ports of components A and B are connected to the inputs of the two-input AND gate C and the signal transitions generated by each of A and B between t = 0 and t = 30 are shown in Figure 1. Assuming models A and B are allocated arbitrarily to processors 2 and 3, A and B are executed asynchronously and, as a result, the real time during simulation at which the signal transitions are propagated from A to C and B to C may not relate to each other. The individual transitions from A to C – n_1: 0 at t = 0, n_2: 1 at t = 10, n_3: 0 at t = 20, and n_4: 1 at t = 30 where t represents the simulation time are guaranteed to be asserted in order that is represented in logical time [11] as shown in Figure 2a. Figure 2b represents a similar ordering of the transitions from B to C – m_1: 1 at t = 0, m_2: 0 at t = 20, and m_3: 1 at t = 30 in logical time. For the purpose of explanation, assume that the ordering of the transitions in real time is represented by Figure 3. The correctness of the distributed approach is invariant to the ordering of the events in real time given that the logical ordering specified in each of the Figures 2a and 2b is preserved. In Figure 3, assume m_1, n_1, n_2, n_3, m_2, m_3, and n_4 are asserted at C at real times T = s_1, T = s_2, T = s_3, T = s_4, T = s_5, T = s_6, and T = s_7 respectively, where T represents the progress of real time during simulation on the parallel processor.

Corresponding to the assertion of an input transition at T = s_1, C is unable to schedule for execution as the signal transition at port 2 is yet to be specified for t = 0 where t represents the progress of simulation time and corresponds to the hardware execution. At T = s_2, C schedules itself for execution and an output transition l_1: 0 at t = 0 + 5 = 5 is determined. Given that the previous value of the output was 0, l_1 does not imply any new information and is ignored. Corresponding to each of n_2 and n_3 at T = s_3 and T = s_4 respectively, the behavior description of C is not executed as the signal transition at port 2 has not been asserted beyond t = 0. At T = s_5, signal transitions have been specified at t = 20 at both ports 1 and 2 and C is scheduled for execution. Output transitions l_2: 1 at t = 10 + 16 = 26 and l_3: 0 at t = 20 + 5 = 25 are generated but l_2 is observed to be inconsistent with l_3 and consequently discarded. Assertion of the output transition l_3: 0 at t = 25 is deferred as the transitions at the input port of C are defined at t = 20 and the model is yet unable to conclude with certainty that l_3 may not be discarded in the future. At T = s_6, transition m_3 is asserted at an input port of C but the input signal at port 1 is yet undefined beyond t = 20. Consequently neither C may be executed nor any decision regarding l_3 be finalized. At T = s_7, input signals at ports 1 and 2 are both defined at t = 30 and given that l_3 has not yet been shown inconsistent and 30 > 25, it may be asserted, with certainty, at the output port of C and consequently propagated to other components that are connected to the output port of C. In addition, the behavior description of C is executed and an output assignment l_4: 1 at t = 30 + 16 = 46 is determined and stored within the model.

3 Blocking and Deadlock

The number of active gates during gate-level simulation has generally been observed to be between 5% and 20% and may be assumed to hold true for behavior level simulation. Such a low activity may cause the following scenario during asynchronous distributed simulation. For example, in Figure 1 assume component A executes a number of times due to signal transitions at its input ports between t = 0ns and t = 1000ns (say) and asserts a number of transitions at port 1 of C. Also assume B executes infrequently due to a limited set of transitions at its input port between t = 0ns and t = 1000ns with the consequence that only one transition at t = 2ns (say) is asserted at port 2 of C. The value of the signal at port 2 remains essentially unchanged between t = 2ns and t =1000ns. Consequently, C may not execute beyond t = 2ns and this situation constitutes blocking [12] with the source being component B. In addition, other components, if any, that are connected to the output port of C either directly or indirectly will be blocked implying a possibility of very low overall activity during simulation.

Blocking does not correspond to a physical process in hardware execution and its cause may be explained as follows. Event driven simulation with selective trace requires, for efficiency, that only changes in the logical value of a signal be propagated. Consequently, the value of the signal between two consecutive transitions e_1 and e_2 is identical to the value indicated in e_1 in an uniprocessor environment. Such an assumption is dangerous in the distributed asynchronous simulation on a parallel processor as a message to the input port of a component may be delayed due to asynchrony and the behavior model may erroneously interpret the absence of message to imply "no change" in the logical value at that port. Consequently, a component must execute based on signals at input ports at t = t_1 such that transitions have been asserted at all input ports at t $\geq$ t_1. Such a mechanism as well as the principle of deferred assertion of output assignments may increase the possibility of occurrence of blocking.

In the event that blocking occurs during simulation of a design, it is first detected in the following manner. When the number of input assignments at an input port of a component that have not yet been used to generate output events exceed a threshold, the component raises an exception. As a consequence of the exception, the execution mode of every processor is set to "exception-mode". The execution mode of the processors is reset from exception-mode when the cause of blocking is removed i.e., the number of outstanding input assignments at the input port of the component falls below the threshold. The actual value of the threshold is empirically determined and it influences the relative durations of normal- and exception-modes during a simulation. The characteristics of the exception-mode may be expressed as follows. Signal values are asserted at all input ports of components including the primary input ports even when the logical values are unchanged from their previous values. In addition, when a model is executed at t = t_1, either a previously generated correct signal transition that was not yet asserted at the output port is propagated to the output or the most recent logical value at the output is asserted at t = t_1 plus the minimum of the high-to-low and low-to-high propagation delays of the component.

Assume that the components A and B in Figure 1 are executed on processor I of a parallel processor system while model C is executed on another processor II of the system. Assume further that a significant number of signal transitions are asserted at the input ports of A and that the signals at the input ports of B are virtually unchanged in their logical values. Consequently, A is executed frequently and B is executed very infrequently and very few output transitions are asserted at the input port 2 of C. The model C is unable to execute in the absence of signal transitions at the input port 2 and the number of outstanding input entries at the input port 1 of C may exceed the threshold. Consequently, C raises an exception and the execution modes of all the processors is set to exception-mode. In this mode, signal transitions are asserted at the input ports of B even though they are unchanged in their logical values. Consequently, B is executed more frequently and a modest number of output transitions are asserted at the port 2 of C. The model C is executed and the outstanding entries at the port 1 are utilized to generate output assignments and the cause of blocking is removed.

The possibility of deadlocks during asynchronous distributed simulation of designs with feedback loops and their resolution is addressed in the remainder of this section. Consider simulation of a simple latch shown in Figure 4.

Assume the presence of signal transitions defined between t = 0ns and t = 1000ns at the input ports l and k of components A and B respectively. Neither A nor B may execute as explained subsequently. A may schedule itself for execution when transitions are propagated to its input port R from the output of B following execution of B. However, B may not schedule itself for execution until A has executed and asserted transitions at its input ports. Consequently, a deadlock is achieved. This paper presents an approach that ensures absence of deadlocks and implements the principle of deferred scheduling for correctness of the results. A somewhat similar approach has been proposed by Peacock [12].

Every component on a feedback arc is identified and the behavior descriptions corresponding to such components are modified to perform the following action. Given that t_{phl} and t_{plh} values are associated with every component, execution of a behavior model at t = t_1 may generate an output transition at t = t_1 + t_{plh} or t = t_1 + t_{phl} depending on the nature of the transition. Where $t_{plh} < t_{phl}$ and the assertion time of the output transition is given by t = t_1 + t_{phl}, the transition is stored within the body of the model and its assertion deferred until a later time. Instead, a timestamp with the assertion time given by t = t1 + t_{plh} is generated and propagated through the output port as the logical value of the signal at the output port will, with certainty, remain unchanged up to t < t_1 + t_{plh}. Where the assertion time of the output transition is given by t = t_1 + t_{plh}, it may be asserted at the output port immediately as no future transitions at the input port beyond t = t_1 may cause the output transition to be discarded. A limitation of this approach is that the efficiency of simulation of circuits with feedback loops may be low when the components constituting the feedback loop are distributed over 2 or more processors and the frequency of the signal at the non feedback port is considerably lower as compared to the sum of the propagation delays of the components constituting the feedback loop.

4 Analysis of Performance of the Asynchronous Distributed Approach

The asynchronous distributed simulation approach has been implemented on the Bell Labs hypercube [2] that consists of 64 concurrent processors and provides a protocol-based operating system. The behavior models of VLSI and digital components are described through the C++ [5] language constructs.

In an experiment to estimate the performance of the asynchronous distributed approach, a typical example design - two-bit adder, is considered where the individual gates are replaced by models whose execution times may be parametrically controlled. The model execution times are varied from 0.34ms through 3.4ms, 34ms, 170ms, and 340ms to 3.4 sec and are based on estimates of model sizes of AM2903, Intel 8086, Motorola 6809, and the VHDL benchmarks. First, in the experiment the entire design is simulated on a single processor. Then, the circuit is partitioned into two, four, eight, and sixteen parts and simulated with 2, 4, 8, and 16 processors. For each case, performance data is collected by varying the number of input vectors from 100 to 1000 and the model sizes from 0.34ms to 34sec.

The graphs in Figures 5a, 5b, and 5c present a logarithmic plot of the CPU time versus the input vector size for varying model sizes for the cases of 1, 4, and 8 processors. It may be observed from the graphs that the performance of the algorithm is linear. The graphs in Figure 6 present a logarithmic plot of the CPU time versus the model size for varying input vector sizes for a four processor simulation. The knees of the individual plots corresponding to the model size of 0.34ms reflect the dominance of message communication in the hypercube over model computation for model sizes smaller than 0.34ms and the dominance of the model computation over communication for model sizes larger than 0.34ms. Figure 7 presents a plot of the speedup factor versus the number of processors for three specific pairs of model and vector sizes. The graph corresponding to the model size of 0.34ms and vector size 100 resembles a saturation curve and refelcts the dominance of the message communication over model computation in the hypercube. The other two graphs are both linear indicating that the speed up factor increases linearly with increasing number of processors and, consequently, the performance of the proposed approach is linear. The maximum speedup factor for the example is observed to be 12 when the design is partitioned and simulated with 16 processors. The slope differences of the graphs also indicate that increasing CPU time is spent in model computation as opposed to communication and other overhead for increasing model sizes.

5 Conclusions

This paper has presented a distributed asynchronous approach for the simulation of behavior models on parallel processors. In this approach, a model determines based on the input signal transitions at the input ports whether it may be scheduled for computation and, consequently, scheduling is distributed in all the models. In addition, the principle of deferred scheduling ensures that inconsistent output events are detected and deleted with the consequence that correct signals are generated. The approach guarantees the absence of deadlocks and resolves blocking by temporarily forcing the execution mode of all processors to exception-mode wherein the cause of blocking is removed. The approach has been implemented on the Bell Labs hypercube and the data obtained from the simulation of designs indicate that the performance of the approach is linear.

References

[1] C. Seitz, "The Cosmic Cube," CACM, Jan 1985, pp.22-33.

[2] E. DeBenedictis, "Multiprocessor Programming with Distributed Variables," Proc. of the Conf. on Hypercube Multiprocessor, Aug 1985.

[3] D.C. Luckham, A. Stanculescu, Y. Huh, and S.Ghosh, "The Semantics of Timing Constructs in Hardware Description Languages," Proc. of the ICCD, Oct 1986, pp.10-14.

[4] S. Ghosh and M. Yu, "A Preemptive Scheduling Mechanism for Accurate Behavioral Simulation of Digital Designs," Accepted for publication in the IEEE Trans on Computers.

[5] The C++ Programming Language, B. Stroustrup, Addison Wesley 1986.

[6] M.M. Denneau, "The Yorktown Simulation Engine," Proc. of the 19th ACM/IEEE DA Conference, 1983, pp.55-59.

[7] J.K. Howard, etal, "Introduction to the IBM Los Gatos Logic Simulation Engine," Proc of the ICCD, Oct 1983, pp.580-583.

[8] The ZYCAD Logic Evaluator: Product Description, ZYCAD Corporation, N. Roseville, Minnesota, 1983.

[9] J. Misra, "Distributed Discrete-Event Simulation," Computing Surveys, Vol 18, No 1, March 1986, pp.39-65.

[10] K.M. Chandy, etal, "Distributed Deadlock Detection," ACM Transactions on Computer Systems, Vol 1, No 2, May 1983, pp.144-156.

[11] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," CACM, Vol 21, No 7, 1978, pp. 559-565.

[12] J.K. Peacock, etal, "Distributed Simulation Using a Network of Processors," Computer Networks, Vol 3, No 1, 1979, pp.44-56.

[13] Daisy Megalogician: Product Description, Daisy Systems, Mountain View, 1984.

[14] M.E. Glazier, etal, "ULTIMATE: A Hardware Logic Simulation Engine," Proc. of the 20th ACM/IEEE DA Conference, 1984, pp. 336-342.

[15] M. Bloom, "Behavior Models Take the Pain Out of System Simulation," Computer Design, 15 February 1987, pp. 38-46.

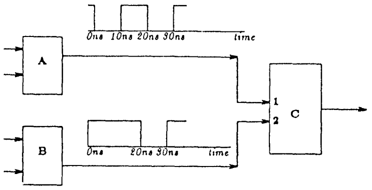


Figure 1: AN EXAMPLE DESIGN.

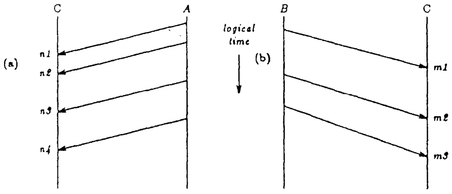


Figure 2: LOGICAL ORDERING OF EVENTS IN AN ASYNCHRONOUS DISTRIBUTED SIMULATION.

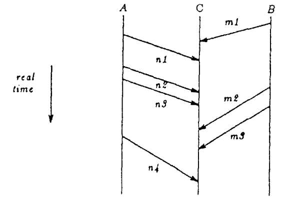


Figure 3: REAL TIME ORDERING OF EVENTS IN AN ASYNCHRONOUS DISTRIBUTED SIMULATION

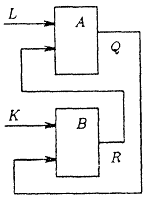


Figure 4: SIMULATION OF A CIRCUIT WITH A FEEDBACK LOOP.

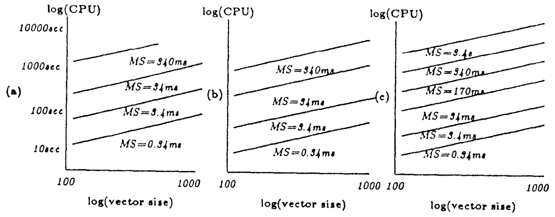


Figure 5: GRAPHS OF CPU TIME VS. VECTOR SIZE.
(a) Uniprocessor (b) Four processors (c) Eight processors

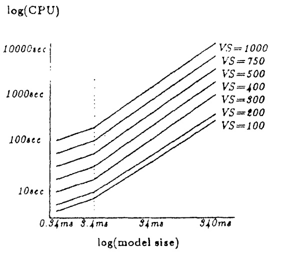


Figure 6: GRAPHS OF CPU TIME VS. MODEL SIZE FOR FOUR PROCESSORS.

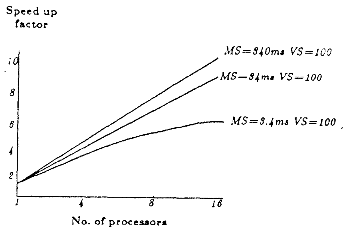


Figure 7: GRAPHS OF SPEED UP VS. NUMBER OF PROCESSORS.

[1] Acknowledgements: The authors wish to express their sincere appreciation to Erik Debenedictis and Boris Lubachevsky of AT&T Bell Labs

Operational Analysis on Hyper-Rectangulars

Teemu Kerola
Univ. of Helsinki
Teollisuuskatu 23
SF-00510 Helsinki, Finland

Alfred Hartmann
MCC
P.O. Box 200195
Austin, Texas 78720-0195

Hyper-rectangulars are a generalization of m-*ary* d-*cube networks (arbitrary radix hypercubes), where the width of the network can be different in each dimension. This gives them configuration flexibility advantages over their single radix constrained subset. Hyper-rectangulars are studied in four classes of configurations, one with all nodes in any one dimensional line in the graph connected to the same channel (bus), and the others with adjacent nodes connected with dedicated links. The dedicated links may be unidirectional or bidirectional, and the nodes can be connected linearly or as a toroid. Given a uniform message rate from each node and uniform targeting to each node, simple formulae are derived for message traffic in all cases. Simplicity of the formulae for most cases does not suffer from the generalization to hyper-rectangular topology, and the results are more broadly applicable. Beyond the operational analysis, stochastic assumptions about the message rates are used to compute overall message latencies and queue lengths within the system. The results have been verified by simulation.*

1. Introduction

Interconnection networks are important in the design of computer and communication systems and have been studied in great detail over many decades. Prior to the rise of computers during the last few decades, most of these studies centered on the telecommunications domain, with its large conglomeration of terminal equipment and intermediate switching stations. Thus much of the early work in this area focused on networks comprised of two node types—*terminal nodes* and *intermediate nodes.* Messages originated at a terminal node (the *source node*) and were routed to another terminal node (the *destination node*) via the intermediate nodes. With the rise of computers, and particularly with the contemporary focus on parallel computing, more attention has been given to networks that simply interconnect computing nodes without making any distinction between terminal nodes and intermediate nodes.

A computer *interconnection network* has an abstract representation as a graph whose nodes are switching points and whose arcs are communication links. Messages originate at a *source node* and pass along one or more links to the *destination node.* If more than one link is employed along a path between the source and destination nodes, then intermediate nodes perform routing functions for the messages along the way. If every node in the network can both originate and absorb messages, as well as serve as intermediate nodes, then we say the network is *static*, whereas if there are some nodes that may only serve as intermediaries (i.e. for routing) then we say the network is *dynamic* [9].

In the following we discuss a general class of static networks which we call *hyper-rectangulars*; these are a direct generalization of the more common hypercube networks.

Consider a connected graph of m^d nodes with the following properties: (i) each node is designated by a d-digit radix-m number, and (ii) there exists an arc between any two nodes whose numbers differ by one in exactly one digit position, and which are equal in all other digit positions. With these properties a network is called an m-ary d-cube, or *hypercube*. If $m=2$ then it is an example of a binary hypercube, of which there are a number of commercial examples [2]. If $d=2$ or $d=3$ then it is typically called a two- or a three-dimensional mesh structure.

Hypercubes are interesting because of the simple routings which are possible (see below) and because of the range of simpler networks which can be topologically mapped onto them [8]. For a given number of nodes, $M=m^d$, the optimal value for d, the hypercube *dimensionality,* is a matter of debate [3]. At least one author has proposed that lower dimensionality hypercubes, say $d \leq 5$, are preferable to higher dimension hypercubes such as the binary hypercube. While we take no position on the matter here, we note that it may be desirable to have some flexibility in the choice of M, the number of network nodes, regardless of the value of d chosen. For example if we choose $d=4$ and $M=256$, then $m=4$ and the very next larger value of m, $m=5$, would multiply the size, and presumably the cost, of the system by over 244%, to 625 nodes. It may also be desireable, perhaps for physical packaging reasons, to add nodes to a system on only one or a few dimensions. For these reasons we broaden our discourse from hypercubic to hyper-rectangular structures.

If we generalize our d-digit node designators to a mixed radix number system, where the node positions in the ith dimension, $0 \leq i \leq d-1$, are in the range 0, ..., m_i-1, so that all dimensions are not necessarily the same width, then the resulting network is a hyper-rectangular network. (Our generalization differs from [1].)

Routing

There are several obvious routing algorithms for m-ary d-cubes, and most of them also apply for hyper-rectangulars. We use the standard left-to-right routing algorithm, which solves the routing problem one dimension at a time, starting from the lowest declared dimension.

One could also turn this order around and do right-to-left routing, or any fixed permutation of the dimension orders could be chosen as long as it is consistent across all nodes. Even random selection of a dimension is acceptable, the important factor being consistent routing by all nodes so that the routing is uniform across the network. If non-uniform routing is employed then it is possible for expected channel message rates to be affected by non-uniformities in the routing strategy the analysis which follows may not apply.

Notation

$$d \overset{\text{def}}{=} \text{number of dimensions, numbered } 0, 1, \ldots, d-1$$

$$m_i \overset{\text{def}}{=} \text{number of nodes in dimension } i\text{, numbered } 0, 1, \ldots, m_i-1$$

$$M \overset{\text{def}}{=} \prod_{i=0}^{d-1} m_i = \text{total number of nodes.}$$

A standard m-ary d-cube is a special case when $m_i = m \ \forall i$. Nodes are labeled according to their d-dimensional position in the structure:

$$\vec{n} \overset{\text{def}}{=} (n_0, n_1, \ldots, n_{d-1}), \quad 0 \leq n_i \leq m_i-1.$$

An index n for node $\vec{n}$ is

$$n = n(\vec{n}) \overset{\text{def}}{=} \sum_{k=0}^{d-1} n_k * \prod_{i=k+1}^{d-1} m_i, \quad 0 \leq n < M$$

and

$$\lambda_{\vec{n}} \overset{\text{def}}{=} \text{message rate originating out from node } \vec{n}.$$

2. Assumptions

The throughput analysis below is based on the following assumptions. Later on we will introduce additional assumptions that are required for computation of average queue lengths and response times.

Assumption A: Each node sends messages to the network at the same average rate, λ:

$$\lambda_{\vec{n}} = \lambda \qquad \forall \vec{n}$$

Assumption B: For every message, the target node is selected from the uniform distribution over *all* nodes:

$$Prob \{ \text{ node } \vec{n} \text{ is target } \} = \frac{1}{M}$$

Assumption C: Message routing is done one dimension at a time, in any order, as long as all nodes use the same method, as discussed earlier. For example, the standard left-to-right routing algorithm can be used.

Assumption D: The system is assumed to be in a steady state. This implies that *flow balance* [4] applies at every server (node or link), i.e., the message rate into a server equals the message rate out of it. It also implies that the network is not saturated.

3. End-to-end Channels

In this section we assume that all nodes on any one dimensional line are connected to the same channel. Technically the channel could be implemented as a bus or an ethernet, for example. For any dimension i, there are now

$$m_0 * m_1 *...* m_{i-1} * m_{i+1} *...* m_{d-1} = \frac{M}{m_i}$$

channels. The channels in dimension i are named

$$c_{ik}\ , \quad 0 \leq k < \frac{M}{m_i},$$

in the order of increasing smallest node index in the channel. The set of nodes on the channel c_{ik} are denoted with the same symbol, c_{ik}; it will be clear from context whether c_{ik} denotes a channel or the nodes on it. We denote

$$r_{ik} = \text{avg. message rate (over time) through } c_{ik}\ , \forall k \ \ 0 \leq k < \frac{M}{m_i}.$$

Consider how the channels in dimension i are used. There are M nodes in the whole system. Each node has message origination rate λ, and so

$$\lambda^{total} = M\lambda = \text{total message origination rate.}$$

It is easy to show that if the hyper-rectangular inter-connection network is implemented with end-to-end channels in each dimension and the assumptions A, B, and C apply, then, given any channel c_{ik}, in dimension i, the message rate on it is

$$r_{ik} = (m_i - 1)\ \lambda \qquad \forall i,k \ \ 0 \leq i < d\ , 0 \leq k < \frac{M}{m_i}.$$

Every channel in any dimension i has the *same* traffic density, and the average rate (over time) through any channel in any dimension i is $(m_i - 1)\lambda$.

Node Traffic

Let $\vec{n}$ be any node in the system. Consider the message rate into $\vec{n}$ from an arbitrary adjacent channel c_{ik}. The probability of a message in c_{ik} arriving from some node other than $\vec{n}$ is $(m_i - 1)/m_i$, and the probability of such a message being targeted to $\vec{n}$ is $1/(m_i - 1)$. So the overall message rate from c_{ik} to $\vec{n}$ is

$$\frac{m_i - 1}{m_i}\frac{1}{m_i - 1} r_{ik} = \frac{1}{m_i} r_{ik}.$$

The total message rate into $\vec{n}$, and because of Assumption D, the total message rate through $\vec{n}$ is thus

$$r_{\vec{n}} = \sum_{i=0}^{d-1}\frac{1}{m_i} r_{ik} = \sum_{i=0}^{d-1}\frac{m_i - 1}{m_i}\lambda = d\left(1 - \frac{1}{\tilde{m}}\right)\lambda < d\ \lambda,$$

where $\tilde{m}$ is the harmonic mean,

$$\tilde{m} = \frac{d}{\frac{1}{m_0} + \cdots + \frac{1}{m_{d-1}}}.$$

In an end-to-end network, the message rate on each channel is relatively large, $(m_i - 1)\lambda$, but the node traffic rate is dependent only on the number of dimensions in the network. For example, for m-ary d-cubes the channel traffic is $O(m)$, whereas the node traffic is $O(d)$.

4. Point-to-Point Channels

Consider now a point-to-point inter-connection network, where adjacent nodes (node addresses differing in one dimension by one) are connected via some type of dedicated link. Adjacent links in the same direction are thought to compose an end-to-end channel, which is denoted as c_{ik}, just as in the earlier case. There are m_i nodes on that channel,

$$\vec{n}_{ikj} \qquad \forall j \ \ 0 \leq j \leq m_i - 1,$$

and $m_i - 1$ links,

$$c_{ikj} \qquad \forall j \ \ 1 \leq j \leq m_i - 1,$$

between them. The nodes and links are numbered as shown in Figure 1.

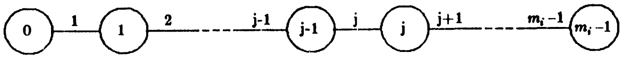


Figure 1: Node and link indexes across an end-to-end channel

Consider any end-to-end channel, c_{ik}, in a hyper-rectangular network. If Assumptions A, B, and C apply, then all possible paths through c_{ik} are equally likely, each with probability

$$\frac{1}{m_i(m_i - 1)}.$$

If c_{ik} is any channel in dimension i in a hyper-rectangular network, and c_{ikj} is any link on it, $1 \leq j \leq m_i - 1$, and if Assumptions A, B, and C apply, then

$$P_{j\,|\,ik} \overset{\text{def}}{=} P\{c_{ikj} \text{ used } \mid c_{ik} \text{ used}\} = \frac{2j(m_i - j)}{m_i(m_i - 1)}, \qquad 1 \leq j \leq m_i - 1.$$

Now, if a hyper-rectangular inter-connection network is implemented with point-to-point links, and the assumptions A, B, and C apply, then the message rate on each link, c_{ikj}, is

$$r_{ikj} = 2j\left(1 - \frac{j}{m_i}\right)\lambda, \ \forall i,k,j \ \ 0 \leq i < d\ , \ \ 0 \leq k < \frac{M}{m_i}, \ \ 1 \leq j < m_i\ .$$

The above equation proves that, for example, the message rate on any link in the dimension i depends only on its distance from the dimension i edge in the rectangular and not on distances from other edges of the network.

The message rate for any dimension i link is largest in the middle of a channel, i.e., when the distance from the dimension i edge is the largest. If the busiest link onto dimension i is denoted as $c_{ikj}^{\max}$, then the corresponding message rate on it is

$$r_{ikj}^{\max} = \begin{cases} r_{i,k,(m_i)/2} = \dfrac{m_i}{2}\lambda, & \text{if } m_i \text{ is even} \\ r_{i,k,(m_i \pm 1)/2} = \left(\dfrac{m_i}{2} - \dfrac{1}{2m_i}\right)\lambda, & \text{if } m_i \text{ is odd.} \end{cases}$$

Node Traffic

Let $\vec{n}$ be any node. The traffic into $\vec{n}$, $r_{\vec{n}}^{\text{in}}$, consists of two parts: traffic destined for $\vec{n}$, $r_{\vec{n}}^{\text{dest}}$, and traffic passing through $\vec{n}$ on its way elsewhere, $r_{\vec{n}}^{\text{thru}}$, or

$$r_{\vec{n}}^{\text{in}} = r_{\vec{n}}^{\text{dest}} + r_{\vec{n}}^{\text{thru}}.$$

Similarly the traffic out of $\vec{n}$, $r_{\vec{n}}^{\text{out}}$, consists of traffic originating at $\vec{n}$, $r_{\vec{n}}^{\text{orig}}$, plus the same through traffic, $r_{\vec{n}}^{\text{thru}}$, i.e.

$$r_{\vec{n}}^{\text{out}} = r_{\vec{n}}^{\text{orig}} + r_{\vec{n}}^{\text{thru}}.$$

In steady state operation (Assumption D) the rates must have the obvious balance:

$$r_{\vec{n}} \overset{\text{def}}{=} r_{\vec{n}}^{\text{in}} = r_{\vec{n}}^{\text{out}}; \quad r_{\vec{n}}^{\text{dest}} = r_{\vec{n}}^{\text{orig}}.$$

We call $r_{\vec{n}}$ the message rate at node $\vec{n}$.

It can be shown that if the hyper-rectangular interconnection network is implemented with point-to-point (non-toroidal) links, and if Assumptions A, B, C, and D apply, then the nodal message rate, $r_{\vec{n}}$, for any node $\vec{n}$ is bounded by

$$\lambda \sum_{i=0}^{d-1} \frac{m_i - 1}{m_i} = \lambda d \left(1 - \frac{1}{\tilde{m}} \right) \leq r_{\vec{n}} < \frac{\lambda}{2} \sum_{i=0}^{d-1} m_i = \frac{\lambda d}{2} \overline{m},$$

where $\tilde{m}$ is the previously defined harmonic mean and $\overline{m}$ is the arithmetic mean,

$$\overline{m} = \frac{1}{d} \sum_{i=0}^{d-1} m_i .$$

For low-dimension hyper-rectangulars this means $r_{\vec{n}}$ can vary between approximately $d\lambda$ and $\frac{d\overline{m}}{2}\lambda$, a very broad range. This makes homogeneous nodes wasteful due to the tapering loads near the perimeter links. This can be corrected by going to toroidal point-to-point constructions which we consider in a moment.

For binary hypercubes, where $m = 2$ and $d = \log_2 M$, the expected value for $r_{\vec{n}}$ is exactly $\lambda d/2$ and all nodes are equally loaded, since a bidirectional end-to-end channel with only two nodes on it is equivalent to a circular (toroidal) channel organization.

If we replace each bi-directional link, c_{ikj}, in the hyper-rectangular with two uni-directional links, $c_{ik\vec{j}}$ and $c_{ik\overleftarrow{j}}$, then the link message rates are halved,

$$r_{ik\vec{j}} = r_{ik\overleftarrow{j}} = j \left(1 - \frac{j}{m_i} \right) \lambda, \ \forall i, k, j \ \ 0 \leq i < d, \ 0 \leq k < \frac{M}{m_i}, \ 1 \leq j < m_i$$

because the message rate on any link is the same in each direction. However, the node message rates remain the same,

$$r_{\vec{n}} = \sum_{i=0}^{d-1} r_{\vec{n},i} .$$

5. Toroids

Suppose now, that, for each dimension i, for each node row in that dimension, there is an additional link from the last node $(m_i - 1)$ to the first node (0) in that row. Further assume that each link is now *uni-directional*, with messages routed only in the index order $0 \to 1 \to \cdots \to m_i - 1 \to 0$. Such circular structures are generally called toroids, and we call the ones described here as uni-directional circular hyper-rectangulars. All the assumptions stated before (Assumptions A, B, C, and D) still apply.

We now derive the message throughput on individual links. Select any dimension i channel c_{ik}, and consider the message rate through the links, c_{ikj}, on it. The total message rate through all of the dimension i channels is

$$r_i^{total} = \frac{m_i - 1}{m_i} M \lambda.$$

Because of the homogeneous structure of the toroid, all channels on dimension i are equally busy, and there are M/m_i channels for dimension i. Thus, the total message rate on c_{ik}, r_{ik}, is

$$r_{ik} = \frac{r_i^{total}}{M/m_i} \lambda = (m_i - 1) \lambda.$$

Let j be any link on c_{ik}. There are $m_i(m_i - 1)$ different routes through c_{ik}, and

$$m_i - 1 + m_i - 2 + \cdots + 2 + 1$$

of them go through j. So, the probability that a message using c_{ik} goes through c_{ikj}, $P_{j \mid ik}$, is

$$P_{j \mid ik} = \frac{\sum_{j=1}^{m_i - 1} j}{m_i (m_i - 1)} = \frac{1}{2}.$$

Now, the message rate on the given link c_{ikj} is

$$r_{ikj} = r_{ik} * P_{j \mid ik} = \frac{m_i - 1}{2} \lambda.$$

Let $\vec{n}$ be any node on any channel c_{ik} in dimension i, and let

$$j_i \overset{def}{=} \text{dimension } i \text{ link leading to } \vec{n}, \ \forall i \ \ 0 \leq i < d.$$

The message rate through $\vec{n}$ is the sum of the message rates on all links leading to $\vec{n}$:

$$r_{\vec{n}} = \sum_{i=0}^{d-1} r_{ikj_i} = \frac{\lambda}{2} \sum_{i=0}^{d-1} (m_i - 1) = \frac{\lambda d}{2} (\overline{m} - 1).$$

Bi-Directional Toroids

Another possibility is to use bi-directional links to connect the nodes in the hyper-rectangular toroid. Assumption C requires a balanced routing algorithm. The deductions below are made based on the assumption that, if two paths of equal lengths exist from node $\vec{n}_1$ to node $\vec{n}_2$, then either one of them is selected with probability 1/2. If some other balanced method is selected, similar deductions can still be made.

Let j be any link, between nodes $\vec{n}_1$ and $\vec{n}_2$, on any channel c_{ik}. The message rate through it is now

$$r_{ikj} = \begin{cases} (m_i - 1)\lambda & \text{if } m_i = 2, \\ \dfrac{m_i}{4} \lambda & \text{if } m_i \text{ even, } m_i > 2, \\ \dfrac{m_i^2 - 1}{4 m_i} \lambda & \text{if } m_i \text{ odd, } m_i > 2. \end{cases}$$

The message rate through any node $\vec{n}$ is again half of the message rates of adjacent links,

$$r_{\vec{n}} = \sum_{\substack{c_{ikj} \text{ adjacent} \\ \text{to } \vec{n}}} \frac{r_{ikj}}{2} = \sum_{i=0}^{d-1} \left[r_{ikj_i} * (1 - 0.5 * Id^{m_i = 2}) \right],$$

where j_i is an index for a dimension i link adjacent to $\vec{n}$, and

$$Id^X = \begin{cases} 1 & \text{if } X \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

6. Queue Lengths and Response Times

The total hyper-rectangular network has an arrival rate of $M\lambda$. We have already derived the overall message rates on individual nodes and links. To obtain the queue lengths and response times we need more information of the system. We need the processing speed for every device (node, bus, or link) in the system. Also for the analytic solution to be tractable, i.e., for the network to be *separable* [7], we need two additional assumptions:

Assumption E: The system can be defined as a sequence of events that occur at distinct times.

Assumption F: The completion rate from a server does not depend on the load at other servers.

One additional assumption is needed for a simple solution to be available:

Assumption G: The completion rate from a *busy* server must not depend on the queue length for that server.

We also need some new notation. Let

$$S_i \overset{def}{=} \text{average service time at device } i \text{ per message (sec/msg).}$$

For further analysis, we transform the message rates into *visit ratios*, which define the average number of times that any message routed through the system passes through a device. Given the actual message rate, r_i, through a device (here a node, bus, or a link), and the total message rate arriving to (originating at) the system, $M\lambda$, the corresponding visit ratio at device (server) i is

$$V_i = \frac{r_i}{M\lambda} \quad \text{(visits/msg).}$$

We can now use well known operational analysis theory for open queueing networks ([7]) and compute the queue lengths and response times for every device i in the system.

The average work *demand* per message for device i is

$$D_i = V_i S_i \quad \text{(sec/msg),}$$

and the processing capacity of the system is determined by the device with the largest demand,

$$D_{\max} = \max_i D_i .$$

The maximum system throughput, i.e., the *network capacity,* is

$$C = \frac{1}{D_{\max}},$$

and thus, we must have

$$M\lambda \le \frac{1}{D_{\max}}, \quad \text{i.e.,} \quad \lambda \le \frac{1}{MD_{\max}},$$

to avoid saturating the network. If $\lambda > 1/MD_{\max}$, Assumption D is violated, the system becomes saturated, and queue lengths and response times "explode" to infinity.

In general, to compute the total average system response time, we need to consider all nodes, busses, and/or links. However, node delays can often be ignored in practice, because they are often included in the link service times.

If two uni-directional links replaced each bi-directional link, and the maximum message rate for the uni-directional link were half of that for bi-directional links, then the maximum visit ratio would be half of that before, but the average demand would be the same. Thus, all other performance measures given above would be the same as they were for the network with bi-directional links.

One-Dimensional End-to-End Channels

For hyper-rectangulars with end-to-end channels, the response times and queue lengths can be expressed in a more condensed form because

$$V_{c_{ik}} = \frac{m_i - 1}{M} \text{ for all } \frac{M}{m_i} \text{ channels } c_{ik} \text{ in dimension } i, \ \forall i \ 0 \le i < d.$$

The average total channel residence time (time spent in all channels per average message) is

$$R^{links} = = \sum_{i=0}^{d-1} \frac{1}{m_i \left[\frac{1}{(m_i - 1)S^{link}} - \lambda \right]}$$

Similarly, for each node, $\vec{n}$,

$$V_{\vec{n}} = \frac{1}{M} \sum_{i=0}^{d-1} \left(1 - \frac{1}{m_i} \right) = \frac{d}{M} \left(1 - \frac{1}{\tilde{m}} \right)$$

and the average total node residence time is

$$R^{nodes} = \sum_{\vec{n}} R_{\vec{n}} = MR_{\vec{n}} = \frac{S^{node} d \left(1 - \frac{1}{\tilde{m}} \right)}{1 - \lambda S^{node} d \left(1 - \frac{1}{\tilde{m}} \right)}.$$

The average total message latency is $R^{links} + R^{nodes}$.

Uni-Directional Toroids

For uni-directional toroids and for any link c_{ikj} on any channel c_{ik} in dimension i,

$$V_{c_{ikj}} = \frac{m_i - 1}{2M}, \ and \ R^{links} = \sum_{i=0}^{d-1} \frac{1}{\frac{2}{(m_i - 1)S^{link}} - \lambda}.$$

For each node $\vec{n}$,

$$V_{\vec{n}} = \frac{1}{2M} \sum_{i=0}^{d-1} (m_i - 1) = \frac{d}{2M} (\overline{m} - 1),$$

and

$$R^{nodes} = \frac{1}{\frac{1}{S^{node} d (\overline{m} - 1)} - \lambda},$$

The average total response time is $R^{links} + R^{nodes}$. Ignoring R^{nodes} for the moment and taking the network response time to be R^{links} we can get a simplified form for the case of m-ary d-cubes (arbitrary radix hypercubes):

$$R^{links}_{hypercube} = \frac{\frac{m-1}{2} d}{\frac{1}{S^{link}} - \frac{m-1}{2} \lambda}.$$

We can recognize the numerator as $\overline{h}$, the average number of link traversals (hops) for a message in the network [3], the first term in the denominator as C^{link}, the single link capacity in messages per second, and the second term in the denominator as r^{link}, the single link traffic rate (r_{ikj} is the same everywhere in a uni-directional toroidal hypercube under our assumptions). Thus

$$R^{links}_{hypercube} = \frac{\overline{h}}{C^{link} - r^{link}},$$

indicating that the network response time (exclusive of node service) in a hypercube is just the average message distance divided by the idle link capacity (in messages per second). This result is reported by earlier authors [5, p.272].

Note that the expression for the average number of hops (link traversals) per message, $\overline{h}$ used above, is easily derivable as

$$\overline{h} \overset{def}{=} \text{E[link traversals]} = \sum_{i,k,j} V_{c_{ikj}} = \frac{d(\overline{m} - 1)}{2}.$$

Note also that we may be justified in ignoring R^{nodes} in our analysis if the node is implemented so that traffic on all d dimensions is handled in parallel within the node. The above expression for $R_{\vec{n}}$ assumes that a node acts as a single server device. If, in fact, all of a node's ports to the network operate in parallel, then the node service time, S^{node} can just be treated as part of the link service time, S^{link}, and R^{nodes} then becomes zero.

Bi-Directional Toroids

For bi-directional toroids and for any link c_{ikj} on any channel c_{ik} in dimension i,

$$V_{c_{ikj}} = \begin{cases} \frac{m_i -1}{M} & \text{if } m_i = 2, \\ \frac{m_i}{4M} & \text{if } m_i \text{ even, } m_i > 2, \\ \frac{{m_i}^2 -1}{4Mm_i} & \text{if } m_i \text{ odd, } m_i > 2, \end{cases}$$

and

$$V_{\vec{n}} = \sum_{i=0}^{d-1} \left[V_{c_{ikj}} * (1 - 0.5 * Id^{m_i=2}) \right],$$

where c_{ikj} is any link on any channel c_{ik} in dimension i. Overall message latencies and queue lengths are then computed with the standard formulae.

7. Conclusions

We have derived simple formulae for channel and link throughput in end-to-end and point-to-point networks under generally applicable assumptions. The analytical link and node throughputs are summarized in Table 1.

Queueing delays within the network will slow down any individual message, but they do not affect the message rates. Queueing behavior cannot be anticipated with operational analysis alone; stochastic assumptions are needed. For separable open networks we have relatively simple closed form solutions for queue lengths and response times in the system.

An important part of the analysis was the decision to include the sending node as a possible target node, so that every node was equally likely to receive *every* message. This simplified the analysis in many places. If one rules out messages to the sending node, and denotes the actual message rate out from each node as β, then all the formulae given earlier apply for

$$\lambda = \frac{M+1}{M} \beta.$$

As a special case, we can use the formulae described earlier to derive link and node throughput for m-ary d-cubes and hypercubes. The derived formulae are given in Tables 2 and 3. Also, the response times and device queue lengths reduce to simpler forms for these special cases. For example, when $m_i \equiv 2$, the link queue lengths and link residence times become

$$Q_{c_{ik}} = \frac{S^{link}\lambda}{1 - S^{link}\lambda} \quad \text{and} \quad R^{links} = \frac{d}{2\left(\frac{1}{S^{link}} - \lambda\right)}.$$

Remark: This paper is a shortened version of a technical report [6], which contains proofs for all the results presented here in addition to examples, supporting simulation results, and additional discussion.

Acknowledgements: Herb Schwetman gave constructive criticism and we also thank Bill Alexander and Bonnie Kerola for their helpful comments.

List of References

[1] Bhuyan, L.M., D.P. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Network", *IEEE Trans. on Computers,* C-33, 4 (April 1984), pp. 323-333.

[2] Bond, J., "Parallel-Processing Concepts Finally Come Together in Real Systems", *Computer Design,* June 1, 1987, pp. 51-74.

[3] Dally, W.J., "Wire-Efficient VLSI Multiprocessor Communication Networks", Proc. 1987 Stanford Conf. on Advanced Research in VLSI, pp. 391-415, 1987.

[4] Denning, P.J., J.P. Buzen, "The Operational Analysis of Queueing Network Models", *ACM Computing Surveys,* 10, 3 (September 1978), pp. 225-261.

[5] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," Computer Networks, 3 (1979), pp. 267-286.

[6] T. Kerola and A. Hartmann, "Operational Analysis on Hyper-Rectangulars," MCC Technical Report PP-199-87, 22 p.

[7] Lazowska, E.D., J. Zahorjan, G.S. Graham, K.C. Sevcik, *Quantitative System Performance,* Prentice-Hall, New Jersey, 1984.

[8] Wu, A.Y., "Embedding of Tree Networks into Hypercubes", *J. of Parallel and Distr. Comp.,* 2 (1985), pp. 238-249.

[9] Wu, C.-l., T.-y. Feng, "Chapter 1: Introduction", *Tutorial: Interconnection Networks for Parallel and Distributed Processing,* pp. 1-3. IEEE Computer Society Press, 1984.

Table 1: Hyper-Rectangular Throughput

Message rate out from each node: λ m_0–m_1– $\cdots$ –m_{d-1} Hyper-Rectangular j is link index from edge		
Connection Topology	Link Rate	Node Rate
one-dimensional bus	$(m_i - 1)\lambda$	$\left(1 - \frac{1}{\tilde{m}}\right) d\lambda$
node-to-node	$2j\left(1 - \frac{j}{m_i}\right)\lambda$	$\frac{1}{2} \sum_{\substack{c_{ikj} \\ \text{adjacent} \\ \text{to } \vec{n}}} r_{ikj}$
uni-dir. toroid	$\frac{m_i - 1}{2}\lambda$	$\frac{\overline{m} - 1}{2} d\lambda$
bi-dir. toroid	$\frac{m_i}{4}\lambda$	$\frac{\overline{m}}{4} d\lambda$

Table 2: Throughput in m-ary d-cubes

Message rate out from each node: λ $m_i \equiv m$ $M = m^d$		
Connection Topology	Link Rate	Node Rate
one-dimensional bus	$(m-1)\lambda$	$\left(1 - \frac{1}{m}\right) d\lambda$
node-to-node	$\frac{m}{2}\lambda$	$\leq d\frac{m}{2}\lambda$
uni-dir. toroid	$\frac{m-1}{2}\lambda$	$d\frac{m-1}{2}\lambda$
bi-dir. toroid	$\frac{m}{4}\lambda$	$d\frac{m}{4}\lambda$

Table 3: Throughput in binary hypercubes

Message rate out from each node: λ $m_i \equiv 2$ $M = 2^d$		
Connection Topology	Link Rate	Node Rate
one-dimensional bus	λ	$\frac{d}{2}\lambda$
node-to-node	λ	$\frac{d}{2}\lambda$
uni-dir. toroid	$\frac{\lambda}{2}$	$\frac{d}{4}\lambda$
bi-dir. toroid	λ	$\frac{d}{2}\lambda$

DISTRIBUTED TERMINATION ON A MESH

Jianjian Song and Larry Kinney

Department of Electrical Engineering
University of Minnesota
Minneapolis, MN 55455

Abstract

A method for distributed termination detection is proposed that naturally fits the structure of an array of mesh-connected processing elements. Two sufficient conditions are given which guarantee that any one of the processing elements may detect the termination of computation on the mesh. The method is fully distributed, symmetric, asynchronous, and efficient in that it combines termination detection with the computation process and it does not require any global information transmission until termination of computations has been detected. The method was originated for use with parallel processing for finite element analysis on a mesh of processing elements, but it is applicable to any asynchronous iterative computations on the mesh. The method can also be used for termination detection when execution of successive tasks are overlapped on the mesh.

1. Introduction

Computation of the finite element analysis can be distributed on an array of processing elements (PEs). The PEs are usually connected as a mesh since a mesh is a good match to the grid patterns used in the finite element analysis. The computation can be carried out by either direct or iterative solution techniques. Iterative solutions have been found more suitable for utilizing the power of parallel processing [1] [2]. Iterative methods can be either synchronous or asynchronous. Asynchronous iterations have been attracting more attention [3] [4]. Being asynchronous, the computation has all the attributes of other distributed computations: a PE is either active in doing its computation or passive when is is done with its computation; passive PEs may be activated again by messages from active PEs; and the pattern of message transmission can not be decided a priori. One of the challenges in using iterative solutions is to determine when the computation is completed. The solution to this problem can be a centralized or distributed one. An ideal solution should have the following properties:

(1) It does not interfere with the computation process (transparency).
(2) It does not require dedicated communication channels.
(3) It does not use a predesignated processor (host, root etc.) that observes the states of all the PEs, i.e., the solution should be fully distributed and symmetric.
(4) Message transmission may be delayed in communication channels, i.e., the transmission is not instantaneous.

In [1], [2] and [4] termination detection was solved by appealing to a global synchronization mechanism. When a PE is finished with its current computation, it will report its state to a predesignated PE (called the host or root). The host collects the states of all the PEs and decides if the computation is terminated. Global termination detection is hard to implement in case of asynchronous iteration, since a passive PE may be activated again by a message from other PEs and this change in PE status must be made known to the host. The host may never know the real status of the computation due to communication delays unless message transmission is assumed instantaneous. Global synchronization may also take much extra time since global communication is usually slower than local communication; hence, PEs must wait for synchronization.

Techniques for distributed termination must be used when global synchronization is not a good choice. The problem of distributed termination has been discussed in the literature [5] [6] [7] [8] [9]. The previous approaches have the following characteristics.

(1) A dedicated communication network, CN, is assumed for the purpose of termination detection (tree in [5] [9]; ring in [6] [7]).
(2) Termination detection is a continuous, trial and error process. A detecting probe (tokens, control message) or detecting wave is initiated and circulated periodically in CN until the probe has detected the system termination.
(3) All algorithms, except the one in [7], use a predesignated processor to detect tęrmination. While the algorithm in [7] is distributed and symmetric in the sense that any processor may detect termination, it uses a common clock which is not desirable in practice.

There are drawbacks in these approaches. First, CN can not be used for transmitting a computation message (data message). Otherwise, the speed of termination detection and computation would both be reduced. There must be another network for computation messages; thus, CN adds hardware and complexity to a parallel computer system. The second problem with these methods is the large number of messages travelling in CN. Some messages that will eventually be destroyed must pass through several processors even though they have become obsolete at the beginning of the journey [7]. The problems are inherent in the ring structure and the assumption that data messages can be passed between any processor pair.

The situation is different in a mesh-connected array of processors. Communication in the mesh is through nearest-neighbor connections which are fixed and local. Only local messages between adjacent PEs exist. Taking advantage of the mesh structure, a token (or probe) passing method is proposed for distributed termination of computation on a rectangular mesh. When finished with its computation, each PE sends its termination state to its neighbor PEs. One token from each side of the rectangular mesh is initiated once. The tokens will be travelling in the mesh according to rules derived below. Their traces and positions will indicate the computation status of the mesh. One or more of the PEs will eventually be able to detect termination of the computation by examining its own record of the token arrival, its state, and its neighbor's states. The method is fully distributed and symmetric in the sense that no PE has more responsibility than the others [7].

The following paragraphs describe the proposed method. Some assumptions about the rectangular mesh and PEs as well as some definitions are given in Section 2. The case where there is no data message (values for actual computation) is considered in Sec. 3. In this case a PE will keep passive once it is in passive state, which makes it easier to detect termination. The method is improved further in Section 4 so that it works even if data messages exist. Correctness proofs, time analysis and discussions are given in Sections 5 and 6.

2. Assumptions and Definitions

The definitions of array, states, arrows, tokens, data messages and control messages are given in this section. Some assumptions about the mesh are also listed.

As shown in Fig. 1, the array consists of mesh-connected processing elements (PEs) with each PE connected to its four nearest neighbors. A PE may be in one of two states: "active" and "passive". A PE is active when it is contributing to the computation and passive when it is finished with its assigned computation. The two states are indicated by circles and black dots, respectively. Token "North passive" indicates that all PEs north of (not including) the west-east line where the token resides have been passive. The definitions of the other tokens are similar. The tokens

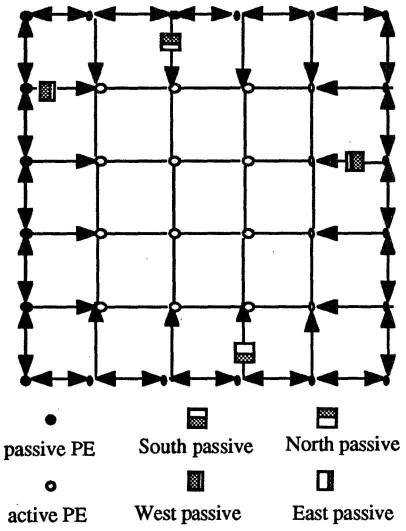


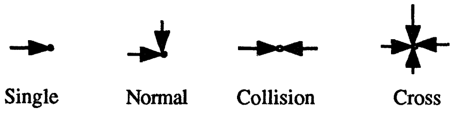
Fig. 1 Structure and notations.

are represented by black-and-white squares. They are messengers travelling in the mesh to collect global information about the state of the mesh. Totally, there are four tokens, one for each side of the mesh. The state of passiveness of a PE can be passed to other PEs. This message is represented by arrows pointing from the sender PE to the receiver PE. An arrow is a passive state messenger. Messages are grouped under data and control. Data messages are those that carry values used in computation. Control messages carry information on the state of a PE, e.g., an arrow or a token. A passive PE may be activated again by data messages from other PEs. A passive PE may communicate with other PEs by control messages to decide the status of the mesh as a whole.

Computation is completed when all PEs are passive and there is no data message in any communication channel. Detection of this state by one of the PEs is called the distributed termination problem.

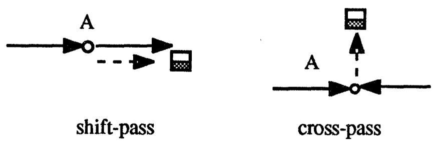
One assumption is that message transmission is instantaneous, which was an assumption made in all previous papers on distributed termination known to us. Another assumption is that there is a continuous communication process that handles messages between PEs no matter whether computation is in process or not.

To simplify the discussion, some combinations of the arrow reception patterns are named as shown below.



Single and Collision are the most important patterns.

There are two ways in which tokens may be passed. One is called shift-pass and the other cross-pass as illustrated below:



For example, consider the south passive token which tends to travel to the north in the mesh. Assume that the token is impending on PE A. Shift-pass simply shifts the token along the horizontal line through PE A. This movement is caused by the single or normal reception patterns. Cross-pass sends the token across its horizontal line; this movement is caused by the collision reception pattern that is east-west oriented.

3. A Distributed Termination Solution In the Absence of Data Messages

We describe a method for detecting, in distributed fashion, termination of computations without a data message on the mesh. The absence of data messages makes the detection problem simpler since a PE can not be activated again once it is passive. (This is removed later in Section 4 by a modification of the method). The method is based on three sets of rules: state transition rules, arrow passing rules, and token passing rules. A PE will maintain a record of the arrival of tokens and arrows from its neighbors. The principle behind the method is that a token will never cross a line of PEs if any PE on this line has never been passive. A tokens location indicates that PEs on lines passed by the token have been passive. The rules are listed below.

State Transition Rules:

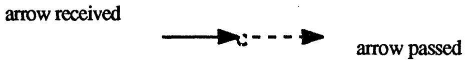
(0) Boundary PEs are always passive and ready to pass control messages (the arrows and tokens).
(1) A PE is activated by initialization or program loading.
(2) A PE becomes passive if it is finished with the assigned computation.

Arrow Passing Rules:

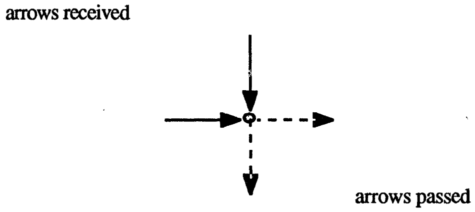
An active PE does not pass any arrow. A passive PE does not pass arrows before receiving arrows from its neighbors.

(0) A boundary PE passes its arrow to all its neighbors as soon as the computation starts.
(1) A PE passes its own arrows in the opposite directions of the arrows it has received. There may be two cases as depicted below.

Case 1: Single state



Case 2: Normal state



(2) A PE passes its arrow to both neighbors if it receives two opposite arrows (Collision) from its neighbors.

Rule (0) assures starting the termination detection process. Rule (1) makes the state of passiveness propagate along a line. And Rule (2) is to send the line passiveness information to all the PEs on the same line. This information will be needed to decide whether a token should be cross-passed.

Token Passing Rules:

There are four different tokens residing initially on the four boundaries of the mesh. The initial locations of the tokens are not important as long as they are on their corresponding boundary sides. A passive PE may deliver tokens to other PEs when it receives tokens. The following is a list of rules for token passing.

(0) Boundary PEs cross-pass their tokens as soon as the computation starts.
(1) A PE in Single or Normal states shift-passes the tokens it has received in the direction of its arrow.
(2) A PE in Collision or Cross states cross-passes the tokens it receives.

(3) A boundary PE keeps any tokens it has received.

The arrival of the tokens will be recorded by PEs that they have visited in order to detect termination of the computation. Termination may be detected by any PE. The following are two sufficient terminating conditions :

CONDITION 1 Any PE can declare that the computation is terminated when the PE has a record of the arrival of all four tokens. The tokens may or may not be with the PE at the time of the decision.

CONDITION 2 A boundary PE can declare that the computation is terminated when it receives one token since this token must be from the side opposite to the PE's side.

The correctness of the conditions can be explained using the definitions of the tokens. There are four sides on a rectangular mesh. Each token indicates that its corresponding side is passive. The whole mesh is obviously passive once all four tokens meet at one PE, i.e., all four sides are passive. The first condition will occur if there is one and only one Cross on any line of the mesh and there is no Collision. The second condition will occur when there is no Cross or more than one Cross on any lines of the mesh.

An alternative technique is to use one token and let it travel from one side to the opposite side of the mesh. This will be sufficient for termination detection according to Condition 2. Detection time may be shorter if four tokens and Condition 1 are used.

4. A Solution with the Existence of Data Messages

When data messages exist, a passive PE may be activated by data messages from another PE. One fact is that a PE that sends a data message destroys the arrow from a passive PE that will receive the message. Termination can also be decided when data messages are present if another state transition rule is added:

State Transition Rule (3):

(3) If a PE sends a data message to a passive PE, the sender may not declare itself passive until the receiver becomes passive again.

This rule guarantees that the token indicating that the receiver was passive will never be cross-passed to the next level by the sender or any PEs on the same line as the sender unless the receiver becomes passive again. Now we can use all the rules and the two sufficient conditions listed in Section 3 to detect termination when data messages exist.

5. Correctness Proof and Primitive Time Analysis

We prove that Condition 1 and 2 in Section 3 are sufficient even in the presence of data messages. Proof of the first condition reads as follows: assume that a passive PE receives four tokens while the system is not terminated. There must exist one originally active PE (i.e., it has never been passive before) according to the state transition rule (3) specified in Section 4. But then the tokens should not have crossed the two lines on which this PE resides, which implies that no PE could have received four tokens. This contradicts the assumption that the passive PE has received four tokens. Proof of Condition 2 follows the same path. After any token has crossed the mesh from one side to the opposite side, there would exist no originally active PEs, which implies that every PE has been passive and the system is terminated.

Consider a square mesh of n PEs ($\sqrt{n}$ on each side). Assuming that the system is already terminated, the worst time for the algorithm to detect the termination is O(n), which is the time for one token to travel through a ring that connects every PE. The best time is O($\sqrt{n}$) (order of square root of n) which is the time for a token to travel across the mesh or four tokens meet at the center of the mesh.The above time estimates represent the worst case situation. Since the token passing is performed in parallel with actual computations, the execution of our termination algorithm may be completely overlapped with actual computations so that termination detection could be completed as soon as computation is over. In comparison, the algorithm developed in [7] will always take O(n) time where n is the number of PEs on a ring since it is the last token (the counter in the paper) that starts the terminating wave.

6. Discussion

The proposed method discussed in this paper has all the merits that the other methods for distributed termination detection claim: asynchronousness, distributiveness, symmetry, etc. It is also more efficient, faster and better than the ring-based algorithms. It is efficient since no additional communication network is needed; messages are only passed the shortest distance possible when they are needed, and global communication is not required until the computation terminates. It is faster since messages may be travelling in parallel with each other unlike sequential message passing in the ring approach. It is better since the mesh structure is fully utilized.

The application of the method is not limited to the finite element analysis. It is useful for any distributed termination detection on an array of mesh-connected processing elements. An immediate example is to solve nonlinear partial differential equations, where iterative solutions techniques are essential [10] [11]. The method is also applicable to multi-task cases. Tokens, control and data messages may be colored to represent different tasks so that a few finite element analysis computations may be running simultaneously. Another application of the algorithm could be to the solutions of three-dimensional partial differential equations.

References

[1] R. Morison and S. Otto, " The Scattered Decomposition for Finite Elements", Technical Report C^3P 286, Caltech Concurrent Computation Group, Caltech, Pasadena, CA 91125, May 1985, pp1-22

[2] David D. Loendorf, Advanced Computer Architecture for Engineering Analysis, PhD Dissertation, The University of Michigan, 1983

[3] Gerard M. Baudet, "Asynchronous Iterative Methods for Multiprocessors", Journal of the ACM, Vol 25 No. 2, April 1978, pp226-244.

[4] Daniel A. Reed and Merrell L. Patrick, "A Model of Asynchronous Iterative Algorithms for Solving large, Sparse, Linear Systems:", Proceedings, 13th International Conference on Parallel Processing, 1984, pp402-409.

[5] Edsger W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations", Information. Processing Letters 11-1, 1980, pp1-4.

[6] Edsger W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, " Derivation of a Termination Detection Algorithm for Distributed Computations", Information Processing Letters 16, 1983, pp217-219

[7] S.P. Rana, " A Distributed Solution of the Distributed Termination Problem", Information Processing Letters 17, 1983, pp43-46

[8] Rodney W. Topor, "Termination Detection for Distributed Computations", Information Processing Letters 18, 1984, pp33-36.

[9] Ron Cytron, " Useful Parallelism in a Multiprocessing Environment", Proceedings, 14th Inter. Conference on Parallel Processing, Aug. 1985, pp450-457.

[10] John R. Rice, "Parallel Methods for Partial Differential Equations", The Characteristics of Parallel Algorithms, The MIT Press, 1987, pp209-231.

[11] Garrett Birkhoff and Robert E. Lynch, Numerical Solution of Elliptic Problems, SIAM Philadelphia 1984.

ON ENHANCING HYPERCUBE MULTIPROCESSORS[†]

Abdol-Hossein Esfahanian,[1] *Lionel M. Ni,*[1,2] *Bruce E. Sagan*[3]

[1]Department of Computer Science
Michigan State University
East Lansing, MI 48824

[2]Division of Mathematics and Computer Science
Argonne National Laboratory
Argonne, IL 60439

[3]Department of Mathematics
Michigan State University
East Lansing, MI 48824

ABSTRACT

We show that by exchanging any two *independent edges* in any shortest *cycle* of the n-cube ($n \geq 3$), its *diameter* decreases by one unit. This leads us to define a new class of *n-regular* graphs, denoted TQ_n, with 2^n vertices and diameter $n-1$, which has the $(n-1)$-cube as subgraph. Other properties of TQ_n such as *connectivity* and the lengths of the disjoints paths are also investigated. Moreover, we show that the *complete binary tree* on $2^n - 1$ vertices, which is not a subgraph of the n-cube, is a subgraph of TQ_n. Finally, we discuss how these results can be used to enhance existing hypercube multiprocessors.

1. INTRODUCTION

The possibility of interconnecting a number of processors together to solve very large problems in scientific computation has been extensively considered in the past [HwBr84]. *Distributed-memory multiprocessor systems* have proven to be one of the most straightforward and the least expensive methods to build such arrays with hundreds or even thousands of processors [Seit85]. In such networks, each processor has its own memory and message passing is the means of information exchange between processors.

It is well-known that the topology of the interconnection network plays a significant role in system performance, especially for large scale distributed-memory multiprocessors [SaSc85]. Several efforts on designing interprocessor communication networks have been reported [WuLi81]. Among various architectural configurations, the point-to-point topology has attacted a great deal of attention due to its simpler communication protocols and direct communication paths among the nodes [HwGh87]. Several features have to be considered when evaluating a point-to-point interconnection network. These features include the ability to embed other problem topologies, the ability to meet the demands of massive parallelism, the connectivity, the worst case communication delay between two nodes, the tolerance of faulty components, the communication bandwidth of each node, and the ease of routing between any two nodes.

Among point-to-point topologies, the hypercube has been a dominating topology used in the first generation of distributed-memory multiprocessors [ShFi88]. The strong connectivity of hypercube and its regularity, symmetry, and ability to embed many other topologies, have made it a powerful candidate for a wide class of applications [Foxg86]. Many other interconnection topolgies have been proposed for distributed-memory multiprocessors, such as tree [DePa78], cube-connected cycle [PrVu81], block-shuffle hypercube [HsYZ87], and hypernet [HwGh87]. These various interconnection topologies have their own advantages and disadvantages based on the above evaluation criteria. In this paper, we present the least expensive approach to enhance the hypercube interconnection scheme.

An *n−dimensional* hypercube multiprocessor consists of $N = 2^n$ processors interconnected as follows. Each processor is labeled by a different n-bit binary number $(b_{n-1}b_{n-2} \cdots b_1 b_0)$. Two processors are connected by a full duplex link if and only if their binary labels differ in exactly one bit position. The popularity of hypercube multiprocessors is due to its underlying topology which is known as the *n−cube graph* Q_n. The n-cube graph has been the subject of many research projects in recent years, mainly because of the availability of hypercube multiprocessors [SaSc85]. As a result, many properties of the n-cube have been discovered [BrSc85].

The rest of this paper is organized as follows. Our notation and terminology are given in the next section. A new interconnection topology, denoted TQ_n, which is based on a simple modification of the n-cube is given in Section 3. We will show in Sections 4 through 7 that TQ_n has certain topological advantages over Q_n. In particular, it is shown that the *diameter* of TQ_n is one less than that of Q_n, and its *vertex-connectivity* is the same as that of Q_n. It is known that the *complete binary tree* on $2^n - 1$ vertices, T_n, is not a subgraph of Q_n [SaSc85]. However, T_{n-1} is contained in Q_n [BrSc85]. We prove that TQ_n has the complete binary tree T_n as subgraph. Other subgraphs of TQ_n are also identified. Finally, practical implications of our results are given in Section 8.

2. NOTATION AND TERMINOLOGY

We will closely follow the graph theoretical terminology and notation of [Hara72]; terms not defined here can be found in that book. Let $G(V,E)$ represent a *graph* with *point* or *vertex set* $V(G) = V$ and *edge set* $E(G) = E$. If an edge $e = uv \in E$ then vertices u and v are said to be *adjacent*, the edge e is said to be *incident* to these vertices, and u and v are the *end points* of edge e. Two edges are said to be *independent* if they do not share an end point. For a vertex $v \in V$, $I(v)$ represents the set of all edges incident to v in G, and its cardinality $|I(v)|$ is the *degree* $deg(v)$ of vertex v. We denote by $\delta(G)$ and $\Delta(G)$ the *minimum* and *maximum degrees* respectively of vertices of G. If $\delta(G) = \Delta(G) = k$, then G is said to be *k−regular*. For a set $X \subset E$ (or $X \subset V$), the notation $G - X$ represents the graph obtained by removing the edges (vertices) in X from G. The *vertex-connectivity*, $\kappa(G)$, of a graph G is the least cardinality $|X|$ of a set $X \subset V(G)$ such that $G - X$ is either *disconnected* or consists of a single vertex.

The *distance* $d(u,v)$ between two distinct vertices u and v is the length (in number of edges) of a shortest path between these vertices. The *diameter* $d(G)$ of graph G is then defined to be $d(G) = \max\{d(u,v) \mid u,v \in V\}$. If H and G are graphs then H is *isomorphic* to a *subgraph* of G if there is a one-to-one function $f: V(H) \rightarrow V(G)$ such that each edge $uv \in E(H)$ is carried to an edge $f(u)f(v) \in E(G)$. By an abuse of language we will often merely say that H is a subgraph of G (where in reality it is $f(H)$ which is a subgraph of G) and will write $H \subseteq G$.

Two specific graphs with which we will be concerned are complete binary trees and n-cubes. As indicated before, T_n will represent the complete binary tree on $2^n - 1$ vertices. The *root* of T_n is the unique vertex whose degree is 2. If Q_n is the n-cube then $\delta(Q_n) = \Delta(Q_n) = n$, $d(Q_n) = n$, and $\kappa(Q_n) = n$. The binary label of a vertex $v \in V(Q_n)$ will be referred to by an n-bit binary number $b(v)$. Also, $\mathbf{O}(b(v))$ and $\mathbf{Z}(b(v))$ will denote the number of ones and zeros, respectively, in the binary number $b(v)$.

[†] This research was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38 and in part by the DARPA ACMP project.

3. THE TWISTED N-CUBE

Let C be any shortest cycle (*i.e.*, a cycle of four vertices) in Q_n. Also, let ux and vy be any two independent edges in C. The *twisted n-cube* graph TQ_n is then constructed as follows. Delete edges ux and vy from Q_n. Then, connect, via an edge, vertex u to vertex y, and vertex v to vertex x. That is, $TQ_n = Q_n - \{ux, vy\} + \{uy, vx\}$. Figure 1 shows Q_3 and TQ_3. Note that by construction, TQ_n is n-regular just as Q_n is. Also, observe that TQ_n has two disjoint Q_{n-1} as subgraphs.

Although the cube can be twisted around any 4-cycle, we will usually use the *canonically twisted* Q_n where vertices u, v, x, and y have the labels $b(u) = 000 \cdots 0$, $b(v) = 010 \cdots 0$, $b(x) = 100 \cdots 0$, and $b(y) = 110 \cdots 0$. In the subsequent sections we describe some of the properties of the twisted cube TQ_n.

4. DIAMETER OF TQ$_n$

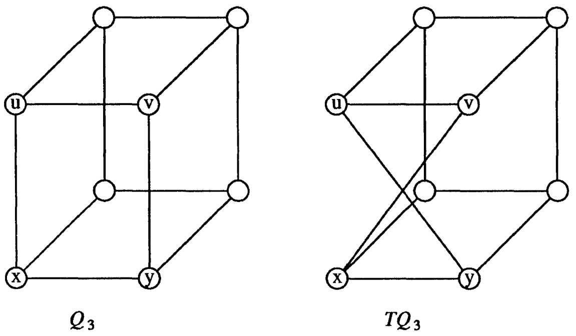


Figure 1.

It is well-known that $d(Q_n) = n$. Also, between any pair of vertices u and v in Q_n there are n disjoint paths, of which $d(u,v)$ are of length $d(u,v)$ and the rest are of length $d(u,v)+2$ [Kuhl80, SaSc85]. As a result, if $d(u,v) \le n-1$ then there are at least $n-2$ disjoint paths between u and v, each of which is of length at most $n-1$. This property of Q_n will be used shortly.

Theorem 1: $d(TQ_n) = n-1$, for $n \ge 3$.

Proof: Let TQ_n be the canonically twisted cube. The theorem can easily be verified when n equals 3 and 4. Thus assume that $n \ge 5$. Now, let s and t be any two vertices in TQ_n. We will show that in TQ_n we have $d(s,t) \le n-1$ for all s, t with equality for at least one pair. Depending on the value of $d(s,t)$ in Q_n, the following two cases are considered.

Case 1: In Q_n we have $d(s,t) \le n-1$. Then there are at least $n-2 \ge 3$ disjoint paths between s and t in Q_n, each of which is of length at most $n-1$. Thus, removal of edges ux and vy from Q_n can destroy at most two of such paths. This implies that in TQ_n we have $d(s,t) \le n-1$.

Case 2: In Q_n we have $d(s,t) = n$. Let $b(s) = (b_{n-1}b_{n-2}b_{n-3} \cdots b_1b_0)$ so that $b(t) = (\bar{b}_{n-1}\bar{b}_{n-2}\bar{b}_{n-3} \cdots \bar{b}_1\bar{b}_0)$ where $\bar{b}_i$ is the binary complement of b_i. A shortest $s-t$ path in TQ_n can be constructed as follows.

First concentrate on the ones of $b(s)$ in positions $n-3, n-4, \cdots, 0$. We can change these ones to zeros by traveling over a single edge for each exchange. Thus, after traveling $\mathrm{O}(b_{n-3}b_{n-4} \cdots b_0)$ edges we will arrive at one of the vertices u, v, x, or y (which one is determined, of course, by the two leading bits $b_{n-1}b_{n-2}$ of s). Next, we can change $b_{n-1}b_{n-2}$ to $\bar{b}_{n-1}\bar{b}_{n-2}$ by using a single edge of TQ_n. That edge will be uy or vx depending upon which of the four vertices we were led to by the first part of the path. Finally, all the zeros in $(b_{n-3}b_{n-4} \cdots b_0)$ must be turned to ones. Again a single edge is used for each of the $\mathrm{Z}(b_{n-3}b_{n-4} \cdots b_0)$ bits involved. Hence the total number of edges in our $s-t$ path is

$$\mathrm{O}(b_{n-3}b_{n-4} \cdots b_0) + \mathrm{Z}(b_{n-3}b_{n-4} \cdots b_0) + 1 = (n-2) + 1 = n-1.$$

It is easy to see that there is no shorter $s-t$ path: traveling over any edge of TQ_n changes only one bit with the exception of uy and vx which change two. It can be easily seen that edges uy and vx cannot both appear in any shortest path. Since $b(s)$ and $b(t)$ differ in all n positions, at least $n-1$ edges are needed to transform all bits. It follows that $d(s,t) = n-1$ by the construction above. Combining this fact with Case 1, we see that $d(TQ_n) = n-1$ as desired. □

5. VERTEX-CONNECTIVITY OF TQ$_n$

It is known that $\kappa(Q_n) = n$ [ArGr81, Kohl80]. We next prove that $\kappa(TQ_n) = n$. In fact we prove a more general connectivity theorem. Let G_1 and G_2 be two connected graphs with the same number p of vertices. Furthermore, let $V(G_1) = \{u_1, u_2, \cdots, u_p\}$ and $V(G_2) = \{v_1, v_2, \cdots, v_p\}$. Then $H = G_1 \odot G_2$ represents the graph obtained by taking G_1 and G_2 and connecting, via a new edge, vertex u_i to vertex v_i, for $1 \le i \le p$. That is,

$$V(H) = V(G_1) \cup V(G_2)$$

and

$$E(H) = E(G_1) \cup E(G_2) \cup \{u_iv_i | \; u_i \in E(G_1),\ v_i \in E(G_2),\ 1 \le i \le p\}.$$

The u_iv_i edges will be referred to as *cross* edges. Note that operation $\odot$ may generate different H graphs depending on how the vertices in graphs G_1 and G_2 are labeled [Hede69].

Theorem 2: Let G_1 and G_2 be connected graphs defined as above, and let $H = G_1 \odot G_2$. Then $\kappa(H) \ge 1 + \min(\kappa(G_1), \kappa(G_2))$.

Proof: Let $k = \min(\kappa(G_1), \kappa(G_2))$, and let X be an arbitrary subset of $V(H)$ such that $|X| = k$. We prove the theorem by showing that $H - X$ is connected. Observe that H contains at least $k + 1$ cross edges since k must be smaller than the number of vertices in each of the graphs G_1 and G_2. Therefore, romoval of k vertices from H cannot cause deletion of all cross edges. Now if $X \cap V(G_1) = \varnothing$ (respectively $X \cap V(G_2) = \varnothing$) then G_1 (respectively G_2) is a connected subgraph of $H - X$. Furthermore, every remaining vertex of G_2 (respectively G_1) is connected to this connected subgraph. Hence $H - X$ is connected.

Now suppose $X \cap V(G_1) = X_1 \ne \varnothing$ and $X \cap V(G_2) = X_2 \ne \varnothing$. We must then have $1 \le |X_1| \le k-1$ and $1 \le |X_2| \le k-1$. This implies that both $G_1 - X_1$ and $G_2 - X_2$ are connected by definition of k. Since there is at least one cross edge, say e, in $H - X$, the end points of e lie in $G_1 - X_1$ and $G_2 - X_2$, and therfore $H - X$ must be connected. □

Theorem 3: $\kappa(TQ_n) = n$.

Proof: Clearly it is possible to take two copies of Q_{n-1} and label their vertices such that $TQ_n = Q_{n-1} \odot Q_{n-1}$. Since $\kappa(Q_{n-1}) = n-1$, Theorem 2 implies that $\kappa(TQ_n) \ge n$. Also for any n-regular graph G, $\kappa(G) \le n$, hence the desired result. □

6. LENGTHS OF DISJOINT PATHS IN TQ$_n$

It is well-known that if G is a graph with $\kappa(G) = n$, then given any two distinct vertices $s, t \in V(G)$ we can find n disjoint $s-t$ paths in G [Hara72]. The following theorem gives an explicit description of such paths in TQ_n. Its proof, which is long, is omitted due to space constraint, but can be found in [EsNS88].

Exception	Possible Paths Lengths				
	$d-1$	d	$d+1$	$d+2$	$d+3$
1. There is one fixed 1 in $suf(s)$ and					
(a) $b_{n-1}b_{n-2} = \bar{c}_{n-1}\bar{c}_{n-2}$, or		d	1	$n-d-1$	
(b) either s, t are adjacent to u, x or s, t are adjacent to v, y		d		$n-d-1$	1
2. There is no fixed 1 in $suf(s)$ and					
(a) $b_{n-1}b_{n-2} = \bar{c}_{n-1}\bar{c}_{n-2}$, or	1	$d-1$		$n-d$	
(b) $b_{n-1}b_{n-2} = c_{n-1}c_{n-2}$ with exactly one of s, t equal to u, v, x, y, or		d		$n-d-1$	1
(c) $b_{n-1}b_{n-2} = c_{n-1}\bar{c}_{n-2}$ with exactly one of s, t equal to u, v, x, y, or		d	1	$n-d-1$	
(d) $b_{n-1}b_{n-2} = \bar{c}_{n-1}c_{n-2}$ with s or t equal to u, v, x, y		$d-1$	2	$n-d-1$	

Theorem 6: Let TQ_n be the canonically twisted n-cube and consider $s, t \in V(TQ_n)$ with $b(s) = b_{n-1}b_{n-2}, \cdots, b_0$ and $b(t) = c_{n-1}c_{n-2}, \cdots, c_0$. If $d(s, t) = d$ in Q_n then a set of n disjoint paths consisting of d of length d and $n-d$ of length $d+2$ continues to exit in TQ_n with the exception of the cases noted in the following table. If the entry in row i and colunm $d+j$ is k this means that there are k disjoint $s-t$ paths of length $d+j$ for exception i. A blank indicates no such paths. All paths for a given row can be taken to be disjoint. □

Note that in all exceptional cases but two (specifically 1(b) and 2(a)) the average length of the n paths in the table is at least as short as the average length between the same two points in Q_n. In fact for some of the cases above the paths from Q_n still exist in TQ_n but the listed set of the paths will be shorter.

7. SOME SUBGRAPHS OF TQ$_n$

In this section, we will identify some of the subgraphs of TQ_n. By construction, Q_{n-1} is a subgraph of TQ_n and thus all its subgraphs are contained in TQ_n. In fact any subgraph of Q_n which does not contain two independent edges belonging to some 4-cycle of Q_n, is also contained in TQ_n. This implies that TQ_n contains a 2^n-cycle, and any 2-dimentional *mesh* which is a subgraph of Q_n. While Q_n contains only even cycles, TQ_n conatins odd cycles as well.

In what follows we will show that the complete binary tree on $2^n - 1$ vertices, T_n, is a subgraph of TQ_n. It is known that T_n is not a subgraph of Q_n [SaSc85]. However, T_{n-1} is contained in Q_n [BrSc85]. To present our result, we need to show that two disjoint copies of T_{n-1} can be found in Q_n. This was first demonstrated by Prada, rediscovered independently by Bhatt and Ipsen, and then re-rediscovered by us [Prah74, BhIp85]. We include the proof for the sake of completeness.

Let S_n denote the graph obtained by taking two disjoint complete binary tree T_{n-1} and connecting their roots by a path of length 3. A picture of S_4 is given in Figure 2.

Theorem 7: For $n \geq 2$, S_n is a subgraph of Q_n. Furthermore, for $n \geq 3$ the roots r and u of the

two copies of T_{n-1} can be labeled so that $b(r)$ and $b(u)$ differ in exactly three positions.

Proof: Figure 3 gives labelings which embed S_n in Q_n for $n = 2, 3$. Note that in the latter case the labels along the path $r-s-t-u$ of length 3 are 001, 011, 111, and 110 respectively. By induction we may assume that S_{n-1} is isomorphic to a subgraph of Q_{n-1} with

$$b(r) = 001\cdots 1$$
$$b(s) = 011\cdots 1$$
$$b(t) = 111\cdots 1$$
$$b(u) = 11\cdots 10.$$

Now we can find two disjoint subgraphs isomorphic to S_{n-1}, call them S^0_{n-1} and S^1_{n-1}, in Q_{n-1} as follows. S^0_{n-1} is obtained by prefixing every label of S_{n-1} with a 0. Thus the labels of the corresponding path of length 3 are

$$b(r^0) = 0001\cdots 1$$
$$b(s^0) = 0011\cdots 1$$
$$b(t^0) = 0111\cdots 1$$
$$b(u^0) = 011\cdots 10.$$

If $v \in S_{n-1}$ is labeled $b(v) = (b_{n-2}b_{n-3}\cdots b_0)$ in Q_{n-1} then in S^1_{n-1} we let $b(v^1) = (1b_0b_1\cdots b_{n-2})$. In particular,

$$b(r^1) = 11\cdots 100$$
$$b(s^1) = 11\cdots 110$$
$$b(t^1) = 11\cdots 111$$
$$b(u^1) = 101\cdots 11.$$

A schematic drawing of these subgraphs is displayed in Figure 4(a). Now, the graph S_n is created by letting

$$S_n = S^0_{n-1} \cup S^1_{n-1} + \{s^0u^1, t^0t^1, u^0s^1\} - \{t^0u^0, t^1u^1\}$$

as in Figure 4(b). Finally the new roots are s^0 and s^1 with labels $001\cdots 1$ and $11\cdots 10$ respectively, which differ in exactly three positions. □

Theorem 8: T_n is subgraph of TQ_n.

Proof: Find a subgraph of Q_n which is isomorphic to S_n with the path $r-s-t-u$ labeled as in Theorem 7, that is

$$b(r) = 001\cdots 1$$
$$b(s) = 011\cdots 1$$
$$b(t) = 111\cdots 1$$
$$b(u) = 11\cdots 10.$$

If $v \in V(Q_n)$ has label $b(v) = 101\cdots 1$ then we can construct

$$TQ_n = Q_n - \{rs, tv\} + \{rt, sv\}.$$

Clearly $T_n = S_n + \{rt\} - \{s\}$ is a subgraph of TQ_n (note that $tv \notin E(S_n)$ so that the removal of this edge from Q_n causes no difficulties). □

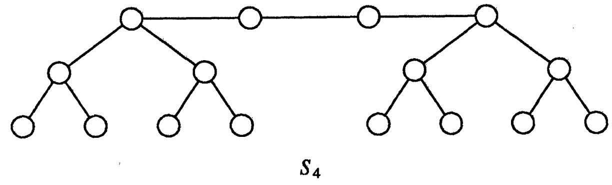


Figure 2.

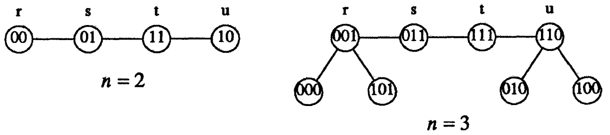
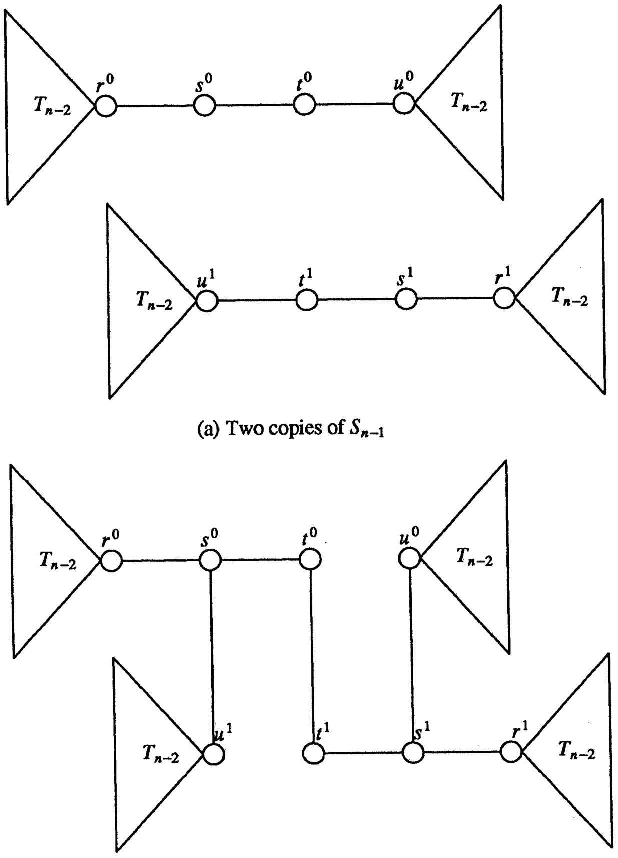


Figure 3: Embedding S_n in Q_n for $n = 2, 3$



(a) Two copies of S_{n-1}

(b) Constructing S_n using two copies of S_{n-1}

Figure 4.

8. CONCLUDING REMARKS

The hypercube interconnection topology, due to its powerful topological properties, has been widely adopted in the construction of distributed-memory multiprocessors. In this paper, we have shown that by exchanging any two independent edges in any shortest cycle of the hypercube, an interconnection topology, namely TQ_n, can be achieved which has some nice properties. Existing hypercube multiprocessors can be modified to take advantage of this new topology in two ways. A hypercube can be converted to TQ_n by exchanging two of its physical links. Second, two extra physical links can be added to a hypercube multiprocessor to obtain a topology which has both Q_n and TQ_n as subgraphs. In both cases, other components of the system should be modified accordingly. One major component is the *router* at each processing node. In what follows we address this issue for both cases.

Each processor (vertex) in the hypercube multiprocessor has a router to handle the interprocessor communication [LaNE87]. The function of the router may be performed by the processor or by a dedicated router chip. In a hypercube multiprocessor, upon receiving a message, a *routing tag* $(r_{n-1}r_{n-2}r_{n-3} \cdots r_0)$ is obtained by taking a bit-wise exclusive-OR operation between the router's local address $(c_{n-1}c_{n-2} \cdots c_0)$ and the destination address $(d_{n-1}d_{n-2} \cdots d_0)$ of the message. The message can then be forwarded to one of the neighboring processors through the j-th link if $r_j = 1$ for $0 \le j \le n-1$.

To support the TQ_n topology, the function of the routers should be slightly modified. For these routers, the routing tag is computed as above. Suppose TQ_n is the canonically twisted n-cube. Let's first consider the four routers at vertices u, v, x, and y; we will refer to these routers as *twisted* routers. If $r_{n-1}r_{n-2} = 01$ then the message is forwarded through the $(n-2)$-nd link, that is, either uv or xy. If $r_{n-1}r_{n-2} = 11$ then the message is forwarded through the $(n-1)$-st link, that is, either uy or vx. Note that in this case one routing step is saved compared with that in Q_n. If $r_{n-1}r_{n-2} = 10$ then the message is forwarded through the $(n-2)$-nd link if $r_j = 0$ for all $0 \le j \le n-3$, otherwise, the message is forwarded through some j-th link with $r_j = 1$ where $0 \le j \le n-3$. Note that in the former case, it will take two routing steps rather than one as required in Q_n. However, this additional routing step may not be necessary if the message is forwarded through other links first as in the latter case.

The function of the remaining $2^n - 4$ routers will also have to be slightly modified in order to take advantage of a possible saving of one routing step. If $r_{n-1}r_{n-2} = 11$ then one routing step can be saved by first forwarding the message to the node $d_{n-1}d_{n-2}00 \cdots 0$, one of the four twisted routers, and then the message is forwarded to the final destination. If $r_{n-1}r_{n-2} = 10$, then the message has to be forwarded through the $(n-1)$-th link if there exists only one j $(0 \le j \le n-3)$ such that $r_j = 1$. This is to avoid having an additional routing step. For all other cases, the message can be forwarded to any j-th link so as long as $r_j = 1$.

For the case where two edges (i.e., uy and vx) are added to Q_n, the routers are modified as follows. For the four twisted routers, now each with $n+1$ links, if $r_{n-1}r_{n-2} = 11$ then the message should be forwarded through the added link. Thus, one routing step is saved. For all other cases, the normal routing procedure should be followed. For the remaining $2^n - 4$ routers, if $r_{n-1}r_{n-2} = 11$, then one routing step can be saved by first forwarding the message to the node $d_{n-1}d_{n-2}00 \cdots 0$, one of the four twisted routers, and then the message is forwarded to the final destination.

In summary, the twisted n-cube, TQ_n, has the following properties as the n-cube Q_n. TQ_n consists of two disjoint Q_{n-1} subgraphs. Even rings and 2-dimensional mesh are subgraphs of TQ_n. TQ_n is n-regular and its vertex connectivity remains n. In addition, TQ_n has the following unique properties not possessed by Q_n. Any odd length ring with $2^n - 1$ or fewer vertices is contained in TQ_n. A complete binary tree with 2^n-1 vertices, which is a highly demanded topology by many applications, is a subgraph of TQ_n. The worst case number of routing steps is reduced from n to $n-1$. Furthermore, the average number of routing steps is also reduced. This implies improvement on communication delay which is critical to system performance.

9. REFERENCES

[ArGr81] J.R. Armstrong and F.G. Gray, "Fault diagnosis in a Boolean n-cube array of microprocessors," *IEEE Trans. on Comput.*, Vol. C-30, No. 8, pp. 587-590, August 1981.

[BhIp85] S.N. Bhatt and C.F. Ipsen, "How to embed trees in hypercubes," *Research Report YALEU/DCS/RR-443*, Department of Computer Science, Yale University, December 1985.

[BrSc85] J.E. Brandenburg and D.S. Scott, "Embeddings of communication trees and grids into hypercubes," *Technical Report*, Intel Scientific Computers, 1985.

[DePa78] A.M. Despain and D.A. Patterson, "X-Tree: A tree structured multiprocessor computer architecture," *Proc. of the 5th Annu. Symp. Computer Architecture.*, pp.144-151, August 1978.

[EsNS88] A.-H. Esfahanian, L.M. Ni and B.E. Sagan, "On enhancing hypercube multiprocessors," *Technical Report, MSU-ENGR-88-012*, Department of Computer Science, Michigan State university, January 1988.

[Foxg86] G.C. Fox, "Caltech concurrent computation program annual report 1985-1986," *Caltech Concurrent Computation Program Technical Report C^3P-290B, October 1986.*

[Hara72] F. Harary, *Graph Theory*, Addison Wesley, 1972.

[Hede69] S. Hedetniemi, "On Classes of Graphs defined by Special Cutsets of Lines," *The Many Facets of Graph Theory, Lecture Notes in Mathematics*, Vol. 110, Springer-Verlag, 1969, pp. 171-189.

[HsYZ87] W.T. Hsu, P.C. Yew and C.Q. Zhu, "An enhancement scheme for hypercube interconnection networks," *Proc. of the 1987 Int'l Conf. on Parallel Processing*, pp.820-823, August 1987.

[HwBr84] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[HwGh87] K. Hwang and J. Ghosh, "Hypernet: A Communication-Efficient Architecture for Constructing Massively Parallel Computers," *IEEE Trans. on Comput.*, Vol. C-36, No. 12, pp. 1450-1467, Dec. 1987.

[Prah74] L.N. Praha, "On cubes and dichotomic trees," *Casopis Pro Pestovani Matematiky*, roc. 99 (1974).

[PrVu81] F.P. Preparata and J. Vuillemin, "The cube-connected cycles: A versatile network for parallel computations," *Commun. ACM*, pp.300-309, May 1981.

[SaSc85] Y. Saad and M.H. Schultz, "Topological properties of hypercubes," *Technical Report*, YALEU/DCS/RR-389, Department of Computer Science, Yale University, June 1985.

[Seit85] C. Seitz, "The cosmic cube," *Commun. of ACM*, Vol. 28, No. 1, pp. 22- 33, January 1985.

[ShFi88] Y. Shih and J. Fier, "Hypercube systems and key applications," *Parallel Processing for Supercomputing and Artificial Intelligence*, K. Hwang and D. DeGroot, eds., New York: McGraw-Hill, 1988.

[WuLi81] S.B. Wu and M.T. Liu, "A cluster structure as an interconnection network for large multimicrocomputer systems," *IEEE Trans. on Computers*, pp.254-265, April 1981.

RELIABILITY OF THE HYPERCUBE

Seth Abraham
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Krishnan Padmanabhan
Distributed Systems Research Department
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Several analytical models are presented and solved in this paper for the subcube reliability problem *associated with the hypercube multiprocessor architecture. This problem refers to the ability of a binary d-cube, in the presence of component failures, to embed disjoint functional subcubes of various sizes in the damaged structure. Partitioning of a fault-free hypercube into subcubes that can be allocated to different tasks is a common practice, so that the degradation in this ability is a good measure of the effect of failures. We provide models that account for node failures only, link failures only, or both node and link failures. These show that the architecture is quite resilient to failures in terms of the ability to salvage functional subcubes out of a damaged hypercube.*

1. INTRODUCTION

The boolean d-cube network [7,9,10] consisting of processing elements placed at the vertices of a d-dimensional hypercube, has proved to be a popular structure for direct-connected multiprocessor systems. The reader is referred to the various papers in [5] for the basis of this popularity. In this paper, we are concerned with a particular aspect of the hypercube architecture, viz., its reliability. A hypercube system has 2^d nodes (processing elements) and $d2^{d-1}$ (full duplex) connecting links. In large systems of this kind, it is obvious that some components of the system will fail before long, so that characterization of the degraded system is important to determine how many of the failures can be tolerated. In the next three sections of this paper, we provide analytical models for a particular version of this reliability problem. The rest of this section will be devoted to terminology and problem definition.

Let us begin with a brief summary of the structural and topological properties of the hypercube that are relevant to our analyses. A complete discussion of this subject can be found in [2,8]. The $N=2^d$ nodes of the d-dimensional hypercube can be labeled using d-bit addresses and the connections between them specified as follows: two nodes whose addresses differ in exactly one bit position i, $0 \le i \le d-1$, are connected by a link. This link is said to span dimension i of the cube, so that each of the d dimensions has $N/2$ links spanning it. We refer to d as the *order* of the hypercube. Each node in a d-cube has degree d and the distance between two nodes x and y, whose addresses differ in j bit positions, is given by the Hamming distance $\mathrm{H}(x,y) = j$. Fig. 1 shows the structure of a binary 4-cube.

A j-subcube of a d-cube is a subgraph consisting of 2^j nodes (and the connecting links between them) obtained by choosing $d-j$ dimensions i_1, $i_2,\ldots,i_{d-j}$, and considering all the nodes that have the same address bit in each of these bit positions. Fig. 1 illustrates two 3-subcubes (highlighted) in a binary 4-cube. Such a j-subcube can be thought of as being generated by the following process: split the d-cube across dimension i_1, separating it into two $(d-1)$-subcubes, each consisting of nodes containing the same bit in position i_1; choose one of these cubes and split it across dimension i_2, resulting in two $(d-2)$-subcubes, etc., continuing until a $(j+1)$-subcube is split across dimension i_{d-j} to result in two j-subcubes.

This recursive construction of the hypercube from smaller subcubes proves very useful in task allocation and partitioning the cube for applications. The subcubes have all the structural properties of the larger cube so that many of the algorithms designed for hypercubes may be written with the order of the available cube as a runtime parameter. The AXIS operating system for the NCUBE multiprocessor, for instance, permits the main cube array to be shared among two or more tasks, allocating the subcube of the appropriate size to each task [4]. Because the subcubes are disjoint from each other, allocation of the partitions is particularly simplified and each task considers itself as working on an i-cube (with nodes relabeled accordingly). It is possible to view an incoming task as a set of interacting modules that have to be assigned to the nodes of a subcube with adjacencies between modules in the task graph being preserved in the subcube; algorithms have been developed to determine the size of the subcube required for each task under this condition [3]. In addition, efficient algorithms for many applications are designed to exploit the subcube partitioning ability of the hypercube, quite often in a recursive or divide-and-conquer fashion [6,11]. It is useful to see how much of this ability is lost when failures begin to occur in the system.

So the problem that we will address in this paper can be expressed as follows. We seek expressions for the reliability and mean time to failure of a d-cube system for a variety of breakdown conditions. These conditions will be defined in terms of the ability to embed disjoint subcubes of different sizes in a hypercube. In all but one of the cases that we consider, we analyze conditions in which at least a $(d-1)$-subcube, the largest proper subcube, is functional. Additional disjoint subcubes of smaller sizes could coexist, and these will be captured in the definitions of various system states. Section 2 considers this problem under the node failure model, in which only the effect of node failures will be considered. Section 3 develops a similar model for the link failure case, and Section 4 analyzes the system when both link and node failures are permitted in the model. In Section 5, we present schemes for remapping the node addresses so that functional subcubes can be salvaged from the damaged hypercube system.

For an alternate formulation of the reliability problem for the hypercube, and some issues related to its fault tolerance, see [1].

2. NODE FAILURE MODEL

Consider first the embedding of a functional $(d-1)$-subcube in the presence of failures. While it is true that a single node failure must always leave an undamaged $(d-1)$-subcube, as few as two failures could destroy all such subcubes. For example, if node 0 and node $N-1$ fail, there is no way to embed an undamaged $(d-1)$-subcube in the damaged d-cube. Given an arbitrary set of node failures, a fault free $(d-1)$-subcube exists if and only if all the faulty nodes may be contained in an i-subcube, $i < d$. (Necessity follows from the example presented above; sufficiency becomes obvious when considering that a d-cube may be divided into two disjoint $(d-1)$-subcubes and all faulty nodes positioned totally within one of the subcubes.)

We define S_i to be the system state in which all node failures in the cube are contained in an i-subcube, but not in an $(i-1)$-subcube for $0 \le i < d$. In terms of functional subcubes, state S_i can be characterized as embedding exactly $d-i$ disjoint subcubes of order $d-1$, $d-2,\ldots,i$, respectively. (To see this, split the d-cube into two $(d-1)$-cubes with one of these cubes containing the faulty i-subcube; split the latter $(d-1)$-cube into two $(d-2)$-cubes, with one of these cubes containing the faulty i-subcube, etc.) This sequence of functional subcube sizing is unique if we insist on a maximal disjoint subcube at each point in the sequence. (Note that while the size sequence is unique, there may be many ways to generate a subcube of a particular size.) Also worth noting is the fact that even though no additional disjoint subcube of order $\ge i$ may be embedded, it might be possible to embed functional subcubes of order $< i-1$ inside the faulty i-subcube. In state S_d, no embedding of a functional $(d-1)$-subcube is possible. Finally, S_* represents the initial, fault-free state of the d-cube. We assume that all nodes have an identical exponential failure distribution with constant failure rate λ. The transitions between these states are shown in Fig. 2.

The transition from state S_i to state S_{i+j} $(0 < j \le d-i)$ occurs when an additional fault has occurred outside the damaged i-subcube and the new damaged cube will be of size $i+j$. To determine the rate, imagine the d-cube as split across i dimensions such that each node in the original damaged i-cube is in a separate partition. Each of the resultant 2^i partitions is a $(d-i)$-cube containing exactly one node from the damaged i-

cube. The new failure must be in one of these partitions, say C; further, all paths within region C cross none of the i dimensions used during the original splitting process. Therefore, if the new failure in C is distance j away from the node in C that belongs to the damaged i-cube, all damaged nodes in the system may be contained in an $(i+j)$-subcube. There are $\binom{d-i}{j}$ such nodes in each C, leading to a transition rate of $\lambda 2^i \binom{d-i}{j}$.

Let us denote the probability of being in state S_i as a function of time as $P_i(t)$. Then the state equations for this system are given by

$$\frac{\partial P_*}{\partial t} = -\lambda N P_* , \quad \frac{\partial P_0}{\partial t} = -\lambda (N-1) P_0 + \lambda N P_* ,$$

$$\frac{\partial P_i}{\partial t} = -\lambda (N - 2^i) P_i + \sum_{j=0}^{i-1} \lambda 2^j \binom{d-j}{i-j} P_j , \quad 0 < i \le d .$$

The initial conditions are $P_*(0) = 1$, and $P_i(0) = 0$ for all i. It can be shown (by induction on i) that the solution to this system of equations can be written as:

$$P_i(t) = (-1)^{i+1} \binom{d}{i} 2^{d-i} e^{-\lambda N t} + \binom{d}{i} \sum_{m=0}^{i} (-1)^{i-m} \binom{i}{m} 2^{d-m} e^{-(N-2^m)\lambda t} .$$

Let us now define the cumulative probabilities $R_i(t)$, $0<i<d$, as

$$R_*(t) = P_*(t), \quad R_0(t) = P_0(t) + R_*(t), \quad R_i(t) = P_i(t) + R_{i-1}(t), \quad i>0 .$$

Thus, $R_i(t)$ is the probability that all node failures (if any) up to time t are contained within an i-subcube, leaving $d-i$ functional disjoint subcubes of order $d-1$, $d-2$,..., i. These functions are plotted in Fig. 3 for a 10-cube. (A conservative node failure rate of $\lambda = 10^{-5}$ per hour is assumed.) $R_i(t)$ represents the reliability of the system, if our system breakdown condition is based on a *fault-containment criterion*, viz., all the node failures are not contained in an i-subcube. The system's mean time to failure (T) can be evaluated under this criterion by integrating the expression for $R_i(t)$. It is easy to see that for $0 < i < d$,

$$T_* = \frac{1}{N\lambda} , \quad T_0 = T_* + \frac{1}{(N-1)\lambda} ,$$

$$T_i = T_{i-1} + (-1)^{i+1} \binom{d}{i} \frac{1}{2^i \lambda} + \binom{d}{i} \sum_{m=0}^{i} (-1)^{i-m} \binom{i}{m} \frac{2^{d-m}}{(N-2^m)\lambda}$$

These are evaluated in Table 1 for various system sizes.

All the numbers in Table 1 would scale linearly with λ. Let us interpret these numbers using the 10-cube as an example. The mean time to the first node failure is 98 hours. If the system can stay operational with disjoint subcubes of order 9,8,7,6,5, and 4, (i.e., all the failures are confined to a 4-cube), then the MTTF increases to about 10 days. Note that much of this increase materializes from just the ability to tolerate one node failure — T_0 = 195 hours. However, as we relax our fault containment criterion beyond a 5-cube (a ($d/2$)-cube in general), the increases in MTTF are much more substantial. For very large cubes ($d \ge 12$), insistence on a functional $(d-1)$-subcube is indeed a stiff condition to satisfy, as the MTTF numbers testify.

3. LINK FAILURE MODEL

We now consider the state of the damaged cube under a link failure model. As in the last section, we are interested only in system configurations that can embed at least a $(d-1)$-subcube. Link failures affect the topology of the cube in a fundamentally different way than node failures do. Two disjoint $(d-1)$-cubes are formed each time a d-cube is split across one of its dimensions. Since there are exactly d ways to perform this split, there are only $2d$ possible $(d-1)$-subcubes that may be embedded in a d-cube. Let us label these subcubes $C_{i,j}$ for $0 \le i < d$ and $0 \le j \le 1$. (Refer to Fig. 1.) $C_{i,j}$ will denote the subcube comprised of nodes with a j in their i^{th} bit (and the links between these nodes). Obviously, subcube $C_{i,0}$ is disjoint from $C_{i,1}$ for $0 \le i < d$. Any other pair of subcubes $C_{i,j}$ and $C_{k,l}$ $(i \ne k)$ will share a $(d-2)$-cube comprised of exactly those nodes (and interconnecting links) with a j in their i^{th} bit *and* an l in their k^{th} bit. If one of the links in this $(d-2)$-cube fails, then both $C_{i,j}$ and $C_{k,l}$ will be damaged. In fact, the first link failure to occur in a cube will damage $d-1$ subcubes; subsequent failures may or may not affect the remaining undamaged subcubes. To determine subcube reliability, we must characterize the effect of the failure of each link in the system for all relevant system states.

For the moment, consider the failure of the link between nodes 0 and 1. This damages only subcubes $C_{i,0}$, $i > 0$ leaving the remaining subcubes undamaged. While any link in the system is equally likely to fail, for the purposes of analysis, we can relabel all the nodes in the cube so that the faulty link is mapped into the link between nodes 0 and 1. Since only one dimension is fixed by this procedure (the dimension which must be labeled 0), there are $(d-1)!$ equally satisfactory ways to label the remaining dimensions. When additional links fail, an appropriate labeling for the remaining non-fixed dimensions may be chosen from this set of $(d-1)!$ labelings so that the subcubes $C_{i,1}$, $(i > 0)$ may be thought of as being damaged *in order*. That is, $C_{1,1}$ is damaged first, then $C_{2,1}$, etc., and finally $C_{d-1,1}$. (The cubes $C_{0,0}$ and $C_{0,1}$ may be damaged at any point in this sequence; we will consider them presently.) This ordering of damaged subcubes does not constitute an ordering of the link failures; it is only the selection of a particular labeling for the nodes as each failure occurs to describe the system state compactly.

We now characterize the state of the system by the set of undamaged $(d-1)$-subcubes in the system. The possible system states (for $0 \le i < d$) are:

$$S_* = \{ C_{k,j},\ 0 \le k < d,\ 0 \le j \le 1 \}, \quad S_{2,i} = \{ C_{0,0}, C_{0,1}, C_{j,1},\ i < j < d \},$$

$$S_{1,i} = \{ C_{0,1}, C_{j,1},\ i < j < d \}, \quad S_{0,i} = \{ C_{j,1},\ i < j < d \} .$$

Note that the $(d-1)$-subcubes in any state above are not all disjoint. It is possible to characterize these states in terms of disjoint subcubes, in a manner similar to the state definitions for the node failure model. The states $S_{2,i}$ each support *two* disjoint $(d-1)$-subcubes and these are the only states for which there are no equivalent states under the node failure model. State $S_{1,i}$ embeds $d-i$ fault-free disjoint subcubes of order $d-1$, $d-2$,..., i. (These are $C_{0,1}$, $C_{d-1,1} \cap \overline{C}_{0,1}$, $C_{d-2,1} \cap \overline{C}_{d-1,1} \cap \overline{C}_{0,1}$,..., respectively.) State $S_{0,i-1}$, $0 < i < d$, is an equivalent state in terms of disjoint functional subcubes, supporting $d-i$ subcubes of order $d-1$, $d-2$,..., i. (These are $C_{d-1,1}$, $C_{d-2,1} \cap \overline{C}_{d-1,1}$,..., respectively.) Thus it is possible to summarize the correspondence between the node and link failure models (Figs. 2 and 4) as follows: states $S_{2,i}$ in the link failure model do not have equivalent states in the node model. State $S_{1,0}$ corresponds to state S_0 in the node model; and states $S_{1,i}$ and $S_{0,i-1}$ together correspond to state S_i, $0 < i < d$, in the node model. Finally, state $S_{0,d-1}$ in the link model corresponds to S_d in the node model, the state in which no functional $(d-1)$-subcube embedding is possible.

As before, we assume an identical, exponential failure distribution (rate λ) for each component (link) in the system. In Fig. 4 we see the state diagram with the transitions between the $3d+1$ states. The transition rate from S_* to $S_{2,0}$ is clearly $\lambda d 2^{d-1}$ as the failure of any link in a fault free system will result in a system state of $S_{2,0}$. A detailed description of the transition rates between the remaining states now follows.

Let us first consider the transitions from state $S_{2,0}$ to state $S_{2,j}$; $C_{0,0}$ and $C_{0,1}$ are both undamaged in these transitions. Any link failure contributing to this transition must span dimension 0. Without loss of generality, let the new failed link be incident to node x, $x \in C_{0,0}$. Consider such nodes with exactly j 1 bits in the node address. We may remap all the nodes in the cube so that the j 1 bits in the address of node x will be in bit positions 1, 2,..., j. Note that x and its new faulty link (across dimension 0) are now in $C_{k,1}$ for all k, $0 < k \le j$. Therefore, exactly j additional subcubes have been damaged. We may count the number of nodes x by counting the nodes in $C_{0,0}$ with exactly j 1 bits; this results in a transition rate from state $S_{2,0}$ to state $S_{2,j}$ of $\lambda \binom{d-1}{j}$.

To generalize this for the $S_{2,i}$ to $S_{2,i+j}$ transition, $(0 \le i < d$, $0 < j < d-i)$ we are again concerned only with faulty links that span the 0^{th} dimension. Since the subcubes $C_{k,1}$ $0 < k \le i$ are already damaged, only the $d-1-i$ subcubes $C_{k,1}$, $i < k < d$ need be considered. Let us consider a faulty link incident to node x, $x \in C_{0,0}$, with a node address containing exactly j 1 bits in the bit positions greater than i. Since the only dimension labels fixed from previous mappings are those $\le i$, we may remap all the nodes in the cube so that the j 1 bits just described are in bit positions $i+1$, $i+2$,..., $i+j$. Clearly x and its new faulty link are in $C_{k,1}$ for all k, $i < k \le i+j$. Thus, we count the number of nodes in $C_{0,0}$ with an address containing exactly j 1 bits in bit positions k, $i < k < d$ and obtain a transition rate of $\lambda 2^i \binom{d-1-i}{j}$. This rate (among others) is shown in Fig. 5a.

The remainder of the transitions from state $S_{2,i}$ involve the links which do not span dimension 0; failure of any of the non-zero dimensioned links will damage either $C_{0,0}$ or $C_{0,1}$ plus j additional subcubes $0 \le j < d-i$. Since we may remap the cube so that $C_{0,0}$ is always damaged first, we may consider links in $C_{0,0}$ without loss of generality. To account for link failures in $C_{0,1}$ the final rate will be doubled.

Consider some node x in $C_{0,0}$ with an address containing j 1 bits in the $d-1-i$ bit positions greater than i. As before, we may remap all the nodes in the cube so that the j 1 bits just described are in bit positions $i+1$, $i+2, \ldots, i+j$. We need to determine all the links incident to x (other than the link spanning dimension 0) such that x and this link will be in $C_{k,1}$ for all k, $i < k \le i+j$. The only links which fit this description span the dimensions 1, 2,..., i, $i+j+1$, $i+j+2, \ldots, d-1$. We may determine the number of nodes x by counting the number of nodes in $C_{0,0}$ with an address containing exactly j 1 bits in bit positions k, $i < k < d$. However, we may not simply multiply this figure by $d-1-j$ to obtain the number of links since the links which span dimensions 1, 2,..., i are incident to two nodes with exactly j 1 bits in bit positions k, $i < k < d$. Thus, half of these links must be subtracted out so that the total number of links is $2^i \binom{d-1-i}{j}(d-1-i/2-j)$. To take into account links in $C_{0,1}$, we double this figure to obtain the rate for the $S_{2,i}$ to $S_{1,i+j}$ transition as $\lambda 2^{i+1} \binom{d-1-i}{j}(d-1-i/2-j)$ for $0 \le i < d$, $0 \le j < d-i$.

The transitions from state $S_{1,i}$, $0 \le i < d$ (Fig. 5b), are similar to the transitions out of state $S_{2,i}$. For the $S_{1,i}$ to $S_{1,i+j}$ $(0 < j < d-i)$ transition, start with the rate $\lambda 2^i \binom{d-1-i}{j}$ for links spanning dimension 0. Since $C_{0,0}$ is already damaged, we must add some of the links within $C_{0,0}$. (In the $S_{2,i}$ case, failure of these links caused a transition from state $S_{2,i}$ to state $S_{1,i+j}$.) Thus we add $\lambda 2^i \binom{d-1-i}{j}(d-1-i/2-j)$ for a total rate of $\lambda 2^i \binom{d-1-i}{j}(d-i/2-j)$. The transition from state $S_{1,i}$ to state $S_{0,i+j}$ $(0 \le j < d-i)$ is identical to the $S_{2,i}$ to $S_{1,i+j}$ case except that since $C_{0,0}$ has already been damaged, we do not need to double the rate. Thus the transition rate is simply $\lambda 2^i \binom{d-1-i}{j}(d-1-i/2-j)$.

Finally, the transitions from state $S_{0,i}$ $0 \le i < d$ (Fig. 5c) may be obtained by adding the $S_{1,i}$ to $S_{1,i+j}$ transition rate to the $S_{1,i}$ to $S_{0,i+j}$ transitions rate for $0 < j < d-i$. Thus we obtain a rate of $\lambda 2^i \binom{d-1-i}{j}(2d-2j-i-1)$.

Now that we have completely specified the rates of all transitions in the state diagram, we may write the state equations. Let $P_{i,j}(t)$ be the probability of being in state $S_{i,j}$ at time t. Solutions for $P_{i,j}(t)$ and $P_*(t)$ may be expressed as sums of exponentials, just as the solutions for the node model were. However, we have been unable to obtain closed form solutions for the coefficients in this case and have evaluated these numerically. Based on the results, one can define the following probabilities:

$P_*(t)$ = Probability that no failures have occurred

$P_{2-(d-1)\text{cubes}}(t)$ = Probability that link failures have occurred, but two disjoint $(d-1)$-subcubes can be embedded $= \sum_{i=0}^{d-1} P_{2,i}(t)$

$P_i(t)$ = Probability that link failures have occurred leaving $d-1$ functional disjoint subcubes of order $d-1, \ldots, i$.

$$= \begin{cases} P_{1,0}(t) & i = 0 \\ P_{1,i}(t) + P_{0,i-1}(t) & i > 0 \end{cases} .$$

Reliability measures are then given by:

$$R_*(t) = P_*(t)\ , \quad R_{2-(d-1)\text{cubes}}(t) = R_*(t) + P_{2-(d-1)\text{cubes}}(t)\ ,$$

$$R_0(t) = R_{2-(d-1)\text{cubes}}(t) + P_0(t)\ , \quad R_i(t) = R_{i-1}(t) + P_i(t)\ , \ \ 0 < i < d\ .$$

The mean time to failure figures, T, corresponding to these reliability measures are shown in Table 2 for a 10-cube with $\lambda = 10^{-6}$ per hour.

Note that we have used a lower failure rate for the links than that for the nodes, to account for their lower logical complexity. It is interesting to see from Tables 1 and 2 that the increase in MTTF over the node model is not commensurate with the decreased failure rate. For $d = 6$, the increase in T_i's is by a factor of 4 to 5, whereas for $d = 12$, the increase is about a factor of 2. The effects of link and node failures on the system are indeed different. A single node failure damages d possible $(d-1)$-subcubes, while a link failure destroys only $d-1$ of these. As few as two node failures can destroy all $(d-1)$-subcube embeddings in a d-cube, whereas this requires three failures under the link model. The more catastrophic effect of node failures on the system is offset by the fact that there are $d/2$ times as many links as there are nodes. This second effect begins to dominate as the size of the network increases (beyond $d = 3$). In particular, if identical failure rates were assumed for nodes and links, the mean time to failure under the link model would be less than that under the node model for $d > 3$.

Other observations made in the last section regarding Table 1 also apply here. One way to relax the size of the functional subcubes is to insist on 2^{d-i} disjoint i-subcubes, $(d-i) > 1$, for the system to be functional. The mean time to failure for this condition can be derived in a straightforward manner. We define $d - i + 2$ states S_j, $0 \le j \le (d-i+1)$, with all the faulty links in state S_j spanning exactly j dimensions. State S_{d-i+1} is the breakdown state. The number of link failures that can cause a transition from state S_j to S_{j+1} is $(d-j)2^{d-1}$. The state equations for this new system can be solved and it can be shown that the mean time to failure is given by

$$T_i = \frac{2}{N} \sum_{j=0}^{d-i} \binom{d}{j} \sum_{m=0}^{j} \frac{\binom{j}{m}(-1)^{j-m}}{(d-m)\lambda} .$$

This is evaluated in Table 3 for $d = 10$, 11, and 12, and various values of i. We can see a significant increase in MTTF as the size of the functional portions is reduced. Thus in a 12-cube, the MTTF for two functional 11-cubes is 85 hours, whereas it is over 300 hours for 32 functional 7-cubes.

4. COMBINED NODE AND LINK FAILURE MODEL

While we have explored the probability of embedding functional subcubes in a damaged d-cube under a node failure model and a link failure model, we have not considered what happens when both node and link failures may occur. Under the classic node failure model we considered earlier, link failures may be disregarded since a link failure may be modeled as the failure of one of its terminal nodes. Thus the node failure rate encompassed the failure of the node *and* all its incident links. However, to apply this technique to the link failure model would necessitate modeling a node failure as the simultaneous failure of all the node's incident fault free links. This would violate the assumption that failures are independently distributed. For this reason, a combined node and link failure fault model is developed here.

The node failure rate will be denoted by λ_n and the link failure rate by λ_l. Both rates are constant and independent. For our new model, let us begin with the link failure model developed in the last section. The question now arises as to what happens to the links incident to a failed node. A link failure is significant to the analysis of the previous section if and only if it belongs to some (as yet) undamaged $(d-1)$-subcube. Links incident to failed nodes are not a part of any undamaged subcube and thus may be ignored in the combined model. Thus the state diagram and link failure transitions (with $\lambda = \lambda_l$) depicted in Figs. 4 and 5 accurately describe the transitions due to link failures (alone) for the new model. All that remains is to add the node failure transition rates. These rates are developed in the following paragraphs.

When a node failure occurs in a previously fault free system, the fault may be mapped to node 0. Clearly this damages d subcubes ($C_{i,0}$ for $0 \le i < d$) and corresponds to the system state $S_{1,0}$. Thus we have a new transition from state S_* to state $S_{1,0}$ with rate $\lambda_n 2^d$ (see Fig. 6a). Any node failure in state $S_{2,i}$, $0 \le i < d$, puts the system into state $S_{1,i+j}$, $0 \le j < d-i$, since it damages either $C_{0,0}$ or $C_{0,1}$, and j subcubes $C_{k,1}$, $i < k < d$. The rates of these transitions may be derived in a manner similar to the way link failures were derived, i.e., counting the number of nodes with addresses containing j 1 bits in the $d-1-i$ bit positions greater than i. The rate must be doubled to account for failures in both $C_{0,0}$ and $C_{0,1}$, leading to the transitions depicted in Fig. 6a.

In state $S_{1,i}$, $0 \le i < d$, a node failure will damage all, some or none of the remaining subcubes. First considering the nodes in $C_{0,0}$, we note that only those nodes with addresses containing 1 bits in bit positions greater than i will damage additional subcubes. These are the transitions from state $S_{1,i}$ to state $S_{1,i+j}$, $0 < j < d-i$ shown in Fig. 6b. Next, if the faulty node is in $C_{0,1}$, at least one subcube ($C_{0,1}$) will be damaged; j additional subcubes $0 \le j < d-i$ will be damaged for nodes with addresses containing j 1 bits in bit positions greater than i. These are the transitions from state $S_{1,i}$ to state $S_{0,i+j}$ depicted in Fig. 6b. Note that only these $2^d - 2^i$ node failures are relevant for the transitions to state $S_{0,i+j}$.

Finally, the transitions from state $S_{0,i}$, $0 \le i < d$, (Fig. 6c) correspond to the failure of nodes with addresses containing j 1 bits, $0 < j < d-i$, in bit positions greater than i.

All the node transition rates described above can now be combined with the link failure rates developed in the previous section to derive the system state equations. When $\lambda_n = 0$, this system is obviously identical to that developed in the preceding section for the link failure model. When $\lambda_l = 0$, the states $S_{2,i}$ are no longer a part of the model. This removes the difference between the link and node failure models referred to in the previous section so that states $S_{1,i}$ and $S_{0,i-1}$ combined can be renamed S_i, to derive the node model. (State $S_{1,0}$ would correspond to S_0, and state $S_{0,d-1}$ to S_d.) We can solve the state equations for the combined model numerically to obtain the state probability distributions. Reliability measures defined in the last section can be computed under this model, and corresponding MTTF's are shown in Table 4.

Clearly the reliability and MTTF of the network are lower when both link and node failures are taken into account, than when only one of them is. Node failures have the dominant effect, but only due to their higher failure rate as discussed at the end of Section 3. However the results of this section show that neither component can be neglected (unless one has a much lower failure rate than the other) and the combined model gives a more accurate picture of system degradation.

We close this section by presenting a model that would seem initially to be a gross approximation to the combined model, but whose accuracy relative to its simplicity turns out to be excellent. Let us go back to the node failure model of Section 2 and define a *supernode* to be a node plus half its incident links. (Each link thus "belongs" to one node.) The failure of any of the $(1+d/2)$ components of the supernode would lead to its failure. (This is where the approximation lies.) The failure rate for each supernode is then given by $\lambda = \lambda_n + d\lambda_l/2$. When d is odd, we will use this expression as an estimate for the supernode failure rate, even though we cannot associate the same number of links with each supernode. Substituting this value of λ into the node failure model of Section 2, we derive the mean time to failure figures in Table 5. The approximation is clearly conservative in that not every link failure would in reality affect the node (say, A) to which it is assigned; if the other end of the link is connected to a node which has already failed, then node A need not be brought down when the link fails. However, it provides results very close to those from the exact model.

5. REMAPPING THE CUBE NODES

All the mappings described in previous sections were intended solely for the purposes of analysis; in this section we describe a procedure for selecting an appropriate mapping for the nodes in the cube so that undamaged subcubes may be used by application tasks. The first step is to determine which of the possible embedded $(d-1)$-subcubes are undamaged. Next, the number and orders of disjoint subcubes must be identified. The final chore of remapping is then reduced to fixing some k bits in the node address resulting in a $d-k$ subcube. We first describe the procedure for the case where we have subcubes of order $d-1$, $d-2$,..., i; subsequently, the procedure for 2^{d-i} functional subcubes is explained.

We label the possible $(d-1)$-subcube embeddings $C_{i,j}$, $0 \le i < d$, $0 \le j \le 1$ as described earlier. When some node x fails, it damages exactly d of these embeddings. The identity of the damaged cubes is determined using the following algorithm:

```
for all bits $x_i$ in the node address $x$, set $C_{i,x_i}$ = DAMAGED;
```

As each link fails, it damages exactly $d-1$ embeddings. For embedding $C_{i,j}$ to be damaged, both nodes incident to the faulty link must lie in $C_{i,j}$. If we describe the damaged link as incident to node x across dimension j, we may use the following algorithm to determine what embeddings are no longer possible:

```
for all bits $x_i$ in the node address $x$, if ($i \ne j$) set $C_{i,x_i}$ = DAMAGED;
```

After the list of current failures has been evaluated, the system may support two disjoint $(d-1)$-subcubes if and only if there exists an integer k such that both $C_{k,0}$ and $C_{k,1}$ are undamaged. (This corresponds to a system state of $S_{2,i}$.) Otherwise, there are k undamaged, non-disjoint $(d-1)$-subcubes. Let these subcubes be labeled C_{i_1,j_1}, C_{i_2,j_2},..., C_{i_k,j_k}. From these, k disjoint subcubes of order $d-1$, $d-2$,..., $d-k$ may be obtained as follows: the $(d-1)$-subcube is comprised of all nodes x such that $x_{i_1} = j_1$; the $(d-2)$-subcube is comprised of all nodes x such that $x_{i_2} = j_2$ and $x_{i_1} \ne j_1$; etc. In general, the $(d-l)$-subcube is comprised of all nodes x such that $x_{i_l} = j_l$, and $x_{i_m} \ne j_m$ for all $m < l$. This completely identifies the disjoint, functional subcubes.

The procedure for 2^{d-i} embedded subcubes of order i is somewhat simpler. Each node need only keep track of dimensions that have no faulty links. That is, when a link failure occurs across some dimension j, then that dimension is no longer considered fault free. As long as the number of fault free dimensions is at least i, the embedding is possible. The subcubes may be identified by fixing all but the i lowest fault free dimensions yielding the 2^{d-i} subcubes.

6. SUMMARY

We have attempted to characterize in this paper the reliability and degradation of the hypercube structure as network components begin to fail. The analysis was based upon the damaged subcube's ability to embed functional subcubes of different sizes. Three different models were presented: one that considered node failures alone, another that considered only link failures, and a third that permitted both node and link failures. The latter two models have comparable complexity, so that the real choice is between the relatively simple node model and the more realistic, but complex combined node and link model. We have also suggested a technique to incorporate link failures into the nodes, that yields a simple but effective approximation to the combined model. While the effect of a single link failure is less catastrophic than that of a node failure, the larger number of links results in their having the dominant effect on system reliability, if nodes and links have comparable failure rates. These studies show that even though the d-cube structure is destroyed by the very first component failure, the cube is quite resilient in terms of its ability to support several smaller subcubes in the damaged structure.

REFERENCES

[1] S. Abraham and K. Padmanabhan, "Fault Tolerance and Reliability Analysis of the Hypercube Structure," To be published.

[2] J.R. Armstrong and F.G. Gray, "Fault Diagnosis in a Boolean n-Cube Array of Microprocessors," *IEEE Trans. on Computers*, Vol. C-30, No. 8, Aug. 1981, pp 587-590.

[3] M-S. Chen and K.G. Shin, "Embedding of Interacting Task Modules into a Hypercube," in *Hypercube Multiprocessors 1987*, pp 122-129.

[4] J.P. Hayes, T.N. Mudge, Q.F. Stout, S. Colley, and J. Palmer, "Architecture of a Hypercube Supercomputer," *Proc. 1986 Int. Conf. on Parallel Processing*, pp 653-660, Aug. 1986.

[5] M.T. Heath (Ed.), *Hypercube Multiprocessors 1987*, SIAM, Philadelphia, 1987.

[6] J. Lee, E. Shragowitz, and S. Sahni, "A Hypercube Algorithm for the 0/1 Knapsack Problem," *Proc. 1987 Int. Conf. on Parallel Processing*, pp 699-706, Aug. 1987.

[7] M. C. Pease, III, "The Indirect Binary n-Cube Microprocessor Array," *IEEE Trans. on Computers*, Vol. C-26, No. 5, pp. 458-473, May 1977.

[8] Y. Saad and M.H. Schultz, "Topological Properties of Hypercubes," Yale University, Department of Computer Science, Research report 389, June 1985.

[9] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, Vol. 28, No. 1, pp. 22-33, Jan. 1985.

[10] H. Sullivan and T. R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I," *Proc. 4th Symp. on Computer Architecture*, pp 105-117, Mar. 1977.

[11] B. Wagar, "Hyperquicksort: A Fast Sorting Algorithm for Hypercubes," in *Hypercube Multiprocessors 1987*, pp 292-299.

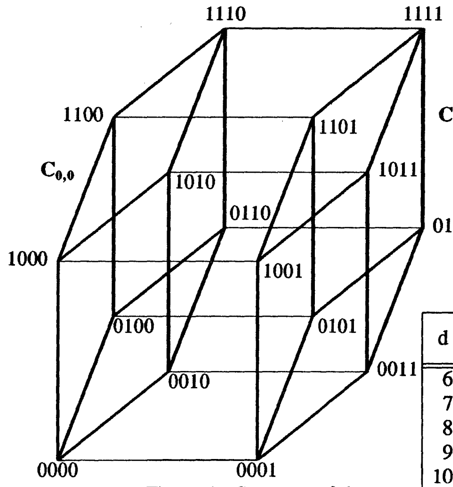


Figure 1. Structure of the binary 4-cube

Table 1. MTTF (Hours) for node failure model ($\lambda = 10^{-5}$ / hour)

d	N	T_*	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
6	64	1563	3150	3303	3726	4468	5611	7867						
7	128	781	1569	1612	1750	2014	2400	2968	4094					
8	256	391	783	795	839	934	1079	1272	1554	2118				
9	512	195	391	394	408	442	498	573	668	808	1090			
10	1024	98	195	196	201	212	234	264	301	348	418	559		
11	2048	49	98	98	99	103	111	124	139	157	180	216	286	
12	4096	24	49	49	49	51	54	59	65	73	81	93	111	146

Table 2. MTTF (Hours) for link failure model ($\lambda = 10^{-6}$ / hour)

d	N	T_*	T_{2-}	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
6	64	5208	11458	11740	12957	15298	18552	23434	33361						
7	128	2232	4836	4897	5213	5924	6945	8273	10284	14353					
8	256	977	2093	2106	2188	2406	2753	3184	3745	4601	6324				
9	512	434	922	925	947	1013	1132	1288	1471	1715	2088	2835			
10	1024	195	412	413	419	438	479	537	605	685	793	958	1287		
11	2048	89	186	187	188	194	207	229	256	285	321	369	443	591	
12	4096	41	85	85	85	87	92	100	110	122	135	151	173	206	273

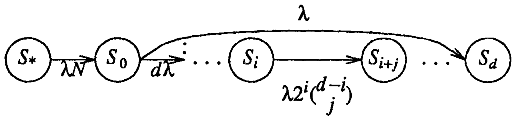


Figure 2. System state diagram (Node failure model)

Table 3. MTTF (Hours) for 2^{d-i} i-subcubes

d	N	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
10	1024	1652	1261	935	656	412		
11	2048	915	719	557	417	295	186	
12	4096	498	400	319	249	188	134	85

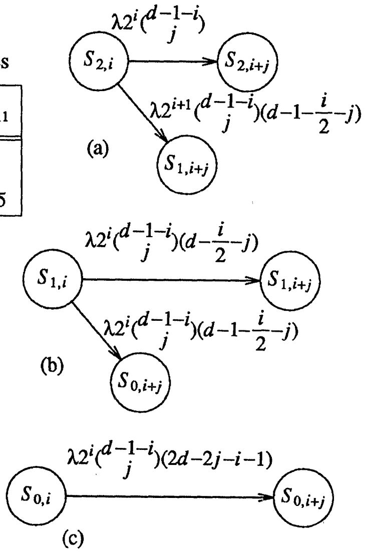

$\lambda 2^i \binom{d-1-i}{j}$
$S_{2,i}$
$S_{2,i+j}$
$\lambda 2^{i+1} \binom{d-1-i}{j} (d-1-\frac{i}{2}-j)$
(a)
$S_{1,i+j}$
$\lambda 2^i \binom{d-1-i}{j} (d-\frac{i}{2}-j)$
$S_{1,i}$
$S_{1,i+j}$
$\lambda 2^i \binom{d-1-i}{j} (d-1-\frac{i}{2}-j)$
(b)
$S_{0,i+j}$
$\lambda 2^i \binom{d-1-i}{j} (2d-2j-i-1)$
$S_{0,i}$
$S_{0,i+j}$
(c)

Figure 5. Transition rates for Fig. 4.

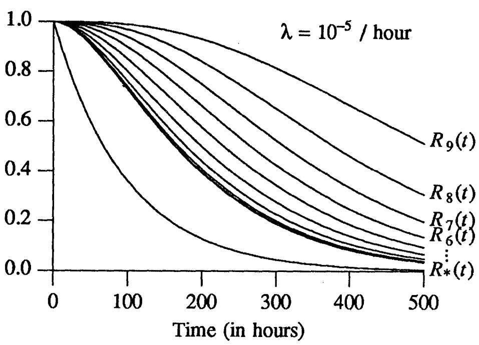


Figure 3. 10-cube reliabilities $R_i(t)$ (Node model)

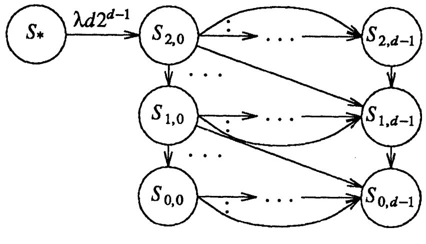


Figure 4. State diagram (Link model)

Table 4. MTTF (Hours) for combined model ($\lambda_l = 10^{-6}$ / hour, $\lambda_n = 10^{-5}$ / hour)

d	N	T_*	T_{2-}	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
6	64	1202	1490	2443	2599	2982	3603	4532	6369						
7	128	579	735	1170	1215	1341	1559	1862	2305	3187					
8	256	279	362	563	575	616	695	806	950	1162	1587				
9	512	135	178	271	275	288	316	359	412	481	583	787			
10	1024	65	88	131	132	136	146	163	184	209	242	291	390		
11	2048	32	43	63	64	65	68	75	83	93	105	121	145	193	
12	4096	15	21	31	31	31	32	35	38	42	47	53	61	72	95

Table 5. MTTF (Hours) for supernode failure model ($\lambda = \lambda_n + d\lambda_l/2$)

d	N	T_*	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
6	64	1202	2423	2541	2866	3437	4316	6051						
7	128	579	1162	1194	1296	1491	1778	2198	3033					
8	256	279	559	568	600	667	771	909	1110	1513				
9	512	135	270	272	282	305	344	395	461	558	752			
10	1024	65	130	131	134	142	156	176	201	232	279	373		
11	2048	32	63	63	64	67	72	80	90	101	116	139	185	
12	4096	15	31	31	31	32	34	37	41	45	51	58	69	91

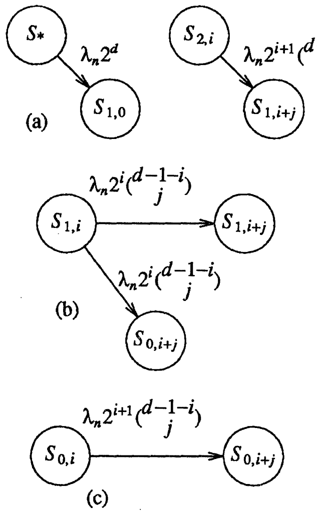

S_*
$\lambda_n 2^d$
(a)
$S_{1,0}$
$S_{2,i}$
$\lambda_n 2^{i+1} \binom{d-1-i}{j}$
$S_{1,i+j}$
$\lambda_n 2^i \binom{d-1-i}{j}$
$S_{1,i}$
$S_{1,i+j}$
$\lambda_n 2^i \binom{d-1-i}{j}$
(b)
$S_{0,i+j}$
$\lambda_n 2^{i+1} \binom{d-1-i}{j}$
$S_{0,i}$
$S_{0,i+j}$
(c)

Figure 6. Additional rates for combined model

SOLVING VISIBILITY PROBLEMS ON MCC'S

Mi Lu

Department of Electrical Engineering
Texas A&M University
College Station, TX 77843

Abstract — In this paper, we present MCC algorithms to solve the visibility problem for a set of disjoint simple objects such as line segments, circles, and simple polygons in the plane. For a collection of n such objects, our algorithms show how to compute, on a $\sqrt{n} \times \sqrt{n}$ MCC, a view of these objects in $O(\sqrt{n})$ time. Both parallel and perspective view are considered. The previous algorithms for computing the views are sequential and have $O(nlogn)$ time complexity [1].

For the above tasks, we also describe methods to solve problems of size n on MCC's with p processors, where $p < n$. Analysis will be given on the time complexity and the limitations imposed by the computational and communication requirements.

I. Introduction

The Mesh-Connected Computer operates as a single instruction stream, multiple data stream (SIMD) computer in which each PE can directly communicate with at most four neighbors. A $\sqrt{n} \times \sqrt{n}$ MCC consists of n identical PE's arranged on a two dimensional grid with processors at the grid points and connections between every horizontal and vertical pairs of PE's. Each PE has a constant number of storage registers, and each can perform standard arithmetic or boolean operations in $O(1)$ time. MCC have been widely used in different areas, and MCC algorithms have been designed to solve various problems [2-6].

An important and fundamental algorithmic problem in computer graphics is the following: Given a set of objects in three-dimensional space, compute the view from some fixed direction or point. The main issue is to eliminate all parts of the objects that cannot been seen (i.e., that lie behind some other object). It is a generalization of the hidden line problem in which objects have straight line edges. The problem also has numerous applications in the motion planning of robotics, and VLSI layout which have attracted considerable attention in the recent years. In our consideration, we simplify the preceding problem conceptually to two-dimensional space, not only because that it is often used as a subproblem in other geometric problems, (the shortest-path problem, for example,) but also due to that the solution for two-dimensional problem is the main part of the tree-dimensional solution, and will show directions for further research on the corresponding problems in three and higher-dimensional space.

In the rest of the paper, we present in Chapter II the MCC algorithms for solving visibility problems, and in Chapter III the problem solution on MCC's of smaller size. Chapter IV will give the conclusion remarks.

II. Solving Visibility Problem on MCC'S

The approach we use to solve the visibility problem is divide-and-conquer. Applying on the mesh, we divide the mesh into two submeshes of equal size, called left and right (or upper and lower respectively) submeshes. We recursively solve the two subproblems on two submeshes in parallel, and then combine the two subsolutions to obtain the final result. Elegant data movement need to be designed for the merge step to exploit the inherent concurrency.

2.1 The visibility problem

A *view* is a picture one sees looking from a direction or a point. A view from a point is a *perspective view*. In this case the view consists of a circle on which the parts of the objects one can see from the given point are projected. A view from a direction is a *parallel view*. In this case, the view consists of a line on which the parts of the objects that are visible from the direction are projected. Perspective views correspond to our natural way of viewing from a place close to the object set, and parallel views correspond to the viewing from a place far from the object set.

A *simple object* is a bounded convex object with the following properties:

1. Any parallel or perspective view of it can be computed in constant time.
2. If the views of two simple objects overlap, then constant time suffices to decide which one of the two objects can be seen entirely.
3. The up to four common tangents of two simple objects can be computed in constant time. Two objects that touch each other (i.e., objects whose boundaries overlap but do not cross) are by definition nonintersecting. Typical examples of simple objects are line segments, disks, and convex polygons with a bounded number of edges. A set of simple objects is shown in Figure 1. Figure 2 is the perspective view of them, and Figure 3 is the parallel view of them.

The *visibility problem* can be formally stated as follows: Given a set of n disjoint simple objects, and a point or a direction, report in order the parts of the objects that are visible from the given point or the given direction. A point p is *visible* from q if the line segment $\overline{pq}$ intersects no objects in the set.

The research work done on visibility problem are included in the following papers. For the case of a single polygon El Gindy and Avis [7] and Lee [8] presented $O(n)$ algorithms. Asano [1] gave an $O(n+hlogh)$ time algorithm for the case where the h disjoint polygons are convex, and an $O(nlogh)$ time algorithm for the general problem. It has been proved that for h disjoint polygons with n edges, the optimal time complexity to find the visibility polygon using $O(n)$ space is bounded by $O(n + hlogn)$ [9]. In [10], Edelsbrunner et al. used $O(n)$ search time, $O(n^2logn)$ preprocessing time and $O(n^2logn)$ space to solve the visibility polygon problem. Recently, Asano et al. [9] solved the visibility polygon problem in $O(n^2)$ preprocessing time, $O(n^2)$ space and $O(n)$ time. The visibility graph of disjoint polygons with n edges can be found by solving the visibility problem from each vertex of those polygons. This problem had been previously solved in time $O(n^2logn)$ by Lee [11] and recently by Welzl [12] and Asano et al. [9] independently in $O(n^2)$ time. The shortest path between two points in the plane with polygonal obstacles can be computed by applying Dijkstra's algorithm to the visibility graph of the obstacles. This problem is of current interest because it is an instance of a general class of important problems in robotics, known as *collision avoidance* problems (see, for example, Lozano-Perez and Sesley [13]).

2.2 Computing the parallel view

A parallel view of a set of objects consists of a partition of a line. Each part of the line corresponds to an object in the set (from the direction of view). The lowest part of the objects in the interval $(k, k+1)$ is visible, and we want to find the lowest part in all intervals, i.e., the *lower envelope* of the set of objects. (See Figure 4.)

To each part of the line we assign the index of the corresponding object. If the part corresponds to a place where one can look through the set we assign $NULL$ to it. It is possible that different parts of the line correspond to the same object, and hence, are assigned the same index (see, for example, object 2 in Fig. 4). A *partition point* corresponds to a leftmost point of an object (Fig. 5(A)) or a rightmost point of an object (Fig. 5(B)), with respect to the direction of view. (A part of the line might consist of one point if it corresponds to a line segment in the direction of view. In this case we treat it as a double partition point). It can be proved that the parallel view of a set of n objects from a fixed direction consists of at most $2n + 1$ parts and at most $2n$ partition points [10].

For computing the view of a set of objects from a fixed direction we will use a divide-and-conquer technique. Divide the set S of n objects into subsets A and B, each containing approximately equal number of objects. Let the partition points of a view of A be $\{a_0, a_1, \cdots, a_k\}$, $k \leq 2n-1$, and the parts of the view of A be $\{\overline{a_0a_1}, \overline{a_1a_2}, \cdots, \overline{a_{k-1}a_k}\}$. Similarly, let the partition points of a view of B be $\{b_0, b_1, \cdots, b_k\}$, $k \leq 2n-1$, and the parts of the view of B be $\{\overline{b_0b_1}, \overline{b_1b_2}, \cdots, \overline{b_{k-1}b_k}\}$. A part of the view of A, $\overline{a_ia_{i+1}}$, is a part of the view of $S = A \cup B$ iff projecting $\overline{a_ia_{i+1}}$ to the view does not cross any other objects, that is, no part of the view of B falls (even partially) in the interval of (a_i, a_{i+1}), or part $\overline{b_jb_{j+1}}$ falls in the interval but $\overline{a_ia_{i+1}}$ is "closer" to the observer than $\overline{b_jb_{j+1}}$. We can compute the view of $S = A \cup B$ by checking at each partition point of both views whether or not this point is also a partition point of the total view. As a result of the definition of a simple object, this checking can be done in constant time. Recursively partition the problem into two equal-sized subproblems, compute the views of the two subsets simultaneously, and merge the results. Assuming that the merge of two subproblems, whose sizes sum to n, needs time $M(n)$, the total time needed, $T(n)$, is given by the following recurrence:

$$T(n) = T(n/2) + M(n)$$

We show below that $M(n)$ is bounded by $O(\sqrt{n})$ in the mesh-connected computer implementation.

Distribute the partition points on an MCC, one point per PE. Since there are $2n$ partion points in a view of n objects, $2\sqrt{n} \times \sqrt{n}$ or $\sqrt{n} \times 2\sqrt{n}$ PE's are sufficient. The PE containing partition point a_i maintains the part $\overline{a_ia_{i+1}}$. Let a submesh of size 2^r be A, and its adjacent submesh be B. Submesh A contains the view of the subset $A \in S$, and submesh B contains the subset $B \in S$. The merging of the views of subsets A and B is performed recursively on submeshes A and B, which is of size 2^{r+1}. In iteration i, 2^{i+1} PE's are involved. The two phases involved in the merge are as follows:

(i) Each partition point a_i in the view of A finds the part $\overline{b_jb_{j+1}}$ such that a_i falls in the interval of $\overline{b_jb_{j+1}}$, and vice versa. (Fig. 6 is a reference.)

(ii) Decide whether $\overline{b_jb_{j+1}}$ prevents the part $\overline{a_ia_{i+1}}$ from being visible, and vice versa.

Phase (i) can be done by finding the difference of *global_rank* and *local_rank* of a_i, which tells the *local_rank* of b_j. We complete phase (ii) by a transformation of the coordinate system. Let the angle of the observing direction be α. Rotate the coordinate axes by α to obtain the new axes system. The point with the coordinates (x', y') under the old axis system will have coordinates (x, y) under the new axis system such that

$$\begin{cases} x' = x\ cos\alpha\ -\ y\ sin\alpha \\ y' = x\ sin\alpha\ -\ y\ cos\alpha \end{cases}$$

Then the view of a single object is just the horizontal line segment with the leftmost point and the rightmost point of the object as its end points. The projection of the object in subset A, say i, will across the object in subset B, say j, iff $y_i \geq y_j$, where y_i is the y coordinate of the leftmost point or rightmost point of object i and y_j is the y coordinate of the leftmost point or rightmost point of object j. The MCC algorithm will be given in algorithm *Parallel View*.

The record maintained in each PE includes the following field:

VIEW 1: Record

i /* index of the PE, can be also used as index of the object */

x, y /* coordinates of the leftmost or the rightmost point of the object */

see /* the index of the part in the view, $NULL$ if no object is visible */

local_rank /* indicating in r^{th} iteration the rank of the partition point in the view of 2^r objects */

```
    global_rank /* indicating in $r^{th}$ iteration the rank of
                   the partition point in the view of $2^{r+1}$
                   objects */
    base /* recording in $r^{th}$ iteration the $(logn + 1 - r)$
            MSB's of the PE index */
    biase/* temporary variable to record the local_rank of
            a partition point in the view to be merged */
end (/* of record */)
```

Algorithm Parallel View

```
1. Distribute the $n$ objects on the MCC, so that each consecutive two PE's contain the same object, say $i$.
2. PE($2k$) finds the leftmost point of the object it contains and PE($2k + 1$) finds the rightmost point of the object it contains, for $k = 0, 1, \cdots, n-1$. /* initialize the partition points */
Record the coordinates of the points found as $(x, y)$.
PE($2k$) sets $see = i$. PE($2k + 1$) sets $see = NULL$.
for $r$:=1 to $logn$ do the following:
3. $base = (logn + 1 - r)$ MSB's of $i$. /* represented by $b_k \cdots b_1 b_0$ */
4. Sort the partition points in non-decreasing order by $x$, on the $2^{r+1}$ submesh. Find the global_rank of each point.
5. Sort the partition points in non-decreasing order by $y$, on the $2^r$ submesh. Find the local_rank of each point.
6. Each PE compute
        $bias = global\_rank - local\_rank - 1.$
   and concatenates it to base with the LSB complemented (denoted as $base(\bar{b}_0)$).
7. Each PE performs a RAR from PE(addr) to get $x_{addr}$ and $see_{addr}$.
8. if $((y > y_{addr}) \wedge (see_{addr} = \neg NULL)) \vee ((y \leq y_{addr}) \wedge (see = NULL))$
        $see = see_{addr}$.
end (/* of algorithm Parallel View */)
```

Step 1 takes $O(\sqrt{n})$ time. Step 2 needs only constant time since that the objects are simple. Step 3 needs constant time also. Sorting in step 4 and step 5 requires $O(\sqrt{2^{r+1}})$ time and $O(\sqrt{2^r})$ time respectively. The time needed by steps 6 and 8 are constant. The RAR performed in step 7 uses time $O(2^r)$. Thus, the time required in iteration r is bounded by $O(\sqrt{2^r})$ and the total time needed in all the iterations is:

$$T(n) = \sqrt{2} + \sqrt{2^2} + \cdots + \sqrt{2^{logn}} = O(\sqrt{n}).$$

We have considered the view from a given direction, i.e., a *parallel view*. However, there is another interesting type of view, called a *perspective view*, which consists of the portions of the objects that are visible from a given point. The problem of finding the perspective view from an arbitrary point is discussed in the following section.

2.3 Computing the perspective view

Let S be a set of n simple objects, and q be an arbitrary query point. We want to find the parts of the objects in S that are visible from q, that is, find the *perspective view*.

A perspective view of a set of objects consists of a partition of a circle. Each circle segment corresponds to a part of an object that one can see from the fixed point or to a place where one can look through the set (see Fig. 7 as an example).

It is easy to verify that a perspective view contains at most $2n$ partition points and hence at most $2n + 1$ parts.

Consider a polar coordinate system with the point q as the origin and the positive y-axis as the reference. Denote the polar angle of a point p_i by $\theta(p_i)$, where the polar angle increases counterclockwise around q. The polar coordinates of a point can be represented as $(\rho,\ \theta)$. The leftmost point p_l or the rightmost point p_r of an object is the tangency point such that the line emanating from q is tangent to the object at it and $\theta(p_l) > \theta(p_r)$.

Computing the perspective view is similar to the computing of parallel view. A similar divide-and-conquer technique is adopted. If $p_0,\ p_1,\ \cdots,\ p_k,\ k \leq 2n-1$, denote the partition points, then $p_i \frown p_{i+1}$ indicates the part of the view. We will distribute the partition points on the mesh with each PE containing one partition point and maintaining the record of the part $p_j \frown p_{i+1}$.

The problem of finding the perspective view can be decomposed into the following two subproblems.
(i) Each partition point a_i in the view of A finds the part $b_j \frown b_{i+1}$ such that $\theta(b_j) \leq \theta(a_i) < \theta(b_{j+1})$, and vice versa.
(ii) Decide the visible part of the objects in the interval $(p_i,\ p_{i+1})$.
The visibility can be checked by determining which is the nearest to q among those objects with the ray emanating from q, extended from $\theta(p_i)$ to $\theta(p_{i+1})$, passed through its interior. We will show below that we can find, in polar order, the parts of the given set of n objects that are visible from q in $O(\sqrt{n})$ time. In fact, if we cut the plane along the ray emanating upwards from q and spread it out according to the angular and radial orders, the spread-out view is similar to the one we discussed in last section.

The transformation of the coordinate system needs not only a rotation but also a translation of the axes. Let (x', y') be the coordinates before the rotation, (x'', y'') the ones before the translation, and (h, v) the position of the observer in the old coordinate system. We have

$$\left\{\begin{array}{l} x' = x\ cos\alpha - y\ sin\alpha \\ y' = x\ sin\alpha - y\ cos\alpha \end{array}\right. ,$$

and

$$\left\{\begin{array}{l} x'' = x' + h \\ y'' = y' + v \end{array}\right.$$

In addition,

$$\left\{\begin{array}{l} \rho^2 = x^2 + y^2 \\ tan\ \theta = \frac{x}{y} \end{array}\right.$$

will complete transforming Cartesian system to polar system. We consider below the algorithm *Perspective View*, an

MCC algorithm for computing a perspective view from a given point.

In *Algorithm Perspective View*, the record *VIEW 2* maintained in each PE is similar to record *VIEW 1* given in section 2.2, except that the field "x, y" is changed to "ρ, θ". *Algorithm Perspective View* is identical to *Algorithm Parallel View* except for step 8. Following is a modified version of step 8.

8'. **if** $((\rho > \rho_{addr}) \wedge (see_{addr} = \neg NULL)) \vee ((y \leq y_{addr}) \wedge (see = NULL))$

$$see = see_{addr}.$$

The same analysis will show that the time needed to find a perspective view of a set of n simple objects is bounded by $O(\sqrt{n})$.

The visibility problem from a point for a set of h (not necessarily convex) disjoint polygons with n edges in total can be solved with the same time complexity by applying the above algorithms to the edges of those polygons. We first compute the visible portion of the boundary of each polygon from the point. The result is a sequence of edges from each polygon. Then the sequence can be decomposed where the visible parts in the view are found.

A visibility graph of n arbitrary oriented segments is a graph whose vertices are endpoints of those segments and whose edges are the straight line segments joining vertices that are visible from each other. This graph can be constructed by solving the visibility problem from each vertex for the given segments. As an application of this result, the shortest path between two points in the plane with polygonal obstacles having n edges can be solved by Dijkstra's algorithm, provided that the visibility graph is available.

III. Solving Problem on MCC'S of Smaller Size

The described results in the previous chapter were obtained using the unbounded model of parallel computation, i.e. we imposed no limit on the number of processors used by our algorithms. We discussed in several papers [14-17] the methods to solve some geometrical problems using MCC's of the same size as the size of the problem, that is, the number of the elements to be processed is equal to the number of the processors in an MCC. Obviously, in any practical situation we will be required to handle varying problem sizes with a fixed number of processors. The situation that the size of the MCC we have is smaller than the problem size occurs very often. We introduce in this chapter the algorithms to solve problems on the smaller size MCC's, and analyze their time complexity and the limitations imposed by the computational and communication requirements.

3.1 Basic approach

If a problem has n pieces of data initially distributed one per PE in a mesh of size n, we now consider what happens when we try to solve the problem on a mesh of size p, $1 \leq p \leq n$, where each PE is initially given $\frac{n}{p}$ pieces of data. This requires that the processors have sufficient memory to handle the largest problem size that will be encountered. A processor with its local storage is referred to as a node.

The basic approach to solve problems using MCC's of smaller size is to combine parallel and sequential processing on the MCC's. Previously developed MCC algorithms are used for inter-node processing, while sequential algorithms are used for intra-node processing. The algorithms include two phases:

(i) Each PE operates on the $\frac{n}{p}$ elements it contains independently. Each finds the partial result of the problem using the sequential algorithm.

(ii) PE's distribute their partial results to other processors, using the parallel merging algorithms discussed previously.

Of course, the time taken by each PE to broadcast its initial data is $\frac{n}{p}$ times as much as before.

As described above, n pieces of input data are distributed on a $\sqrt{p} \times \sqrt{p}$ MCC, with $\frac{n}{p}$ pieces of data per PE. Two sorting orders are considered, *consecutive order* and *cyclic order*. In *consecutive order*, $\frac{n}{p}$ successive elements of the sorted sequence are stored in each node, with successive sets of $\frac{n}{p}$ elements being stored in nodes in order of increasing node address (see Fig. 8(a)). In cyclic order, node i stores the elements in the set $\{j \mid i = rank(j) \bmod p\}$ such that $i = j \bmod p$. We describe below the details of sorting into consecutive order (see Fig. 8(b)). Rearrangement of consecutive to cyclic storage order (or vice versa) can be carried out in time $O(\frac{n}{p} + p)$, by pipelining the data transfers.

The sorting is carried out first by intra-node processing and then inter-node processing. A local sort is performed initially. Batcher's [18,19] odd-even merge is then mapped onto the MCC. When doing the local sort, each PE sorts the data it contains independently. This can be done by using sequential algorithms within $O(\frac{n}{p} log \frac{n}{p})$ time. After that, the $\frac{n}{p}$ elements stored in a node is a sorted sequence. The merge of the sorted sequences can be accomplished by inter-node processing in $O(\frac{n}{\sqrt{p}})$ time [20].

Similar to algorithms for performing RAR and RAW on MCC's with constant memory [3], our RAR and RAW algorithms on the MCC's with $\frac{n}{p}$ memory are performed using the well-defined operations *SORT, RANK, CONCENTRATE, DISTRIBUTE* and *GENERALIZE.* The time complexity for both RAR and RAW is bounded by $O(\frac{n}{\sqrt{p}} + \frac{n}{p} log \frac{n}{p})$ [20].

Since the previously described MCC algorithms for solving computational geometry problems are based on sorting, RAR and RAW techniques, the results of sorting, RAR and RAW on MCC's of smaller size provide the solutions of solving computational geometry problems on MCC's of smaller size.

3.2 Lower bound time and optimal size

We have presented the parallel algorithms for solving computational geometry problems on the mesh of unbounded model and on the mesh of smaller size. The new questions brought to our attentions are: what is the trade off between the time complexity and the number of processors? Is it true that the more processors we have, the less time are required?

Given n elements distributed on p processors with $\frac{n}{p}$ elements per PE, where $p < n$, Figure 9 shows the relationship of T versus $\sqrt{p}$. The line $T = \sqrt{p}$ indicates the influence of the diameter of the mesh. Two curves, $T = \frac{n}{p} logn$ and $T = \frac{n}{\sqrt{p}}$, are also given which indicate the time needed for inner-node processing and for intra-node processing respectively.

Since moving a data from, say, the upper left corner of a mesh to the lower right corner needs time no less than $2\sqrt{n}$, T should be greater than $\sqrt{n}$. In the meantime, we can find that $\frac{n}{\sqrt{p}} > \frac{n}{p} logn$ in general. Thus our working area is above the line $T = \sqrt{p}$ and the curve $T = \frac{n}{\sqrt{p}}$. It can be observed that the minimal T is given at the point $(n, \sqrt{n})$. That means, if n processors are provided, we can obtain the optimal time complexity which is $O(\sqrt{n})$.

The sequential algorithms to solve the previous geometric problems have an optimal time complexity of $O(nlogn)$. With p processors, $O(\frac{n}{p} logn)$ time performance is desired. However, it can be realized only when $p \leq log^2 n$, where $\frac{n}{\sqrt{p}} > \frac{n}{p} logn$. The number of the processors used are very few in that case and the utility of each processor is 100%, although the time needed is greater than $O(\sqrt{n})$.

When $p > log^2 n$, our working area is bounded by $T = \frac{n}{\sqrt{n}}$ and the processors can not be utilized with 100% efficiency. This is because of the bandwidth limitation. The machine model we used is a limited-connectivity processor network. We avoided the complicate interconnections in the machine building at the cost of the loss in time performance.

When more than n processors are given where n is the size of the problem, we can find in surprise that T increases as p increases. It demonstrates that to put more than n processors in operation is just a west. We can not gain anything in time performance, because of that the time complexity is bounded by the diameter of the mesh.

In a word, $0 < p < log^2 n$ corresponds to the computation bound region in Fig. 9, $log^2 n < p < n$ corresponds to the connection bound region, and $p > n$ corresponds to the diameter bound region.

IV. Conclusions

Parallel MCC algorithms for solving visibility problems are presented. Given a set of n simple objects in the plane, our algorithms can find a parallel view or a perspective view of them on a $\sqrt{n} \times \sqrt{n}$ MCC, and have the optimal $O(\sqrt{n})$ time complexity. Methods for solving above tasks on MCC's with p processors, is also described, where $p < n$. We analyzed their time complexity and the limitations imposed by the computational and communication requirements. The result is considerable significant since it provides a well performed approach for solving a general problem often occured in practical MCC applications.

References

[1] T. Asano, "Efficient algorithms for finding the visibility polygons for a polygonal region with holes," manuscript, Dept. of Electrical Engineering and Computer Science, University of California at Berkeley, 1984.

[2] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," Commun. ACM, vol. 20, no. 4, Apr. 1977, pp. 263-271.

[3] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," IEEE Trans. on Computers, Vol. C-27, no. 2, 1979, pp. 2-7.

[4] A D. Nassimi and S. Sahni "Finding connected components and connected ones on a mesh-connected Parallel Computer," SIAM J. Comput., Vol. 9, No. 4, 1980, pp. 744-757.

[5] M. J. Atallah and S. R. Kosaraj, "Graph problems on a mesh-connected processor array," J. of ACM, Vol. 31, no. 3, July, 1984, pp. 649-667.

[6] R. Miller and Q. F. Stout, "Computational geometry on a mesh-connected computer," Proc. of 1984 Int. Conf. on Parallel Processing, 1984, pp. 66-73.

[7] H. El Gindy and D. Avis, "A linear algorithm for computing the visibility polygon from a point," J. of Algorithms, Vol. 2, 1981, pp. 186-197.

[8] D. T. Lee, "Visibility of a simple polygon," *Computer Vision, Graphics, and Image Processing*, Vol. 22, 1983, pp. 207-221.

[9] Takao Asano, Tetsuo Asano, L. Guibas, J. Hershberger and H. Imai, "Visibility-polygon search and Euclidean shortest paths," Proc. of Symp. Found. Comput. Sci., 1985, pp. 155-164.

[10] H. Edelsbrunner, M. H. Overmars and D. Wood, "Graphics in flatland: a case study," Advances in Computing Research, F. P. Preparata, ed., Vol. 1, JAI Press Inc., 1983, pp. 35-59.

[11] D. T. Lee, "Proximity and reachability in the plane," Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1978.

[12] E. Welzl, "Constructing the visibility graph for n line segments in $O(n^2)$ time," Inform. Processing Lett., Vol. 20, 1985, pp. 167-171.

[13] T. Lozano-Perez and M. A. Wesley: "An algorithm for planning collision-free paths among polyhedral obstacles," Commun. ACM, Vol. 22, 1979, pp. 560-570.

[14] M. Lu and P. Varman, "Solving geometric proximity problems on mesh-connected computers," Proc.of 1985 IEEE Comp. Soci. Workshop on Comp. Architec. for Pattern Analysis and Image Database Management, Miami Beach, Florida, Nov. 1985, pp. 248-255.

[15] M. Lu, Ph.D. Dissertation, Department of Electrical and Computer Engineering, Rice University, 1987.

[16] M. Lu and P. Varman, "Optimal algorithms for rectangle-intersection problems on a mesh-connected computer," Journal of Parallel and Distrib. Comput., 5, 1988, pp. 154-171.

[17] M. Lu and P. Varman, "Geometric problems on two-dimensional array processors," Circuit, Control and Signal Processing, Vol. 7, No. 2, 1988, pp. 191-211.

[18] K. E. Batcher, "Sorting networks and their applications," Proc. AFIPS 1968 SJCC, Vol. 32, AFIPS Press, Montvale, N. J., pp. 307-314.

[19] D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[20] M. Lu, "Solving problems on MCC's of smaller size," to be published.

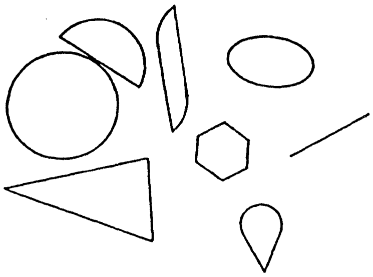


Figure 1

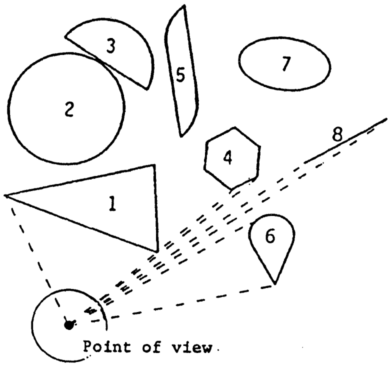


Figure 2. A perspective view of a set of objects.

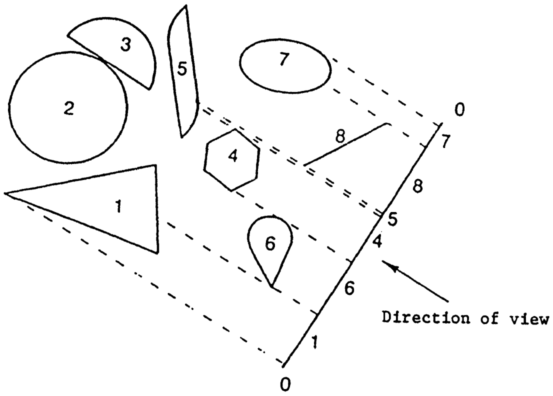


Figure 3. A parallel view of a set of objects

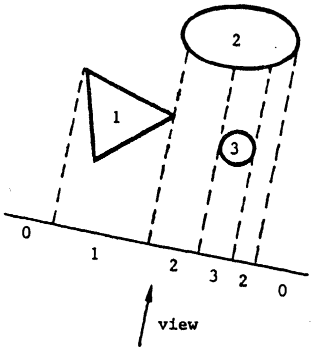


Figure 4

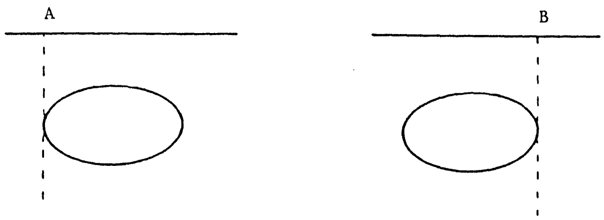


Figure 5. Two partition points: (A) at the leftmost point of an object and (B) at the rightmost point of an object.

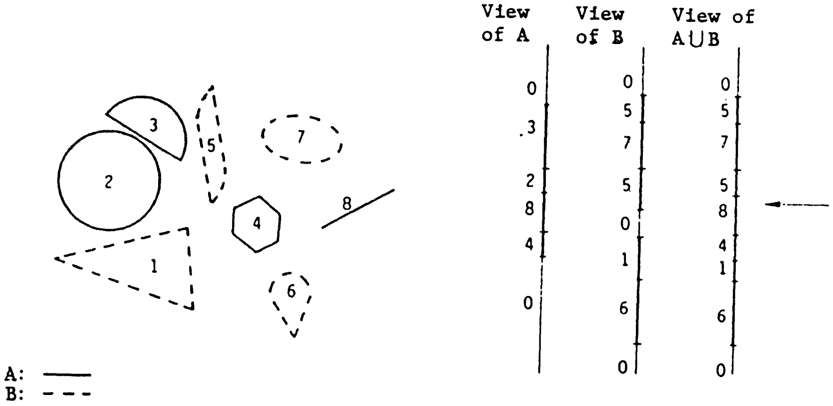


Figure 6

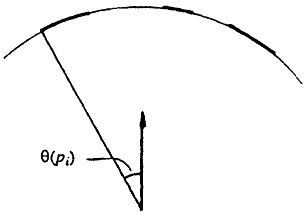
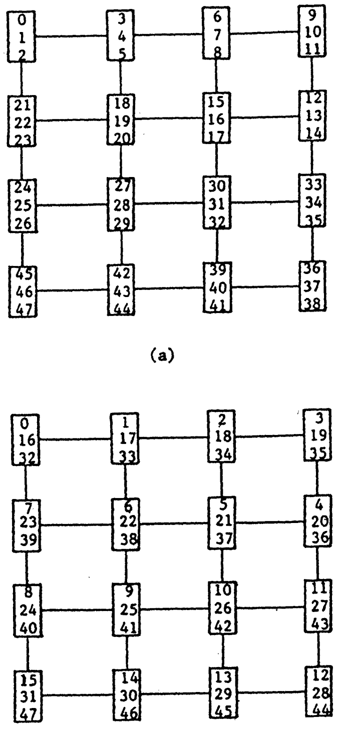


Figure 7



(a)

(b)

Figure 8

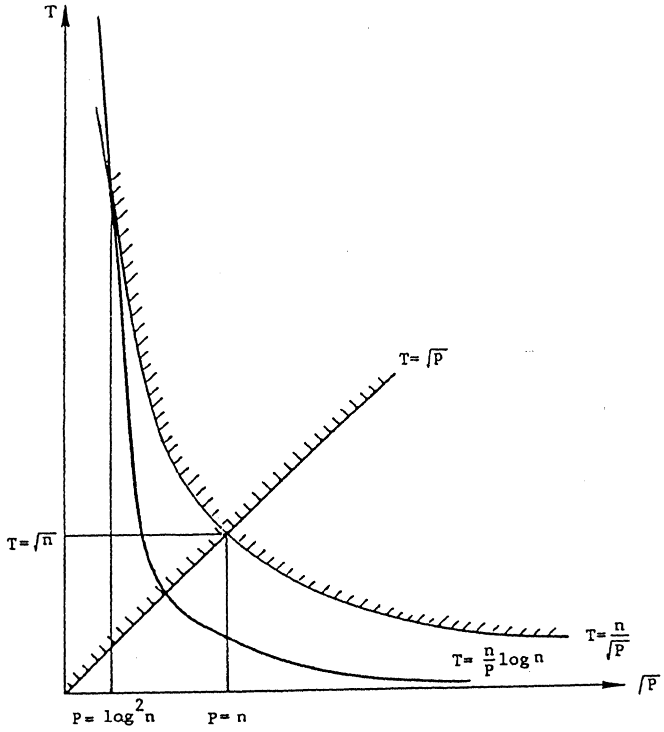


Figure 9

On Array Storage For Conflict-Free Memory Access For Parallel Processors[1]

Meera Balakrishnan, Rajiv Jain and C. S. Raghavendra
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-0781

Abstract

In a parallel processing system with N processors sharing N memory modules, storing a matrix for conflict free access is an important problem. In this paper we propose a method for storing an $N \times N$ matrix in N memory modules such that any row, column, forward or backward diagonal of the matrix can be accessed by the processors without conflicts. It is shown that this problem is similar to the Magic Square Puzzle, and algorithms are presented for storing the matrix when $N = 2^n$ for $n \geq 2$ and for odd N.

1 Introduction

Pipelined machines and array processors depend on an uninterrupted flow of data for high performance and hence the organization of vector elements in the memory modules is of prime importance. Any conflicts in data fetch can severely degrade the performance of these machines. For example, a matrix application program may generate an access request for a row of an $N \times N$ matrix (vector) stored in N memory banks. If all the N vector elements are in one memory module, N separate memory accesses will be required to retrieve an N element vector. On the other hand, a single access is sufficient if the vector elements are spread across N memory modules. In the case of memory conflicts additional cycles are required to resolve the conflict. The processor-memory speed will no longer be balanced, reducing the processor speed by a factor proportional to the number of conflicts in any memory module.

In our model of a parallel processing system, we consider $N = 2^n$, $n \geq 2$, processors connected to N memory modules by an interconnection network, as shown in Fig. 1. Given an $N \times N$ matrix, we have to find a storage organization for each element a_{ij} of the matrix, such that the elements of any row, column, forward or backward diagonal of the matrix are in different memory modules and can be accessed simultaneously. The problem, thus, is to find a mapping f such that for every element a_{ij}, $f(i,j) = m$. Here m is the memory module number in which a_{ij} is stored and f is such that rows, columns and diagonals can be retrieved in a single memory access.

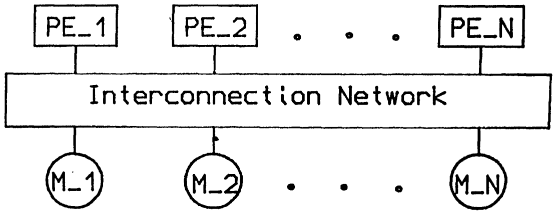


Fig.1 Parallel Processing System Model

2 Definitions And Related Research

In the remainder of this paper, $N = 2^n$ denotes the number of processors and M denotes the number of memory modules.

Many authors [3,4,5,6,7,8] have addressed the issue of conflict-free access to various vectors (ex. rows, columns, diagonals) of a matrix. The function f which assigns a memory module to each element of the matrix is called the *skew function*. f is a linear skew function if the indices of the element a_{ij} form a linear combination of the type $k_1 i + k_2 j$, where k_1 and k_2 are integer constants. Lawrie [5] has shown that for $M = N$, and even value of M, it is not possible to find a linear skew function for conflict-free access to the rows, columns and diagonals of an $N \times N$ matrix. Budnick and Kuck [3] have shown that for an $N \times N$ matrix ($N = 2^{2k}$, k is a positive integer), a linear skew function cannot provide conflict-free access to the rows, columns, diagonals and $N^{\frac{1}{2}} \times N^{\frac{1}{2}}$ blocks when stored in $N = M$ memory modules.

Furthermore, Shapiro [6] has claimed that for rows, columns and diagonals (forward and backward), if there does not exist a linear skew function for $M = N = 2^n$ for providing conflict-free access, then, there is no valid skewing scheme of any type whatsoever. Deb [4] has provided a counter example (Fig. 2) which shows that it is indeed possible to store a 4×4 matrix (i.e. $M = N = 2^n$, $n = 2$) with conflict-free access to rows, columns and the main forward and backward diagonals.

In the remainder of the paper we show that an $N \times N$ matrix can be skewed for conflict-free access to rows, columns and the main forward and backward diagonals for $N = 2^n$ for $n \geq 2$ and for odd N by mapping the problem to a variation of the famous Magic Square Puzzle. We assume that $M =$

[1] This research was supported by the NSF (Contract No. MIP-8452003) with matching funds from the AT&T Information Systems.

N. The following paragraph defines the mapping to the Magic Square Puzzle.

Fig. 3a shows a 4 × 4 matrix *without* skewing and with the linear address of each element marked. Fig. 3b shows the elements in their skewed positions as determined by some (unknown) skew function. Let us reduce the linear addresses of Fig. 3b to their *mod-4* values, add 1 and generate the matrix in Fig. 3c. It can be seen that the numbers 1, 2, 3 and 4 occur in each row, column and main diagonals exactly once. If we superimpose Fig. 3c on Fig. 3a to generate the matrix in Fig. 3d, we see that associated with each element of the unskewed matrix is a number that assigns it to a memory module. Notice that the skewing has made no change in the row index of the element. This assignment results in the skewed matrix in Fig. 3b.

We see that for any $N \times N$ matrix, if there exists another $N \times N$ matrix A whose elements only take on values $1, 2, \cdots, N$ such that each value occurs exactly once in any row, column or diagonal (or any other desired vector), then each of these vectors of length N can be accessed without conflict when stored in N parallel memory modules. The matrix A is called the $N \times N$ *assignment matrix.* Fig. 3c is an example of a 2 × 2 assignment matrix.

The following definitions are valid for $N = 2^n$.

Definition 2.1 *Let a vector* $(1, 2, ..., N)$ *be partitioned into equal segments of size* $k = 2^e$, *for some integer* $e \geq 0$. *Then the* k-segment image permutation *of the vector is defined for* $1 \leq x \leq N$, *as*

$$\pi_k(x) = 2k\lceil \frac{x}{k} \rceil + 1 - (x + k)$$

As an example, the following is a *2-segment image* permutation for $N = 8$.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 1 & 4 & 3 & 6 & 5 & 8 & 7 \end{pmatrix}$$

In the following definition a sequence of integers which occurs in the construction of the assignment matrix is defined.

Definition 2.2 *Let the bit representation of integer* x, $(1 \leq x \leq N-1)$, *be* $x_n x_{n-1} \cdots x_2 x_1$. *Consider the representation of all integers from 1 to* $2^n - 1$ *arranged in ascending order such that each integer occurs exactly once. Let* x_i *bit of the representation be given a weight of* 2^i. *Then the* j^{th} *element of the* sequence *is defined as the weight of the least significant occurrence of 1 in the binary representation of the* j^{th} *integer in the sorted list.*

For example, if $n = 3$, the sequence of length 7 is (2, 4, 2, 8, 2, 4, 2). Also, *sequence(i)*, $1 \leq i \leq N-1$, returns the i^{th} element of this sequence.

0	1	2	3
a11	a12	a13	a14
a23	a24	a21	a22
a34	a33	a32	a31
a42	a41	a44	a43

Fig.2 Counter Example

$\boxed{0}$ a11	$\boxed{1}$ a12	$\boxed{2}$ a13	$\boxed{3}$ a14
$\boxed{4}$ a21	$\boxed{5}$ a22	$\boxed{6}$ a23	$\boxed{7}$ a24
$\boxed{8}$ a31	$\boxed{9}$ a32	$\boxed{10}$ a33	$\boxed{11}$ a34
$\boxed{12}$ a41	$\boxed{13}$ a42	$\boxed{14}$ a43	$\boxed{15}$ a44

(a) Unskewed Matrix with Linear Addresses

$\boxed{0}$ a11	$\boxed{1}$ a12	$\boxed{3}$ a14	$\boxed{2}$ a13
$\boxed{6}$ a24	$\boxed{7}$ a23	$\boxed{5}$ a21	$\boxed{4}$ a22
$\boxed{9}$ a32	$\boxed{8}$ a31	$\boxed{10}$ a33	$\boxed{11}$ a34
$\boxed{15}$ a43	$\boxed{14}$ a44	$\boxed{12}$ a42	$\boxed{13}$ a41

(b) Skewed Matrix with Linear Addresses

1	2	4	3
3	4	2	1
2	1	3	4
4	3	1	2

(c) Linear Addresses After Taking Modulus 4 and Adding 1

$\boxed{0}$ a11	$\boxed{1}$ a12	$\boxed{3}$ a13	$\boxed{2}$ a14
$\boxed{2}$ a21	$\boxed{3}$ a22	$\boxed{1}$ a23	$\boxed{0}$ a24
$\boxed{1}$ a31	$\boxed{0}$ a32	$\boxed{2}$ a33	$\boxed{3}$ a34
$\boxed{3}$ a41	$\boxed{2}$ a42	$\boxed{0}$ a43	$\boxed{1}$ a44

(d) Matrix with Memory Module Assignment

Fig. 3 Memory Module Assignment for 4*4 Matrix

The rows of the assignment matrix A are numbered from 1 (topmost) through N (bottom) and the columns are numbered from 1 (leftmost) through N (rightmost).

3 Proposed Solution

In the *Magic Square Puzzle* [2] numbers ranging from 1 to N^2 are entered in an $N \times N$ matrix such that the sum of each row, column or diagonal is a constant. Our problem requires numbers ranging from 1 to N to be entered such that each row, column and diagonals add up to $\frac{N(N+1)}{2}$. We solve the problem by breaking it into two cases, namely, $N = 2^n$ for an integer $n \geq 2$ and an odd value of N. We shall consider the former case first as the latter case has a trivial solution.

3.1 $N = 2^n$, $n \geq 2$

Benson and Jacoby [2] have generated all possible magic squares of the 4^{th} order (ie. 4×4 matrix) using numbers ranging from 1 to 16 exactly once. This was done in the 1970's with the aid of the computing facilities at Dickinson College, Carlisle, Pennsylvania [2]. Exactly 880 such magic squares were generated by the computer. This verified the result claimed by Frenicle who published the 880 magic squares in 1693.

These squares have been classified into twelve basic types, depending on the relation between numbers of the same row, column, forward diagonal or back-diagonal [2]. One of the basic types is shown in Fig. 4. Arcs have been drawn between elements whose sum is a constant. Also, each column adds up to a constant. This type is the most relevant of the squares-types to our problem of generating the assignment matrix for conflict-free access. Compare Fig. 4 with Fig. 3c. Using this clue we were able to systematically generate the skewed matrix for $N = 2^n, n \geq 2$.

Fig. 5 shows the assignment matrices for $N = 4$, $N = 8$, and $N = 16$. These matrices are divided into four quadrants, of $\frac{N}{2} \times \frac{N}{2}$ cells each. We observe that some properties of the assignment matrix help to simplify its construction.

Property 1: Exactly 1 through N numbers are used as valid entries.

Property 2: Each of the numbers occur exactly once in any row, column or diagonal.

Property 3: Every two numbers specified by the arcs in Fig. 4 add up to $N+1$. Thus all entries in any of the upper (lower) quadrants can be generated if the entries in the corresponding lower (upper) quadrants are known. That is, $A[N+1-i][j] = N+1-A[i][j]$, where $(1 \leq i \leq \frac{N}{2})$.

Property 4: Every column i, $(1 \leq i \leq \frac{N}{2})$, in the *left* quadrant (upper or lower) is a $\frac{N}{2}$ - *segment image permutation* of the $(N+1-i)$ column of the corresponding (upper or lower) *right* quadrant.

It is therefore sufficient to generate any one quadrant of the $N \times N$ matrix. The remaining quadrants can be easily generated from this quadrant as shown by the above properties. Algorithm 1 gives the construction of the $N \times N$ assignment matrix A. The algorithm is coded in the C programming language.

Lemma 3.1.1 *If any vector X of size 2^n is a permutation in the range 1 to 2^n then, a vector Y which is computed as*

$$Y[i] = 2^{n+1} + 1 - X[i]; 1 \leq i \leq 2^n$$

is a permutation in the range $(2^n + 1)$ to 2^{n+1}.

Theorem 3.1.1 *Algorithm 1 generates an $N \times N$ assignment matrix such that each integer between 1 and N occurs exactly once in every row, column, forward and backward diagonal.*

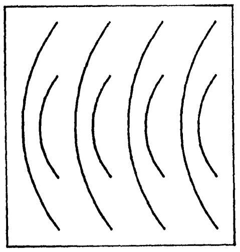


Fig.4 A Basic Type of a 4*4 Magic Square

Proof : We shall individually prove that every row, column and the forward and backward diagonal is a permutation. Each of these proofs is done using the induction technique.

We shall prove that the forward diagonal elements form a permutation. The proof for the backward diagonals can be done using similar reasoning. Fig. 6a shows the structure of the assignment matrix for $N = 2^i$, and Fig 6b. for $N = 2^{i+1}$. Tracing through the algorithm, we observe that it maps the shaded areas of the Fig. 6a to the corresponding shaded areas in Fig. 6b.

Basis : $N = 4$. The assignment matrix given in Fig. 5a is generated from the algorithm. The diagonal elements form a permutation on numbers in the range 1 to 4 in the $2^2 \times 2^2$ assignment matrix.

Hypothesis : The diagonal elements form a permutation on numbers in the range 1 to 2^i in a $2^i \times 2^i$ assignment matrix (Fig. 6a).

Induction Step : Consider the blocks along the forward diagonal of Fig. 6b. The diagonal elements of the shaded blocks form a permutation in the range 1 to 2^i (induction hypothesis). We shall now show that the values along the forward diagonal of the unshaded blocks form a permutation in the range $2^i + 1$ to 2^{i+1}. Notice that the diagonal elements of the unshaded blocks are generated from the elements of the back diagonal of the matrix of Fig. 6a according to the following equation:

$$A[j][k] = 2^{i+1} + 1 - A[2^i + 1 - j][k]$$

$\forall j, k \in$ diagonal elements of the unshaded blocks.

Since $A[2^i + 1 - j][k]$ lists the elements of the back diagonals of the assignment matrix of dimension $2^i \times 2^i$ and is a permutation in the range 1 to 2^i, the diagonal elements of the unshaded blocks must be a permutation in the range $2^i + 1$ to 2^{i+1} (by Lemma 3.1.1). This completes the proof that the forward diagonal is a permutation.

We shall now prove that every column of Fig. 6b forms a permutation.

Basis : For $N = 4$, every column of the 4×4 matrix of Fig. 5a is a permutation.

Hypothesis : The columns of the $2^i \times 2^i$ matrix shown in Fig. 6a are permutations in the range 1 to 2^i.

Induction Step : Consider a single column of blocks in Fig. 6b. From the induction hypothesis the columns of the shaded blocks form a permutation in the range 1 to 2^i. From Property 3 (Section 3) it is clear that each shaded block generates one unshaded block according to

$$A[j][k] = 2^{i+1} + 1 - A[2^i + 1 - j][k]$$

$\forall j, k \in$ unshaded blocks. Therefore $A[j][k]$ are unique in the range 2^i+1 to 2^{i+1} and by Lemma 3.1.1 form a permutation.

Finally, we shall prove that every row of the assignment matrix A also forms a permutation.

Basis : By inspection of Fig. 5a, for $N = 4$, every row is a permutation.

Hypothesis : The rows of a $2^i \times 2^i$ assignment matrix (Fig. 6a) are permutations. We first show that rows of the lower right quadrant when concatenated with the rows of the upper left quadrant form a permutation in the range 1 to 2^i (Fig. 6a). The reasoning is as follows. By Property 4 (Section 3), the lower left quadrant is generated from the lower right quadrant by performing an $\frac{N}{2}$- segment image permutation on the columns of the lower right quadrant. This transformation maps row j of the lower right quadrant to row $(N + 1 - j)$ of the lower left quadrant $(1 \leq j \leq \frac{N}{2})$. Thus the numbers in row $(\frac{N}{2} + 1 - j)$ of the lower left quadrant are identical (though not in the same order) with those in row j of the lower right quadrant. The numbers in row j of upper left quadrant are obtained from row $(\frac{N}{2} + 1 - j)$ of lower left quadrant by Property 3. Therefore, row j of the upper left quadrant cannot have numbers in common with the numbers in row $(\frac{N}{2} + 1 - j)$ of the lower left quadrant, and hence are also different to the numbers in row j of the lower right quadrant. Furthermore, row j of the upper left quadrant is a permutation. This proves that the rows of the lower right quadrant when abutted with the rows of the upper left quadrant, form a permutation in the range 1 to 2^i. We will use this result in the induction step to show that the rows of Fig. 6b are indeed permutations.

Induction Step : We have proved in the hypothesis step that the rows of the upper left quadrant and the lower right quadrant of Fig. 6a when abutted form a permutation. Thus, the shaded top row of Fig. 6b is a permutation. By Lemma 3.1.1, the bottom unshaded row in Fig. 6b is a permutation in the range $2^i + 1$ to 2^{i+1}. Thus the whole of the bottom row, the shaded as well as the unshaded, together form a permutation. Using a similar argument, every row of the assignment matrix can be shown to be a permutation.

Hence every row, column and the forward and backward diagonal of the assignment matrix A generated by Algorithm 1 is a permutation. ■

Algorithm 1 has a complexity of $O(N^2)$ as each cell of the matrix A is visited exactly once and there are N^2 cells. The assignment matrix is superimposed on the data matrix to obtain the memory module assignment for each element of the data matrix for conflict-free access to rows, columns and main diagonals. That is, if the number corresponding to a_{ij} is m in the assignment matrix, then element a_{ij} is stored in memory module m.

```
Nby2 = N / 2;
/*Generate the lower-right quadrant*/
for (i = 1; i ≤ N; i += 2) {
    i1 = Nby2 + ((i - 1) / 2);
    A[i1][Nby2] = N - i;
    A[i1][Nby2 + 1] = N - i + 1; }
for (i = 1; i < (Nby2 - 1) / 2 + 1; i++) {
    i1 = sequence[i - 1];
    x = Nby2 + (i - 1) * 2;
    y = Nby2 + (i * 2);
    for (a = 0; a < (Nby2 / i1); a++)
        for (b = 0; b < i1; b++) {
            A[Nby2 + (a + 1) * i1 - b - 1][y] =
            A[Nby2 + (a * i1) + b][x];
            A[Nby2 + (a + 1) * i1 - b - 1][y + 1] =
            A[Nby2 + (a * i1) + b][x + 1]; }
}
/* Generate the remaining quadrants */
for (i = 0; i < Nby2; i++)
    for (i1 = 0; i1 < Nby2 / 2; i1++) {
        A[i1 + Nby2 / 2][i] = A[i1 + Nby2][i + Nby2];
        A[i1][i] = A[i1 + (3 * Nby2) / 2][i + Nby2];}
for (i = 0; i < Nby2; i++)
    for (i1 = 0; i1 < Nby2; i1++) {
        A[i1 + Nby2][i] = N + 1 - A[Nby2 - 1 -i1][i];
        A[i1][i + Nby2] = N + 1 - A[N - 1 - i1][i + Nby2];}
```

Algorithm 1: Generation of Assignment Matrix

```
procedure generate-quadrant($N$)
{
   $y = 1; i = 0; j = \lceil \frac{N}{2} \rceil$:
   $A[i][j] = y$:
   while (there is an empty cell) {
      $y = (y)mod(N) + 1$:
      if ($A[(i-1)mod(N)][(j+1)mod(N)]$ is empty ) {
         $i = (i-1)mod(N)$:
         $j = (j+1)mod(N)$:
         $A[i][j] = y$:
      }
      else {
         $i = (i+1)mod(N)$:
         $A[i][j] = y$:
      }
   }
}
```

Algorithm 2: Procedure to generate assignment matrix for odd N

3.2 Odd Values of N

In this case, where N is odd, the solution of the magic square problem is trivial. Let the cells of the $N \times N$ matrix be labelled as a_{ij}, where $0 \leq j \leq (N-1)$. Algorithm 2 generates the desired assignment matrix. The proof that Algorithm 2 generates a permutation can be found in [2]. An example of an assignment matrix for $N = 7$ is given in Fig. 7.

4 Conclusion

In this paper we have presented an algorithmic solution to the problem of aligning data for conflict-free access to rows, columns and the main diagonals. The method presented in this paper for generating module numbers can be used in the table lookup technique as employed in *GF*11 [1]. However, it is not practical for large matrices. A skew function therefore needs to be extracted from the information in the assignment matrix so that, given the indices, the function assigns the memory module. It appears that this function is nonlinear. An alignment interconnection network that will implement the permutation defined by the skew function can then be synthesized. The feasibility of modifying the assignment matrix to accommodate conflict-free access to broken diagonals and $N^{\frac{1}{2}} \times N^{\frac{1}{2}}$ sub-matrices, also needs to be studied. Furthermore, the possibility of N taking any even value, not necessarily a power of 2 needs to be considered.

References

[1] J. Beetem et. al, **The GF11 Supercomputer**, *12th. Annual Inter. Symp. on Comp. Arch.*, 1985.

[2] W. H. Benson and O. Jacoby, **New Recreations With Magic Squares**, *Dover Publications Inc.*, New York, 1976.

[3] P. Budnick and D. J. Kuck, **The Organization and Use of Parallel Memories**, *IEEE Trans. Comp.*, Vol.C-20, No.12, Dec. 1971.

[4] A. Deb, **Conflict-Free Access of Arrays - A Counter Example**, *Inf. Proc. Letters*, Vol.10, No.1, Feb 1980.

[5] D. H. Lawrie, **Access and Alignment of Data in an Array Processor**, *IEEE Trans. Comp.*, Vol.C-24, No.12, Dec. 1975

[6] H. D. Shapiro, **Theoretical Limitations on the use of Parallel Memories**, *Ph.D. Thesis*, University of Illinois at Urbana-Champaign, 1976.

[7] H. D. Shapiro, **Theoretical Limitations on the Efficient Use of Parallel Memories**, *IEEE Trans. Comp.*, Vol.C-27, No.5, May 1978.

[8] H. A. G. Wijshoff and J. Van Leeuwen, **On Linear Skewing Schemes and *d*-Ordered Vectors**, *IEEE Trans. Comp.*, Vol C-36 No.2, Feb. 1987.

1	2	4	3
3	4	2	1
2	1	3	4
4	3	1	2

(a) N = 4

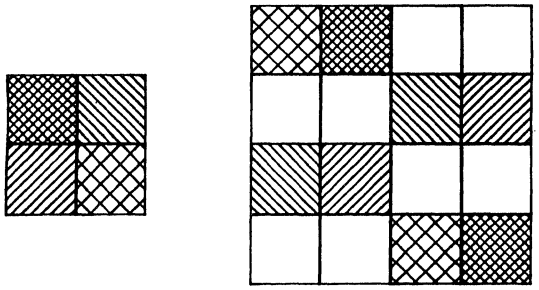
3	4	1	2	8	7	6	5
1	2	3	4	6	5	8	7
7	8	5	6	4	3	2	1
5	6	7	8	2	1	4	3
4	3	2	1	7	8	5	6
2	1	4	3	5	6	7	8
8	7	6	5	3	4	1	2
6	5	8	7	1	2	3	4

(b) N = 8

7	8	5	6	3	4	1	2	16	15	14	13	12	11	10	9
5	6	7	8	1	2	3	4	14	13	16	15	10	9	12	11
3	4	1	2	7	8	5	6	12	11	10	9	16	15	14	13
1	2	3	4	5	6	7	8	10	9	12	11	14	13	16	15
15	16	13	14	11	12	9	10	8	7	6	5	4	3	2	1
13	14	15	16	9	10	11	12	6	5	8	7	2	1	4	3
11	12	9	10	15	16	13	14	4	3	2	1	8	7	6	5
9	10	11	12	13	14	15	16	2	1	4	3	6	5	8	7
8	7	6	5	4	3	2	1	15	16	13	14	11	12	9	10
6	5	8	7	2	1	4	3	13	14	15	16	9	10	11	12
4	3	2	1	8	7	6	5	11	12	9	10	15	16	13	14
2	1	4	3	6	5	8	7	9	10	11	12	13	14	15	16
16	15	14	13	12	11	10	9	7	8	5	6	3	4	1	2
14	13	16	15	10	9	12	11	5	6	7	8	1	2	3	4
12	11	10	9	16	15	14	13	3	4	1	2	7	8	5	6
10	9	12	11	14	13	16	15	1	2	3	4	5	6	7	8

(c) N = 16

Fig. 5 Assignment Matrices



(a) $N = 2^i$ (b) $N = 2^{i+1}$

Fig 6. Proof for Diagonal and Back Diagonal

2	4	6	1	3	5	7
3	5	7	2	4	6	1
4	6	1	3	5	7	2
5	7	2	4	6	1	3
6	1	3	5	7	2	4
7	2	4	6	1	3	5
1	3	5	7	2	4	6

Fig. 7 Assignment matrix for N = 7

THE IMPACT OF RUN–TIME OVERHEAD ON USABLE PARALLELISM

Constantine D. Polychronopoulos

Center for Supercomputing Research and Development
and Dept. of Electrical and Computer Engineering
University of Illinois at Urbana–Champaign
Urbana, Illinois 61801

ABSTRACT

During the execution of a program on a parallel machine run–time overhead incurs from activities such as scheduling, interprocessor communication and synchronization. This overhead is added to the execution time in the form of processor latencies and busy waits. As overhead increases, the amount of parallelism that can be exploited decreases. We consider two models of run–time overhead. In the first model overhead increases linearly with the number of processors assigned to a parallel task. In the second case, overhead is logarithmic on the number of processors. We discuss ways of computing optimal or close to optimal number of processors for each case, as well as critical task size.

1. Introduction

The overhead involved with the simultaneous application of many processors to the same task can be very significant [PoKu87], [Poly86], [Rein85], [Cytr85]. So far most of the existing parallel processor systems have not addressed the overhead issue adequately nor have they taken it into account either in the compiler or in the hardware. On the Cray X–MP, for example, microtasking can be applied at any level, although it has been shown that below a given degree of granularity microtasking results in a slowdown [Cray85].

In this paper we analyze two widely used models of overhead and their impact on the degree of parallelism that we can exploit. Using these models we can compute an approximation to the optimal number of processors for a given parallel task. This is also equivalent to computing the minimum size of an allocatable task. With these models we then perform some measurements using simple parallel loops. Finally we discuss ways of computing approximate execution times of tasks at compile time.

2. Overhead of Parallel Tasks

As our machine model we choose a p–processor shared memory or message passing system with homogeneous processors. If T_1 and T_p are the serial and parallel execution times (on p processors) for a program *PROG* respectively, then we define the *speedup* S_p of *PROG* on a p processor system to be $S_p = T_1/T_p$. The *efficiency* of execution of *PROG* is then defined by $E_p = S_p/p$ [Kuck78]. It is clear that for each program and for each system $1 \leq S_p \leq p$ and $0 < E_p \leq 1$ [Bane79]. It is usually hard to precisely compute T_1 and T_p at compile–time. However, close approximations are adequate in estimating overhead and performing related optimizations.

A program is composed of a collection of *tasks* where tasks can be *serial* or *parallel.* Any pair of tasks can be data dependent or independent on each other. The tasks and the dependence relationships among tasks define the *task graph* for a given program. Parallel tasks are composed of a set of *independent processes.* Processes are serial entities i.e., they always execute on a single processor. We assume that the execution of a process is *nonpreemptive.*

We will consider the worst–case overhead incurred with the parallel execution of processes. This is the familiar fork/join operation which is employed, for instance, in generating several processes from a parallel DO loop. Such parallel loops can be specified by the programmer or can be the result of program restructuring. A particular type of parallel loops which is used in this paper is the DOALL loop [KLPL81]. The iterations of a DOALL loop are data independent and therefore can be assigned to different processors and can be executed in any order. A DOALL loop defines a task; one or more iterations executing concurrently on the same processor define a process. Parallelism at the task level can be utilized by executing different tasks simultaneously. This is also known as *high level spreading.*

The question of interest to us is the estimation of the *critical process size* or CPS. Informally, the CPS can be defined as the minimum size of a process for which the execution time on a single processor is equal to the associated overhead. When a parallel task is distributed to several processors at run–time, it incurs a penalty or overhead that limits the degree of exploitable task granularity. Consider the parallel execution of a DOALL loop whose iterations are spread across processors at run–time. Run–time overhead may include several activities that do not occur during serial execution. All processors involved, for example, will have to access the ready–task queue in a serial mode since it is a critical section. Different processors will get different iterations of the same loop. At the end of the loop all processors involved must pass through a barrier to determine that the loop has been executed and that they are allowed to proceed with the next task [PoKu87]. The fetching of instructions at run–time can also be considered part of the overhead. Espe-

This work was supported in part by the **National Science Foundation** under Grant No. NSF DCR84–10110 and NSF DCR84–06916, the **U. S. Department of Energy** under Grant No. DOE DE–FG02–85ER25001, and by donations from the **Control Data Corporation** and the **IBM** Corporation.

cially with self-scheduling, instruction prefetching cannot work since, by definition, it is impossible to predict which processor will execute the next task or the next iteration of a loop. All these activities prolong the parallel execution time of a program. None of the above occurs during serial execution. This overhead, as would be expected, makes it inefficient to execute in parallel small tasks or to use a very large number of processors on even large parallel tasks. If the task is not large enough to amortize the overhead, we may end up with a parallel execution time which is larger than the serial execution time.

The tasks involved in an instance of high level spreading can be thought of as iterations of a DOALL loop whose loop-body contains conditional statements, and therefore different iterations have different execution times. Therefore high level spreading can be reduced to the parallel loop case where the number of iterations equals the number of independent tasks in that set. Since it is impossible to precisely estimate the execution time of a loop body with conditional statements, either at compile-time or at run-time, we assume an average or a worst case value as discussed in Section 5. For the moment let us assume that the loop-body for a given parallel loop has a constant execution time.

Consider a DOALL loop with N iterations whose body execution time is B units, and which is to be executed on a system with p processors. Let us see how one can compute an approximation to the CPS, i.e., the minimum number of iterations (*chunk*) allocated to each idle processor. Each time a processor dispatches one or more iterations of the loop it incurs an overhead σ. The question is to determine the minimum chunk size k for which $S_p > 1$, or equivalently,

$$S_p = \frac{NB}{\frac{N/k}{p}(kB + \sigma)} > 1.$$

After simplification we get

$$k > \frac{\sigma}{B(p-1)}.$$

As one would expect, the chunk size is inversely proportional to the number of processors executing the loop. For example, if $\sigma = B$ and $p = 2$ then at least $k = 2$ iterations should be allocated each time. In what follows we concentrate on determining the optimal number of processors that should be allocated to a given parallel loop.

3. Two Run-Time Overhead Models

To analyze the run-time overhead we use two conjectures that have been backed by empirical results [Cytr85], [LeKK86], [Ston87]. The first conjecture states that during the parallel execution of a task the run-time overhead is linearly proportional to the number of processors involved. The second conjecture states that the run-time overhead is logarithmically proportional to the number of processors. Let us consider two examples where these two conjectures are valid.

Consider the execution of a DOALL loop on a set of p processors connected to a common bus. If the iterations of this DOALL are spread among the p processors, then all p processors must execute a join operation before they are allowed to proceed with the next task. If two lexically adjacent DOALLs L_1 and L_2 operate on the same array, it will be necessary (in general) to execute a barrier synchronization (join) between L_1 and L_2. Thus all processors executing L_1 must finish before they start on L_2. Clearly the execution of a barrier operation on a bus-based multiprocessor involves $O(p)$ steps in the worst case [Ston87]. In a dynamic scheduling environment this overhead will also occur during dispatching of iterations, assuming all processors start on a loop at the same time.

If the same example is used for p processors interconnected in a tree structure, the barrier operation will take $O(\log p)$ steps to complete. A more real-world example of logarithmic run-time overhead are shared memory multiprocessors such as the Cedar and the Ultracomputer which employ multistage interconnection networks. If no special hardware is used and if synchronization is done through the shared memory, then the logarithmic overhead case applies here as well. The results presented in this section can be used by the compiler to draw exact or approximate conclusions for each task in a program, and can be used at run-time to avoid inefficient processor allocations.

3.1. Run-time Overhead is $O(p)$

As mentioned above we can identify a parallel task with a DOALL loop without loss of generality. Let T_1 and T_p denote, as usual, the serial and parallel execution time of a given task. Let N be the number of iterations of a DOALL loop and B the execution time of the loop-body. If the loop-body has a varying execution time the procedure of Section 5 can be used to derive a worst case or average value for B.

In this section we consider the case where the run-time overhead is linearly proportional to the number of processors assigned to a parallel loop. Let σ_o be the run-time overhead constant which in general depends on the characteristics of the code and the machine architecture. The compiler can supply the value of σ_o for each loop (parallel task) in the program. The serial execution time of a loop with N iterations and a loop-body execution time of B would be $T_1 = NB$. The parallel execution time then on p-processors would be

$$T_p = \left\lceil \frac{N}{p} \right\rceil B + \sigma_o p. \tag{1}$$

Consider (1) as a function of p. If overhead was zero, (1) would be an integer-valued decreasing function. Since (1) is not continuous it is not amenable to analytical study. We can approximate the function in (1) by a continuous function, by eliminating the ceiling. We thus get

$$T(p) = NB/p + \sigma_o p. \tag{2}$$

$T(p)$ is a continuous real function in the interval $(0, +\infty)$, with continuous first and second derivatives. Therefore we can study its shape and determine the point where overhead becomes minimal. In other words we want to find the value of p for which (1) becomes minimum and therefore the speedup of that task is maximized. The minimum value is given by the following theorem.

Theorem 1. $T(p)$ in (2) is minimized when the task is executed on a number of processors given by

$$p_o = \sqrt{NB / \sigma_o}. \tag{3}$$

Proof: First we show how (3) is derived and then prove that it is indeed the optimal value for that task (loop). Consider (2) which is an approximation to the parallel execution time defined by (1). $T(p)$ has a first derivative

$$\frac{dT(p)}{dp} = T^{'}(p) = -\frac{NB}{p^2} + \sigma_o. \tag{4}$$

The local extreme points of (2) are at the roots of its first derivative, that is, at

$$p_{o,1} = \pm\sqrt{NB \,/\, \sigma_o} \tag{5}$$

and since we are only interested for values in the interval $(0, +\infty)$, we discard the negative root p_1. The second derivative of $T(p)$ is

$$\frac{d^2T(p)}{dp^2} = T^{''}(p) = \frac{2NB}{p^3} > 0. \tag{6}$$

$T^{''}(p)$ is always greater than zero and therefore the extreme at $(p_o, T(p_o))$ is a minimum, where p_o is given by (5). If p_o is an integer that divides N, then the parallel execution time T_p is also minimized and it is given by

$$T_{p_o} = \frac{NB}{\sqrt{NB/\sigma_o}} + \sigma_o\sqrt{NB \,/\, \sigma_o} =$$

$$\sqrt{NB\sigma_o} + \sqrt{NB\sigma_o} = 2\sqrt{NB\sigma_o}. \tag{7}$$

Indeed if T_p is the parallel execution time for any other p, then p can be expressed as $p = c\sqrt{NB/\sigma_o}$ where c is a positive rational number. Then $T_{p_o} < T_p$, or equivalently,

$$2\sqrt{NB\sigma_o} < \sqrt{(NB)^2\sigma_o \,/\, c^2(NB)} + \sqrt{c^2\sigma_o^2(NB) \,/\, \sigma_o} \tag{8}$$

and if we substitute $x = NB\sigma_o$ in (8) we have

$$2\sqrt{x} < \sqrt{x/c^2} + \sqrt{c^2x} \;\rightarrow\; 0 < x(1 + c^4 - 2c^2)$$

and since $x > 0$, we get $(1 - c^2)^2 > 0$ which is always true. ■

Therefore p_o is the optimal value for $T(p)$ and in certain cases the optimal value for T_p.

Corollary 1. For $\sigma_o \geq (NB)/4$ the approximation function $T(p)$ defined in (2) satisfies

$$T(p) \geq NB \tag{9}$$

for any integer $p \neq 0$.

Proof: By substituting $T(p)$ from (2) in (9) we have

$$\frac{NB}{p} + \sigma_o p > NB \quad \text{or} \quad \sigma_o p^2 + p(NB) + NB > 0 \tag{10}$$

(10) is a quadratic equation of p and since $\sigma_o > 0$, the inequality in (10) is always true if the determinant D of the equation in (10) is negative, i.e.,

$$D = (NB)^2 - 4\sigma_o(NB) < 0 \quad \text{which gives us} \quad \sigma_o > \frac{NB}{4}. ■$$

Corollary 2. If $\sigma_o \geq (NB)/k$ then the parallel execution time for $p \geq k$ is greater than the serial execution time, i.e., $T_p > T_1$.

3.2. Run-Time Overhead is $O(logp)$

Let us assume that the run-time overhead is logarithmically proportional to the number of processors assigned to a parallel task. Therefore, in this case the parallel execution time is given by

$$T_p = \left\lceil \frac{N}{p} \right\rceil B + \sigma_o \log p. \tag{11}$$

To determine the optimal number of processors that can be assigned to a parallel task, we follow the same approach as in the previous case. Again since (11) is not a continuous function we approximate it with

$$T(p) = \frac{NB}{p} + \sigma_o \log p \tag{12}$$

which is continuous in $(0, +\infty)$, with continuous first and second derivatives. The corresponding theorem follows.

Theorem 2. The approximate parallel execution time defined by (12) is minimized when

$$p_o = \frac{NB}{\sigma_o}.$$

Proof: The first derivative of (12) is given by

$$\frac{dT(p)}{dp} = T^{'}(p) = -\frac{NB}{p^2} + \frac{\sigma_o}{p}. \tag{13}$$

$T^{'}(p)$ has an extreme point at

$$p_o = \frac{NB}{\sigma_o}. \tag{14}$$

The second derivative of (12) at p_o is

$$T^{''}(p) = \frac{2NB - \sigma_o p}{p^3} \quad \text{and} \quad T^{''}(NB/\sigma_o) = \frac{\sigma_o^3}{(NB)^2} > 0$$

Therefore $T(p)$ has a minimum at $p = p_o$. ■

However p_o is not necessarily a minimum point for (11). We can compute an approximation to the optimal number of processors for (11) as follows. Let $\epsilon = \lceil p_\rho \rceil - p_o$ where $0 < \epsilon < 1$. Then the number of processors p_o that "minimizes" the parallel execution time T_p in (11) is given by

$$p_o^{'} = \begin{cases} \lfloor p_o \rfloor & \text{if } \epsilon \leq 0.5 \\ \lceil p_o \rceil & \text{if } \epsilon > 0.5 \end{cases} \tag{15}$$

where $p_o = NB/\sigma_o$. In the next section we see that (15) is a very close approximation to the optimal number of processors for (11). The overhead problem was studied in a similar context in [Cytr85].

4. Measurements

We can use the above models to derive an approximate estimate of the effect of run-time overhead on the degree of usable parallelism, and thus on execution time. We used (1) to compute the actual execution time of a parallel task, and (2) to compute its approximation function for the linear overhead case. Similarly (11) and (12) were used for the logarithmic overhead case.

Figure 1 illustrates the execution time versus the number of processors for a DOALL with N=150 and B=8 under (a) linear overhead, and (b) under logarithmic overhead. Figures 2, 3, and 4 illustrate the same data for three different DOALLs, whose N and B values are shown in each figure. The solid lines plot the values of T_p, the actual parallel execution time. Dashed lines give the approximate execution times $T(p)$. For these measurements a value of $\sigma_o = 4$ was used. The overhead constant although optimistically low, is not unrealistically small for (hypothetical) systems with fast synchronization hardware. In all cases we observe that as long as $p \leq N$ (which is the case of interest), the difference between the values of the approximation function $T(p)$ and the actual parallel execution time T_p is negligible.

Looking at Figures 1a and 3a we observe that when the loop body is small, the associated overhead limits severely the number of processors that can be used on that loop. For these two cases for example, only 1/10 and 1/40 of the ideal speedup can be achieved. When B is large however the overhead has a less negative impact on performance. For the case of Figure 2a for instance, 1/2 of the maximum speedup can be obtained in the presense of linear overhead. The same is true for Figure 4a. In all cases the logarithmic overhead had significantly less negative impact on speedup.

5. Deciding the Minimum Unit of Allocation

Estimating the projected execution time of a piece of code (on a single processor) can be done by the compiler or the run–time system with the same precision. Let us take for example the case of a DOALL loop without conditional statements. All that needs to be done is estimate the execution time of the loop body, and let it be B. For our purpose, the exact number of loop iterations need not be known at compile–time. Since we know the overhead for the particular machine and the structure of a particular loop, we can find the critical block size for that DOALL that is, the minimum number X of iterations that can be allocated to a single processor such that $S_p > 1$. This number X can be "attached" to that DOALL loop as an attribute at compile–time. During execution the run–time system must assign to an idle processor X or more iterations of that loop (but no less). In case $X \leq N$ the loop is treated as serial.

Let us consider the code inside a DOALL loop. The control–flow graph of a code module with conditional statements can be uniquely represented by a directed graph. Consider for example the code module of Figure 5 which constitutes the loop body of some DOALL. The corresponding control–flow graph is shown in Figure 6. Since there is no hope of accurately estimating the execution time either in the compiler or at run–time, we choose to follow a conservative path. The execution time of each basic block $B_1, \ldots, B_7$ can be estimated quite precisely. We take the execution time of the loop body to be equal to the execution time of the shortest path in the tree.

The shortest path can be found by starting from the root of the tree and proceeding downwards labeling the nodes as follows. Let t_i be the execution time of node v_i, and l_i be its label. The root v_1 is labeled $l_1 = t_1$. Then a node v_i with parent node v_j is labeled with $l_i = l_j + t_i$. As we proceed we mark the node with the minimum (so far) label. In case we reach a node that has already been labeled (cycle) we ignore it. Otherwise we proceed until we reach the leaves of the tree. Note that the labeling process does not have to be completed: If at some point during the labeling process the node that has the minimum label happens to be a leaf, the labeling process terminates. The path π that consists of the marked nodes is the shortest execution path in that code. The number of iterations required (conservatively) to form the critical size is a function of the number of processors as shown in Section 3. B, the execution time of π, is given by the label of the last node of path π. A less conservative approach would be to take the average path length, assuming all branches in the code are equally probable. In the example of Figure 6 the above procedures give us $B = 12$ and $B = 33.33$ respectively.

```
1:  B1
    if C1 then B2
    else B3
    B2
    if C2 then goto 1
    else if C3 then B4
         else B5
    exit
    B3
    if C4 then B6
    else B7
```

Figure 5. An example of conditional code.

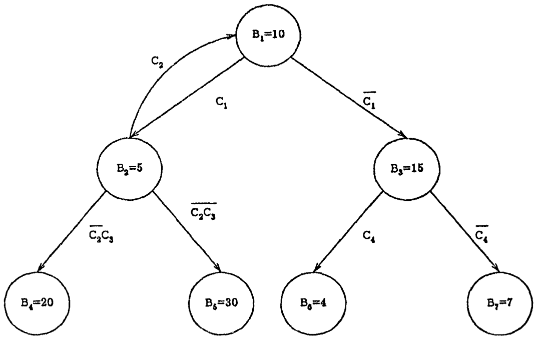


Figure 6. The control flow tree of Figure 5.

6. Conclusions

Run–time overhead is an important issue for parallel processor machines. Even moderately low run–time overhead can significantly limit the amount of program parallelism that can be exploited. In this paper we analyzed two models of run–time overhead and we computed the optimal number of processors that can be used for each case. The measurements indicated that the approximations used model closely the exact formulation of the problem.

Acknowledgements: The author would like to thank the anonymous referees for many useful suggestions.

REFERENCES

[Amda67] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Computer Conference Proc.*, Vol. 30, 1967.

[Bane79] U. Banerjee, "Speedup of Ordinary Programs," Ph.D. Thesis, University of Illinois at Urbana–Champaign, DCS Report No. UIUCDCS–R–79–989, October 1979.

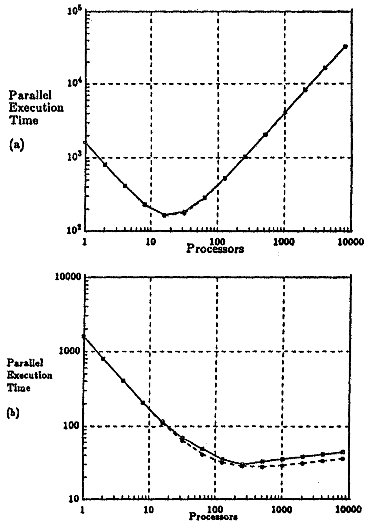


Figure 1. (a) linear and (b) logarithmic overheads for *N*=200, *B*=8.

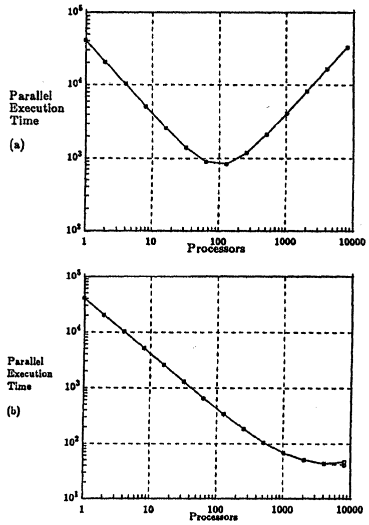


Figure 3. (a) linear and (b) logarithmic overheads for *N*=4096, *B*=10.

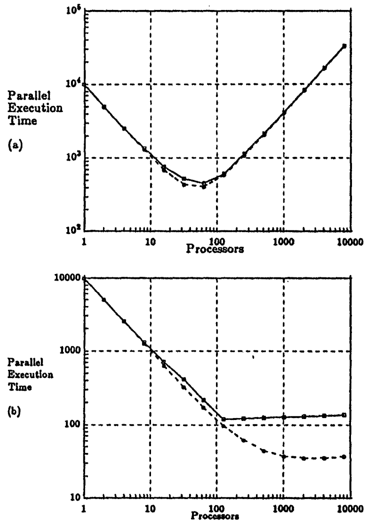


Figure 2. (a) linear and (b) logarithmic overheads for *N*=100, *B*=100.

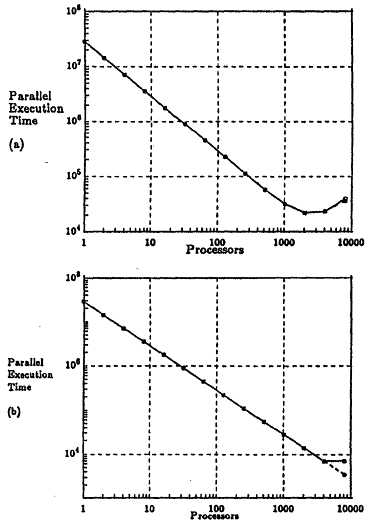


Figure 4. (a) linear and (b) logarithmic overheads for *N*=4096, *B*=7000.

[Cray85] "Multitasking User Guide," Cray Computer Systems Technical Note, SN-0222, January, 1985.

[Cytr85] R.G. Cytron, "Useful Parallelism in a Multiprocessing Environment," *Proceedings of the 1985 International Conference on Parallel Processing,* St. Charles, IL, pp. 450-457, August, 1985.

[KLPL81] D.J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proceedings of the 8-th ACM Symposium on Principles of Programming Languages,* pp. 207-218, January 1981.

[Kuck78] D.J. Kuck, *The Structure of Computers and Computations,* John Wiley & Sons, New York, 1978.

[LeKK86] G. Lee, C. Kruskal, and D. J. Kuck, "The Effectiveness of Combining in Shared Memory Parallel Computers in the Presence of 'Hot Spot'," *Proceedings of the 1986 International Conference on Parallel Processing,* St. Charles, IL, August, 1986.

[PoKu87] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", *IEEE Transactions on Computers,* Vol. C-36, No. 12, December 1987.

[Poly86] C. D. Polychronopoulos, "On Program Restructuring, Scheduling and Communication for Parallel Processor Systems", Ph.D. Dissertation, TR No. CSRD 595, Center for Supercomputing R & D, University of Illinois, August 1986.

[Rein85] S. Reinhardt, "A Data-Flow Approach to Multitasking on CRAY X-MP Computers," *Proceedings of the 10th ACM Symposium on Operating Systems Principles,* December, 1985.

[Ston87] H. Stone, "High Performance Computer Architecture", *Addison Wesley,* Boston, 1987.

THE MICROFLOW ARCHITECTURE

Jon A. Solworth

Department of EECS (M/C 154)
University of Illinois at Chicago
P. O. Box 4348
Chicago, Illinois 60680

(312) 996-0955

Abstract

An MIMD architecture dubbed Microflow is presented which combines very low cost communication and synchronization with the latency avoidance techniques of uniprocessor architectures. The communication and synchronization is implemented with extremely fast message passing by having targets of messages be general purpose registers. Communication between adjacent nodes can be accomplished in the time it takes to execute one instruction.

A Microflow processor contains multiple windows, each containing a context. This mechanism enables high performance servers to be constructed in software while enabling the server to have high priority and low overhead.

The message passing elements integrate smoothly with RISC or even moderately horizontal instruction sets, enabling Microflow to perform well even on those parts of the code which do not parallelize well.

1. Introduction

In this paper, an MIMD architecture dubbed Microflow is presented which combines very low cost communication and synchronization with the latency avoidance techniques of uniprocessor architectures. The name Microflow is derived from extremely fine-grained message passing which extends functional unit style data- and control- flow synchronization and communication across processors. The peak communications performance per processor in *M*illions of *T*ransmissions *P*er *S*econd (*MTPS*) is equal to the processor MIPS rate.

Assume a switch transition rate of four times the processor instruction rate. This message transmission rate means that:

- Neighboring processors can communicate and synchronize in about a few instruction cycles.
- Monitors (and other forms of remote procedure calls) can be invoked in a few instruction cycles

And, on a computer with a thousand processors:

- Most distant processors can communicate and synchronize in 4 machine instructions.
- Computer-wide synchronization can be achieved in tens of instructions.
- Summing, enumeration, and-trees and or-trees can also be achieved in tens of instructions.

The speed at which a parallel processor performs communications and synchronization is an important metric of performance. However, when the degree of parallelism is larger than the number of processors, then the computation can be broken into parallel chunks, each chunk can be most efficiently run sequentially. Also, when the degree of parallelism is smaller than the number of processors then it is the speed of the uniprocessor which will increasingly determine computation performance. Unlike other fine-grained architectures that we are aware of, Microflow performs serial code at uniprocessor speeds. Hence, all of the traditional latency avoidance speedup techniques are applicable, including: memory hierarchy, prefetching, pipelining, word parallel arithmetic and so forth. In addition, compiler optimizations enable each processor to execute only those instructions which are necessary for its part of the computation.

The Microflow design conservatively extends the techniques for high-performance uniprocessor by integrating switching hardware and augmenting processor design with message passing. Although the hardware extensions are conservative, both new programming language constructs and new compilation techniques are necessary to fully exploit the Microflow Architecture [Sol87].

Our benchmark computation is to efficiently operate on pointer-based data structures, although Microflow is also very effective at both systolic processing and coarse grain parallel processing. Pointer-based computations arise in symbolic processing as in compiler optimization, computer-aided design, data bases, and artificial intelligence applications.

2. Design

We shall define an *integrated* architecture as one which combines both message passing and shared memory. Logically, message passing and shared memory are equivalent in the sense that either technique can be used to simulate the other. Integrated architectures have advantages over either shared memory or message passing only if the hardware performance of each is significantly better than simulation in software. We describe how these performance advantages are attained in Microflow.

2.1. Implementation of integrated architecture

In the succeeding sections, the implementation of loads (shared memory) and of sends (message passing) are described. All large scale parallel processors have an interconnection network, as shown in Figure 1. If the computer is a message passing system, then the interconnection network routes network messages from processors to nodes; if it is a shared memory system, then the interconnection network routes network messages between the memory modules and the processors. Hence, in either case it is network messages which are routed on the network.

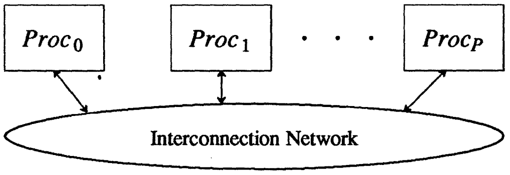


Figure 1 — General large scale parallel processor scheme

2.1.1. Implementation of load

Loads are implemented in Microflow the same way as on other high performance processors. Associated with each register is a *Full/Empty* bit. Registers with valid contents are marked *Full*; *Empty* denotes that the register is reserved for the result of an outstanding load.

A load performs the following functions:

(1) The register is marked as *Empty*.

(2) A network message is sent off-chip and routed to the addressed memory module.

(3) The memory module fetches the value and replies with a return network message which is routed to the originating processor.

(4) The value is put in the register and the register is marked as *Full*.

As in other high-performance computers, the processor continues to issue instructions until an instruction is encountered which requires a register which is marked *Empty*. Two advantages are obtained:

- Multiple loads with different target registers can execute simultaneously
- The use of both control flow pipelining (instruction counter) and data flow (returning load values) is more efficient than either mechanism.[1]

2.1.2. Implementation of send

Before we discuss the architectural implementation, message passing is presented on a more abstract level. The semantics are that $proc_i$ sends a value to a *communications variable* (*cv*) at $node_j$. As with registers, communications variables are marked with *Full/Empty*. The sequence for a send is:

(1) The originator, $proc_i$ sends a network message to *cv* at $node_j$.

(2) The processor at $node_j$, $proc_j$ issues a receive instruction which marks the receiving *cv* as *Empty*.

(3) When the message is at $node_j$, and the *cv* to which it is destined is marked *Empty*, the value is loaded into the *cv* and the *cv* is marked *Full*.

Note that since the originating and destination processors operate asynchronously, steps 1 and 2 may be interchanged. Nevertheless, the *Full/Empty* interlock ensures correct operation.

Architecturally, the issue arises whether to map these *cvs* to memory locations or to registers. If the *cvs* are mapped to memory, then the *cv* space is as large as the variable space, but access is slow. Alternatively, if *cvs* are mapped to registers, there is only a small number of them available, but they are at least an order of magnitude faster. Moreover, if the receiver is ready before the sender, neither memory nor network bandwidth is consumed while the receiver busy waits. Given sufficiently fast context switching, work could even be done in the interim.

In Microflow, communications variables are mapped to registers. The additional hardware is negligible since, in effect, a send looks like a remote load.[2] It is also worth noting that the target of a send is a general purpose register; this enables the compiler to choose the allocation between computation and communication registers that best suit the program.

The implementation of send ensures that a processor can send or receive a message every cycle. Hence, peak transmission rate is equal to instruction rate, enabling computations to exhibit speedups when executing only 1-3 instructions between transmissions.

The above arguments show that message passing and shared memory easily integrate into the same architecture. The judicious use of message passing also reduces network traffic. For example, a shared memory implementation of producer-consumer requires three uni-directional network messages (one for the write, and two for the read). Using message passing, the same function is performed by one network message.[3]

2.1.3. Communication variable queues

Associated with each communications variable is a queue which can contain up to *P* elements. The queue serves two purposes:

(1) It allows each processor to send to a *cv* without first getting permission. This avoids an extra round trip of network latency.[4]

(2) It reduces the amount of work each processor performs by requiring it to look one place for "work". Hence, no polling of *cvs* is required.

There is insufficient room on the processor chip to store the queue. Therefore, both the initial segment of the queue and the queue management control are implemented in the cache. Once the queues grow beyond a certain size, the tail of the queue is stored in main memory. This size is chosen so that main memory references are infrequent.

Another issue that arises is how the processor and cache chip communicate about the queue. Every time a processor issues a *receive* or a *load*, the register and window specified are provided to the cache. The cache maintains a duplicate

[1] The instruction counter, while limiting choice, speeds up the execution by fetching and decoding instruction prior to the arrival of data. In pure data flow schemes, it is the arrival of data which triggers an instruction fetch, thereby adding a delay to data execution.

[2] Since it is illegal for the software to use a register for both a *cv* and a load target, one set of *Full/Empty* bits is sufficient for both purposes.

[3] The example is simplified since a) multiple reads might be needed if the consumer started reading before the producer produced the data and b) that there is no handshaking to ensure the message is not lost. The cost of handshaking can be made arbitrarily small by associating queues of length *L* and handshaking after every *L* items.

[4] Large fan-in *cvs* do not imply a serialization point since the number of sends to a *cv* may be of size *P* in the worst case, but is usually much smaller. For example, breadth first search must assume a graph-node at a processor node is connected to a node at every other processor, even if this will rarely happen.

set of *Full/Empty* bits. Whenever the cache receives a value for a register which is empty, that value is right away sent to the processor chip, otherwise it is cached.

2.2. Context switching

Rapid context switching in Microflow is used for two principle reasons; latency avoidance and the implementation of servers. The use of fast context switching for latency avoidance is well known. However, we believe its application to servers is new.

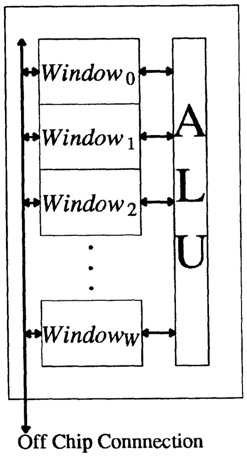


Figure 2 — Multiple windows per processor

The rapidity of context switching is achieved by replication of hardware resources. Each processor chip contains *W* sets of registers, called *windows* and one ALU (see Figure 2). Independent instruction streams are executed in Microflow's windows, unlike the overlapping windows used in some RISC architectures. Each window contains the entire processor state for a context; at most one context (or window) is designated as current. The multiplicity of registers means that not only can context switching be performed without saving or restoring registers but that the pipeline does not even need to be flushed.

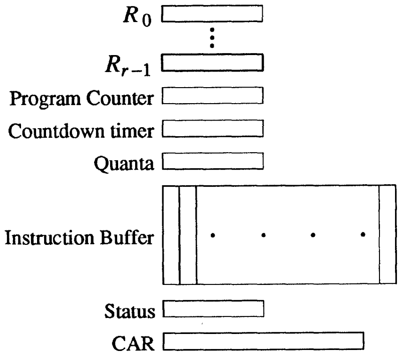


Figure 3 — Registers per window

Each window contains a set of general purpose registers, a program counter, a countdown timer, a quanta size, an instruction buffer, a status register, and a *C*ommunications and *A*ddressing *R*egister (*CAR*). See figure 3. The CAR is a multifield register which enables the construction of very complex network messages; however, most network messages are produced by a single 3-address RISC instruction.

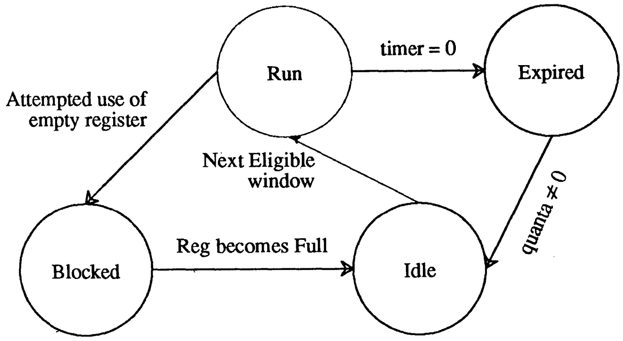


Figure 4 — State diagram for windows

A window which is currently active remains active until either its countdown timer reaches zero (**expired**) or it attempts to use a register which is marked *Empty* (**blocked**). The context then switches to the next window which is neither blocked nor expired and the countdown timer on the previous window is reset to the quanta size. This context switch and reseting of timer the takes place concurrently with instruction execution. Hence, no cycles are lost due to context switching; and it is possible, for example, to run HEP style instruction interleaving [Smi81] by setting the quanta to one in all windows.

The processor instruction set is a normal RISC instruction set augmented with instructions for constructing complex message (using the CAR) and the instructions:

send dest_processor,dest_register,value
receive R_i

Where *dest_processor* and *value* are register contents and *dest_register* is a literal. To send a message to a different window, the window field of the CAR is set immediately before the send.

The first window contains the application code and is expected to consume the lion's share of the processor cycles. Other windows contain server code that are invoked by the application and proceed independently. Example of servers include fetch&add, and-trees, and or-trees.

The server acts on a demand basis. Since context switching is free (if no other window has any work to do, a window will "switch" to itself), a small quanta is assigned to the application thread, with larger quanta assigned to the server thread. The application thread will execute only when the server has nothing to do, and the server (which is infrequently busy) will respond almost at once.

Of course, server code could be executed in the application window. But message arrival is asynchronous requiring the application window to performed only message code (and be idle waiting for messages) or to continue running the application and interrupt on message arrival (implying a large overhead). Neither of these alternatives are acceptable. By using separate windows, the application can request a service before the service needs it, thereby overlapping the application with the server and avoiding latency.

The organization of separate server windows (and the low cost of embedding trees in the Microflow network) makes servers sufficiently fast to perform combining operations in software (for fetch&add operations), and to replace combining of reads with broadcast operations. We believe that this should speed up switch operation by a factor of two over combining switches [DKS85], while more easily enabling high fan-in/fan-out switches. Not only will the network run faster, but the software servers enable the construction of trees that perform blocking functions. For example, the **and**-tree requires all processors to supply a vote before any result is returned. Such algorithms require busy waiting with network combining techniques.

2.3. Node design

Each node consists of a processor, memory, and a switch as shown in figure 5.

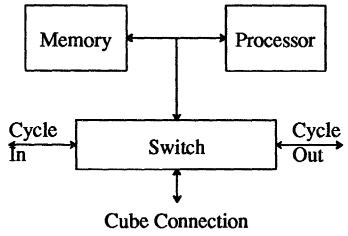


Figure 5 — A Microflow Node

Each memory module is physically adjacent to some processor, although all memory is globally addressable. Since the processor node has a cache, the memory can be constructed from inexpensive dynamic RAM chips. Since the cache is snoopy, the processor can cache shared variables whose location is in the local memory. This enables the accessing of "hot-spot" variables to occur at cache rates rather than at the much slower memory speeds. The switch design performs only simple routing most of the time; when the network becomes congested the switch starts to perform a deadlock avoidance algorithm. This enables the switch to be built for the maximum speed possible, and as we shall see, this is particularly important on high fan-in/fan-out switches.

2.4. Network

The target applications and granularity of Microflow require that trees (for the purposes of control) be inexpensively embedded in the network, that is adjacent nodes in the tree would be adjacent in the network. We have chosen Cube-Connected Cycles (CCC) as our network [PrV81]. CCCs maintain many of the properties of Hypercubes, including logarithmic diameter and good performance on many Hypercube algorithms, but require hardware proportionate to the number of processors *P*. (A Hypercube requires hardware *P*log*P*).

Fast switches play an important role in fine-grain parallel processing. The switch shown in Figure 3 conceptually contains two cycle connections, one cube connection and one processor connection; however, *k* adjacent switches in a cycle can be coalesced into one *k*-ary switch.

The number of wires in the interconnection network and especially switch design will require messages to be packetized. However, it is advantageous for a processor to have parallel access to its memory. Therefore, the switch design merges *k* packetized processor ports into a single shared bus. Studies in uniprocessor systems with snoopy caches indicate that busses can support at least eight processors. For Microflow this number is likely to be somewhat lower since the bus is also being used for network traffic and because more bus activity is required because of the fine-grain of the application.

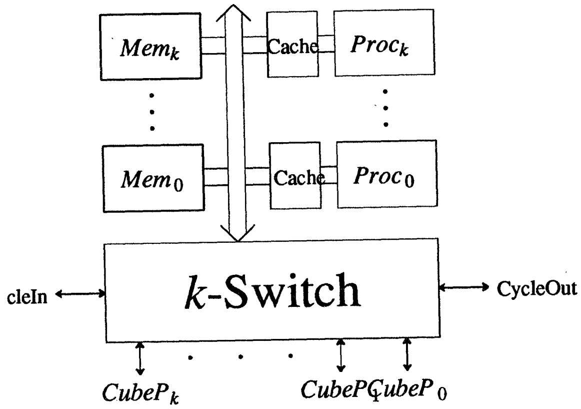


Figure 6 — *k*-ary switch with shared bus

We note that in addition to the single cycle bus access, by low interleaving the memory modules, a processor (or network switch) can perform stores at peak performance. This design enables this balance performance to be achieved with a smaller number of total memory modules (over the design in Figure 6).

3. Comparison to other architectures

The technique of multiple windows for latency avoidance dates back at least to the peripheral processor units (PPU) on the CDC 6600 [Tho70] and was first (and very elegantly) proposed for parallel processors in CHOPP [SûB77], in which the logarithmic random access delay was masked by log(P) windows. Snoopy caches were first proposed by Goodman [Goo83].

The use of *Full/Empty* bits on registers, or more complex schemes, also dates back at least to the CDC 6600, and probably much earlier. The first commercial computer to use *Full/Empty* bits on memory is the Denelcor HEP [Smi81], which used the bits as a means of performing fine-grained dataflow handshaking. However, a consuming process would have to busy wait if it was ready before the corresponding producing process. To eliminate busy waiting, I-structures [ArT80] provide a queue of waiting reads, which were automatically triggered when a write occurred. Microflow's message structure is more general than I-structures (and is faster). For example, if a large number of reads are pending, I-structures produce a serial bottleneck — in Microflow, a broadcast tree can be constructed resulting in no bottlenecks.

The Denelcor HEP also had *Full/Empty* bits on registers, and processes running on the same processor could communicate through sharing of register address spaces. However, there was no way for processes running on different processors to effect each others register set, so that this mechanism was not very heavily used.

The designs which Microflow is mostly closely related to in spirit are the Connection Machine [Hil85] and Message-Driven Processor (MDP) [DCC87]. The Connection Machine achieves synchronization by two means: its SIMD instruction execution and a global network empty signal which ensures that all messages have been delivered. In Microflow, fast point-to-point synchronization is the basic mechanism. When global synchronization is needed, trees are constructed in software. This mechanism enables a heterogeneous set of operations to occur (MIMD), for network accesses to be pipelined, for each processor to execute only those instructions relevant it, to take advantage of memory hierarchy (both faster and cheaper access).

The MDP communicates only by messages, but simulates shared memory by having a message unit which is able to simulate reads and writes. The MDP messages are sent to objects which then must be invoked, resulting in delays to fetch instructions and a very limited register set which caused accesses to be made to local memory. To make these accesses as fast as possible, memory is implemented on chip thus restricting its size. The constraint of on-chip memory also limits the ability to take advantage of the memory hierarchy, which reduces average access time, and its small size increases communication and latency requirements.

4. Performance Parameters

The Performance Parameters for a Microflow architecture are described in terms of the basic instruction rate of a processor. Using current technology, a Microflow processor would be implemented as a RISC augmented with multiple independent windows and message passing instructions. Our unit of time, the cycle time, is the rate of instruction execution. A network switch hop can be conservatively implemented in .25 cycles. This means that adjacent node can communicate in about 1 instruction time, for a k=4, *packets*=4 switch. Table 1 shows the Microflow parameters.

Operation	Cycles
processor instruction	1
switch hop	.25
adjacent node communication	1
furthest node communication (P = 2K)	3
cache speed	.5
memory module speed	2
memory read from furthest node (P = 2K)	8

Table 1 — Microflow architecture speed parameters

We have coded up a number of algorithms, including summing, enumeration, sorting (parallel quicksort), breadth-first search, parallel prefix on linked-lists, matrix multiply and inversion, and transitive closure. The number of server windows was between 0 and 3 (with an average of 1/2). Moreover, the number of registers needed were never larger than logarithmic in the number of processors. Hence a limited number of communications variables seem to be needed in an *integrated* architecture.

5. Conclusions

We have described some of the effects of implementing integrated architectures (those containing both shared memory and message passing) and discussed their performance advantages over either shared memory or message passing. The Microflow architecture is a very efficient implementation of an integrated architecture. Microflow:

- Provides extremely fast message passing by having targets of messages be general purpose registers.
- Provides enqueuing of messages destined to a single register both to reduce handshaking delays across the network and to eliminate polling by the receiving processor.
- Uses windows not only for latency toleration, but for latency avoidance by extending hardware latency avoidance techniques to software.
- Is extremely fine grained while maintaining the performance advantages on serial code.

References

[ArT80] Arvind and R. H. Thomas, "I-Structures: An Efficient Data Type for Functional Languages", *Laboratory of Computer Science, Tech. Memo. 178*, 1980.

[DCC87] W. J. Daley, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty and S. Wills, "Architecture of a message-driven processor", *14th Annual Symposium on ARCH*, Pittsburgh, June, 1987.

[DKS85] S. Dickey, R. Kenner, M. Snir and J. A. Solworth, "A VLSI combining network for the NYU Ultracomputer", *Proc., ICCD*, October, 1985.

[Goo83] J. R. Goodman, "Using cache memory to reduce processor-memory traffic", *Proceedings of the 10th Annual Conf. on Computer Architecture*, Stockholm, 1983.

[Hil85] W. D. Hillis, *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.

[PrV81] F.. Preparata and J. Vuillemin, "The Cube-connected cycles: a versatile network for parallel computation", *Communications of the Association for Computing Machinery 24*, 5 (May 1981), 300-309.

[Smi81] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system", *Real Time Signal Processing IV, Proceedings of SPIE 298* (1981), 241-248, The International Society of Optical Engineering.

[Sol87] J. A. Solworth, "Epochs", *C. S. Dept*, Jan. 1987.

[SuB77] H. Sullivan and T. R. Brashkow, "A large scale homogeneous machine I & II", *Proceedings of the 4th Annual Symposium on Computer Architecture*, 1977, 105-124.

[Tho70] J. E. Thornton, *The Design of a Computer: The Control Data 6600*, Scott, Foresman and Company Publishers, Glenview, Illinois, 1970.

Concurrent Miss Resolution in Multiprocessor Caches*

C. Scheurich and M. Dubois
Department of Electrical Engineering
University of Southern California
Los Angeles, California, 90089–0781
(213) 743-8080, dubois@priam.usc.edu

Abstract—The performance of cache-based, shared-memory multiprocessors can suffer greatly from moderate cache miss rates because of the usually high ratio between memory-access and cache-access times. In this paper we propose a cache design in which the handling of one or several cache misses occurs concurrently with processor activity. In multiprocessors, such lockup-free caches aggravate the memory coherence problem. The proposed design relies on a cache-block size of one word and as a result is simple and efficient. A multiprocessor architecture, using lockup-free caches, is described and shown to be correct. Through performance models, we identify system configurations for which lockup-free caches are effective. Compiler techniques, to take advantage of the proposed design, are illustrated at the end of the paper.

1. INTRODUCTION

Cache memories are commonly used to reduce the memory access latency for both data and instruction accesses. Caches can do this very effectively and economically [1]. In shared-memory multiprocessors caches are more important than in uniprocessors because the individual processors of a multiprocessor must be connected to the shared memory through an interconnection. Increased memory access latency and conflicts reduce the efficiency of each processor. Prefetching, which can reduce the apparent access latency visible to the processors, is also more difficult in multiprocessors because of the coherence problem [2].

It is possible to design caches which do not block the processor on an access miss—they are called *lockup-free caches.* In such designs the processor may continue sending requests to the cache both for data and instructions while the cache and the main memory system are resolving one or several previous misses. Such a scheme was described by Kroft in [3] for a uniprocessor. When processors are part of a shared-memory multiprocessor, the design of lockup-free caches becomes difficult because of the added problem of maintaining cache coherence.

In this paper we describe a multiprocessor architecture which allows caches to be lockup-free and in which synchronization is enforced by means of "hardware-guarded" primitives. We show that the operation of the caches is, with a few exceptions, very similar to that of lockup-free caches of uniprocessors. Furthermore, a straight-forward, snoopy cache coherence protocol can be used to enforce inter-cache consistency. The benefit of this architecture is that the overall cache miss penalty (i.e., the average time a processor is blocked because of a cache miss) is reduced and hence a small block size can be used to minimize overall cache-memory traffic. Goodman [4, 5] has argued that a block size of one word reduces memory traffic. Reduced memory traffic increases the possible system throughput for a given interconnection. In a different context, Lee *et al.* [6] have shown that in a multiprocessor it is preferable to use a small block size and to offset the penalty due to increased misses through processor prefetching. In such systems "blind" prefetching of instructions and data resulting from a larger block size is replaced by "smart" prefetching performed in the processor and assisted by the compiler.

The performance advantage which can be reaped from a lockup-free cache depends on the average shared memory access time and on the dependencies within each instruction stream. A performance model including these two parameters is developed and design trade-offs are discussed.

2. MULTIPROCESSOR CACHE OBSTACLES

2.1 Multiprocessor Cache Performance

Even small cache miss rates can have detrimental effects on the throughput of high-speed processors. As the following evaluation demonstrates, the efficiency of the processor can go down rapidly when the memory access time is large or when the hit ratio is low.

Let us assume a processor system with the following characteristics:

X is the maximum throughput of the processor in MIPS (Million of Instructions Per Second) if all accesses can be resolved by the cache (i.e., a cache hit rate of 1.0). X can often be easily estimated for a given processor architecture and instruction mix.

d is the average number of accesses per instruction execution, including instruction fetches, operand fetches and resultant stores. It is called the demand rate.

T_m is the average time to resolve a miss via main memory.

h is the average cache hit rate.

$t_{i,0}$ is the average time it takes to execute one instruction if all accesses are cache hits (i.e., $t_{i,0} = 1/X$).

If the processor blocks on every miss, the average time to execute one instruction, t_i is given by $t_i = t_{i,0} + (1-h)dT_m$. Hence, the average performance of the system, in MIPS, is reduced to $X' = X\frac{t_{i,0}}{t_{i,0}+(1-h)dT_m}$. Dividing by $t_{i,0}$ and letting $T_m^0 = T_m/t_{i,0}$, yields $X' = X\frac{1}{1+(1-h)dT_m^0} = XF_s$. F_s is called the *slowdown factor.* It varies between 0 and 1 and the closer it is to 1, the better the processor efficiencies. In Figure 1, the performance degradation of a cache-based system is shown for different values of

*This research is supported by an NSF Research Grant No. CCR-8709997.

h and T_m^0. As can be seen in Figures 1, even a system with a relatively high hit rate of 0.98 can suffer substantially, if the average main memory access time is high relative to the cache access time. Particularly in the case of fast processors, the ratio T_m^0 is likely to be in the upper ranges shown in Figure 1.

In uniprocessors hit rates of 1 could not be obtained even if the cache size was infinite, because of the initial loading of instructions and data. After a context switch the cache experiences a *cold start* period when most of the blocks of the new process have to be reloaded [7, 8]. In multiprocessor caches, data must also be invalidated because of the modification of cached data by other processors. These invalidations reduce the cache hit rate as compared to the uniprocessor case [9, 10]. Finally, it is well known [11], that a high average hit ratio hides wide variations in the hit ratio of individual programs. A truly general-purpose system should exhibit more uniform performance for different workloads.

2.2 Multiprocessors vs. Uniprocessors

The easiest way to build cache-based multiprocessors is to interconnect them with one or several buses. Since the buses provide a broadcast medium which automatically serializes all accesses, maintaining cache coherence is simplified. The problem with using buses lies with their limited bandwidth. Even if processors have large private caches, a great deal of memory to cache communication is still necessary due to the need to propagate data updates (usually in the form of an (1) invalidation, (2) fetch block sequence). Frequent updates and consequent invalidations have two effects—they strain the bandwidth capabilities of the bus and they lower the invalidated caches' hit rates. While small block sizes can reduce bus traffic, there is the detrimental effect of lowering overall hit rates since caches cannot benefit from the *spatial locality* of code and data. The effect is particularly bad during *cold start* periods—after a context switch, for example—when the cache misses on a large number of consecutive accesses.

The benefits of small block sizes can be reaped if processors are not constrained to wait for individual misses to be resolved before initiating another access. The use of lockup-free caches in multiprocessors is restricted, though, by the fact that logical problem can arise when accesses are performed out of program order. This may be the case in a lockup-free cache if, for example, a miss is followed by a hit, with the result that the access which hits is performed before the access which misses because of the longer time to resolve the miss. The restriction on the order in which accesses must be performed is due to the possibility of *inter-process* dependencies [12].

If a system makes no attempt to enforce *all* inter-process dependencies, and the programmer and compiler are aware of this fact, then the out-of-program order execution of accesses by a single process is allowable. (Of course intra-process dependencies must still be preserved.) In such a system it is possible to design caches to be lockup-free. However, after having removed the possibility of using shared variables to implement synchronization, there must be an alternate method available to allow processes to synchronize. Special hardware recognized primitives, such as the `test&set` instruction, can be used to implement synchronization.

2.3 Restrictions On Ordering

With respect to the *ordering of events* within a multiprocessor, the user (or compiler) may expect the system to adhere to one of two logical models of behavior. In [12] these models are called the *strongly ordered* and the *weakly ordered* model of behavior. In a strongly ordered system, processors must initiate memory accesses one-by-one, in program order [13]. Furthermore, all processors must "observe" all other processors' `write` operations in the same order. (By "observe" it is meant that an update becomes readable.) A system that is strongly ordered is sequentially consistent and can implement synchronization by software alone.

A weakly ordered system assumes three types of shared data.

1. Instructions[1], private data, and non-writable data can be accessed and cached by all processors in any possible order. Since non-writable data are never modified, no interprocess dependencies can exist on such data. This is also true for private data which are only modfied and `read` by one processor.

2. All other ordinary shared writable data can only be modified in *mutual exclusion*. These are data used to transfer information from one process to another.

3. Synchronization variables are data used to enforce mutual exclusion on `write` accesses to ordinary shared writable data. Synchronization variables are hardware recognizable as such.

Data of type (1) pose no difficulty and will not be further discussed. Data of type (2) are user/compiler generated. Accesses to such data must be protected by critical sections or semi-critical sections [10]. In the first case data may only be `read` or `written` by one processor at a time—the processor which has gained access to the appropriate critical section. In the second case, several processes are allowed to `read` the same data at the same time but updates must occur in a critical section.

Accesses to data of type (3) are synchronization primitives such as `test&set` operations. Since data modified within a critical section cannot be read by another processor, while the modifying processor is still executing the critical section, the order in which the data are modified within the critical section is immaterial. The only constraint for correctness is that all updates have properly propagated when the critical section is exited. However, a processor is not "aware" whether it is presently executing a critical section or not. Since critical sections may be nested or overlapped, keeping track of critical sections by the processor is not simple. For correctness, though, it is sufficient that all accesses of type (2) have propagated and completed, before an access of type (3) can complete. If a synchronization variable access is encountered, either a critical section is entered into or one is exited from—in either case all previous accesses must have been performed.

We summarize this section by defining four properties that must be maintained for a weakly ordered system to remain correct:

P1: Intra-process dependencies must be observed at all times. Such dependencies must be treated as in any conventional uniprocessor.

[1] We assume separation of instructions and data so that instructions can never be modified.

P2: Memory coherence must be maintained at all times. This is true for all three data types. Any processor's `read` operation will always reflect the most recent `write` operation to the datum. (Memory coherence of type (2) data only has to be restored *before* the processor that modifies it exits the critical section. However, since it is easy, we assume that memory coherence is maintained at *all* times.)

P3: All modifications of type (2) data must be performed from within critical sections. The compiler or the programmer must ensure that no two or more processes can be in the same critical section at the same time.

P4: All pending misses must be resolved before an access to type (3) data can proceed, within a processor.

3. LOCKUP–FREE CACHES IN WEAKLY ORDERED SYSTEMS

We limit our discussion here to weakly ordered bus-based systems which adhere to property **P2**, decribed in the above Section (i.e., memory coherence is maintained at all times for all data).

3.1 Basic Operation

Most of the time the cache responds to processor requests of type (1) and type (2) data. With respect to such data, the operation of the cache is equivalent to the operation of a uniprocessor lockup-free cache. Kroft described the implementation of such a cache in [3]. A basic overview of Kroft's principle of operation is given here; for more details the original paper should be consulted.

Multiple misses are resolved by storing information about each and then forwarding the miss request, packaged along with some vital return information, to the main memory. This is accomplished with the following in mind.

- Local dependencies must be observed.
- If a missed and to-be-returned block is to be allocated, space must be reserved in the cache for that block and, if necessary, a replacement must be made.
- Miss requests must be tagged such that:
  1. The word of the block which caused the miss is known.
  2. The functional unit which the word is to be forwarded to is known.
  3. The slot in cache which is reserved for the block is known.

Most of the above qualities are implemented by a set of associatively accessible registers, called MSHR registers (Miss Information/Status Holding Register), which keep track of the status of all pending misses. Returned blocks are buffered in a stack which can either be emptied, as contention allows, or can directly be accessed to speed up immediate demands.

3.2 Multiprocessor Issues

Multiprocessor caches must react in a specific way upon type (3) data accesses. An access to type (3) data means that a synchronization point has been encountered. To adhere to **P3**, a type (3) access must be disallowed until all pending accesses have been resolved. In Kroft's type of architecture, this implies that all MSHR registers must be empty before the synchronization access can proceed. Once the access has been performed, the cache continues to operate as usual, resolving misses and hits concurrently, until another synchronization is encountered.

Type (3) data may be cached and are subject to coherence control as are all other data. The only constraint on type (3) data is that they are recognizable by the hardware. In modern processors this is done by accessing these data through special synchronization primitives, such as `test&set` operations. The processor can inform the cache at the time of the access that type (3) data are the target. A `test&set` instruction can be executed indivisibly at the cache if an ownership scheme [14] is used to implement cache coherence.

4. A SAMPLE ARCHITECTURE

Figure 2 depicts a sample architecture. The multiprocessor is bus-based, with N processors connected to M memory modules by one or several packet-switched buses. The programmer and the compiler assume that the system behaves in a weakly ordered manner. The block size of the cache is one word (32 bits).

4.1 Processors

The specifics of the individual processor architectures are not important. For the system to be useful, processors should be able to prefetch operands and instructions as far ahead as possible. Since the block size is one word, `write` operations can *always* proceed, whether or not the block is in the cache. That is, since a `write` redefines the value of an entire block, the block need not be fetched before modification. (How coherence is maintained on such `write` operations is discussed in the next Section.) However, this means `write` operations to bytes, or half-words must be compiled to preserve the correct outcome of the intended operation. We believe that this drawback is far outweighed by the fact that `write` operations always hit at the cache.

4.2 Caches and Cache Coherence

The complexity of the cache architecture depends on both the number of MSHR registers implemented and on the cache coherence protocol used. The coherence protocol we suggest is a bus-based, "snoopy cache" protocol. This protocol works either for single or multiple packet-switched buses. In the absence of better evidence, we assume that the number of MSHR registers is four.

The cache block size is one word. Each block can be in one of three states. Namely, **RO** (Read Only) to indicate that the block may be shared with other caches and may not be modified without broadcasting invalidations; **RW** (Read Write) indicates that the block is private and may be modified without delay; **I** (Invalid) indicates that the block is not valid (not in cache) and must be requested over the bus if it is to be read.

The coherence protocol is very simple. A `read` hit may proceed at any time if the block is in state **RO** or in state **RW**. If a `read` miss occurs on a block in state **I**, a request for the block is placed on the bus (assuming that an MSHR register is available, otherwise the cache locks and the request will be posted after the first MSHR register becomes available). A `write` operation *always* appears to hit. If the block to be written is indeed in state **RW** it may be modified without further action. If the block is in state **RO** it is modified, and an invalidation for the block is placed on the bus. The state of the block is changed from **RO** to **RW**. If the block is either initially invalid or not present it is immediately allocated and an invalidation is broadcast. The modification of a block without having previously been fetched is allowed, since a single `write` operation always modifies the entire block. Note that it is not possible for two processors to update the same block concurrently because mutual exclusion prevents this from happening.

Only in three cases (read miss, broadcast invalidations, and write back) is the bus accessed. In the first case, the `read` request may either proceed to the memory for service, or the `read` request is interrupted by the cache which contains the requested block in state **RW**. The previous "owner" cache forwards the block to the requesting cache, and changes the block's state from **RW** to **RO**. During the transfer of the block, memory is updated as well. The second type of bus activity (an invalidation) causes all caches that contain the block (either in states **RO** or **RW**) to invalidate their copies. Write-backs are only necessary if a to-be-replaced block is in state **RW**. Blocks in state **RO** can be overwritten.

The cache performs four basic tasks:

1. It responds to and services access requests from the processor.
2. It monitors the buses for invalidations and accesses it must respond to.
3. It receives returned blocks on which it previously missed, either from the memory or from another cache.
4. It replaces cache blocks as necessary.

4.2.1 Task 1: Processor Requests

If a `read` request of type (1) or type (2) data hits at the cache, the word is immediately supplied to the processor. A `read` miss results in the following activity:

- Check all MSHR registers whether the requested block is already in transit (the processor is in this case accessing the same word twice in a row): If this is the case it is allocated to another MSHR register along with the target register of the processor. If no MSHR register is available, the cache locks. When the block is returned, it is allocated in the cache, the MSHR register is deallocated, and the word is forwarded to the processor.
- If there is no MSHR match, then the block is allocated in a reserved cache frame (a replacement is triggered if necessary) and a MSHR register stores the necessary return information. A reserved cache frame is marked as reserved.
- If no MSHR register is available, the processor locks until one becomes available.

For any `write` request, the block is written, whether or not it is present. If the block was not present or was in state **RO**, an invalidation with the address of the block is broadcast. No MSHR register is allocated to the access in this case. If the block was not present it may cause a replacement and a write back. The block will be in state **RW**. It is not possible for a block causing a `write` miss to be already present in an MSHR register, since this would imply an intra-processor dependency.

If the request is to type (3) data, the cache locks until all MSHR registers are empty (all pending misses have been resolved). Then the access to type (3) data is resolved like any other access. If it causes a miss, the cache remains locked until the access is completely resolved (i.e., the data are returned). Then the cache is unlocked and may proceed to resolve misses to type (1,2) data concurrently.

4.2.2 Task 2: Bus Watching

The buses are monitored continuously. If an invalidation of a present block is detected, the block is invalidated. It is not possible for a bus invalidation to occur for a block presently being fetched (i.e., a block referenced in one of the MSHR registers) since this would violate property **P3** of Section 2. If a read request for a block present in state **RW** is detected then the appropriate bus is interrupted and the block is placed on the bus. The block is forwarded to both the requesting cache and to main memory. The block remains allocated but the state is changed to **RO**.

4.2.3 Task 3: Returned Blocks

When a miss is resolved, either by the memory or by another cache, the MSHR registers are checked to find out where the block is allocated and which processor register it is intended for. The block is placed in the reserved cache frame and forwarded to the processor. One more contingency must be taken care of. The cache frame which was reserved for the miss may have been "replaced"[2]. This is handled by marking the appropriate MSHR register if a reserved block is replaced. In this case the word is only forwarded to the processor but does not get allocated at the cache.

4.2.4 Task 4: Block Replacements

A block is replaced when the cache frame it resides in is needed. Two flags, associated with the cache frame need to be checked. If the block is in state **RW** the block needs to be written back to main memory. Otherwise it may be overwritten. A block marked as reserved means that the word of a pending miss is intended to reside in the frame. In this case, the cache frame may be used but only after checking the MSHR registers and flagging the appropriate MSHR register. The flag indicates that the returned block has lost its reserved cache frame and is only to be forwarded to the processor and *not* written to cache.

4.3 Memory System

The memory is interleaved into M modules. FIFO (First In First Out) memory buffers queue requests for each module. The coherence of the system is not affected by the buffers [12].

The architecture allows for virtual memory addressing. In this case, each processor has a TLB (Translation Lookaside Buffer) which caches the most recently performed virtual-to-physical address translations. The cache, however, must lock if a TLB miss occurs. The TLB miss may result in a page fault. If a page fault occurs, prefetching and writing memory words beyond the access causing the page fault must be prevented.

5. ANALYSIS

Two models are presented here. Both models are approximate, but useful information can be derived from them. We make the following assumptions in our model:

1. A processor makes d memory references per instruction.

[2] It can not be invalidated, as mentioned before, but in the case of a direct mapped cache may have been overwritten by either another `read` miss or a `write` which mapped to the same cache frame.

2. The distance between references with dependencies in the reference string of a process is fixed and is equal to l. Hence, after a miss, up to l references can be made before the processor blocks and has to wait for the miss to be resolved.

3. For each access the probability of a hit is h and the probability of a miss is $(1-h)$. Successive accesses are independent. Therefore, the number of references between two consecutive misses is geometrically distributed with mean $1/(1-h)$. (Figure 3 illustrates the concept.)

4. The memory access time is constant and equal to T_m.

5. The time to execute an instruction if all accesses hit in the cache is constant and equal to $t_{i,0}$. We associate a time of $t_{i,0}/d$ with each reference.

6. Effects of synchronization and of TLB misses are neglected.

There are several approximations in the model. First of all, in a practical system the dependency distance is usually variable and the memory access time is random because of memory conflicts. These two approximations were made here to facilitate the solution of the model. Second, successive accesses to the cache are correlated; however, the hypothesis of independent accesses is often made in cache models (see for example [15]) and is as good as any other hypotheses in the absence of real program traces. Overall, we feel that the models include the most important parameters affecting the performance of the lockup-free caches and should give indications as to system configurations for which the complexity of lockup-free caches is warranted.

5.1 Model 1: One MSHR Register

It is assumed that only one MSHR register exists in the cache. A single miss does not cause the cache to lock immediately. The cache will block either if a second miss occurs, or if a dependency with the reference of the first miss prohibits any further prefetching. We have to consider two different cases. In the first case, the memory access time is larger than the time during which the processor can continue prefetching without encountering a dependency with a previous miss. That is, $\frac{t_{i,0}l}{d} < T_m$. In the second case, we assume the opposite, that is $\frac{t_{i,0}l}{d} \geq T_m$.

Case 1: When a miss is encountered, it is immediately forwarded to the memory. The number of references which can be overlapped while the miss is being resolved is $\bar{a}$. This number is governed by the probability that a miss occurs during the next l accesses before the processor blocks due to a dependency with an access that missed (Figure 3). Hence $\bar{a}$ is given by:

$$\bar{a} = (1-h) + 2h(1-h) + 3h^2(1-h) + \cdots$$
$$+(l-1)h^{l-2}(1-h) + lh^{l-1}(1-h) + lh^l$$

which can be reduced to:

$$\bar{a} = \frac{1-h^l}{1-h}$$

If T_N is the time to execute N instructions then

$$T_N = N(t_{i,0} + (1-h)dT_m) - N\frac{1-h^l}{1-h}(1-h)t_{i,0}$$

The time to execute one instruction is:

$$t_{i,0}h^l + (1-h)dT_m$$

Hence the slowdown factor is:

$$\frac{1}{h^l + (1-h)dT_m^0}$$

Case 2: In this case the amount of overlap of hits is only limited by the probability that a second miss occurs before the first miss is resolved. This case may result if the code is restructured by the compiler such that the distance between dependencies which cause the processor to block is very large. Let N be the number of references between two successive misses. $\bar{N} = 1/(1-h)$ and the average time to execute between two misses becomes

$$T_N = \frac{t_{i,0}}{d}\sum_{k=0}^{\infty} kP[N=k] = \frac{t_{i,0}}{d}\sum_{k=1}^{\infty} P[N \geq k]$$

$$= \frac{t_{i,0}}{d}\left[\sum_{k=1}^{dT_m^0} 1 + h^{dT_m^0} + h^{dT_m^0+1} + \cdots\right]$$

$$= T_m + \frac{t_{i,0}}{d}\frac{h^{dT_m^0}}{1-h}$$

This result comes from the fact that if the inter-miss distance N is less than dT_m^0, the cache blocks the processor for a time $T_m - \frac{Nt_{i,0}}{d}$. The average time per instruction is given by:

$$\frac{dT_m + t_{i,0}\frac{h^{dT_m^0}}{1-h}}{\frac{1}{1-h}}$$

The slowdown factor for this case is:

$$\frac{1}{h^{dT_m^0} + (1-h)dT_m^0}$$

5.2 Model 2: Infinite Number of MSHR Registers

In this model, it is assumed that the number of MSHR registers is infinite. As for the previous model, we consider two cases.

Case 1: In this case we assume $t_{i,0}l/d < T_m$. After the first miss, at reference $1/(1-h)$, l references can be generated by the processor while the first miss is resolved by the memory system in time T_m. Therefore, a processing time of $t_{i,0}l/d$ can be overlapped with the first miss. If any one of these l references causes a miss, it can be serviced immediately because there are an infinite number of buffers. The misses occurring for the l references do not cause additional blocking due to dependencies because the processor has to wait until the first miss is resolved before initiating new references. When the first miss is completed, the processor can continue execution, and because of the geometric distribution assumption, the next miss occurs $1/(1-h)$ references later, on the average. The sequence of events after the first miss is repeated.

Therefore, in a time

$$\frac{t_{i,0}}{d}\frac{1}{1-h} + T_m$$

a number of $\frac{1}{1-h} + l$ references are performed, corresponding to

$$\left(\frac{1}{1-h} + l\right)\frac{1}{d}$$

instructions.

The average time per instruction is

$$\frac{t_{i,0} + T_m d(1-h)}{1 + l(1-h)}$$

and the slowdown factor is

$$\frac{1 + l(1-h)}{1 + T_m^0 d(1-h)}$$

Case 2: In this case we assume that $t_{i,0} l/d \geq T_m$. When a reference has a dependency with a previous access that missed, the miss has had the time to complete and therefore, the dependencies do not block the processor. We have achieved total overlap of miss handling and the slowdown factor reaches its maximum value of 1.

6. DISCUSSION

6.1 Performance Interpretations

Figure 4 shows the ratios of MIPS rates of a system with one MSHR buffer per cache and of a system with locking caches. The improvement is greatest for low hit rates and memory access times in the range of $T_m^0 < l/d$. This is due to the fact that if $T_m^0 > l/d$, most misses will cause some blocking, because before a miss can be resolved a dependency will occur with the reference that misses. If $T_m^0 < l/d$, however, the cache will only lock if a second miss occurs during the service of the first miss; some misses will be totally overlapped with other references and the amount of overlap also increases with T_m^0.

For larger hit rates, such as h=0.98, the improvement due to the lockup-free cache with only one MSHR is low because misses are rare and the only savings per miss, with respect to a locking cache, are l references. These results confirm Kroft's results for the average access time as a function of the the number of MSHR registers. Kroft's results (derived from a prototype) indicate very poor performance for a cache with a single MSHR register and very good performance for a cache with up to four MSHR registers. Kroft further states that very little is to be gained by using more than four registers.

Figure 5 shows the ratio of MIPS rates of a system with an infinite number of MSHR buffers and a system with locking caches for different hit rates, as a function of T_m^0. The improvement of performance is highest for a low hit rate of h=0.8. This is to be expected, since a high miss ratio results in multiple misses in a streak of l consecutive references and these misses can be overlapped. In the ideal case, for example, l misses occur sequentially, until the system blocks due to a dependency with the first miss (we assume $l/d < T_m^0$). The misses are resolved one after another and the total penalty paid for the sequence of l misses is only T_m^0. Hence, the system benefits from frequent misses.

For $l/d > T_m^0$, the system with an infinite number of buffers operates at maximum speed, with a slowdown factor of 1. In this case no miss ever causes a penalty, since the system does not block on multiple misses and all misses are always resolved before a dependency can occur. For higher hit rates the probability of overlapping misses decreases and performance ratio degenerates to the case of the cache with only one MSHR buffer. The number of buffers used, on average, is an interesting parameter. It is possible to estimate the average number of busy buffers in the cache as follows. Let t_i be the average time per instruction. The average time to execute N instructions is Nt_i and a total of $Nd(1-h)$ misses must be processed, requiring a total service time of $Nd(1-h)T_m$ from the miss buffers. Therefore the average number of busy buffers is

$$\frac{d(1-h)T_m}{t_i} = \frac{d(1-h)T_m^0}{1 + d(1-h)T_m^0}[1 + l(1-h)] \quad (case\ 1)$$

$$or\ = d(1-h)T_m^0 \quad (case\ 2)$$

This value is upper-bounded by $1 + l(1-h)$. For all of the examples of Figure 5, the average number of busy buffers is less than 2.75.

6.2 Consequences

For an otherwise efficient lockup-free cache to be of consequential benefit, multiple MSHR registers must be implemented. Only in the case when the hit rate is low, and $T_m^0 < l/d$ does a single MSHR buffer offer worthwhile improvement.

A cache with a *number* of MSHR buffers can, however, be very useful. Such a cache can offer substantial improvement over a locking cache when the hit rate is not very high. This fact can benefit systems in three particular circumstances.

1. Any system with a low hit rate benefits from lockup-free caches consistently. In our sample architecture, two benefits are derived from the fact that the cache block size is one word. Namely, the bus traffic is minimized and `write` operations *always* hit at the cache. The drawback of the small block size is, however, a lowered hit rate. The system can accommodate this lower hit rate because a lockup-free cache is used.

2. Context switches always cause very low transitory hit rates. A lockup-free cache can help "smoothen" the performance dip in the system behavior after such context switches.

3. A cache which usually exhibits a good hit rate, may be sensitive to particular "pathological" workloads which can lower the hit rate for particular applications, or, especially, for operating system calls. As in the case of context switches, the cache can adapt to such workload changes if it is lockup-free.

It is interesting to note, that a system with single word blocks, while likely to lower the hit rate, also experiences a favorable shift in the distribution of misses. Misses are much more likely to appear in bursts, since an initial sequential access to an array or other data structures will cause a miss for every access. This is also the case when double- or quad-words are accessed. A system with lockup-free caches, will exhibit better performance if misses are clustered together than if they are homogeneously distributed. This characteristic can be taken advantage of if the compiler can generate `load` instructions for data to be used in the future, ahead of time. In this case the "blind" prefetching associated with larger block sizes, has been replaced with selective "smart" prefetching under compiler control.

6.3 Dependency Effects

Figure 6 shows the MIPS ratio of the infinite buffer system and the locking cache system as a function of l. For this case, $T_m^0 = 20$ and $d = 1.5$. As is to be expected the performance ratio increases linearly until $l/d = T_m^0$ up to a point where the lockup-free cache system operates at peak speed and remains constant. In the case of a hit rate $h = 0.8$, the maximum performance improvement over the locking cache system is 700%. Since the inter-dependency distance l affects the performance of the system greatly, techniques on how to increase l are important.

(1) The compiler can attempt to increase l within the instruction stream by reordering instructions in a more favorable way.

(2) All `load` instructions should be non-blocking; as they are implemented in the IBM RT processor [16].

(3) The compiler can generate special instructions which explicitly load data into cache, long before they are needed. Such non-blocking `load` operations enable the compiler to control selective prefetching of data. These loads should be generated in bursts.

(4) For instruction access misses not to cause blocking, several consecutive instructions can be prefetched ahead of instruction decode time. Only `branch` instructions will cause blocking in this case. If branches are delayed as in RISC processors, l can be increased.

(5) Some architectures can naturally exhibit a high value for l. For example, if a vector processor is attached to the system, long strings of vector register `load` instructions are likely to be executed frequently.

7. CONCLUSION

In this paper we have shown how a multiprocessor can be configured with lockup-free caches. Vital to such a system are three key concepts:

1. The correctness of the system.
2. The efficiency of the interconnect.
3. The efficiency of the cache architecture.

We have shown that the processors of a multiprocessor system may resolve multiple misses concurrently if the system is weakly ordered. Weakly ordered multiprocessors require that shared writable data are modified exclusively from within critical sections. This restriction can be enforced by the compiler.

The interconnect we propose consists of packet-switched buses. By using a cache block size of one word, the bus traffic is minimized. Hence, more processors can be connected to the buses and contention is lower. The fact that a one word block size decreases the cache hit rate is overcome by the fact that the caches are lockup-free.

We have shown that in a weakly ordered system, maintaining cache coherence is very simple, when the cache block size is one word. The assumption of a one word cache block size, eliminates all cache `write` misses and minimizes bus traffic since prefetching is controlled by the compiler and is not done blindly.

Overall, we believe that bus-based multiprocessors with lockup-free caches are both viable and useful. One of the most interesting features of such a system is the adaptability to the cache miss rate it exhibits. When hit rates are high, the improvement due to overlapping misses is low. However, when the hit rate declines, the efficiency of the lockup-free caches improves rapidly. This characteristic makes lockup-free caches a particularly appealing feature for systems with a large variety of types of workloads, and hence varying hit rates.

8. REFERENCES

[1] A.J. Smith, "Cache Memories", *ACM Computing Surveys*, Vol. 14, No.3, Sept. 1982, pp.473-530.

[2] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, Vol. C-27, No.12, December 1978.

[3] D. Kroft, "Lockup-free Instruction Fetch/Prefetch Cache Organization," *Proceedings of the 8th International Symposium on Computer Architecture*, June, 1981.

[4] J.R. Goodman, "Using Cache Memory to reduce Processor-Memory Traffic", *Proceedings of the 10th International Symposium on Computer Architecture,*, June 1983, Stockholm, Sweden, pp. 124-131.

[5] J.R. Goodman, "Cache Memory Optimization to Reduce Processor/Memory Traffic," *Journal of VLSI and Computer Systems*, 2,2 (1987), pp.61-86.

[6] R.L. Lee, P.-C. Yew, and D.H. Lawrie, "Multiprocessor Cache Design Considerations," *Proceedings of the 14th International Symposium on Computer Architecture*, Pittsburgh, June 1987.

[7] M.C. Easton and R. Fagin, "Cold-start vs. Warm-start miss ratios," *CACM*, Vol. 21, No. 10, Oct. 1978.

[8] H. S. Stone, "Footprints in the Cache," *Proceedings of Performance '86 and ACM Sigmetrics 1986 Joint Conference on Modelling, Measurement and Evaluation*, May 1986.

[9] M. Dubois and F.A. Briggs, "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers*, Vol. C-31, No.11, November 1982.

[10] M. Dubois, "Effect of Invalidations on the Hit Ratio of Cache-Based Multiprocessors," *Proceedings of the 1987 International Conference on Parallel Processing*, Aug. 1987.

[11] A.J. Smith, "Cache Evaluation and the Impact of Workload Choices," *Proceedings of the 12th International Symposium on Computer Architecture*, 1985.

[12] M. Dubois, C. Scheurich and F. Briggs, "Memory Access Buffering In Multiprocessors," *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986.

[13] C. Scheurich and M. Dubois, "Correct Memory Operation of Cache-Based Multiprocessors," *Proceedings of the 14th International Symposium on Computer Architecture*, Pittsburgh, June 1987.

[14] J.A Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, Vol. 4, No. 4, Nov. 1986.

[15] J.H. Patel, "Analysis of Multiprocessor with Private Cache Memories," *IEEE Transactions on Computers*, Vol. C-31, No. 4, April 1982.

[16] C.E. Gimarc and V.M. Milutinovic,"A Survey of RISC Processors and Computers of the Mid-1980's," *IEEE Computer*, Sept. 1987.

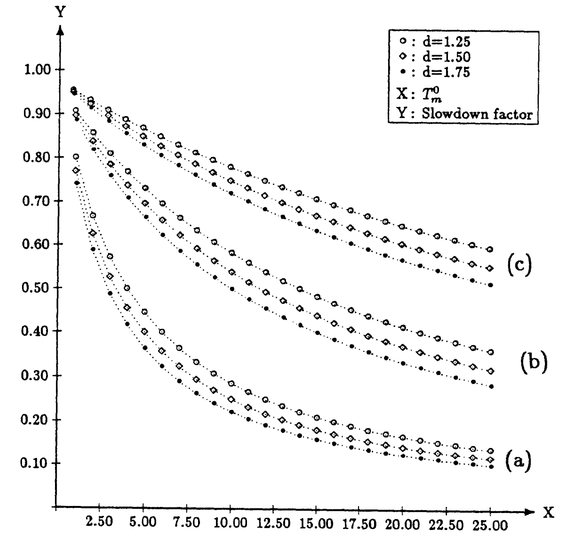


Figure 1: Slowdown factor vs. T_m^0 for hit rates of (a) h=0.8, (b) h=0.95, and (c) h=0.98.

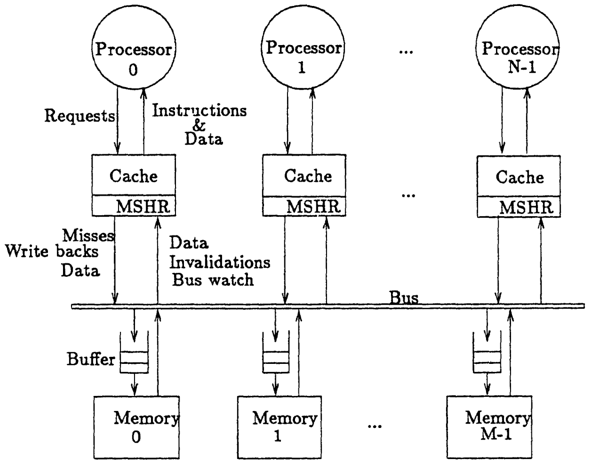


Figure 2: A cache-based multiprocessor with lockup-free caches.

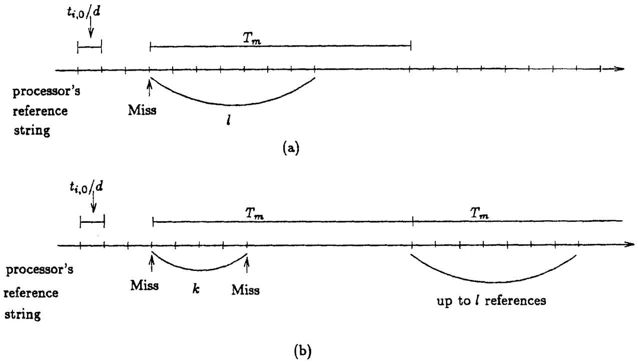


Figure 3: Timing of a lockup-free cache with one buffer. (a) No miss occurs in the l references following the first miss; maximum overlap is achieved. (b) A miss occurs. Only $k < l$ references can be overlapped.

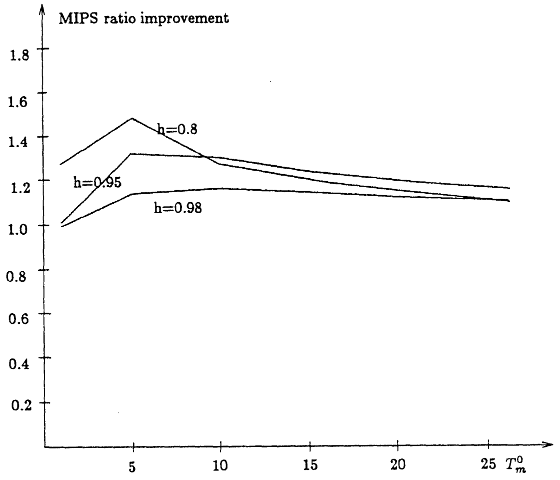


Figure 4: MIPS ratio improvement with one miss buffer, $d = 1.5, l = 10$.

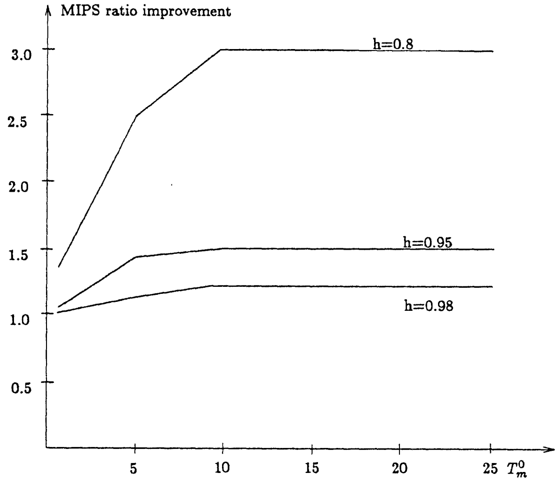


Figure 5: MIPS ratio improvement with an infinite number of buffers, $d = 1.5, l = 10$.

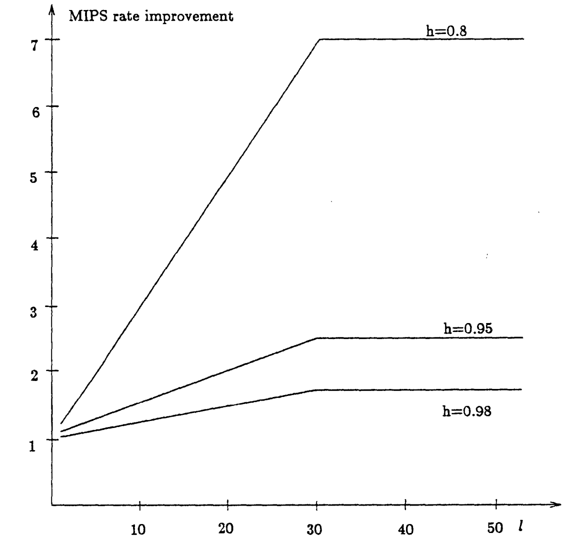


Figure 6: MIPS ratio improvement as a function of $l, T_m^0 = 20, d = 1.5$.

POET: A Tool for the Analysis of the Performance of Parallel Algorithms*

Anselmo A. Lastra and C. Frank Starmer
Departments of Computer Science and Medicine
Duke University
Durham, NC 27710

Abstract

A tool to aid in the analysis of the execution time of parallel algorithms is presented. The tool consists of a simple language for describing the algorithms and an interpreter that determines the execution time on a given number of processors.

The key concept is the separate specification of local computation and remote memory access in the algorithm description. This allows the interpreter to simulate the communications of the target parallel machine. An accurate simulation of the memory access delays coupled with the specified amount of local computation results in the predicted parallel execution time. Operation of the system has been validated by comparing predicted versus observed execution times for numerical algorithms on the Butterfly and Butterfly Plus Parallel Processors.

Introduction

The exploration of the time performance of algorithms is ubiquitous in computer science. Practitioners range from the novice programmer deciding upon a sorting algorithm to the computer scientist investigating the theoretical complexity of algorithms. The tool described in the paper is aimed at an investigator between these two extremes. It is designed for the practical and speedy performance analysis of algorithms for shared memory parallel architectures.

There have been parallel performance analysis tools described in the literature. In particular, those for analyzing the speedup of FORTRAN code[6], and as part of a program development environment[5], as well as theoretical[8] and probabilistic[3] models of parallel performance. These, however, don't address the needs of someone trying to decide on a particular algorithm or investigating the suitability of a particular architecture. Presumably at that stage programs have not been written and may never be. What is needed is a tool that analyzes performance given a simple description of an algorithm. This paper describes such a tool. POET (Prediction of Execution Times) is a system to facilitate execution time analysis of the parallel implementations of algorithms. POET consists of a language for specifying algorithms and an interpreter for the language that predicts the time performance of the algorithm on an MIMD shared memory parallel machine.

On a single processor, a good way to predict the execution time of a section of code is to "count operations". In other words, combine an analysis of the flow of control with an estimate of the run time of small sections of code, such as inner loops or individual operations. This yields an analytical expression, or perhaps just a value, for the execution time of the algorithm.

This method fails on a parallel machine because of interactions between processors. On a shared memory machine the execution time of code on a particular processor is affected not only by memory access to remote memories, but also, on some architectures, by accesses of the local memory by other processors. Other memory related phenomena that affect execution time are cache utilization or bus saturation.

Since the factors that perturb execution time are memory related, it is reasonable to assume that by separating purely local computation from remote memory access, one could use a modified version of the method that worked for the uniprocessor. This is what POET accomplishes. The algorithm specification language allows for the separate specification of local computations from remote fetches and stores. The

* Supported in part by grant HL-32994 from the National Institutes of Health and contract ONR-4414804 from the Office of Naval Research

interpreter then uses this information to simulate the memory transfer behavior of the target parallel machine. The estimated times of local computations combined with the delays incurred by the memory access patterns then constitute the total run time.

Currently simulation modules for the Butterfly[1] and Butterfly Plus[2] parallel processors have been implemented and validated. We believe that implementations simulating other shared memory MIMD machines, such as the RP3[4], will require modifications of only one module.

The Language

To determine flow of control, the simulator interprets a language with a syntax similar to that of C. New statements were added to describe the parallelism and memory transfers. Only a restricted subset of the data types and expressions in C were retained, mainly to keep the implementation of the interpreter more manageable. An example program is shown in Figure 1. Note that whenever "processor" is mentioned, it denotes a simulated processor, not one on which POET is running.

There are two statements for specifying the use of time by the algorithm, *compute* and *transfer*. The compute statement specifies an amount of purely local computation. It has one argument which is an expression that, when evaluated, yields the time that the computation adds to the clock. For example, for matrix multiplication an individual computation may consist of the sum of the execution times of an add and a multiply. The execution time specified for the compute statement may be an estimated time, instruction execution times from the manufacturer of the hardware, or perhaps time from a benchmark.

The *transfer* statement represents a store or fetch operation to a remote memory. There are three arguments. The first argument specifies which remote memory is to be accessed. The second argument specifies the time that a memory access adds to the local machine and the third the time consumed by the remote processor (due to one or more lost memory cycles perhaps). These times are variables because one may want to specify different types of transfers, such as an integer, double precision floating point, or maybe a block of bytes. If simulating single memory architectures, the first and third arguments, the remote memory number and time penalty on remote processor, are not applicable. Times for data transfers are obtained from the manufacturer's specifications or from benchmarks.

```
for(i = 0; i < n; i = i + procs)
    for(j = 0; j < n; j++){
        for(k = 0; k < n; k++){
            transfer(mem1, FetchDouble, RemoteFetchDouble);
            transfer(mem2, FetchDouble, RemoteFetchDouble);
            compute(DoubleAdd + DoubleMult);
            }
        transfer(mem3, StoreDouble, RemoteStoreDouble);
```

Figure 1. Simplified code fragment for matrix multiply. The two matrices of size n are on memories mem1 and mem2 with the result going to mem3. DoubleAdd and DoubleMult are the execution times of an add and a multiply, respectively. FetchDouble is the time consumed by the local processor for fetching a double precision floating point number while RemoteFetchDouble is the time that the remote processor is delayed because of the fetch. StoreDouble and RemoteStoreDouble are analogous times for storing a number. Procs is the number of processors. This simple model does not include the overhead of index or pointer calculations.

The *parallel* and *wait* statements control the parallel execution of the simulator. They are used to begin simulated execution on the user specified number of processors, to implement critical regions, to synchronize, and to return execution to one processor. The *wait* statement places the processor executing it into an idle state until the expression specified as an argument becomes true. *Wait* is a useful construct for synchronizing groups of processors.

Initial simulated execution is on one processor. The *parallel* statement causes the simulator to begin simulated execution on the number of processors, or perhaps just different processes, specified as an argument. Upon reaching the end of the parallel block, simulated processors are set to an idle state. When all processors have completed the block, processing is continued on processor number 0, the processor on which original execution began.

All variables in the POET language must be declared. There are two main classes of variables, *local* and *shared*. An individual copy of each of the local variables is made for each of the simulated processors while there exists only one copy of each of the global variables.

Other statements in the POET language are identical to those in C. They include statements for flow of control such as *if* and *for*. The more commonly used arithmetic and logical operators have been implemented.

The Simulator

The simulator loads a program, in the language described above, that specifies the flow of execution to be modelled. The program is parsed and converted into an intermediate language. The intermediate language consists of a restricted subset of instructions. For example, the *for* statement is broken up into an *if* statement, two arithmetic expressions, and a *goto* statement. In a manner similar to an assembly language without the word size restrictions. Having such a restricted set of instructions made the interpreter easier to design and modify.

The interpreter works on one simulated processor at a time and continues working on that processor until an instruction is reached that consumes execution time or affects another processor. The processor on which to work is determined by a scheduler described below. State information is kept for each of the simulated processors. This includes the contents of all of the local variables, a program counter, and the current processor state such as transferring data, computing, idle, and waiting.

A clock is also kept for each simulated processor. A processor's clock is incremented when a *compute* or *transfer* instruction is interpreted. The scheduler uses these clocks to determine which processor to simulate next. The process to schedule next is the one that has the smallest value on its execution clock. If there is more than one clock with the same minimum time, they are scheduled for the interpreter in a round robbin fashion. This scheduling algorithm should yield an execution trace very close to that of the processors executing in parallel. A global *clock* is kept which shows how far the simulation has come. This clock eventually yields the overall run time.

The key to the simulator is the module in the interpreter that mimics the processor to memory communication of the target machine. For the current target machine, the BBN Butterfly, it is particularly easy to implement. When a processor requests access to another memory (the memory of another processor) with a *transfer* instruction, the simulator examines the state of the remote processor. If the remote processor is executing a local computation (a *compute* statement) or is idle, the simulated transfer occurs immediately. The remote processor is penalized for lost memory cycles and the transfer time is added to the execution time of the local processor. Both of these times are specified as arguments to the *transfer* instruction.

If the remote processor is busy transferring data, a retry would occur after the remote processor completes the transfer or transfers. In the simulator instead of executing a retry, a queue is kept of data requests for each processor. This queue is examined when the processor completes a data transfer. The requesting processor blocks until the simulated transfer completes.

Note that we do not model individual paths through the Butterfly switch, we assume that one is available. This turns out to be a good simplifying assumption because the switches on the Butterfly do not block when a transfer fails[2,7]. Rather, the path is released and the transfer is retried a short random time in the future. Another reason that a particular path need not be simulated is that many Butterfly processor configurations have alternate paths between nodes.

It is true that a high volume of data transfers would produce path saturation and cause this implementation of POET to produce inaccurate timings. However, when simulating extreme volumes of interprocessor transfers, high accuracy was not deemed necessary since the degree of parallelism would have presumably reached a plateau. This model trades a slightly limited range for simplicity.

The memory access simulator is the only one which varies with target architectures. Simulators for some architectures would be more difficult to implement. For example, on an architecture with a single memory and a large cache for each processor, a model of the steady state distribution of cache hits would have to be developed. Also a characterization of the delay on the global bus would be important.

The simulator maintains timing information for individual processors and can output the results in a variety of ways. Control is also provided over the number of processors in a simulation. Optional tracing is provided as an aid in debugging or to visualize bottlenecks in the algorithms being studied.

Performance of the BBN Butterfly version of POET was validated by comparing predicted execution times generated by POET with measured execution times of numerical algorithms. The testing was conducted on both a Butterfly Parallel Processor, and the newer, faster, Butterfly Plus.

The algorithms used are representative of those used for scientific calculations. They included matrix manipulation algorithms, and those for the solution of ordinary differential equations.

Agreement between predicted and measured execution times was generally excellent. The only exceptions were algorithms designed to cause the path saturation effect described above. One memory was accessed repeatedly by all of the processors. When this occurred, the simplified Butterfly data transfer model of this version of POET resulted in optimistic execution times. For our purposes this was acceptable since these were artificially poor algorithms designed for testing POET. If more accurate results were desired under these conditions, a more complex data transfer model would be necessary.

Conclusions

We believe that there are many advantages to this technique for exploring algorithm performance. First, it is easy to specify the algorithms. Working code does not have to be produced before estimates can be made of the execution time. Many different algorithms may be tested experimentally in the time it would take to code and debug only one. Second, the simulator runs quickly. Even if code is available, benchmarking it is likely to take much longer than testing it on POET. Third, the parallel hardware does not have to be available. One may explore ideas when the parallel machine is busy, or down, or even while awaiting delivery.

Another use for POET might be in testing ideas for parallel machines. By varying the times specified for computation and for data transfer, one can predict the increase or decrease in the degree of parallelism for a given change in the speed of the processor, memory, or interprocessor switch. With modifications to the interconnect module of POET, the performance of experimental network topologies may be studied.

An interesting addition to POET would be a graphical display of simulated execution. This would aid in detecting bottlenecks and in understanding the parallel execution of algorithms. Another, more difficult, enhancement would be automatic benchmarking of existing code.

REFERENCES

1. *Butterfly Parallel Processor Overview,* BBN Laboratories Inc., 1985.

2. *Inside the Butterfly Plus,* BBN Advanced Computers Inc., 1987.

3. Lester, B. P., A System for Computing the Speedup of Parallel Programs in *Proc. of the 1986 Conference on Parallel Processing,*1986, 145-152.

4. Pfister, G. F., Brantley, D. A., George, D. A., Harvey, S. L., Kleinfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A., Weiss, J., The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture in *Proc. of the 1985 Conference on Parallel Processing,* 1985, 764-771.

5. Purtilo, J., Reed, D. A., Grunwald, D. C., Environments for Prototyping Parallel Algorithms, University of Illinois Department of Computer Science Technical Report, 1987.

6. So, K., Bolmarcich, A. S., Darema, F., Norton, V. A., A Speedup Analyzer for Parallel Programs in *Proc. of the 1987 Conference on Parallel Processing,* 1987, 653-661.

7. Thomas, Robert, H., Behavior of the Butterfly^{TM} Parallel Processor in the Presence of Memory Hot Spots in *Proc. of the 1986 Conference on Parallel Processing,* 1986, 46-50.

8. Vrsalovic, D., Siewiorek, D. P., Segall, Z. Z., Gerhinger, E. F., Performance Prediction for Multiprocessor Systems in *Proc. of the 1984 Conference on Parallel Processing,* 1984, 139-146.

A Queueing Network Model For A Cache Coherence Protocol On Multiple-bus Multiprocessors*

Qing Yang and Laxmi N. Bhuyan

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504-4330.

Abstract

The memory latency in a shared memory multiprocessor system can be reduced by either the use of a high bandwidth interconnection network or the incorporation of private cache memories. This paper presents the performance analysis of a system that employs a high bandwidth multiple-bus network and private cache memories. The cache coherence protocol is a modified version of the Write-once protocol proposed for single bus architecture and the multiple-bus network is asynchronous packet s\.itched. A queueing network model consisting of mixed multiple class customers has been developed. The model captures the effects of both multiple-bus contention and the cache coherence protocol on the system performance. To reduce the computational complexity of the model, a simplified algorithm based on flow equivalence technique has been developed. Numerical results obtained from our model show that a high bandwidth network such as multiple-bus is necessary for a large system because the single bus gets saturated very rapidly and creates system bottleneck.

1. Introduction

The problem of memory latency has been considered as a major obstacle in the evolution of shared-memory multiprocessor systems. Extensive studies aiming at reducing the memory latency have been carried out in the past. There are two basic ways of dealing with this memory latency problem: 1) design of a more cost-effective interconnection network that offers high communication bandwidth [2]; 2) use of private cache memory to reduce memory access time and memory bandwidth requirement [10].

A great deal of work has been done in design and analysis of various interconnection networks such as crossbar, multistage interconnection networks, and multiple-buses. Compared to crossbar and multistage interconnection networks, multiple-bus interconnection provides several advantageous features such as flexibility, expandability and fault tolerance [3, 8, 16]. One can configure the multiprocessor system in a variety of ways to provide a range of trade-offs among bandwidth, connection cost and reliability. However, as was pointed out by Winsor and Mudge [15], the cache based multiple-bus multiprocessor may suffer from difficult synchronization problems. Hence, asynchronous multiple-bus systems seem to be more attractive for large size cache based multiprocessors. In [17], queueing network models have been developed to analyze the performance of asynchronous, packet-switched multiple-bus system with buffers. It has been shown [17] that a packet switched multiple-bus system provides high bandwidth and flexibility. None of the previous studies on multiple-bus system takes into account the incorporation of private cache memories, except for [5] where some theoretical bounds for the system throughput are developed. The use of private cache memories in a multiprocessor system introduces the complex cache coherence problem [4, 10] because multiple copies of a memory block may reside in different caches at any given time. Modification of any copy of a shared memory block by a processor in its local cache may cause an obsolete value of the shared data in the main memory and other caches that are currently having a copy of this block. The avoidance of this cache inconsistency problem is vital to the correct operation of the shared-memory multiprocessors.

A number of cache coherence protocols have been proposed in the literature recently, which can be broadly classified into two groups. The first group employs a centralized global directory scheme while allowing a general interconnection network to be used [13]. The second group allows the cache consistency to be maintained in a decentralized manner but limiting the interconnection to be only single shared bus [1, 6]. The write-once protocol, proposed by Goodman[6], was the first distributed protocol to appear that provides a good compromise between write-through and write-back protocols which are used in commercial machines [9]. However, both of these two classes of protocols have potential problems. The central directory protocols allow a general IN to be used but the performance of the system is highly dependent on the central directory. The distributed protocol, on the other hand, tries to remove this bottleneck, but instead it is passed from the central directory to the single shared bus. As has been shown in [1] and [14], the single bus creates system bottleneck when the number of processors exceeds 10. Hence, it seems to be necessary to develop cache coherence protocols that allow high bandwidth interconnection network to be used while keeping the advantage of distributed control. In this paper, we consider a cache coherence protocol based on Goodman's Write-once protocol that can be applied to high bandwidth packet switched multiple-bus systems [17].

Archibald and Baer have presented a comprehensive performance comparison of various single bus protocols by means of simulation [1]. An analytical model for the single bus protocol based on write-once

*This research is supported by NSF Grant #DMC-8513041 and a grant from Louisiana Board of Regents.

as well as its variants has been reported by Vernon and Holliday in their paper [14]. Their model is exact and is based on Generalized Timed Petri Nets technique. However, it can not be easily extended to large system sizes because of the complexity. In this paper, a queueing network for cache based asynchronous multiple-bus multiprocessors will be developed that consists of both open and closed customer classes [7]. The effects of both multiple-bus contention and the cache protocol on the system performance will be studied. The model can be solved by using standard MVA algorithm and flow equivalence technique[7] to obtain performance values for a variety of system parameters with reasonable computational cost.

In the following section, we will give a brief description of the system organization and the cache protocol for the proposed system. The assumptions that are used in our analysis and the queueing network model of the cache based multiple-bus multiprocessor are presented in Section 3. Section 4 discusses the performance results and Section 5 presents the conclusions.

2. The System Organization and Operational Characteristics.

Fig.1 illustrates the cache-based multiple-bus multiprocessor configuration. Associated with each processor is a private cache memory through which all memory accesses pass. A set of B packet switched buses connect all the N caches with shared main memory which is also divided into M interleaved modules. The communication between the caches and between a cache and the main memory is performed through system buses. A requesting cache (called master) which issues a memory access request releases the bus immediately after the request packet containing the memory address, master id., and desired operation(read/write), etc. has been sent to the slave device. The released bus can be used for other purpose while the desired operation in the slave (memory controller or cache controller) is in progress. After the operation is finished, the slave device acts as a pseudomaster to packetize the response data and sends it back to the requesting cache through a system bus. As a result of this, the system buses can be well utilized and high system throughput can be expected. However, buffers that temporarily hold incoming and outgoing packets from each device are necessary. The packet transmission can be done on any one of the system buses as determined by the arbiter. The cache and memory controllers must be able to receive more than one packets simultaneously from the buses, otherwise a packet may be lost.

The timing of the system buses is asynchronous in the sense that there is no centralized global clock that distributes clock signals to all the devices in the system. The data transfer on a bus is done by means of interlock handshaking. However, the internal cycle times of all processors and caches are assumed to be the same and this cycle time constitutes the basic time unit in our foregoing analysis. In [18], we have presented a somewhat detailed design of the internal organization of each private cache as well as the cache protocol for synchronous packet switched bus systems. It has been shown in [18] that the proposed system satisfies the sequential consistency requirement [12]. For asynchronous buses, concerned in this paper, the design of cache organization and protocol should be similar except for different implementations at the circuit level. For the purpose of completeness, a brief review of the cache protocol follows.

The state of a memory block viewed by a particular cache can be one of the following: 0), not present, 1), valid, 2), written-once, 3), dirty, and 4), invalid (see Figure 2). When a processor read results in a cache hit, the cache controller will supply the requested word to the processor without changing the state of the cache block containing the word. On a read miss, the cache controller locates the place where the requested memory block resides, flushes a cache block to make room for the incoming block, and loads the block through one of the system buses. The memory block is read from one of the main memory modules provided that none of the caches has a dirty copy of the memory block. Otherwise, the memory block is loaded from the cache that has a corresponding cache block with dirty state. Once the block is loaded, the state is set valid. When the cache controller receives a write request from the processor, it first checks the state of the cache block into which the write is to be performed. If the state of the block is dirty or written-once, the write can proceed without any delay except for setting the state to be dirty. If the state of the block is valid, the cache controller has to acquire one of the system buses to broadcast an invalidation signal to other caches and write through the word to be modified into main memory before the write operation can be performed. Upon a write miss, the cache controller performs the same operation as for a read miss except that the requested memory block is loaded with dirty state and other copies in the system are invalidated.

There are B snooping controllers associated with each private cache. Each of the B snooping controllers monitors one of the system buses for read and write from other caches. There are basically four types of bus transactions: Shared Read (SR), Dirty Read (DR), Write Invalidation (WI), and Write Back (WB). An SR transaction is due to a read miss request issued by a cache. Upon a write miss, a cache controller will generate a DR (since the requested block is loaded with dirty state) request. A WI

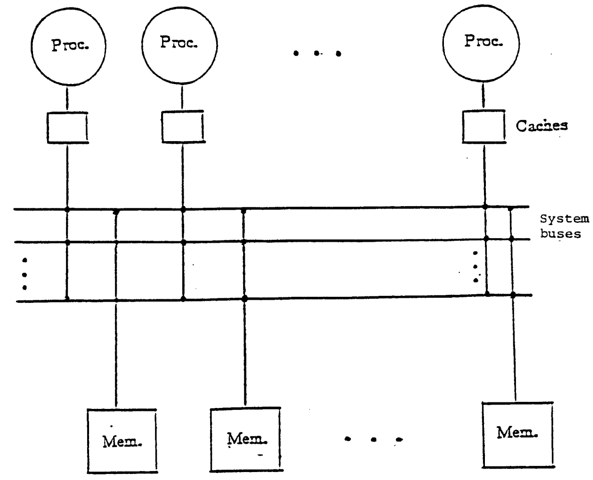


Figure 1. A cache based multiple-bus multiprocessor.

transaction is started by a cache that serves its processor's write request into a valid cache block. If a block to be flushed (in order to make room for a incoming block) is in dirty state, the write back of this block is necessary. The WB transaction is caused by a cache that writes back a replaced block. If a snooping controller detects an SR or DR, it first determines whether its own cache has a dirty copy of the block requested on the bus. If so, the snooping controller must inhibit the main memory from responding to the bus read and provide the data to the requesting cache. The local copy of the block is changed to invalid in case of DR. In case of SR, the block is written back to the main memory and the state is changed to valid. If the local copy of the block is in a state other than dirty, the snooping controller invalidates the copy upon a DR request and sets state to valid upon an SR request. The snooping controller invalidates the corresponding cache block in its local cache when a WI bus transaction is detected. During each of these cache operations requested by snooping controllers, requests from the processor are suspended.

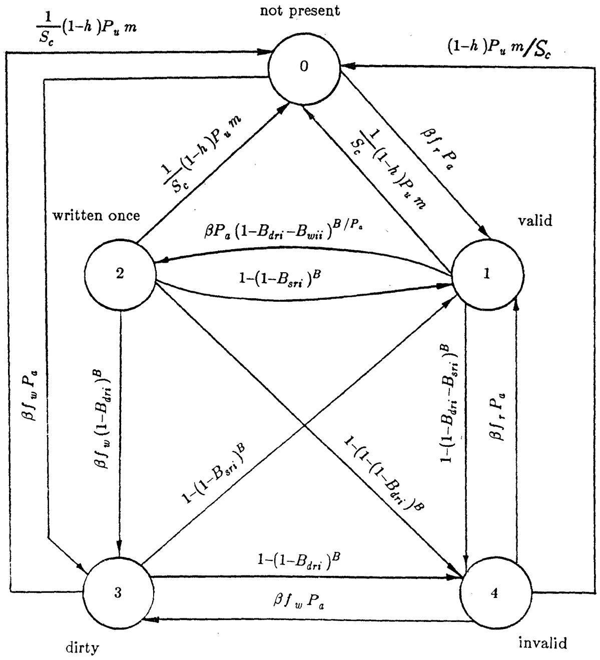
Due to the multiple buses, directly applying the above protocol to the multiple-bus system may result in race and hazard conditions. In [18], we have defined 5 types of hazard conditions, namely DR-SR hazard, DR-DR hazard, DR-WI hazard, SR-WI hazard and WI-WI hazard. An occurrence of any hazard condition described above may cause a program error. These problems can be resolved by using the same technique as described in [18].

3. Queueing Analysis

3.1. Assumptions and The Work Load Model

In the analysis presented in this section, we shall assume homogeneity of processors, memory modules and buses. That is, all processors in the system are considered to be behaviorally identical, so are the memory modules and buses. After an amount of internal computation, called thinking time, a processor generates a memory request which is sent to its private cache. The thinking time of each processor is a random variable, depending on the type of the instruction, and is assumed to be arbitrarily distributed with a mean of Z cycles. Memory access time and the transfer time of a data block on a bus are assumed to be fixed at T and t_r cycles, respectively. The cycle time of a cache memory is assumed to be constant value which is same as a processor cycle. These assumptions are reasonable because for a given design of a machine, the basic data unit that is transferred between the main memory and caches (such as a block) or between a processor and its local cache (such as a word) is fixed in size. However, the model developed in this paper can be easily extended to deal with generally distributed random variables of memory access times and bus transfer times.

The work load model selected for our analysis is similar to the one developed by Dubois and Briggs [4]. The memory reference stream generated by a processor is the merging of two streams: one for private memory blocks that can be accessed only by one processor and the other for shared blocks which can be read or written by any processor in the system. There are totally N_{sb} shared memory blocks in the system and each private cache is assumed to be capable of holding S_c cache blocks(shared or private).



$\beta = p_i q_s m P_u$.

Figure 2. Markov state diagram for a memory block.

The probability that a memory request issued by a processor addresses one of the N_{sb} shared blocks is represented by q_s and the probability of addressing a private block is $1-q_s$. Similarly, a memory request is a read with probability f_r and a write with probability f_w. It is also assumed that a shared memory request addresses any one of N_{sb} blocks equally likely, i.e. a uniform reference model for shared blocks is assumed. If a request is for a private block, it is a cache hit with probability h and miss with probability $1-h$. When a cache miss occurs, the cache controller randomly selects one cache block to be replaced to make room for the incoming block. In most of the existing cache systems, *LRU* (Least Recently Used) replacement policy is primarily employed. The random replacement policy is assumed here to simplify our analysis. One should note, however, that this random replacement algorithm does exist such as in PDP-11/70. With this replacement algorithm, the probability that a shared block is selected to replace is equal to the percentage of shared blocks in the cache at the time when the miss occurs. If the selected block to replace is private, it is modified and needs to be written back with probability md. Hence, the probability that it is not modified and no action is needed is $1-md$.

Consider a particular processor in our multiprocessor system. It can be in one of two states: busy or idle. A processor is said to be busy when it is doing some internal computation and the processor is idle after it issues a memory request until the request is satisfied. If the memory request results in a cache hit, the processor resumes busy again immediately after the cache operation is complete. Additional delay may be incurred if one of the followings is true. 1) The request results in a cache miss so that a bus transaction is needed; 2) The request is a write that modifies a shared and clean cache block. In this case, the processor can not perform the write immediately until other copies of the memory block are invalidated; 3) The cache controller is busy serving requests from snooping controllers that observed bus transactions which require the cache controller to invalidate certain cache block or to supply a cache block on to the buses. The proportion of the time that the processor remains busy is defined to be the processor utilization. Since the system throughput is directly related to this processor utilization, we shall use the processor utilization and processing power (the product of processor utilization and number of processors) as performance metrics in our analysis.

3.2. The Model

The cache based multiple-bus multiprocessor described above is modeled by a mixed queueing network that consists of both closed and open customer classes [7], as shown in Figure 3. The processors are modeled by delay servers labeled P_i's. The memory modules, $M_1\ M_2\ \cdots\ M_M$, are represented by FCFS service centers with service rate $1/T$. The N private caches C_i's are also modeled by FCFS service centers with service rate of 1. The possible "customers" in a private cache queue are the requests from its local processor and the requests from other processors for invalidation and supplying data, etc.. The bus system is represented by a flow equivalence service center [7] as shown in the figure. In case of centralized control, the bus system queue is simply a load dependent service center. There are a total of N distinct customers in the network that belong to closed class, each of which is associated with a specific processor. The routing of each of these closed class customers can be described as follows.

Initially a customer stays in the delay center associated with it for a random amount of time to represent the thinking period of the processor. The processor then requests a cache service (read or write) and the "customer" is threaded to the cache queue. If the request can be satisfied within the private cache, the "customer" goes back to the delay center and the processor stays busy again. If the cache request results in a cache miss or a cache hit but requiring invalidation signal to be sent, then a bus transaction is necessary. In this case, the request will join the bus system queue in which a free bus is to be obtained. Once the processor gets a bus, the request packet will be transmitted to different places in the system depending on the nature of the request. According to the write-once protocol, the requested memory block can be acquired either from one of the memory modules or from one of the remaining caches. The block is supplied by a memory module only when none of the remote caches has a dirty copy of the requested block. In this case, the request packet joins one of the memory queues (equally likely). Once the request packet reaches the head of the queue, the required memory operation takes place. After the operation is finished, the memory controller formats a response packet that contains both the requested data and the identification of the requesting processor. The response packet is then returned to the requesting cache through one of the system buses and the particular data item requested by the processor goes directly to the processor. Similarly, if the missed block is to be supplied by a remote cache, instead of joining a memory queue the request packet joins the corresponding cache queue from where the cache controller supplies the data through the system buses. It is clear that the exact routing behavior of a "customer" depends on a set of routing probabilities. For instance, upon emerging from a cache queue, the customer may go back to its processor with probability R_{cp} and go to bus queue with probability R_{cb}. Determination of these probabilities requires a careful analysis of stochastic sharing behavior of the system. This is the task of next subsection.

Readers may have already noticed that in our cache coherence protocol there are cases that a memory request may spawn into a number of requests that go to different places in the network. For example, when a processor writes into a valid cache block, both write-through into main memory and invalidation signals to other caches are to be sent. Also when a block being fetched upon a miss arrives at a processor, the requested word in the block goes to the requesting processor directly and the block needs to join the cache queue to be written into cache memory. As another example, if a cache observes a DR or SR request on a bus and it has a written-once copy of the requested block, then appropriate state changes are necessary. In all these cases, more than one "customers" are generated by one customer. This phenomenon is called customer forking in the queueing network and is very difficult to analyze, if not impossible [7]. However, a careful examination of the cache protocol shows that the only effect of an invalidation signal on the system performance is its contribution to the load of the cache queue in which the invalidation is to be done. And similarly the effects of write-back and write-through into the main memory are contributions of the traffic load on the buses and memories. Therefore, to capture these

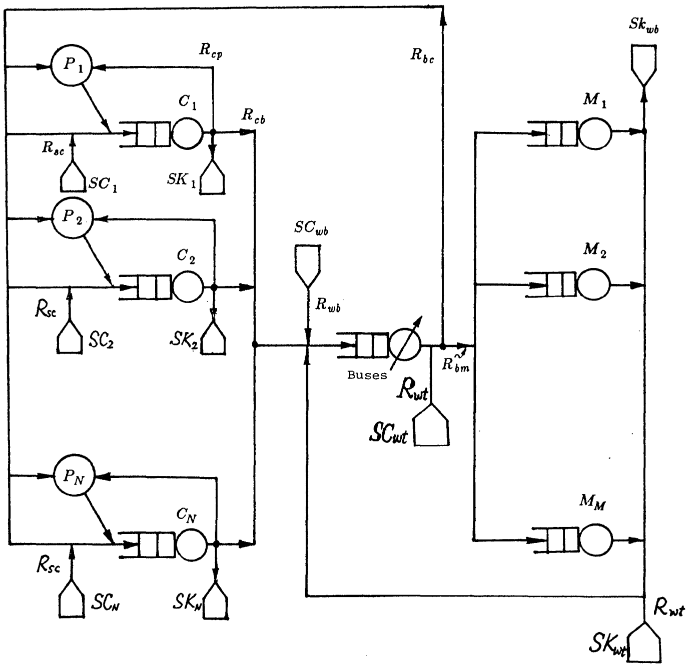


Figure 3. The queueing network model for the cache based multiple-bus multiprocessor system.

effects we shall use open class customers by adding sources and sinks as shown in Figure 3. The sources SC_i's and sinks SK_i's are used to represent the effect of invalidations, block loading and state changing on the caches and SC_{wb}-SK_{wb} and SC_{wt}-SK_{wt} are used to represent the effect of write-back and write-through on the buses and memories, respectively. The arrival and departure rates of these sources and sinks will be discussed shortly.

3.3. Routing Probabilities

Recall that a memory block can be in one of five different states as seen by a processor: not-present(0), valid-and-clean(1), once-written(2), dirty(3) and invalid(4). The Markov state diagram that represents these states is shown in Figure 2. Let π_i denote the steady state probability of a memory block being in state i. Note that the state of a memory block can be changed by a request either from the processor or from a system bus. Let p_i be the probability that the particular memory block i is addressed by a request generated by the local processor. The state transition rate of a shared block i, from state 0 to state 1 is $q_s p_i f_r P_u P_a / Z$ and to state 3 is $q_s p_i f_w P_u P_a / Z$ due to the local processor read and write respectively, where P_a is the probability that a processor can get a bus at a cache cycle. Bus requests, on the other hand, are mainly the effects from remote processors. For example, if the local cache observes an SR request on a bus and it has a dirty copy of the requested block, this SR request brings the block from the original dirty state to valid-and-clean state(i.e. from 3 to 1). This is because the local cache supplies the block to the requesting cache, writes it back to the main memory and changes the block state to valid-and-clean. As another example, the state of a block changes from valid-and-clean(1) to invalid(4) upon receiving a WI for that block on a bus. Let B_{sri}, B_{dri}, and B_{wii} represent the probabilities that there is a bus request on a bus for block i that is of SR, DR, and WI respectively. Then we get a set of transition rates of the Markov process as marked in Figure 2.

It should be pointed out that our Markov state diagram represents discrete Markov process based on cache cycles. However, the state transition between state 0 and other states may take more than one cycle due to the bus contention. This effect is taken care of by incorporating P_a's into the transition rate. Since our purpose at this moment is to derive the steady state probabilities of a memory block being in each state, we will momentarily set P_a to be 1. The bus contention will be modeled at higher level queueing model. Obviously, the Markov process shown in Figure 2. is aperiodic and irreducible. Solving the balance equations yields the following expressions for π_i's.

$$\pi_0 = \frac{(1-h)}{q_s\, p_i\, S_c + 1-h},$$

$$\pi_1 = \frac{\delta \pi_0}{\theta(1-p_{2,2}) - \theta(1-p_{2,2})p_{1,1} - \alpha p_{1,2} - \gamma p_{1,4}},$$

$$\pi_2 = \frac{p_{1,2}}{1-p_{2,2}}\pi_1,$$

$$\pi_3 = \frac{p_{0,3}(1-p_{2,2})(1-p_{4,4})\pi_0 + [p_{0,3}p_{1,4}(1-p_{2,2}) + p_{0,3}p_{1,2}(1+p_{2,4}-p_{4,4})]\pi_1}{\theta(1-p_{2,2})}$$

$$\pi_4 = \frac{p_{0,3}p_{2,4}(1-p_{2,2})\pi_0 + [p_{1,4}(1-p_{2,2})(1-p_{3,3}) + p_{1,2}p_{2,4}(1+p_{0,3}-p_{3,3})\pi_1}{\theta(1-p_{2,2})},$$

where $p_{i,j}$ is the transition rate from state i to state j as shown in Figure 2. And θ, δ, α and γ are given by

$$\theta = (1-p_{3,3})(1-p_{4,4}) - p_{0,3}p_{2,4},$$

$$\delta = (1-p_{2,2})[\theta p_{0,1} + p_{0,3}p_{2,1}(1-p_{4,4}) + p_{0,1}p_{0,3}p_{2,4}],$$

$$\alpha = \theta p_{1,2} + p_{0,3}p_{2,1}(1+p_{2,4}-p_{4,4}) + p_{0,1}p_{2,4}(1+p_{0,3}-p_{3,3}),$$

$$\gamma = p_{0,3}p_{2,1}(1-p_{2,2}) + p_{0,1}(1-p_{2,2})(1-p_{3,3}).$$

The quantities of B_{dri}, B_{sri}, and B_{wii} are given by

$$B_{dri} = q_s \frac{1}{BZN_{sb}} NP_u\, f_w\, \pi_0,$$

$$B_{sri} = q_s \frac{1}{BZN_{sb}} NP_u\, f_r\, \pi_0,$$

$$B_{wii} = q_s \frac{1}{BZN_{sb}} NP_u\, \pi_1.$$

Having obtained the values of π_i's, we are now ready to derive the routing probabilities of queueing network of Figure 3. Let us consider R_{cp}, the probability that a customer will go back to the processor after it comes out from the cache queue. R_{cp} is also the proportion of cache requests that can be successfully served by the local cache. Clearly, it is given by

$$R_{cp} = (1-q_s)h + q_s\, f_r\,(\pi_1+\pi_2+\pi_3) + q_s\, f_w\,(\pi_2+\pi_3).$$

The proportion of cache requests that need to access bus system, R_{cb}, is simply $1-R_{cp}$. A memory request on a bus can be served by either a cache or a memory module as described previously. Hence, a customer coming out of bus system may get into service center representing memory modules or go to one of remaining cache queues. The probability of going to a cache queue, R_{bc}, is given by

$$R_{bc} = q_s \frac{B_{dri}+B_{sri}}{B_{dri}+B_{sri}+B_{wii}}[1-(1-\pi_3)^{N-1}] + q_s \frac{B_{wii}}{B_{dri}+B_{sri}+B_{wii}},$$

and the probability of going to one of memory modules is given by $R_{bm} = 1 - R_{bc}$. It is assumed in our analysis, that once a request goes to the memory queues it will join one of the M memory queues equally likely. Similarly if the request goes to cache queues, it will enter any one of N-1 cache queues with same probability.

The behavior of the open customers is determined by the arrival and departure rates. In order for the system to be stable, the arrival and departure rates for each open class must be equal. Let R_{sc}, R_{wb} and R_{wt} be the arrival rates to the system from source SC_i's, SC_{wb} and SC_{wt}, respectively. They are given by

$$R_{sc} = \pi_1 B_{wii} + (1-h)(1-q_s)\frac{P_u}{Z} + \pi_0 q_s \frac{P_u}{Z} + B_{dri}(\pi_1+\pi_2) + B_{sri}\,\pi_2,$$

$$R_{wb} = \frac{P_u}{Z} N \left[\frac{S_c - N_{sb}(1-\pi_0)}{S_c}(1-h)md + \frac{N_{sb}(1-\pi_0)}{S_c}(1-h)\pi_3\right],$$

and

$$R_{wt} = N q_s\, f_w\, \pi_1 \frac{P_u}{Z}.$$

3.4. Solution of the Model

In the queueing network model defined above, each of N closed class customers has its own routing behavior which differs from others. For example, a customer originating from delay server P_i, will never visit the delay server labeled P_j for $i \neq j$. Thus, it is necessary to use multiple class solution technique [7] to solve the model. Moreover, the bus system queue in the model has load dependent service rate. Therefore, we end up with a mixed, multiple class queueing network containing a load dependent service center. The exact solution of such a model can be obtained by applying exact multiple class MVA algorithm[7] provided that all the service times of FCFS servers are exponentially distributed. In our case where the memory access time, bus transfer delay and cache cycles are constant values, the heuristic algorithm proposed by Reiser [11] can be used. However, the algorithms described above requires large amount of CPU time and memory space. In fact the time and space complexities of the algorithm are of $O(2^N)$ which is an exponential function of the number of processors. In order for our model to be practically useful, we shall develop a less complex algorithm that can be used to estimate the system performance for a variety of system parameters.

We begin with defining an aggregated queue that consists of N-1 processors and cache queues as shown in Figure 4. This queueing network is obtained by shorting out one processor-cache system, bus, and memory system queues. The queueing network of Figure 4 will be solved in isolation for each feasible population. The solutions of this model will then be used to solve the high level queueing model shown in Figure 5. In this high level queueing model, the entire queueing network of Figure 4 is considered as one single flow equivalence center. The processor-cache system (P_1 and C_1) in Figure 5 is then used to capture the detail behavior of the processor and cache operations while the rest of N-1 processors and caches are considered as a flow equivalence queue that captures only their aggregated effects on the entire queueing network.

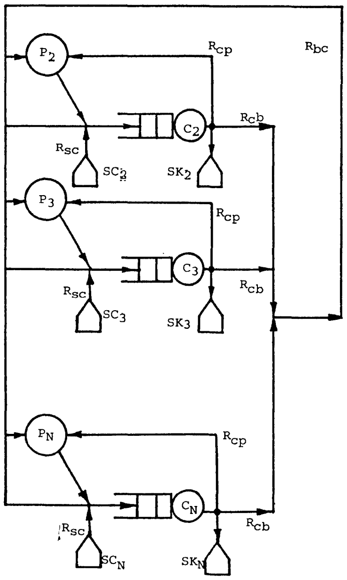


Figure 4. The queueing network representing N-1 processor-cache system.

In solving the low level queueing model of Figure 4, we still use the heuristic multiple class MVA [11] but the simplified approximate one [7] because the model is much simpler than the original model. The solution outputs of this model that are relevant to the high level model are the service rates of class 1 customer from P_1 and the summation of the throughput of the remaining N-1 classes of customers for each possible placement of customers in the queueing network of Figure 4. Once we obtained these service rates, the high level queueing model of Figure 5 is solved with 2 classes. Class 1 consists of 1 customer from P_1 and class 2 consists of N-1 customers in the flow equivalence center. As a result, the time and space complexity can be reduced to approximately $O((1+N)^2)$ instead of $O(2^N)$ provided that the low level solutions are available. The complexity of solving low level model of Figure 4 is hard to quantify due to the iterative nature of the simplified algorithm. However, it is known that the number of operations per iteration is $O((N-1)n)$ for the population of n and empirically less than two dozen iterations are typically required for convergence to less than a 0.1% change in queue length.

In summary, the solution of the queueing network involves the following steps: 1) Obtain the routing probabilities of closed class customers, arrival rates and departure rates of open class customer by setting the processor utilization P_u to 1; 2) Solve the queueing network model of Figure 4 for each feasible population to obtain two set of load dependent service rates; 3) use the results obtained in step 2 to solve the high level queueing model of Figure 5 to obtain the performance metrics including processor utilization P_u; 4) Derive a new set of arrival and departure rates for open class customers by using the new value of P_u and repeat this procedure from step 2 until the required accuracy is reached.

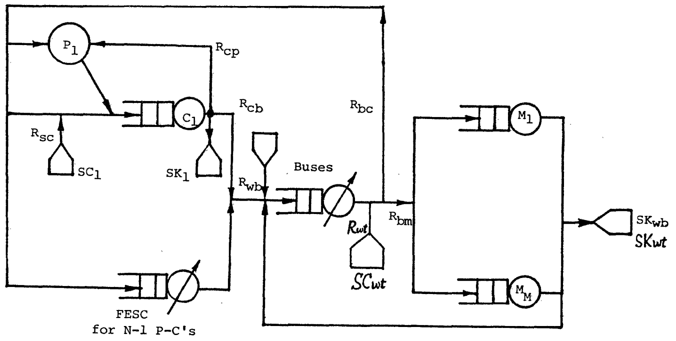


Figure 5. The high level queueing network representing the system.

4. Results and Discussions

Solving the queueing network model defined in the previous sections, we can obtain a set of system performance metrics such as the mean queue length of each service center, the response time of a memory access, processor utilization and processing power. The processing power is the sum of the processor utilizations in the system which takes into account both the number of processing elements and the queueing effects of cache protocol on the multiprocessor system. Hence, we shall use processing power as the performance metrics in our following discussions. The goal of our experiments here is to study the effects of the number of buses and degree of sharing on the performance of the cache based multiple-bus multiprocessor systems for a given set of architectural and work load parameters.

The architectural parameters used in this section are chosen close to the statistic values that appeared in the literature. The size of each private cache is assumed to be 2K blocks and the cycle time of a cache is the same as that of a processor. There are in total 32 shared memory blocks (N_{sb}) in the system. The memory operation takes 4 processor cycles. The transfer time of a packet on a bus is assumed to be 1 cycle. This assumption is reasonable because most of the existing system buses have data bus width of 64 or 128 lines [9]. As a result, a block of 4 or more words can be transferred in one bus transaction. The fraction of read (f_r) is 70% and write (f_w) 30%. The probability of a private block being modified, *md*, is 20%.

Figure 6 shows the processing power as a function of mean processor thinking time for 16 processors and up to 4 buses. The thinking time is also the interrequest time that indicates the offered load of processors to the rest of the system. The memory request rate which is the reciprocal of the thinking time represents the behavior of programs that are run on the multiprocessor system. Obviously, CPU-bound jobs have low values of request rates that generate less memory requests and consequently requires less communication overhead. As a result of this, the performance difference between the systems with different number of buses is not significant at very low value of request rate. However, as the request rate increases, the performance gain of more buses becomes significant. In particular, the processing power is almost doubled by adding one more bus to the single shared bus multiprocessor for the request rate more than 0.4.

Figure 7 shows the system processing power as a function of number of processors and for same numbers of buses. For a small number of processors, the performance difference due to different number of buses is not significant, which indicates that a single bus does not create a severe system bottleneck. This result is consistent with that observed by Archibald and Baer [1] in their simulation studies. However, as the number of processors increases, the difference between the curves becomes large. In other words, the single bus gets saturated very quickly and degrades the system performance. For a large system size, the difference in behavior for different number of buses will be more pronounced.

The effects of the degree of sharing on the system performance are shown in Figure 8, where three curves are illustrated for different values of q_s, the probability that a given request is for a shared memory block. As expected, larger q_s increases the bus traffic in order to enforce the coherence protocol. Moreover, each cache controller dedicates more time to update remote requests from the system buses. As a result, a lower processor utilization is observed.

5. Conclusions

As one type of high bandwidth interconnection network for multiprocessors, multiple-bus structure has drawn a considerable interest among computer architecture community. A great deal of work has been done in design and analysis of such structures. However, the previous analyses of multiple-bus structures did not consider the effect of cache coherence protocols. In this paper, we consider the packet-switched multiple-bus multiprocessors that use a modified write-once protocol to enforce cache coherence. A queueing network model that consists of mixed multiple class customers has been developed. The model captures the effects of both the multiple-bus contention and the cache coherence protocol on

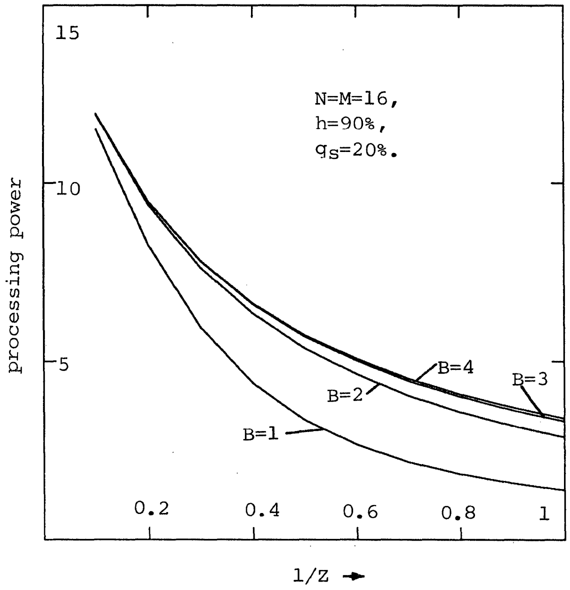


Figure 6. Processing power as a function of request rate.

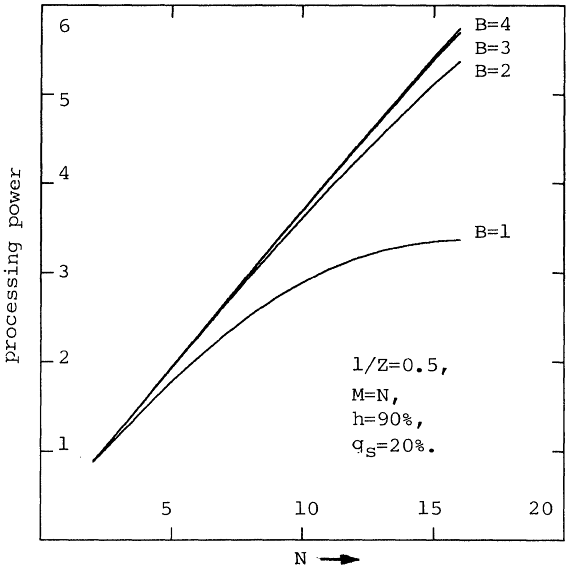


Figure 7. Processing power as a function of the number of processors.

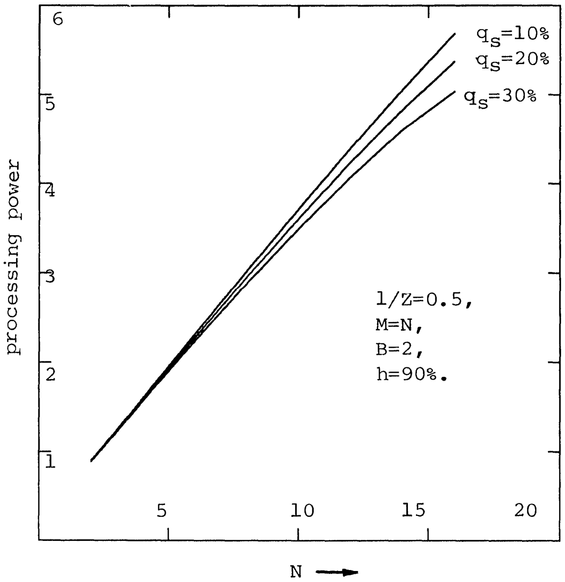


Figure 8. The effect of degree of sharing on the system performance.

the system performance. To reduce the complexity of solution of the model, a simplified algorithm based on flow equivalence technique has been developed.

From the numerical results obtained from the model, we conclude that both the number of buses and the degree of sharing have significant effects on system performance. A high bandwidth network such as multiple-bus is necessary for a large system because the single bus gets saturated very rapidly and creates a system bottleneck. Although the snooping system and cache controller designs become complex with the increase in the number of buses, we believe that the design of a 2 to 4-bus system is quite feasible.

References

[1] J. Archibald and J-L. Baer, "Cache coherence protocols: Evaluation using multiprocessor simulation model," *ACM Tran. on Comput. Systems,* Vol. 4 No.4, pp. 273-298, Nov. 1986.

[2] L. N. Bhuyan, "Interconnection networks for parallel and distributed processing," Guest Editor's Introduction, *IEEE Computer,* pp. 9-12, June 1987.

[3] C. R. Das and L. N. Bhuyan, "Bandwidth availability of multiple-bus multiprocessors," *IEEE Trans. on Comput.,* Vol. C-34, pp. 918-926, Oct. 1985.

[4] M. Dubois and F. Briggs, "Effect of cache coherency in multiprocessors," *IEEE Tran. on Comput.,* C-31, No.11, pp. 1083-1099, Nov. 1982.

[5] M. Dubois, "Throughput analysis of cache-based multiprocessor with multiple buses," *IEEE Tran. on Comput.,* Vol. 37, pp. 58-70, Jan. 1988.

[6] Goodman, J. R., "Using cache memory to reduce processor-memory traffic", *10th Annu. Symp. on Comput. Arch.,* pp. 124-132, 1983.

[7] E.D. Lazowska, et al., *Quantitative System Performance---Computer System Analysis Using Queueing Network Models,* Prentice-Hall, Inc., Englewood Cliffs, 1984.

[8] T. N. Mudge, J. P. Hayes and D. C. Winsor, "Multiple-bus architectures," *IEEE Computer,* Special Issue on Interconnection Networks, June 1987.

[9] *Multimax Technical Summary.* Encore Computer Corporation, Marlboro, Massachusetts 01752, rev a edition, May 1985.

[10] J. H. Patel, "Analysis of multiprocessors with private cache memories," *IEEE Tran. on Comput.,* Vol. C-31, Apr. 1982, pp. 296-304.

[11] M. Reiser, " A queueing network analysis of computer communication networks with window flow control," *IEEE Tran. on Commun.* Vol. COM-27, pp. 1199-1209, Aug. 1979.

[12] C. Scheurich and M. Dubois, "Correct memory operation of cache-based multiprocessors", in *The 14th Ann. Int'l Symp. on Comput. Arch.,* pp. 234-243, June 1987.

[13] Tang, C. K., "Cache System Design in Tightly Couple Multiprocessor System", *AFIPS Proc., Natl. Comput. Conf.,* Vol. 45, pp. 749-753,1976.

[14] M. K. Vernon and M. A. Holliday, " Performance analysis of multiprocessor cache coherency protocols using generalized Timed Petri Nets," *Proc. ACM SIGMETRICS Conf.,* pp. 9-17, 1986.

[15] D.C. Winsor and T.N. Mudge, "Crosspoint cache architecture," *Proc. 87'Intl. Conf. on Parallel Processing,* pp. 266-269, 1987.

[16] Q. Yang and S. G. Zaky, "Communication performance in multiple-bus systems," To appear in *IEEE Trans. on Comput..*

[17] Q. Yang, L. N. Bhuyan, and R. Pavaskar, "Performance analysis of packet-switched multiple-bus multiprocessor systems," *The Eighth Real-Time Systems Symposium* pp. 170-178, Dec. 1987.

[18] Q. Yang and L.N. Bhuyan, "Analysis of a cache coherence protocol for synchronous packet-switched multiple-bus multiprocessors", Technical Report: *TR 88-3-1, CACS, USL.*

Stale Data Detection and Coherence Enforcement Using Flow Analysis

Hoichi Cheong *and* Alexander V. Veidenbaum

Center for Supercomputing Research and Development
University of Illinois at Urbana–Champaign
Urbana, Illinois 61801

Abstract

Software–assisted cache coherence enforcement schemes for large multiprocessor systems with a shared global memory and interconnection network have gained increasing attention. Such schemes rely on software to decide which memory references access potentially stale cache copies of variables. The algorithms used usually overestimate the number of such references. Few have pursued techniques to more accurately identify accesses to stale cache copies. In this paper, we propose an approach based on flow analysis to detect such accesses. Software–based cache coherence schemes that can utilize the detected results are discussed. We then show our recently proposed approach which has less unnecessary invalidation and is faster than other proposed coherence schemes.

1. Introduction

Properly managed private cache memory in multiprocessor systems with shared global memory and interconnection networks can decrease memory access time and reduce network congestion and contention in the shared memory. However, cache coherence has to be enforced before private caches can be effectively used in such systems. The discussion will be restricted to the problems of maintaining cache coherence in large–scale multiprocessor systems with interconnection networks and shared global memory (or simply large–scale multiprocessor systems).

A cache coherence violation occurs when a cache copy of a processor does not have the up–to–date value of the variable. Such a copy is called a *stale* copy and an access to the copy a *stale access.*

Most proposed cache coherence schemes rely entirely on run–time mechanisms to maintain cache coherence. All schemes in this category use some form of shared media, a bus or directories, such that a processor can either monitor or be notified of each modification of a variable by another processor [Tang76, CeFe78, Good83, ArBa84, McCr84, PaPa84, RuSe84, KaEW85]. Cache copies that become stale due to a modification by other processors can then be updated or invalidated, and cache coherence is maintained. However, these schemes are not suitable for large–scale multiprocessor systems since bus–based systems support only a limited number of processors. On the other hand, directory approaches are expensive to construct and cause global memory accesses to be serialized to different degrees, or they incur high communication cost.

Software–assisted approaches have been proposed and present an alternative for cache coherence in large–scale multiprocessor systems. In these approaches, potentially stale cache copies of variables are purged at specific locations in program execution. Copies known to be up–to–date, which are usually kept in the shared global memory, are then accessed [Smit85, BrMW85, EGKM85, McAu86, Veid86, LeYL87, Lee87, ChVe87]. High communication cost associated with run–time detection of stale cache copies is avoided.

Efficient invalidation of stale copies is crucial to software assisted schemes. However, existing schemes also invalidate non–stale cache copies and may result in low hit ratios. This is the price paid for not relying on run–time detection of stale cache copies. To limit the number of invalidations of non–stale cache copies, alternatives to run–time detection of stale accesses are important but have not been fully addressed in software–assisted schemes.

The objectives of this work are (1) to develop a compile–time flow analysis algorithm to detect stale accesses in parallel programs, and (2) to discuss possible enforcement schemes based on the detection results.

In the following sections, the formulation of the cache coherence problem in terms of flow analysis is presented first. Then, an algorithm for stale access detection is introduced. Possible cache coherence schemes based on the detection result are discussed. Finally, a brief discussion of the scheme that we recently proposed [ChVe88] is presented.

2. The Cache Coherence Problem and A Flow Analysis Model

Efficient cache coherence maintenance depends on the stale access detection. In this section, the issues associated with compile–time detection of stale accesses and the proposed solutions are covered.

2.1. Assumptions about Parallel Execution

For clarity, our discussion will focus on parallel programs with *Doall*–type [LuBa80] parallel loops (loops with no dependences across iterations). Barrier synchronization is assumed at parallel loop boundaries. Also, it is assumed that synchronization operations are necessary to preserve the correctness of

This work is supported in part by National Science Foundation under Grant No. US NSF DCR84-10110, the U.S. Department of Energy under Grant No. US DOE DE–FG02–85ER25001, and IBM Corporation.

the program; otherwise, a variable cannot be written and read by distinct processors. All operations and memory accesses of processors need to be completed before they can proceed across a synchronization point. Synchronization variables are not cacheable. It is also assumed that processor assignment to a parallel loop is unknown at compile time. The following discussion focuses on intra-procedural analysis and an empty cache is assumed at the beginning of a procedure.

2.2. Conditions for Stale Accesses Detection

Following the ideas in [Veid86], the occurrence of a stale access to a variable X_k by a processor P_i is determined by the following sequence of events in execution: (e_1) a cache copy of X_k is loaded in the cache of the P_i, (synchronization operations take place), (e_2) the latest write to X_k is executed by a processor P_j, $j \neq i$, (synchronization operations take place), and (e_3) processor P_i reads X_k. The synchronization operations in the sequence are implied under the earlier assumption. Since our discussion is focused on parallel programs with *Doall* loops, synchronization is done where a processor assignment (PA) to loop iterations takes place. However, the approach is general enough to deal with other cases.

Notice that the sequence of e_1 and e_2 constitutes the sufficient condition for the existence of a stale copy of X_k in P_i. A stale copy, unless being accessed, does not cause incorrect computation. However, since events e_1 through e_2 in a sense determine whether processor P_i in e_3 will access a stale cache copy, let us call e_1 through e_2 collectively, *determining sequence* (DS) of the stale access to variable X_k by P_i.

Compile-time detection of stale accesses faces a major obstacle. Namely, the details of processor assignment are unknown at compile-time. For example, it may not be known at compile-time whether processors in e_2 and e_3 are the same. If they are, the access in e_3 will not be stale. The identities of processors involved in the sequence are important. Without this knowledge, a compile-time detection scheme can not accurately predict the occurrence of the DS.

The next best thing to detect is a set of sequences of events which includes the DS. Such a set of sequences serves as a necessary condition under which *some* processor may access a stale copy of a variable X_k. The set is represented by the following *relaxed determining sequence* (RDS) of variable X_k: (E_1) a cache copy of X_k is loaded into the cache of some processor, (E_2) one or more PA's occurs, (E_3) a write to X_k is executed, (E_4) one or more PA's occurs, and the event pair E_3 and E_4 may be repeated.

The DS of a stale access to a variable by a particular processor is a specific case of the RDS of the stale access to the variable. E_1 establishes the existence of an initial cache copy of a variable. Each occurrence of E_3 represents a new value assigned to the variable. The PA separating the writes imply that each such write may be executed by a different processor and can turn existing cache copies stale. The terminating PA of the RDS is necessary to determine that the lastest write in the RDS and the subsequent read after the RDS may be executed by different processors. Hence, a read following these events may use any one of the cache copies which are all stale except the one written last.

When an RDS of X_k precedes a read from X_k with no write in between, the read from X_k is considered a stale access. In some cases, an RDS of X_k, a write to X_k, and a read from X_k occur in a sequence. If the execution of the read implies the execution of the preceding write to X_k, the read from X_k is not stale. This will be discussed in details in later sections.

Notice that an RDS contains no read accesses except possibly in E_1 since a read access does not modify the existing value of a cache item and hence does not cause staleness. To simplify detection, one needs to look only for write events. The following sequence, however, will not be detected: a read, PA, a write, PA, ... This is the case when a read is not preceded by any write to the same variable in a subroutine. For such a read, a dummy write to the variable is assumed in the analysis. The dummy write accounts for the initial loading of the cache copy due to the read.

2.3. Flow Analysis Formulation and the Flow Graph

To detect an RDS at compile time, let us look for an execution sequence of at least two ordered pairs of write – PA events at compile time. Flow analysis is a good tool for such a task.

Let us now formulate the detection in the flow analysis terms [AhSU86]. A write access to a variable is a *definition (def)* and a read access is a *use*. A *def* that is followed by PA's is called a *determining def* (DD). Given a set of DD's of a variable, each DD represents a potentially distinct cache copy. Thus, the detection of the RDS of a variable is nothing but finding consecutive occurrences of DD's (By consecutive occurrences of DD's, the sequence of write – PA, write – PA, ... , is implied in order to be distinguished from the sequence of write, write, ... , PA. The latter does not form an RDS).

The *reaching def* algorithm [AhSU86] can determine whether a value *defined* earlier in program execution can be used by a read reference. It will be applied to determine whether distinct cache copies, represented by DD's, can be *used* by a read access.

To find *reaching defs*, the concepts of *kill*, *generate*, and *reach* are used. A *def* of variable X is *generated* by a statement S_j when the statement contains a write reference to X. A *def* of X, d_j, in S_j is *killed* by another *def* of X, d_i, in S_i if there is a path in the flow graph from S_j to S_i. A *def* of X, d_j, in S_j is said to *reach* S_k if there is path from S_j to S_i and no other *def* of X *kills* d_j on the path.

The goal of using the *reaching def* algorithm is to check if DD's can reach a *use* without being *killed* so that the *use* may access the values associated with such DD's. A *def*, d_i, *kills* another *def*, d_j, with respect to a subsequent *use* when the *use* of the variable accesses *only* the value written by d_i.

The following two facts are important for detecting an RDS and ultimately stale accesses. First, for cache management purposes, it is assumed that a new *def* of an array variable *will* result in generating new values, thus potentially creating new cache copies. Since it is not known for sure if a new assignment *generates* the same array elements as the previous one, let us assume that array *defs* are not killed by subsequent *defs*. Secondly, consider the flow graph. Let d_i be the a *def* of a scalar X between two adjacent PA's, PA_j and PA_{j+1}. Let the execution of *uses* between d_i and PA_{j+1} always be preceded by d_i. Then, these *uses* access only the value written by d_i. However, the *uses* after PA_{j+1} may access the value written by d_i or values by *defs* prior to PA_j. Thus, the "scope" of the *kill* by d_i is limited only to *uses* between d_i and

PA_{j+1}. For ***uses*** subsequent to PA_{j+1}, ***defs*** prior to PA_j are not killed by d_i. The ***defs*** prior to PA_j can thus ***reach*** the ***uses*** subsequent to PA_{j+1}. This is called *reaching* due to processor assignment.

2.4. Modified Flow Graph

A flow graph that models parallel execution is created by adding a ***cobegin*** and a ***coend*** node at the beginning and the end of each parallel loop. These nodes are the places of processor assignments (PA's). The nodes of a flow graph are basic blocks [AhSU86]. Each of the ***cobegin*** and ***coend*** nodes is considered as a special type of basic block, namely, a *processor assignment block* (PAB).

The ***reaching def*** algorithm applied to conventional flow graphs cannot properly handle ***reaching*** due to processor assignment. The flow graph is modified to correct this. An edge serving as a by-pass for ***defs*** prior to a PA_j to ***reach*** the ***uses*** after PA_{j+1} is added to adjacent pairs of ***PA's***. Since scalars are not written in parallel loops, edges are only added from the ***coend*** node of an outermost parallel loop to the ***cobegin*** node of the outermost parallel loop(s) next in the flow direction. These are called augmenting edges. The examples of a flow graph and its modified version are given as G_0 and G_1, respectively, in Figure 1.

2.5. The Detection Algorithm

Detection of stale accesses is performed as follows. First, a flow analysis algorithm is applied to G_1 to compute the DD's *reaching* each block. Secondly, the existence of an RDS for each read reference in the block is determined from the DD's reaching a block. Finally, each read reference in a basic block is checked to determine if it indeed accesses a potentially stale cache copy.

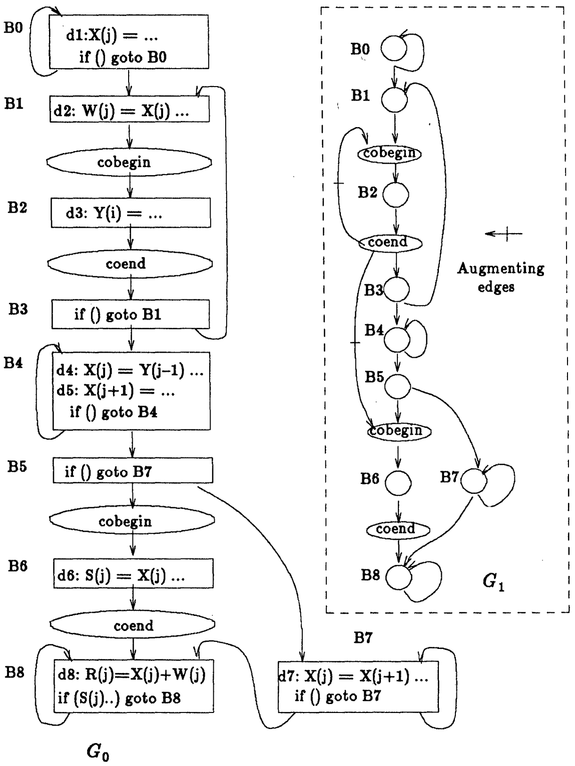


Figure 1. G_0 and G_1 of an example.

First, a ***gen*** set and a ***kill*** set are computed for each basic block, B. The ***gen*** set, ***gen***(B), contains the statements which have a ***def*** of a variable that is not ***killed*** in B. The ***kill*** set, ***kill***(B) contains the set of statements, not in B, with ***defs*** that will be ***killed*** by a ***def*** in B provided there is a path from those statements to B. Each block is also associated with the following sets: ***in*** (B) set, ***in'***(B), ***out***(B), and ***out'***(B). ***in*** (B) contains ***defs*** reaching B. ***in'***(B) contains the DD's reaching B, namely, ***defs*** which reach a ***cobegin*** or ***coend*** node before reaching B. ***Defs*** that ***reach*** the entry of B or are ***generated*** in B are included in ***out***(B) if they also ***reach*** the exit of B. If B is not a PAB, ***out'***(B) contains DD's that ***reach*** B but are not ***killed*** in B. If B is a PAB, all ***defs*** reaching B are turned into DD's and are included in ***out'***(B). The sets are defined by the following equations:

$$out\,(B) = gen\,(B) \cup (\,in\,(B) - kill\,(B))$$

$$out'(B) = \begin{cases} in'(B) - kill\,(B) & \text{if } B \textit{ is not a PAB} \\ in'(B) \bigcup in\,(B) & \text{if } B \textit{ is a PAB} \end{cases}$$

$$in\,(B) = \bigcup_{\text{for } all\ j} out\,(B_j),$$

$$in'(B) = \bigcup_{\text{for } all\ j} out'(B_j),$$

where B_j is an immediate predecessor block of B.

Next, an iterative algorithm is used to find ***in***(B) and ***in'***(B) for each block B in G_1. The algorithm converges when ***in***(B) and ***in'***(B) of every block both stay unchanged for two consecutive iterations. When the algorithm converges, ***in'***(B) contains all DD's reaching B.

For convenience in our discussion, let ***in'***(B) be partitioned into subsets each containing statements defining a specific variable. For a variable X, let ***in'***$_X(B)$ represent the subset in ***in'***(B) such that,

$$in'(B) = \bigcup_{\text{for } all\ X} in'_X(B),$$

where X is a variable that has a ***def*** reaching B. The results of the algorithm on the example program in Figure 1 are given in Table 1.

2.5.1. Finding an RDS from Determining Definitions

The next step is to find out if the ***in'***(B) to a block may result in an RDS. The conditions under which a pair of distinct DD's of the same variable reaching a block may result in an RDS are shown below:

Lemma 1

Two distinct ***determining defs*** of a variable d_i and d_j, reaching a block B_k, may result in an RDS to B_k if $d_i \in$ ***in'***(B_j), where $d_j \in B_j$.

Block	*gen*	*kill*	*in*	*in'*			
				in'_X	in'_W	in'_S	in'_Y
B_0	$\{d_1\}$	nil	$\{d_1\}$	nil	nil	nil	nil
B_1	$\{d_2\}$	nil	$\{d_1,d_2,d_3\}$	$\{d_1\}$	$\{d_2\}$	nil	$\{d_3\}$
B_2	$\{d_3\}$	nil	$\{d_1,d_2,d_3\}$	$\{d_1\}$	$\{d_2\}$	nil	$\{d_3\}$
B_3	nil	nil	$\{d_1,d_2,d_3\}$	$\{d_1\}$	$\{d_2\}$	nil	$\{d_3\}$
B_4	$\{d_4,d_5\}$	nil	$\{d_1,d_2,d_3,$ $d_4,d_5\}$	$\{d_1\}$	$\{d_2\}$	nil	$\{d_3\}$
B_5	nil	nil	$\{d_1,d_2,d_3,$ $d_4,d_5\}$	$\{d_1\}$	$\{d_2\}$	nil	$\{d_3\}$
B_6	$\{d_6\}$	nil	$\{d_1,d_2,d_3,$ $d_4,d_5\}$	$\{d_1,d_4,d_5\}$	$\{d_2\}$	nil	$\{d_3\}$
B_7	$\{d_7\}$	nil	$\{d_1,d_2,d_3,d_4,$ $d_5,d_7\}$	$\{d_1\}$	$\{d_2\}$	nil	$\{d_3\}$
B_8	$\{d_8\}$	nil	$\{d_1,d_2,d_3,d_4,$ $d_5,d_6,d_7,d_8\}$	$\{d_1,d_4,d_5\}$	$\{d_2\}$	$\{d_6\}$	$\{d_3\}$

Table 1. Analysis results for the example in Figure 1.

Proof

By the definition of $in'(B)$, if $d_i \in in'(B_j)$ and $d_j \in B_j$, a path between the block containing d_i and B_j must contain a *cobegin* or a *coend* node. Since d_j is a DD reaching B_k, there is a *cobegin* or *coend* node on the path between B_j and B_k. Then the following sequence is obtained: (E_1) d_i, (E_2) a PA, (E_3) d_j, and (E_4) a PA.

A single *determining def* in the source program may also result in an RDS to a block. This is the case when d_i is enclosed by a serial loop or a backward branch that also encloses a parallel loop. This case is covered by the following Lemma.

Lemma 2

A *determining def* d_i, reaching B_k, results in an RDS to B_k if $d_i \in in'(B_i)$ and $d_i \in B_i$.

Proof

By the definition of $in'(B_i)$, if $d_i \in in'(B_i)$ and $d_i \in B_i$, a backward path to B_i must contain a *cobegin* or a *coend* node. Since d_i is a DD reaching B_k, d_i must reach a *cobegin* or *coend* node before reaching B_k. Thus, to block B_k, there exists the RDS: (E_1) d_i, (E_2) a PA, (E_3) d_i, and (E_4) a PA.

The following theorem can be used to detect the existence of an RDS.

Theorem 1

The in'_X set of B_k results in an RDS to a read access to X in B_k if and only if the following is true: (1) there exists $d_i \in in'_X(B_k)$ such that $d_i \in in'_X(B_j)$ and $d_i \in B_j$, or (2) there exists $\{d_i, d_j\} \subseteq in'_X(B_k)$ such that $d_i \in in'_X(B_j)$, where $d_j \in B_j$.

Proof

The *if* part is straight forward by Lemma 1 and Lemma 2. The proof of the *only-if* part is as follows. Suppose that neither (1) or (2) is true. The case of an empty in'_X is trivial. If in'_X (B_k) is not empty and (1) and (2) are both false, the following must be true: blocks containing members of $in'_X(B_k)$ are not connected by paths containing *cobegin* or *coend* nodes since no member of $in'_X(B_k)$ appears in the in'_X sets of other blocks containing members of $in'_X(B_k)$. Then, there is no PA between the execution of the *def(s)*. As a result, there can be no RDS involving any *def* pair d_i and d_j or multiple d_i*'s*. It has just been shown that the $in'_X(B_k)$ cannot result in an RDS if (1) and (2) are both false. So if $in'_X(B_k)$ results in an RDS, either (1) or (2) has to be true. Q.E.D.

2.5.2. An Implementation of RDS Detection

The following is a possible implementation of RDS detection from the in'_X set. For each variable X, the DD's in the union of in'_X sets of all blocks and the RDS's formed by DD's are represented by an undirected graph $M_{IN'_X}(V,E)$, in which a node $v_j \in V$ represents a DD and an edge $e \in E$, between nodes v_i to v_j, denotes that the DD's represented by v_i and v_j together result in an RDS.

To determine, for block B_j of G_1, if $in'_X(B_j)$ of a variable X results in an RDS of X, $M_{IN'_X}(V,E)$ is examined. For a *def*, $d_i \in in'_X(B_j)$, if there is an edge in $M_{IN'_X}(V,E)$, between the node representing the d_i and a node representing another *def*, $d_k \in in'_X(B_j)$ or d_i, the $in'_X(B_j)$ set results in an RDS to B_j.

$M_{IN'_X}(V,E)$ can be represented by an adjacency matrix $M_{IN'_X}$. The $M_{IN'_X}(V,E)$ graphs and the $M_{IN'_X}$ matrices for the variables *used* in our example are shown in Figure 2 and Table 2. The $in'_X(B_j)$ sets for different blocks containing *uses* of variable X are shown as augmented columns to the matrix.

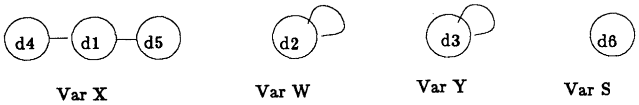


Figure 2. $M_{IN'}(V,E)$ of different variables.

	d_1	d_4	d_5	in'_X			
				B_1	B_6	B_7	B_8
d_1	0	1	1	1	1	1	1
d_4	1	0	0	0	1	0	1
d_5	1	0	0	0	1	0	1

	d_2	in'_W
		B_8
d_2	1	1

	d_3	in'_Y
		B_4
d_3	1	1

	d_6	in'_S
		B_8
d_6	0	1

Table 2. $M_{IN'}$ matrices for the example.

Take B_6 as an example. Its in'_X set contains d_1 d_4 and d_5 (in the augmented column with in'_X and B_6 as headings). From the adjacency matrix, since entries (row d_1, col d_5) and (d_1, d_4) are 1, either one of these DD pairs in $in'_X(B_6)$ forms an RDS.

2.6. Finding *Stale Uses* in a Basic Block

Now, the nature (stale or non–stale) of *uses* in a basic block can be determined. The *uses* of scalars and arrays are treated differently.

For *uses* of scalars in a basic block, *uses* are divided into *upwardly—exposed uses* and *non—upwardly—exposed uses*. An *upwardly—exposed use* is a *use* in a block which is preceded by no *def* of the same variable in the block. An *non—upwardly—exposed use* is preceded by at least one *def* of the same variable.

Theorem 2.1

An *upwardly—exposed use* of a scalar variable X in a block is considered a *stale access* if in'_X of the block results in an RDS.

Proof

If in'_X to the block results in an RDS in the system, the *def* in E_3 in the RDS has turned the cache copy, written by the *def* in E_1, into a stale copy prior to the execution of the block. It is possible that the processor executing the block has the stale copy of the variable in its cache.

Non—upwardly—exposed uses are not stale regardless of whether the *in'* set of the variable results in an RDS, since the preceding *def* in the same block always supplies the *uses* of the scalar with an up–to–date copy.

Recall that a *def* of an array does not kill an earlier *def* Thus, a *use* of an array element may not access only the values written by a preceding *def* in the same basic block. The *use* has to be assumed stale. Therefore, for array variables, no distinction between *upwardly—exposed* and *non—upwardly—exposed uses* is made. The following theorem is for detecting stale accesses to arrays and the proof is similar to the one for Theorem 2.1.

Theorem 2.2

A *use* of an array variable X in a block is considered a *stale access* if in'_X to the block results in an RDS.

In our example, the *uses* of X in B_1 and B_7 are non–stale accesses, and those in B_6 and B_8 are stale. The *use* of W in B_8 and the *use* in Y in B_4 are stale. The *use* of S in B_8 is non–stale.

2.6.1. Optimization Using Data–Dependence Analysis

The algorithm in the above discussion does not use subscript expressions when dealing with array variable accesses. As a result, an array *def* is included in $in'X(B)$ even if the the elements assigned by the *def* are not used in B. This happens when the subset of array elements *used* has no element in common with the subset *defined.* A larger in'_X set to a block is more likely to result in an RDS, hence the *uses* are more likely regarded as stale accesses. Also, when two DD's of an array write two disjoint subsets of array elements, these DD's should not result in an RDS even if the conditions of Theorem 1 are met. If such a case can be detected, fewer stale *uses* will occur. Another result is Theorem 2.2, in which no distinction between *upwardly–exposed* and *non–upwardly–exposed* uses of arrays is made, regardless of whether the *use* will access only up–to–date copies supplied by preceding *defs* in the same block. Using data–dependence analysis, a more accurate detection can be achieved.

Data dependence analysis can be used to refine the in'_X for a *use* of array X or to detect that two DD's in in'_X write no array element in common. A *reaching def* can be deleted from the in'_X for a *use* of X if the *use* does not depend on that *def* of X. A *use* with a smaller in'_X has a better chance of being a non–stale access. Since each *use* of array X may have a different subscript expression than the other *uses* in a basic block, the in'_X set has to be refined each time a *use* of X with a different subscript expression is investigated.

On can also use the analysis of subscript expressions to determine whether the *uses* are preceded by *defs* with the same subscript expression in the same block. Such *uses* will not be treated as stale accesses. This can be done by using direction vectors [Wolf82]. A *use* preceded by a *def*, with an all–equal direction vector in the same block, is not stale.

3. Possible Coherence Schemes by Selective Invalidation

Software–directed cache coherence schemes can benefit from the stale access detection. Possible approaches to invalidate stale cache copies and related issues are discussed. Then, a scheme that uses the detection results and a fast selective invalidation approach is shown.

3.1. Invalidation Schemes

There are several ways to make a reference access only the up–to–date copy of a variable. It can be achieved either by invalidating the stale cache copies, by loading the up–to–date copy by forcing a cache miss (also considered as a form of invalidation), or by updating all cache copies as soon as a modification is done. We believe that the last approach is not suitable for large–scale multiprocessor systems because it necessarily (1) has high communication cost, (2) updates cache copies which are not going to be used later, or (3) requires more run–time bookkeeping. The following discussion will concentrate on invalidation.

The goal is to achieve cache coherence without communication to other processors. It can be accomplished by letting each processor manage its own cache through invalidation. To simplify the discussion, let us assume that the global memory always contains the up–to–date copies of the variables after each synchronization point or cache line replacement.

Invalidation schemes in previous work suffer from a common disadvantage, namely, over–invalidation. They all invalidate at fixed program locations, such as where processor assignments (PA's) or synchronization operations occur. Some methods invalidate all shared variables at each such location. Others invalidate the entire cache (indiscriminate invalidation) in favor of fast invalidation time. Cache variables that are not stale are often invalidated [ChVe88].

Our discussion of invalidation schemes will focus on the following issues: (1) when to invalidate, (2) what to invalidate, and (3) how to invalidate. To address these issues, the invalidation approaches are divided into two classes, the *post-access* invalidation and the *pre-access* invalidation. In one approach, each processor invalidates the cache copies after its references in order to prevent future stale accesses. In the other approach, each processor invalidates before accessing what has been detected as stale. Both approaches depend on the knowledge that an access has been detected as stale. These approaches are fundamental for software-directed invalidation. The division equally applies to selective and indiscriminate invalidations. However, selective invalidation is of primary interest in this study. The indiscriminate approach has been thoroughly discussed in [Veid86, ChVe87, Lee87].

3.1.1. Post-Access Invalidation

Post-access invalidation is a preventive coherence scheme in the sense that cache copies are invalidated before they can become stale due to a write by another processor. Cache copies created by read or write accesses need to be invalidated if they may turn stale before being accessed by future *uses*. The condition under which such cache copies are detected is similar to that for detecting an RDS and is given in the following:

C1. *A cache copy created by any def or use that reaches a* PA, *a def, and a* PA *before reaching a use is considered a stale copy that may be accessed by the use.*

The *post-access* approach puts the responsibility to invalidate on the processor that creates such cache copies. Let us define the *post-access* approach as follows:

In the post-access invalidation, a processor invalidates the cache copies that it created if they satisfy condition C1.

In the *post-access* approach, invalidations can be *immediate* or *delayed.* In *immediate* invalidation, the cache copy is invalidated right after each reference. Otherwise, the invalidation is called *delayed.*

Immediate invalidation results in over-invalidation since the number of invalidations of a variable can be as large as the number of references. As a result, a cache item may be invalidated more than once even though the one after the last reference is sufficient. Temporal locality of multiple references of a variable between a pair of adjacent PA's is destroyed.

Using *delayed* invalidation, one can take advantage of such temporal locality and can invalidate less items. Between a pair of adjacent PA's, *delayed* invalidation can rely on standard optimizations to reduce or eliminate over-invalidation. In the case of scalar variables, what should be invalidated by *delayed* invalidation can be easily determined. Flow analysis can show which scalar variable is referenced on a path between a pair of adjacent PA's. Invalidation of the cache copy can be delayed until the processor is done with the references to the variable. *Delayed* invalidation of array variables has to depend on subscript analysis. However, in cases where subscript analysis is inexact, *immediate* invalidation, in which the exact subscript expression of an array reference is used to determine the elements to be invalidated, can be used (or invalidating indiscriminately before the subsequent PA).

While it is conceivable to delay invalidations beyond the PA immediately following the accesses, the growing complexity and inaccuracy of determining what to invalidate make such a delay impractical. Consider the case in which invalidations of the cache copies created before PA_i are postponed until after PA_i but before PA_{i+1}. At compile-time, it may be impossible, especially when the details of processor assignment are not known, to determine exactly which elements of an array the processor has referenced prior to PA_i. When invalidation is delayed after PA_i, a much larger number of the array elements have to be assumed present in the cache and they will have to be invalidated.

Although such delayed invalidation is more difficult to perform, it does have an advantage; it can preserve temporal locality in the following case. Consider a sequence of references of a variable in the following form: non-stale *use*, PA, ... , non-stale *use*, PA, *def*, PA, stale *use*. The cache copies of the variable do not become stale until the *def* is executed. If invalidation can be delayed until after the last non-stale *use* in the sequence, processors accessing these cache copies before the *def* can have cache hits. When the cache copies of the non-stale *uses* are invalidated before each PA, potential hits by the non-stale accesses separated by PA's are lost.

3.1.2. Pre-Access Invalidation

As a dual to the preventive approach of *post-access* invalidation, cache copies are allowed to turn stale and are left alone. Stale cache copies are invalidated only when they may be accessed. *Pre-access* invalidation is defined in the following:

In pre-access invalidation, a processor invalidates the cache copy before the detected stale access by the processor.

Similar to the *post-access* approach, an invalidation is called *immediate* if it is done right before each stale access. Otherwise, it is called *early.* Since there can be several paths leading to a *use*, *early* invalidation has to make sure that the potentially stale cache copy is invalidated no matter which path is taken.

As in the *post-access* approach, *immediate* invalidation has the disadvantage of over-invalidation. Neither does it exploit temporal locality between a pair of adjacent PA's since newly loaded cache copies due to preceding *uses* or *defs* are also invalidated. *Early* invalidation has the same advantages as *delayed* invalidation in the *post-access* approach. Between a pair of adjacent PA's, *early* invalidation tries to determine 1) the first *use*, or 2) the first *def* (only for arrays) that is always executed prior to the stale *use* of the same variable or array element. If 1) is found, invalidation is needed only before the first *use*. Invalidation is not needed if 2) is found. For scalar variable accesses, flow analysis can handle the above easily. For array variables, subscript analysis is needed. When subscript analysis does not help, the worst case solution is to use *immediate* invalidation, or indiscriminate invalidation after the PA immediately preceding a stale *use*.

Invalidating earlier than the PA immediately preceding a stale *use* is difficult. The difficulties are similar to the ones in postponing invalidation beyond the PA immediately following an access in the *post-access* approach. A processor in *pre-access early* invalidation thus, in practice, invalidates the stale copy no earlier than the PA immediately preceding a stale access. Therefore, potential temporal locality of detected stale accesses across PA's cannot be preserved. Such temporal locality exists in the following partial sequence of a variable: PA, stale *use*, PA, stale *use*, ..., PA, stale *use*.

Pre-access invalidation has an advantage in that cache copies accessed by non-stale accesses are not invalidated. As a result, temporal locality of accessing non-stale cache copies can be extended beyond a pair of adjacent PA's.

3.1.3. Selective Invalidation

Invalidation is usually accomplished by *invalidate* instructions. Since *invalidate* instructions consume processor cycles, it is essential to keep their number at minimum. *Early* and the *delayed* invalidations are better than *immediate* since multiple invalidations of the same cache item can be avoided. However, even though the *early* and the *delayed* approaches can reduce the number of *invalidate* instructions, to selectively invalidate cache items by executing the invalidate instructions is a sequential process, and it increases the execution time. Also, in the worst case, *immediate* invalidation has to be used and the number of *invalidate* instructions executed becomes even larger.

Invalidate instructions can be replaced by reference marking. Stale accesses are detected and marked. Provided the processor can issue a different kind of read accesses for references marked stale, the cache controller will load the up-to-date copies from the global memory at access time upon such read requests. This is essentially a variation of *pre-access immediate* invalidation (this does not apply to *post-access* invalidation). However, it saves processor cycles since no explicit *invalidate* instructions are required.

3.2. Fast Selective Invalidation

This approach was proposed in [ChVe88]. It is a *pre-access* approach with hardware assist. *Invalidate* instructions, which increase execution time, are replaced by reference marking. In addition, status bits in the cache memory help to take the advantages of temporal locality and the reduced over-invalidation associated only with *early* invalidation. Furthermore, this approach is not affected by *over−invalidation* as in cases when subscript analysis does not help *pre-access early* invalidation.

The fast selective invalidation scheme works as follows. References detected as *stale* accesses are marked *memory−read* and the non-stale accesses are marked *cache−read*. Each cache word has a *change* bit and a *clear* bit. It is assumed that the processor sets the *change* bits of the whole cache in one clock upon executing a *change−cache* instruction. The *clear* bit can be similarly set by a *clear−cache* instruction. The *change−cache* instruction is inserted after each PA. The cache controller differentiates a *memory−read* from a *cache−read*. Whenever a *cache−read* is executed, the cache-controller will report a hit if the referenced item is in the cache. Whenever a *memory−read* is executed, the cache controller handles the access in two ways when a cache copy exists:

(1) If the *change* bit of the referenced word is set, the controller reports a miss. The word is loaded from the global memory and the *change* bit is reset.

(2) If the *change* bit is not set and the tags are matched, the controller reports a hit.

A write to a cache word always resets the corresponding *change* bit.

One advantage of this scheme is that it is faster than invalidating page table entries or cache items one-by-one as in previously proposed schemes [Smit85, McAu86]. Also, fewer non-stale cache copies than in any previously proposed scheme are invalidated since both accesses to read-only variables and non-stale accesses to read-write shared variables require no invalidation of the cache copies.

The *change* bit helps preserve temporal locality and helps reduce over-invalidation between a pair of adjacent PA's. In so doing, no subscript analysis is needed as in *pre-access early* or *post-access delayed* approaches. When a stale access is preceded, in execution, by another stale *use* or a *def* of the same element, the *change* bit is reset due to the earlier access. Even though it is marked *memory−read*, the reset *change* bit indicates that a fresh copy has been loaded and prevents it from being invalidated again by another load from the shared global memory.

A restricted version of stale access detection has been implemented. Parallelized Fortran programs by Parafrase, a Fortran preprocessor [KKLW80], are used as inputs. The simulated performance of the fast selective invalidation scheme was evaluated and the results were presented in [ChVe88].

4. Conclusion

In this paper, a flow analysis algorithm for detecting stale accesses is proposed. Even though the detection of stale accesses is an important step in software-assisted cache coherence schemes in multiprocessor systems, it has not been fully addressed in previous work. A modified flow graph for modeling parallel execution and its effects on cache coherence is introduced such that the standard flow analysis techniques can be applied to detect stale accesses.

Possible approaches to coherence enforcement using the result of the detection scheme are discussed. The advantages and disadvantages of such approaches are described. A recently proposed scheme that relies on both hardware and software to enforce cache coherence is shown. The new approach achieves better selective invalidation and does it faster than previously proposed schemes.

The detection algorithm proposed can also be extended to manage other types of memory systems, such as local memory and multilevel cache systems in multiprocessor systems. Also, even though only one type of parallelism has been considered, e.g., *Doalls*, throughout this paper, the results can be extended to other parallel loop types and other types of parallelism.

Acknowledgements

Our thanks to Tim Davis, John Fu, and Elana Granston for their valuable comments.

References

[AhSU86] Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman. **Compilers Principles, Techniques, and Tools.** Addison-Wesley, 1986.

[ArBa84] Archibald, James and Jean-Loup Baer. *An Economical Solution to the Cache Coherence Problem.* **11th Int. Symp. on Comp. Arch.** (June 1984) pp.355-362.

[BrMW85] Brantley, W. C., K. P. McAuliffe and J. Weiss. *RP3 Processor-Memory Element.* **Proc. of the 1985 Int. Conf. on Parallel Processing** (1985) pp. 782–789.

[CeFe78] Censier, L. M. and P. Feautrier. *A New Solution to Coherence Problems in Multicache Systems.* **IEEE Trans. Comput., Vol. C–27** (Dec. 1978) pp. 1112–1118.

[ChVe87] Cheong, H. and A. V. Veidenbaum. *The Performance of Software-Managed Multiprocessor Caches on Parallel Numerical Programs.* **International Conference on Supercomputing** (June 1987).

[ChVe88] Cheong, H. and A. V. Veidenbaum. *A Cache Coherence Scheme With Fast Selective Invalidation.* To appear in **Proc. 15th International Symposium on Computer Architecture** (June, 1988).

[EGKM85] Edler, Jan, Allan Gottieb, Clyde P. Kruskal, Kevin McAuliffe, Larry Rudolph, Marc Snir, Patricia Teller and James Wilson. *Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach.* **Proc. 12th Int. Symp. on Comp. Arch.** (June, 1985) pp. 126–135.

[Good83] Goodman, James R. *Using Cache Memory to Reduce Processor-Memory Traffic.* **Proc. 10th Annual International Symposium on Computer Architecture** (June, 1983) pp. 124–131.

[KaEW85] Katz, R. H., S. J. Eggers, D. A. Wood, C. L. Perkins and R. G. Sheldon. *Implementing a Cache Consistency Protocol.* **Proc. 12th Ann. Int. Symp. on Comp. Arch.** (June, 1985) pp. 276–283.

[KKLW80] Kuck, D. J., R. H. Kuhn, B. Leasure and M. Wolfe. *The Structure of an Advanced Vectorizer for Pipelined Processors.* **Fourth International Computer Software and Applications Conference** (Oct., 1980).

[LeYL87] Lee, R. L., P. C. Yew and D. H. Lawrie. *Multiprocessor Cache Design Considerations.* **The 14th Annual International Symposium on Computer Architecture** (June, 1987) pp. 253–262.

[Lee87] Lee, Roland L. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors.* **Ph.D. Thesis, Tech. Rep 670, Center of Supercomputing Research and Development, U. of Illinois at Urbana–Champaign** (August 1987).

[LuBa80] S. F. Lundstrom and G.H. Barnes, *Controllable MIMD Architecture.* **Proceedings of the 1980 ICPP** (1980) pp. 19–27.

[McAu86] McAuliffe, Kevin K. *Analysis of Cache Memories in Highly Parallel Systems.* **Ph. D. Thesis, Tech. Rep. #269, Courant Institute of Mathematical Sciences, New York University, New York** (1986)

[McCr84] McCreight, Edward M. *The Dragon Computer System: An Early Overview.* **Technical Report** (June, 1984).

[PaPa84] Papamarcos, Mark and Janak Patel. *A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories.* **Proc. 11th Ann. Int. Symp. on Comp. Arch.** (June, 1984) pp. 348–354.

[RuSe84] Rudolph, Larry and Zary Segall. *Dynamic Decentralized Cache Schemes for MIMD Parallel Architectures.* **Proc. 11th Ann. Int. Symp. on Comp. Arch.** (June, 1984) pp. 340–347.

[Smit85] Smith, Alan Jay. *CPU Cache Consistency with Software Support and Using "One Time Identifiers".* **Proceeding of the Pacific Computer Communications '85** (1985) pp. 153–161.

[Tang76] Tang, C. K. *Cache System Design in the Tightly Coupled Multiprocessor System.* **Proc. AFIP National Computer Conference** (1976) vol. 45, pp. 749–753.

[Veid86] Veidenbaum, Alexander V. *A Compiler-Assisted Cache Coherence Solution for Multiprocessors.* **1986 Proc. ICPP** (Aug., 1986) pp. 1029–1036.

[Wolf82] Wolfe, M. J. *Optimizing Supercompilers for Supercomputers.* **Ph. D. Thesis, Tech. Rep No. UIUCDCS–R–82–1105, Department of Computer Science, U. of Illinois at Urbana–Champaign** (October 1982).

SHARED DATA CONTENTION IN A CACHE COHERENCE PROTOCOL

Michel Dubois and Jin-Chin Wang

Department of Electrical Engineering
University of Southern California
University Park, Los Angeles, CA 90089-0781
(213)743-8080, dubois@priam.usc.edu

ABSTRACT

In many shared-memory multiprocessors, private caches are associated with each processor and coherence among caches is maintained in hardware by a cache coherence protocol on the memory bus. *Multithreading*, or the concurrent execution of the multiple processes forming a task is also often supported in these systems. The efficiency of multiprocessor systems for a parallel algorithm depends to a large extent on the amount of sharing in the algorithm and on the effectiveness of the cache protocol for shared data accesses. Even if the cache sizes were infinite, the number of processors which can be connected to a bus would still be limited by the bus traffic due to the initial loading of data and instructions in each cache, and to the active sharing of writable data.

In this paper, we analyze shared data contention in parallel algorithms and its effects on the performance of a cache coherence protocol under the assumption of infinite cache sizes. A simple program model for data sharing is introduced and an analytical closed-form solution is found for all components of the cache coherence overhead. We then study the overhead due to shared data contention in five parallel algorithms: the iterative Jacobi algorithm, the iterative S.O.R. algorithm, the parallel quicksort, and the shuffling and the non-shuffling F.F.T. Finally, these overheads are compared with the predictions of the analytical model.

1. INTRODUCTION

In modern multiprocessors, a given algorithm may be decomposed into cooperating processes that run in parallel [2]; this technique is called *multitasking* or *multithreading.* In a shared-memory multiprocessor processes working in parallel on the same algorithm cooperate through the sharing of data in memory. Usually, in a shared-memory multiprocessor a cache [16] is associated with each processor, in order to reduce both memory access latency and memory-bus traffic. Shared data may be cached provided a hardware protocol maintains consistency among multiple copies of the same data in different caches. By caching shared data one hopes to increase the average hit ratio and to reduce the bus traffic.

Three techniques are possible to analyze the performance of cache-based multiprocessors: measurements on an existing system, trace-driven simulations [13], and simulations or analytical models based on a program behavior model. These three techniques vary in cost, flexibility and accuracy. If an analytical model is shown valid for a number of significant systems and algorithms, then it can be used to predict their performance at reduced cost.

Early work on the analytical evaluation of cache-based systems was done by Patel [14] and Briggs and Dubois [4]; these papers either ignored the cache coherence effect or assumed that shared data are not cached. In order to include the effect of data coherence in the models, several authors have taken the approach of modeling the workload with an analytical program model [17]; the workload model is then used in a simulation or in an analytical model. In [7], Dubois and Briggs introduced a model for multiprocessor program behavior and derived a closed-form solution in the case of a multiprocessor system with finite caches and LRU (Least Recently Used) replacement policy. Archibald and Baer [3] published simulation results comparing various cache protocols. In [18] Vernon and Holliday introduced a timed Petri net model driven by the same program model as in [7]; no closed-form solution was proposed.

In this paper, we introduce a new program behavior model based on the observation that shared data are modified in critical sections; a closed form solution is derived for the average miss ratio and penalty. Finally, the model is applied to several algorithms and effect of cache block size is investigated.

2. INFINITE CACHE MODEL AND GENERAL ASSUMPTIONS

Several simplifying assumptions are made throughout this paper. We first list them, then we discuss their validity.

Assumption 1: The size of all caches is infinite.
Assumption 2: The models are in steady-state. Initial transients are not included.
Assumption 3: Process preemption and migration are disallowed; i.e., a process executes from start to finish on the same processor and without interruption.

The major motivation for studying the infinite cache model is its simplicity. Most parameters of the cache do not affect the model prediction, such as cache size, cache organization or cache replacement policy; the resulting models are therefore parsimonious. Moreover, present trends in memory chip sizes indicate that fast and large caches are becoming possible. In these caches, most of the misses are due to the initial loading of the data, and to coherence invalidations. It is expected that the infinite cache model will become more and more relevant as the level of integration of static RAM chips increases.

Modeling transient effects in an infinite cache is not difficult, but the models are not very interesting. For example, at program start, caches are empty; every block referenced in a parallel algorithm must first be brought into one of the caches; this initial miss is not counted in the models. The number of these initial misses is simply equal to the total number of different blocks accessed during the whole execution of the parallel algorithm.

The third assumption may be the most restrictive. It is realistic in systems supporting group scheduling [11]. Under the group scheduling strategy, all processes participating in a task are scheduled and preempted together. We will reconsider this assumption in Section 7.

3. CACHE COHERENCE PROTOCOL

The cache coherence protocol considered in this paper is an invalidation based protocol described for example in [10](page 522); other protocols have been designed [3], and the techniques described in this paper could be applied to any one of them.

Generally, in a coherence protocol, multiple copies of the same cache block may be present in different caches, provided the copies are Read-Only (RO copies), that is, provided no processor has modified any word in the block. If a processor needs to modify a word in a block, it must obtain a Read-Write copy (RW copy), i.e. a unique copy of the block: this may involve invalidating copies of the block in other caches. Usually, a block containing only instructions, private data or shared constants will be tagged as RO, while blocks containing shared writable data may be tagged as RW. Therefore, we distinguish between S-blocks and P-blocks. An S-block contains at least one shared writable data item while a P-block contains only instructions, private data or shared Read-Only data. In the protocol selected for this study, the following cache events on an S-block may occur in a multiprocessor systems with infinite caches and in steady-state (refer to Figure 1):

1. *Miss:* this event occurs when the data is referenced and is not present in the cache. We denote this event as M (Miss). All misses occurring as a result of the following events are accounted for as M events.

2. *Transition from RO to RW:* this event occurs when a processor needs to modify a block already present in another cache as RO; a miss may occur and an invalidation must be sent to the processor(s) possessing RO copy(ies); we denote this event as IN_RO (for INvalidation of RO copy(ies));

3. *Transition from RW to RO:* this event occurs when a processor reads a block present in another cache as RW; besides the occurrence of a miss, a signal must be sent to the cache possessing the RW copy and this cache must update the main memory; we denote this event as CS_RW (Change State of a RW copy).

4. *Transition from RW to RW* in a different cache: this event occurs when a cache needs to modify a block which is owned as RW by another cache; it implies a miss, an invalidation, and the update of main memory; we denote this event as IN_RW (INvalidation of a RW copy).

When writable blocks are actively shared, copies must be transferred among caches and invalidation signals must be sent. As the number of processors actively sharing a block increases, the invalidation activity usually increases.

4. ANALYTICAL MODELS

The access pattern to shared data in multiprocessor systems depends on the algorithm. We can distinguish between two broad classes of shared variables: synchronization data (such as locks) and other shared operands. Synchronization data are used to coordinate process execution or to protect shared operand accesses.

Kung [12] classifies multitasked algorithms into *synchronized* and *asynchronous* algorithms. In *asynchronous* algorithms, accesses to shared operands are not protected and each processor may access the data as it needs them. In *synchronized* algorithms, accesses to shared data are restricted, either by explicit synchronization or simply by structuring the forking and joining of processes. In *synchronized* algorithms, shared writable data are accessed either in **critical sections** [2] (only one process can access the data at a time either to Read or to Write) or in **semi-critical sections** [5] (multiple processors can read a data item at a time, but if a process has to modify the data item, it must do so in mutual exclusion). Figures 2 (a) and (b) illustrate both access patterns. In these Figures, only accesses to a specific shared datum are shown.

4.1 Analytical Model for one S-block

The program model is derived from the model in [17]. We had to extend this model because it did not capture the locality of references to shared writable data. In another paper [6], we presented two additional program models for which the effect of cache coherence can be solved analytically and which take into account the accesses made in critical sections and semi-critical sections. All these program models can be defined as a special case of the following model. The program model that we are about to define assumes that accesses by one processor to a shared writable block are done in uninterrupted bursts. Besides modeling critical section accesses, the burst model takes into account the locality of reference on shared blocks. We use the same notation as in [7].

The P processors execute independent streams of instructions and generate homogeneous streams of references. S-blocks belong to different *sets*; all S-blocks in a set are accessed with the same pattern, even if they are accessed by different processors; the program model, model parameters and coherence overheads are identical for all the S-blocks in a set.

Let q_s be the fraction of references to S-blocks. The fraction of S-block accesses that are for a particular S-block i is p_i with i=1,...,N_s and N_s is the total number of shared writable blocks. S-block i is shared by J_i processors ($J_i \leq P$, the total number of processors in the system). Processors access an S-block i in bursts. l_i is the average burst size, i.e. the average number of accesses to the block during an access burst. An isolated access is counted as a burst of size one. The average burst size can be found by dividing the total number of references by the number of access bursts. For example, for the examples of Figure 2 the average burst sizes are $l_i = 2$ (Figure 2(a)) and $l_i = 1.75$ (Figure 2(b)), assuming that one cache block contains only one data element.

The fraction of processor references which start an access burst for a given S-block i is $\frac{q_s \cdot p_i}{l_i}$. The basic approximation of the model is that access bursts are independent from one another. When a processor completes an access burst, all the J_i processors have the same probability of starting the next access burst to the shared block. We designate by W_i the probability that the block is modified during an access burst. Because of the infinite cache assumption, there is no interference among cache accesses to different blocks and the events occurring for one block are independent of the events occurring for any other block; the state transitions of S-block i can be observed in isolation.

The *global state* of an S-block i is described by the number of caches possessing a copy of the block and by the status RO or RW of the block. The global states are denoted by 1_RW, 1_RO, 2_RO,..., J_i_RO. We can ignore the identity of specific processors because the multiprocessor is homogeneous and symmetric.

The Markov chain for the state transitions of S-block i is shown in Figure 3 (we have dropped the index i in the Figure for clarity. Note that all parameters are for a given S-block i). The state of the Markov chain is the global state of the block *whenever an access burst is completed* (except for the state MEM, which is the state of the block before the first reference to it). It is clear from Figure 3(a) that states MEM and 1_RO are transient states. Figure 3(b) shows the reduced Markov chain where the transient states have been removed. We will only solve the Markov chain of Figure 3(b). A state transition occurs in this state diagram every time a burst of accesses is completed by one processor. The transition probabilities from state k_RO, $k < J_i$, are found as follows.

1. From state k_RO to state $k+1\_RO$: The probability of this transition is the product of of the probability that the next burst contains only Read accesses, $(1-W_i)$, and of the probability that the access burst is made in one of the $J_i - k$ other caches, $(\frac{J_i-k}{J_i})$.

2. From state k_RO to state k_RO: This is the case when the next access burst contains only Read accesses to one of the k caches. The transition probability is $(1-W_i)\frac{k}{J_i}$.

3. From state k_RO to state 1_RW: This is the case when the next access burst modifies the block. The transition probability is W_i.

The transition probabilities from states 1_RW and J_i_RO are derived from similar arguments.

This finite state Markov chain is aperiodic and irreducible. Let's denote by $Pr(1)$ and by $Pr(k)$, k=2,...,J_i, the state probabilities of state 1_RW and states k_RO respectively. The state probability distribution is given by the set of equations: (see for example [1])

$$Pr(k) \quad = \quad \frac{(J_i-k+1)(1-W_i)}{(J_i-k)(1-W_i)+J_iW_i} \cdot Pr(k-1), \qquad (1)$$
$$for\ k = 2, ..., J_i$$

and

$$Pr(1) = \frac{J_i \cdot W_i}{(J_i-1)(1-W_i)+J_i \cdot W_i} \qquad (2)$$

With these state probabilities, one can compute the probability of occurrence of each coherence event. When there are k copies in k processor caches a miss occurs at the beginning of a new access burst, i.e. at a state transition in Figure 3(b), if the next processor to start an access burst is one of the $(J_i - k)$ processors without a copy in their cache. Therefore, the fraction of references to S-block i which miss in the cache is equal to the fraction of state transitions causing a miss divided by the average burst length l_i.

$$M_i = \frac{1}{l_i} \quad [\quad Pr(1) \cdot \frac{(J_i-1)}{J_i} + Pr(2) \cdot \frac{(J_i-2)}{J_i} + ...$$
$$+Pr(J_i-1) \cdot \frac{1}{J_i}] \qquad (3)$$

After some transformations, one finds simply (see [9]):

$$M_i = \frac{1}{l_i} \frac{(J_i - \ 1) \cdot W_i}{1+(J_i - \ 1) \cdot W_i} \qquad (4)$$

In the Markov graph of Figure 3(b) a transition from state 1_RW to state 1_RW results from three possible sequences of events:

1. the processor owning the RW copy of the block has started a new burst of accesses for the same block; no event is recorded for S-block i in this case;

2. a different processor has started an access burst for S-block i and its first access to the block is a Write; an event of type IN_RW must be recorded for S-block i;

3. a different processor has started an access burst for S-block i and its first access to the block is a Read followed by a Write; one event of type CS_RW followed by one event of type IN_RO must be recorded.

In order to differentiate between the 2nd and the 3rd cases, we have to introduce a new factor f_i, which is the fraction of Write bursts[1] such that the first access is a Write. f_i can easily be computed from a string of references. For example, in Figure 2(a), $f_i = .75$, and, in Figure 2(b), $f_i = .5$.

An invalidation of RO copies occurs whenever an access burst modifies a block in an RO state. It also occurs in a transition from 1_RW to 1_RW, provided the second access burst is executed by a different processor and starts with a Read. Therefore, the fraction of accesses to S-block i invalidating RO copies in other caches is given by:

$$I_i = \frac{1}{l_i}\left[W_i(1-Pr(1)) + W_i(1-f_i)Pr(1)\frac{J_i-1}{J_i}\right]$$

[1] By definition, a Write burst is an access burst containing *at least* one Write access.

A change of state from RW to RO occurs whenever a burst leaving the block in state 1_RW is followed by a burst starting with a Read access by any other processor. Therefore, the fraction of references to S-block i changing the state from RW to RO is:

$$C_i = \frac{1}{l_i}\left[Pr(1)(1-W_i)\frac{J_i-1}{J_i} + Pr(1)W_i(1-f_i)\frac{J_i-1}{J_i}\right]$$

Finally, an invalidation of a RW copy occurs whenever an access burst leaving the block in state 1_RW is followed by a Write from any other processor. The fraction of references to S-block i causing such an event is therefore:

$$D_i = \frac{1}{l_i}Pr(1)f_iW_i\frac{J_i-1}{J_i}$$

In these equations, $Pr(1)$ is given by equation (2).

4.2 System Effects

The results of the previous section are combined to model the effect of cache coherence on the overall system performance under the assumptions of Section 2. Two performance measures are derived: the miss ratio and the average coherence penalty.

4.2.1 Miss ratio

In the infinite cache model, if we neglect the transients, the miss rate is given by the miss rate on shared writable data, i.e.,

$$M = q_s \sum_{i=1}^{N_s} p_i M_i \qquad (5)$$

where N_s is the total number of shared writable blocks. The value for M_i is obtained by applying equation (4) and depends on different values of the parameters for different S-block i. In many cases, the terms in the above sum can be clustered by grouping the shared writable blocks into sets; within a set all blocks are referenced with the same pattern, and therefore have the same value of M_i.

To find M from equation (5), one need to specify the model parameters for all sets of blocks. In the studies presented in [7,3,18], there is only one set of parameters. Implicitly, it is assumed that the models can be applied to a single *average* set including all the shared writable data. Parameters are therefore computed as averages. For the five examples presented in Sections 5 and 6, this approach has proven to be acceptable.

4.2.2 Average Coherence Penalty

A processor runs at maximum speed when no cache misses or coherence events occur. To each coherence event corresponds an average penalty, λ_{EVENT}. The penalty associated with an event is defined as the average time that a processor is blocked at the occurrence of the event. The average coherence penalty per memory reference to S-block i is:

$$\lambda_i = M_i\lambda_M + I_i\lambda_{IN\_RO} + C_i\lambda_{CS\_RW} + D_i\lambda_{IN\_RW}$$

M_i, I_i, C_i, and D_i were defined in Section 4.1.

If we neglect the transients, the average coherence penalty per memory reference is given by the sum of the coherence penalties on each shared writable block i: that is,

$$\lambda_{total} = q_s \sum_{i=1}^{N_s} p_i\lambda_i \qquad (6)$$

As for the system miss rate, S-blocks can be clustered into a few sets in which blocks have the same average coherence penalty. The average penalty could also be approximated by the penalty for an *average* block.

The average coherence penalty adversely affects the processor efficiency. In powerful and expensive main frame multiprocessors, any loss of processor efficiency is critical for the performance/cost ratio of the system.[2]

5. APPLICATION TO MULTITASKED ALGORITHMS (CACHE BLOCK SIZE IS ONE)

If the cache block size is one data element, then the values of the parameters are straight forward. This section deals with a cache block size of one. In Section 6, we will investigate the block size effect.

We compare the model predictions with the simulation of specific algorithms running on shared-memory multiprocessors in which each processor has a private data cache of infinite size. The behavior of the relaxation and FFT algorithms are data independent. To simulate these algorithms a simulation methodology described in [8] was applied. In this methodology, the algorithm is actually executed on a uniprocessor and the multiprocessing effect is obtained by executing the program of each simulated processor in turn. The simulator *switches* from one simulated processor to the next on each shared data access and synchronization primitive execution. A slightly different technique was applied to the quicksort algorithm and will be explained in Section 5.2. Many simulation results can be derived analytically. These analytical derivations whenever they are possible are presented in [9]. Also the computation of the model parameters for each case is given in [9].

5.1 Relaxation Algorithms for Partial Differential Equations

We consider two iterative schemes [20]: the Jacobi and the Successive Over Relaxation (S.O.R.) algorithms. In the Jacobi iterative algorithm, the computation consists in repetitively updating each point of a grid as follows:

$$x_{i,j}{}^{(K+1)} = \frac{1}{4}\left[x_{i+1,j}{}^{(K)} + x_{i-1,j}{}^{(K)} + x_{i,j+1}{}^{(K)} + x_{i,j-1}{}^{(K)}\right]$$

The Jacobi iterative algorithm requires to maintain two grids. In each iteration, each point of one grid is updated by using the values of the 4 neighbors in the other grid. Then the processors synchronize and the two grids are interchanged. For a M × N grid, there are 2(M+N) boundary grid points and these points are not modified during execution: these points

[2] If T_1 is the mean execution time of an instruction in the uniprocessor system (in microsecond), and if r is the average number of memory references per executed instruction, the average instruction execution time in the multiprocessor is $T_1 + r\ \lambda_{total}$, and the MIPS rate (Million of Instructions Per Second) per processor is $MIPS\ rate = \frac{1}{T_1 + r\lambda_{total}}$.

are Read-Only and can be treated as P-blocks. The grid points adjacent to these boundary points are called *outer* grid points. The reference pattern to outer grid points is different from the pattern to *inner* grid points. Consider, for example, two square grids of size 8 × 8 for which we allocate one processor to each subgrid of size 4 × 4, as displayed in Figure 4. The sets of shared writable data are circled in Figure 4. There are only three sets of S-blocks (in the sense of Section 4.1), (1) inner grid points with $J = 2$, (2) outer grid points with $J = 2$, and (3) inner grid points with $J = 3$.

Shared writable data are accessed in semi-critical section in the Jacobi iterative algorithm. In one iteration, the shared data in one grid are Read-Only and are accessed by different processors, and in the next iteration they are modified by a single processor in a critical section phase. Since the models developed in this paper are for infinite caches in steady-state, we consider iterations n and n+1 where n>1 and we assume that the data caches are large enough to contain the two subgrids accessed by each processor.

In the S.O.R. algorithm, only one copy of the grid is needed and iterates are updated according to the red/black ordering: grid elements which are in even positions (the sum of the indexes is even) are tagged as black, others as red. Each iteration proceeds in two sweeps. The red elements are updated in the first sweep, and the black elements are updated in the second sweep. After each sweep, each processor has to synchronize with at most 4 neighbors. Each processor has the same number of red and of black iterates. The equation for the update of an iterate in the (K+1)th iteration is :

$$\begin{aligned} x_{i,j}{}^{(K+1)} = \quad & (1 - \omega)\, x_{i,j}^{(K)} \\ + \quad & \frac{1}{4}\, \omega\, \left[x_{i+1,j}^{(K)} + x_{i-1,j}^{(K)} + x_{i,j+1}^{(K)} + x_{i,j-1}^{(K)}\right] \end{aligned}$$

where ω is the relaxation factor.

During one sweep of the algorithm, some shared grid points are read by multiple processors, and some others are read and modified by one processor. As for the Jacobi iterative algorithm, there are 3 sets of shared writable grid points.

5.2 Quicksort

Quicksort is a *divide-and-conquer* algorithm, which sorts a file A[1], A[2], ...A[N] by rearranging it to make the condition that A[1], ...A[j-1] ≤ A[j] ≤ A[j+1], . . .A[N] hold for some j, and by recursively applying the same procedure to the subfiles A[1], . . .A[j-1] and A[j+1], . . .A[N]. A program for the quicksort on a uniprocessor is given in Figure 5 [15]. In a multiprocessor, at the end of each splitting phase, one subfile is processed by the same processor and the other subfile is sent to a different processor, until all processors are busy. The computation proceeds in a tree-like fashion with as many leaf nodes as there are processors, as displayed in Figure 6. At the leaf of the computation tree, each processor is assigned the quicksorting of one subfile.

In an infinite cache environment, coherence activity occurs mostly while the tree is growing. We only consider the part of the execution from the start of the algorithm until a processor has reached the bottom of the tree and has finished the first iteration of its local quicksort. While a processor splits a subfile no other processor accesses any data item in the subfile. Therefore shared data are accessed in critical sections. The P subfiles obtained at the leaves of the tree (P is the total number of processors), correspond to P different paths in the computation tree; the data in these subfiles are shared to various degrees. For example, one subfile is accessed by the same processor from start to finish, and therefore is not shared. Other subfiles may be shared by J processors, $J = 2, ..., log_2 P + 1$. $P - 1$ sets of shared writable data can be identified. Each set can be associated with a leaf in the computation tree. Figure 6 illustrates the different sets for $P = 8$.

Let's assume that the probability of an exchange is q during each splitting phase. The values of the parameters for the model are $l = 1 + q$, $W = q$, and $J = 2, ..., log_2 P + 1$. The exact values for the miss rate and for the average penalty can be computed and verified by simulation. The simulation of the quicksort proceeds as follows. We take a file of size N and scan it repetitively as in the quicksort algorithm. However, we do not execute the quicksort, because its behavior is data dependent. Rather, in the simulation, shared data accesses are generated as follows. Every time an element of a subfile is visited, we decide to exchange with probability q; a subfile is always split into two equal halves.

5.3 Fast Fourier Transform (FFT)

The one-dimensional non-shuffling FFT algorithm for N data items is represented by a butterfly graph with $log_2 N$ stages. A bit-reversal permutation is applied at some point of the algorithm, so that the results are stored in the same order as the initial data items. Let s(k),k=0,1,2,...,N-1 be N samples of a time function. The DFT (Discrete Fourier Transform) of s(k) is defined to be the discrete function x(j),j=0,1,2,...,N-1, where

$$x(j) = \sum_{k=0}^{N-1} s(k)\, e^{\frac{2\,\pi\, i\, j\, k}{N}}$$

where j=0,1,..N-1 and i= $\sqrt{-1}$.

In the non-shuffling FFT algorithm, we divide the array of N items into P chunks containing $\frac{N}{P}$ consecutive items. Each processor computes the FFT for its chunk, containing $\frac{N}{P}$ data items. For N=16 and P=4, the non-shuffling FFT algorithm is illustrated in Figure 7. In general, each block is shared by $log_2 P + 1$ processors and the algorithm can be divided into two parts. In the first part, i.e., the first $log_2 \frac{N}{P}$ stages of the butterfly, every shared block is accessed by one processor. In the second part, i.e., in each of the last $log_2 P$ stages of the butterfly, each shared block is first read by two processors and then modified by a single processor in a critical section. Since there is no coherence activity in the first part of the non-shuffling FFT algorithm, we examine the second part of the algorithm. Synchronization is necessary in this algorithm and is denoted by dotted lines in Figure 7. In general, 2 $log_2 P$ synchronization points are needed in the second part (If the algorithm used two copies of the array and alternated between the copies only $log_2 P$ synchronization points would be needed.) There is only one set of shared writable data in this problem.

Another algorithm for FFT in multiprocessors is the shuffling FFT. In this algorithm, computations of partial FFTs alternate with shuffling stages in which data are passed among processors. Figure 8 presents the shuffling FFT algorithm for an example where N=16 and P=4. During each butterfly computation and each shuffling stage, each shared block is read and updated by a single processor. There is only one set of shared writable data and each data is shared by $J = 2$ processors.

5.4 System Effects

Table 1 records the overall miss rate on data and the average coherence penalty for the five algorithms. The unit for the penalties is the average penalty for a miss. We have chosen the following penalties for each event: $\lambda_M = \lambda_{CS\_RW} = \lambda_{IN\_RW} = 1$, $\lambda_{IN\_RO} = 0.5$. There are three numbers per entry in the Table. The first one is obtained summing by applying the model to each set and by the contributions of each set. The second number is obtained by computing the average values of the model parameters and by applying the model with these average values. These two numbers are very close because there is only one set of data, or one set dominates, or all sets are accessed with the same probability. The third number is obtained by simulation. Remember that these performance estimates apply to the part of the algorithm where sharing of writable data occurs, and that the first miss and its associated penalty are not counted.

6. EFFECT OF CACHE BLOCK SIZE

Cache block size is an important factor that affects the system performance. When a cache block contains more than one datum, access bursts to an S-block have different characteristics and in general, it is much more difficult to apply the models.

In this section, we show how the cache block size, B, affects the model and simulation results.

We chose the following penalties rate for each coherence event: $\lambda_M = \lambda_{CS\_RW} = \lambda_{IN\_RW} = 0.75 + 0.25B$ and $\lambda_{IN\_RO} = 0.5$. We therefore model the penalty for a block transfer by a simple linear function of the block size.

6.1 Relaxation Algorithms for Partial Differential Equations

In the two iterative algorithms, when the cache block size increases, the number of sets of S-blocks also increase. For instance, there are five sets of S-blocks when B is two, eight sets of S-blocks when B is four, and ten sets of S-blocks when B is eight. For the Jacobi iterative algorithm, Figure 9 illustrates the eight different types of S-blocks, named type 1 to type 8 when the cache block size is four data elements.

In the case of an $M \times M$ Jacobi array, when the cache block size exceeds $\frac{M}{\sqrt{P}} + 1$, processors update S-blocks alternatively, and hence the number of references l in each burst is equal to one; J can be as high as $2 \cdot \sqrt{P}$ or even P.

We have identified all sets and we have applied the model to each set for block sizes from 1 to 256, and for a grid size of 128 × 128. Figures 10 and 11 show the comparison between model prediction(dotted curve) and simulation (plain) for the system miss ratio and the system penalty (obtained by summing the contributions of all sets). These curves are valid for any number of processors P provided $B < \frac{M}{\sqrt{P}} + 1$. These two Figures show that the analytical program model yields very good predictions for the Jacobi iterative algorithm when the cache block size is greater than two (error is less than 2%).

In the S.O.R. algorithm, for any cache block size, the number of sets of S-blocks are the same as in the Jacobi iterative algorithm; however, the reference patterns to the blocks in the sets are different.

In the case of an $M \times M$ S.O.R. array, when the cache block size exceeds $\frac{M}{\sqrt{P}} + 1$, S-blocks are updated alternatively by different processors. In this case, J can be as high as $2 \cdot \sqrt{P}$ or even P.

Figures 12 and 13 illustrate the results of the system miss ratios and system penalities for different cache block sizes for a 128 × 128 grid. These curves are independent of the number of processors provided $B < \frac{M}{\sqrt{P}} + 1$. The model (dotted) is compared to the simulation (plain). Again, the Figures show that the model is very reliable.

6.2 Quicksort

For simplicity we have considered only the *ideal* quicksort: the number of processors and of elements in the file is a power of 2 and each split is perfect, i.e. a subfile of size n is exactly split in two subfiles of size $\frac{n}{2}$. When the number of data elements in one cache block is less than or equal to $\frac{N}{P}$, where N is the total number of data elements, a cache block can only be referenced by one processor in a splitting phase.

Figures 14 and 15 show the results of the system miss ratio and the system penalty for the model (dotted) and for the simulation (plain). In these simulations, the file size was N=64K, and the number of processors was 8. Different curves would be obtained for different number of processors. The relative error in these two Figures is less than 20%.

6.3 Fast Fourier Transform (FFT)

In the non-shuffling FFT algorithm, there is only one set of S-blocks. When the number of data elements in a cache block is less than or equal to $\frac{N}{P}$, each S-block is shared by $log_2 P + 1$ processors.

When the number of data elements in one cache block exceeds $\frac{N}{P}$, S-blocks bounce back and forth among processors at every Write. In this case, each block is shared by more than $log_2 P + 1$ processors.

In the shuffling FFT algorithm, there is also only one set of S-blocks. When the number of data elements in a cache block is less than or equal to $\frac{N}{P}$, each S-block is shared by two processors. Otherwise, when the number of data elements in one cache block is larger than $\frac{N}{P}$, S-blocks move back and forth among processors.

Figures 16-19 show the results for non-shuffling and shuffling FFT algorithms. The file size is 64K and the number of processors is four. These curves would be different but would have the same shape for larger number of processors. The relative errors between model predictions and simulations are between 20% and 30% for the system miss ratio, and between

5% and 20% for the system penalty.

7. DISCUSSION OF RESULTS

It has been observed that the combined effects of critical sections (for all block sizes) and of the spatial locality [16] of accesses (for block sizes larger than one) to shared writable data result in access bursts to such data by different processors. This is the basic premise of the paper. Based on this observation, we have extended a previous program model for the sharing of data, and we have tried to match the model predictions and the predictions of simple simulations of algorithms in multiprocessors with infinite caches.

It appears that iterative algorithms such as the Jacobi or S.O.R. are very well suited to cache-based systems with large data caches, because shared data contention is low (in realistic cases, the number of processors sharing a given writable block is less than four and the fraction of accesses to shared writable data is low). Figures 10-13 show that bigger block sizes do not improve the overall hit rate on shared data and cause more penalty: the average miss rate on each S-block access decreases (i.e. M_i decreases) but the number of accesses to such blocks increases (i.e. q_s increases); the probability of a coherence event per access to S-blocks decreases, but this is more than compensated by the increase of q_s and of the penalty associated with each coherence event. For shared data accesses, the block size should be small (one or two data elements). Note that this conclusion is only valid if the caches are large enough to contain all data across successive iterations; moreover, the first iteration causes large number of misses for the initial load of data and instructions. These transients are helped by a bigger block size. From the Figures, we observe that a block size of 16 data elements is acceptable for shared data accesses. These conclusions are valid for many configurations of processors as explained in Section 6.1.

The results on the quicksort are somewhat *artificial* because we have simulated the *ideal* quicksort only, to simplify. In practice, one would have to run simulations for multiple random input files and take averages. Nonetheless, the stochastic simulation does represent one possible execution of the quicksort. Bigger block sizes are a definite advantage in the ideal quicksort algorithm, up to a size of $B = 16$. If the algorithm used to estimate the median is good, this conclusion should hold also in the general case (we expect however more contention).

Bigger block sizes also improve the performance of the FFT routines (strangely enough up to a block size of 16 data elements, again). While the penalties on individual coherence events increase with the block size, the fraction of shared-data accesses causing these events decreases and the total number of accesses stays constant, as the block size increases.

In all the simulations we ran, there is a maximum block size beyond which the performance drops sharply. This block size depends on the size of the problem and on the decomposition of the algorithm (i.e. the number of processors).

From Tables 1-6 and from Figures 10-18, it appears that for the five algorithms studied in this paper, the precision of the model based on the idea of access bursts is good in many cases. We never expected the models to fit exactly each case: because of the large data reduction in the stochastic models, a given model with given parameter values maps on different algorithms with different behaviors. It appears however that the models and their parameters are sufficient to approximate the shared data contention effect for some important parallel algorithms, and in the case of the infinite cache model.

If we look at the comparisons between model and simulations, it appears that the quicksort and the non-shuffling FFT result in the worst predictions; in the case of the non-shuffling FFT, the model predicts that the coherence overhead will increase with P, the number of processors, while the simulations predict that it remains constant. A closer look at Figure 7 shows that an S-block is not shared by all $log_2 P + 1$ processors at all times but rather that it is shared by different groups of two processors at different stages of the computation. Applying the model with J=2 would yield a much improved prediction of the model.

We have assumed all along that preemptions and migrations of processes were disallowed. Indeed, in all the algorithms we have studied, processes are statically scheduled. We mostly made this assumption to simplify the analysis of the algorithms. However, in many cases, the bursty behavior of accesses would be preserved if preemption, migration, and dynamic scheduling were allowed. The reason is that bursts of accesses are short and unlikely to be interrupted by preemption. Migration and dynamic scheduling would increase the randomness in the selection of the processor to start the next access burst, and therefore would alleviate the problem observed in the quicksort and non-shuffling FFT. While the parameters W, f *and* l would remain the same as in this paper (assuming that time between preemptions is very large compared to the average burst time), the parameter J would have to be different. If migration is allowed, then private writable data will behave like shared writable data and the model could be applied to private data, as well.

8. CONCLUSIONS

In this paper, we have presented and solved a simple model for the caching of shared writable data in multiprocessor systems executing parallel algorithms. The simplicity and generality of the results stem from the infinite cache hypothesis. The infinite cache model is independent of all cache parameters (e.g, organization or replacement policy).

There are many extensions possible to this work. First of all, one could analyze the behavior of more parallel algorithms and compare it to the model predictions. One could investigate the effect of migration, preemption or dynamic scheduling. Some parameters in the model, such as l, are easier to estimate directly than others [9], such as J when migration and preemption is allowed. Besides using the results of simulations or measurements to estimate these parameters, one can use the model to derive upper bounds (for example, if J=P or if W=1, then the miss ratio obtained through equation (4) is an upper bound.) Finite-cache effects should be studied.

Finally, given the simplicity of the program behavior models, one can derive simple results for proposed coherence protocols in order to compare their effectiveness in handling shared writable data [19].

Acknowledgement. This research is supported by an NSF Research Grant No. CCR-8709997.

References

[1] A.O. Allen. *Probability, Statistics, and Queueing Theory.* Academic Press, 1978.

[2] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1), March 1983.

[3] J. Archibald and J.L. Baer. Cache-coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[4] F.A. Briggs and M. Dubois. Effectiveness of private caches in multiprocessor systems with parallel-pipelined memories. *IEEE Transactions on Computers*, C-32(1), January 1983.

[5] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[6] M. Dubois. Effect of invalidations on the hit ratio of cache-based multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987.

[7] M. Dubois and F.A. Briggs. Effects of cache coherency in multiprocessors. *IEEE Transactions on Computers*, C-31(11):1083–1099, November 1982.

[8] M. Dubois, F.A. Briggs, I. Patil, and M. Balakrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 909–916, August 1986.

[9] M. Dubois and J.C. Wang. *Shared Data Contention in a Cache Coherence Protocol.* Technical Report CRI 88-26, CRI Department of Electrical Engineering, University of southern California, 1988.

[10] K. Hwang and F.A.Briggs. *Computer Architecture and Parallel Processing.* Mac Graw-Hill, 1984.

[11] A.K. Jones and P. Schwartz. Experence using multiprocessor systems - a status report. *Computing Surveys*, 12(2), June 1980.

[12] H.T. Kung. *Algorithms and Complexity: New Directions and Recent Results.* J.F. Traub Ed., New York: Academic Press, 1976.

[13] R.L. Lee, P-C Yew, and D.H. Lawrie. Multiprocessor cache design considerations. In *Proceedings of 14th Annual International Symposium on Computer Architecture*, June 1987.

[14] J.H. Patel. Analysis of multiprocessors with private cache memories. *IEEE Transactions on Computers*, C-31(4):296–304, April 1982.

[15] R. Sedgewick. *Quicksort.* New York: Garland Publishing, Inc., 1980.

[16] A.J. Smith. Cache memories. *Computing Surveys*, 14(3), September 1982.

[17] J.R. Spirn. *Program Behavior : Models and Measurements.* Elsevier Computer Science Library, 1977.

[18] M.K. Vernon and M.A. Holliday. Performance analysis of multiprocessors cache consistency protocols using generalized timed petri nets. In *Performance '86 and ACM Sigmetrics 1986 Joint Computer Performance Modelling, Measurement and Evaluation*, pages 9–17, May 1986.

[19] J.C. Wang and M. Dubois. A performance comparison of cache coherence protocols based on the access burst model. In *Second Annual Parallel Processing Symposium*, pages 73-87, April 1988.

[20] D. Young. *Iterative Solution of Large Linear Systems.* Academic Press: New York, 1971.

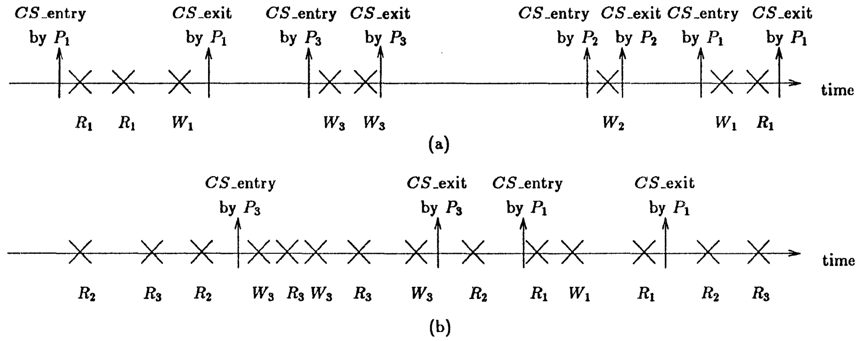


Figure 2: (a) Access pattern to a shared writable datum protected by critical sections. (b) Access pattern to a shared writable datum protected by semi-critical sections. (R_j: Read access to shared datum X by processor j, W_j: Write access to shared datum X by processor j.)

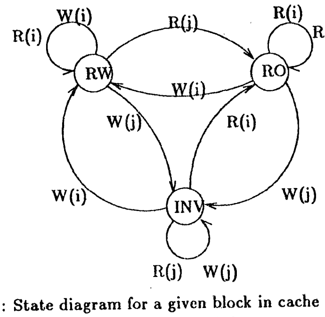


Figure 1: State diagram for a given block in cache i (infinite cache assumption)(R(i) : Read block by processor i, R(j) : Read block by processor j, W(i) : Write to block by processor i, W(j) : Write to block by processor j.)

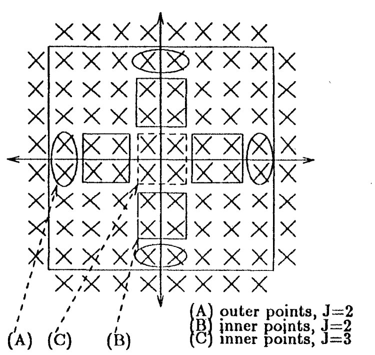
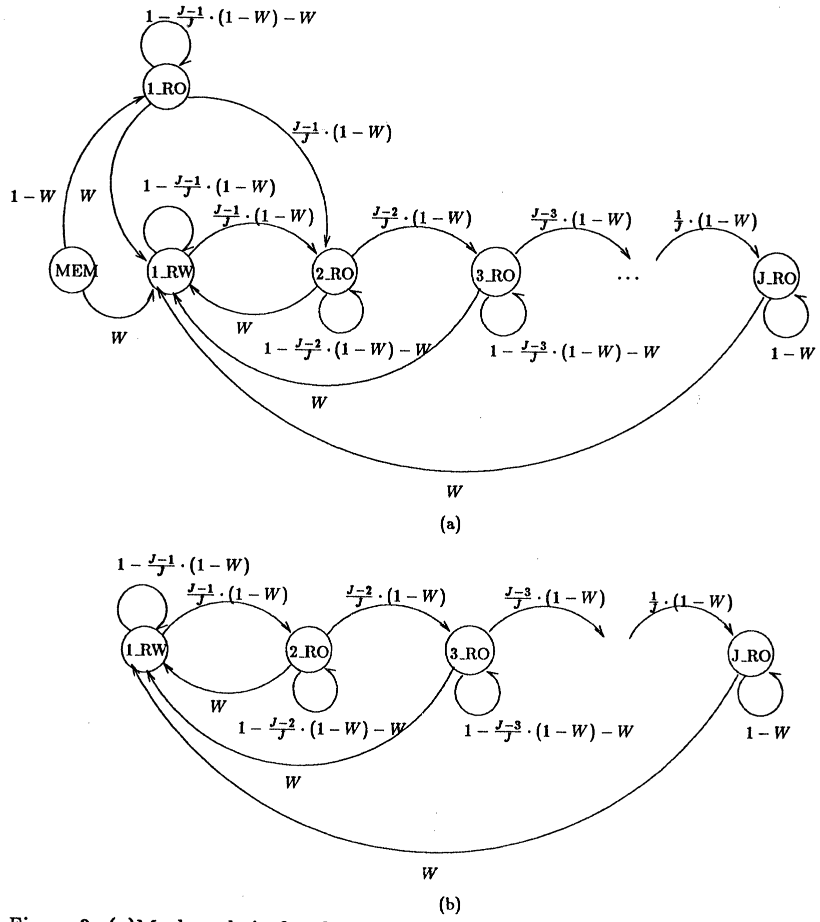


Figure 4: The three data sets in the Jacobi iteration.



$1-\frac{J-1}{J}\cdot(1-W)-W$
1_RO
$\frac{J-1}{J}\cdot(1-W)$
$1-W$
W
$1-\frac{J-1}{J}\cdot(1-W)$
$\frac{J-1}{J}\cdot(1-W)$
$\frac{J-2}{J}\cdot(1-W)$
$\frac{J-3}{J}\cdot(1-W)$
$\frac{1}{J}\cdot(1-W)$
MEM
1_RW
2_RO
3_RO
...
J_RO
W
W
$1-\frac{J-2}{J}\cdot(1-W)-W$
$1-\frac{J-3}{J}\cdot(1-W)-W$
$1-W$
W
W
(a)
$1-\frac{J-1}{J}\cdot(1-W)$
$\frac{J-1}{J}\cdot(1-W)$
$\frac{J-2}{J}\cdot(1-W)$
$\frac{J-3}{J}\cdot(1-W)$
$\frac{1}{J}\cdot(1-W)$
1_RW
2_RO
3_RO
J_RO
W
$1-\frac{J-2}{J}\cdot(1-W)-W$
$1-\frac{J-3}{J}\cdot(1-W)-W$
$1-W$
W
W
(b)

Figure 3: (a)Markov chain for the state transitions of an S-block shared by J processors (including transient states). (b)Markov chain for the state transitions of an S-block shared by J processors (without transient states).

```
Procedure quicksort (l,r:integer);
var n,t,i,j : integer;
begin
   if r > l then
      begin
         n:=a[r]; i:=l-1; j:=r;
         repeat
            repeat i:=i+1until a[i] ≥ n;
            repeat j:=j-1until a[i] ≤ n;
            t:=a[i]; a[i]:=a[j]; a[j]:=t;
         until j ≤ i;
         a[j]:=a[i]; a[i]:=a[r]; a[r]:=t;
         quicksort(l,i-1);
         quicksort(i+1,r)
      end
end;
```

Figure 5: Program for the uniprocessor quicksort.

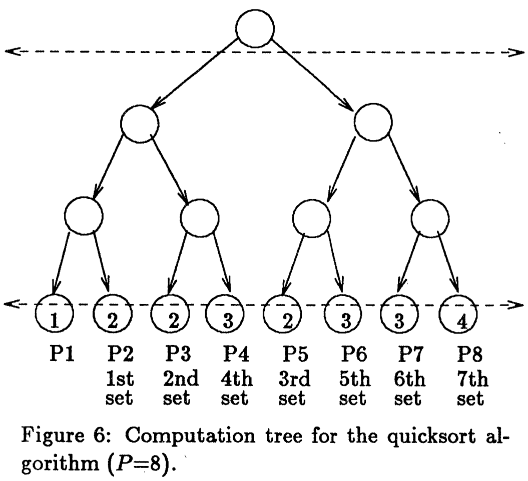


Figure 6: Computation tree for the quicksort algorithm (P=8).

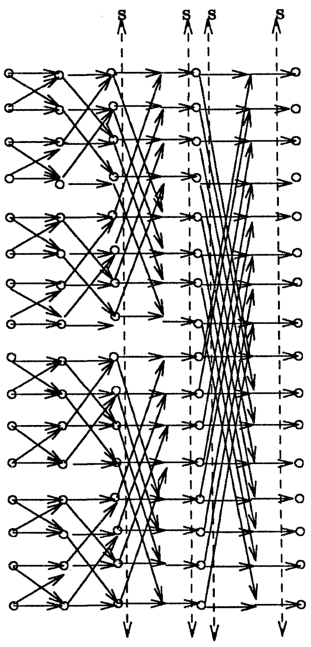


Figure 7: Non-shuffling FFT algorithm for P=4 and N=16.

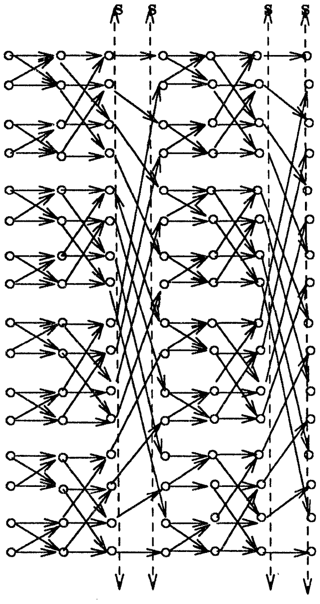


Figure 8: Shuffling FFT algorithm for P=4 and N=16.

Table 1: System Effects (block size of one)

(1) Model using average values of the parameters
(2) Sum of the contributions of each set (model)
(3) Sum of the contributions of each set (simulation)
In the following Table, parameters, J, W, l, f, are average values.

Algorithm	J	W	l	f	q_s	N_s		miss	penalty
Jacobi iteration	2.01	0.201	1	1	0.0309	1016	(1)	0.0052	0.0125
							(2)	0.0052	0.0124
							(3)	0.0063	0.0156
S.O.R. iteration	2.01	0.201	1.201	0	0.0309	508	(1)	0.0043	0.0108
							(2)	0.0043	0.0104
							(3)	0.0053	0.0130
Quicksort	2.72	0.500	1.5	0	0.8750	57344	(1)	0.2692	0.6079
(P=8)							(2)	0.2584	0.5869
							(3)	0.2512	0.5294
Non-shuffling	3.00	0.333	1	1	1.0000	65536	(1)	0.4000	0.7810
FFT							(2)	0.4000	0.7810
(P=4)							(3)	0.3333	0.8333
(P=16)	5.00	0.333	1	1	1.0000	65536	(1)	0.5714	0.9817
							(2)	0.5714	0.9817
							(3)	0.3333	0.8333
Shuffling FFT	2.00	0.600	17.8	0.4	0.7500	49152	(1)	0.0158	0.0348
(P=4)							(2)	0.0158	0.0348
							(3)	0.0171	0.0426
(P=16)	2.00	0.600	15.4	0.4	0.9375	61440	(1)	0.0228	0.0502
							(2)	0.0228	0.0502
							(3)	0.0247	0.0617

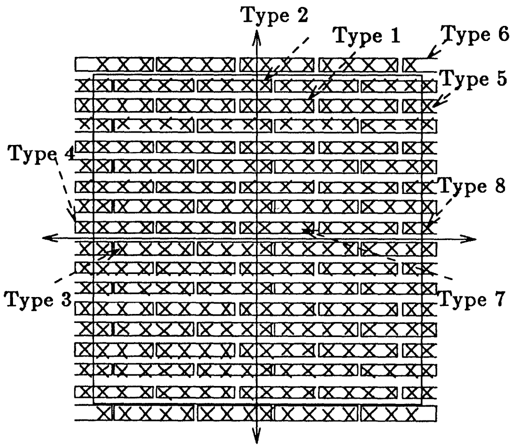


Figure 9: The eight sets of $\dot{S}$-blocks in the Jacobi iteration when B is equal to four.(M=16, P=4)

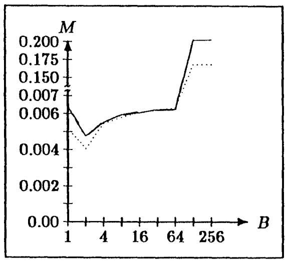


Figure 10: The system miss ratio for Jacobi iteration algorithm

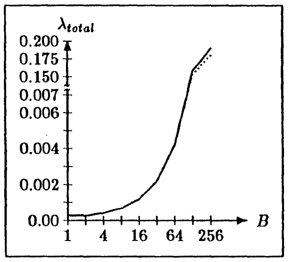


Figure 11: The system total penalty for Jacobi iteration algorithm

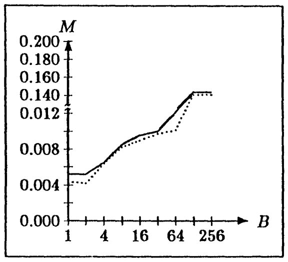


Figure 12: The system miss ratio for S.O.R. iteration algorithm

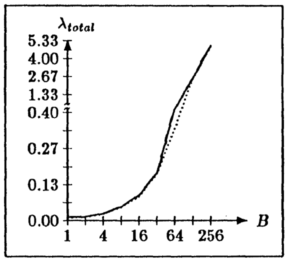


Figure 13: The system total penalty for S.O.R. iteration algorithm

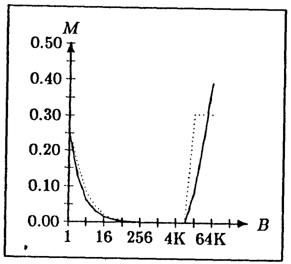


Figure 14: The system miss ratio for quicksort algorithm

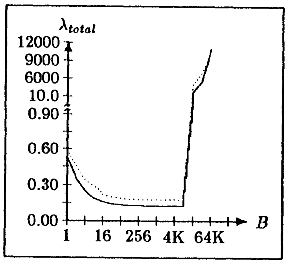


Figure 15: The system total penalty for quicksort algorithm

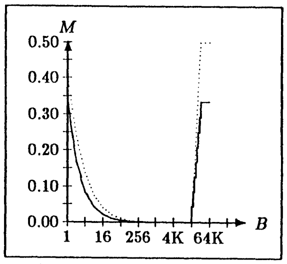


Figure 16: The system miss ratio for Non-shuffling FFT algorithm

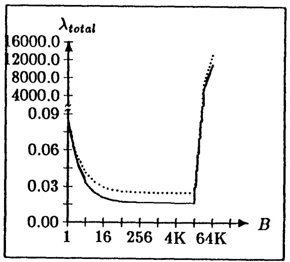


Figure 17: The system total penalty for Non-shuffling FFT algorithm

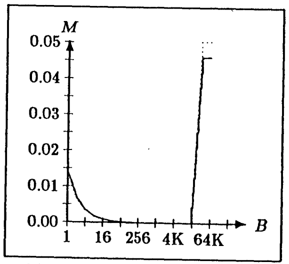


Figure 18: The system miss ratio for Shuffling FFT algorithm

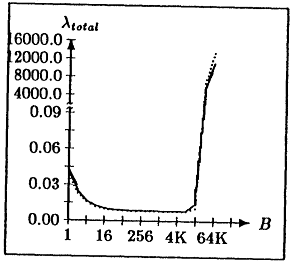


Figure 19: The system total penalty for Shuffling FFT algorithm

plain line : simulation result
dotted line : model prediction

Multiprocessor Performance Measurement Using Embedded Instrumentation

Thomas L. Sterling
Albert J. Musciano
Donald J. Becker
Advanced Technology Department
Harris Corporation
PO Box 37, MS 3A/1912
Melbourne, FL 32902

Randy B. Osborne
MIT Laboratory for Computer Science
545 Tech Square, Room 205
Cambridge, MA 02139

Abstract

The multiprocessor is a powerful medium for conducting empirical studies of parallel processing techniques and architectures. Critical to the success of this approach is the nature, detail, and accuracy of the measurements acquired to evaluate system behavior. While software methods of determining behavior characteristics are flexible, they are intrusive, perturbing the operation of the system and yielding measurements of marginal accuracy. This paper describes two hardware instruments developed for multiprocessor behavior analysis that help circumvent the intrusive properties of software measurement techniques. One instrument, DLA, measures memory access latencies and delays due to contention for shared resources. Another instrument, SySM, monitors processor software activity, providing execution profiling statistics. This paper discusses three examples of their use in parallel processing research on the Concert Multiprocessor Testbed.

1. Introduction

The future of high performance computing will rely on current research into parallel processing architectures and techniques. Effective tools are needed to explore parallel program characteristics and parallel architecture behavior. Two major classes of tools currently being used are simulators and prototype parallel systems.

Simulators[1][2][3] are widely employed due to their flexibility and ease of modification. Unfortunately, they suffer from innate slowness, restricting the applications that can be run on the simulated target system. Another tool is a prototype system on a parallel computer: experimental and commercial multiprocessors[4] and SIMD[5] machines capable of executing significant parallel algorithms in acceptable time. While these systems enable programmers to run significant applications and make coarse measurements with software, they do not permit easy evaluation of behavioral details. Software support for detailed measurements is intrusive, perturbing the behavior of the parallel system by the act of measurement.

Software simulation can provide detailed traces of any activity within a modeled system, often producing enormous amounts of information. Real time instrumentation cannot access all of the elements of a system; it does not have the time and storage resources available to collect exhaustive traces in a nonintrusive manner. Fortunately, some experiments only require a simple set of statistics, rather than a time domain trace. In these cases, instruments can be devised that record these statistics instead of going through the intermediate step of acquiring a time trace.

Harris Corporation's Advanced Technology Department and the MIT Laboratory for Computer Science have each developed a version of a multiprocessor called Concert. Concert[6] features embedded instrumentation, allowing parallel processing experiments to be performed in real time with a minimum of intrusion. The nature and functionality of Concert's instrumentation and examples of its use in parallel processing research will be given.

SySM and DLA are instruments in Concert which measure performance loss parameters. SySM[7] provides hardware support for monitoring software behavior in a nearly nonintrusive manner. The programmer divides the application program into a set of mutually exclusive segments. As execution shifts from one segment to another, the application informs SySM of the transition. SySM accumulates the number of entries and the total time spent in each segment, and tracks nested interrupts. DLA[8] provides nonintrusive measurements of the time lost due to memory access latency and contention for shared communication channels. It determines the amount of time each processor spends waiting for access to shared buses, the amount of time it spends accessing hierarchical memory,

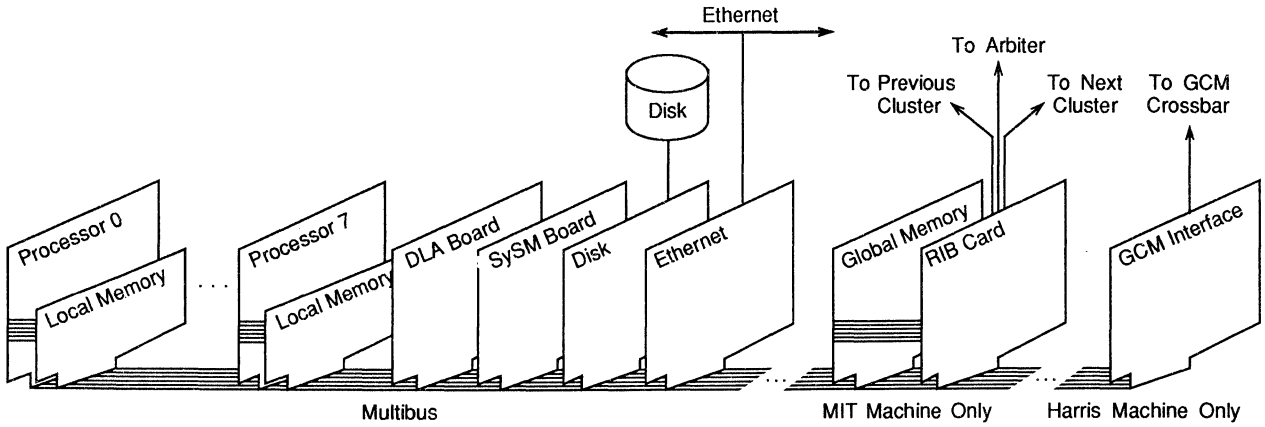


Figure 1. A Typical Concert Cluster

and the number of times atomic test-and-set operations are performed. SySM and DLA provide a powerful mechanism for observing those elements of a multiprocessor that result in performance loss.

Three different research projects have utilized Concert, SySM and DLA: the SpoC[9] parallel execution environment, an interpreter for the Multilisp[10] language, and an emulator of the YARC static dataflow architecture[11]. These experiments illustrate how SySM and DLA can be used to reveal the behavior of both the application software and the multiprocessor hardware. The disparate functionality of the tools is used to advantage, as each application uses the tools in a different manner.

2. Background

The development of SySM and DLA was based upon the needs of the researchers investigating parallel performance degradation, and was shaped by the architecture of Concert Multiprocessor. The details of Concert, and the approach used to analyze degradation, played a significant role in the design and subsequent use of SySM and DLA.

2.1. The Concert Multiprocessor Testbed

The Concert multiprocessor was developed as a flexible facility for empirical research in the field of parallel processing. Two versions of Concert have been implemented: the first at the MIT Laboratory for Computer Science and the second at the Advanced Technology Department of Harris Corporation's Government Systems Sector. They are logically equivalent, supporting the same software environments and executing a shared base of applications. The two Concerts differ in the means by which global memory is shared and system wide communication is performed. Concert has been used to study parallel algorithms, languages, computing models, system run time strategies, and parallel computer architecture. The Concert environment consists of the multiprocessor hardware, a message passing library for parallel program development, and a set of utilities supporting local area network communication and a disk file system.

Concert is a tightly coupled shared memory multiprocessor. It incorporates up to 64 MC68000[12] microprocessors, organized in eight clusters of up to eight processors each. Each processor has 512 Kbytes of local memory; the system also has 8 Mbytes of global memory. In addition, a set of globally accessible registers provide interprocessor interrupts. The eight clusters are connected with the global memory and registers by one of two communication mechanisms.

The cluster is the basic unit of the Concert multiprocessor. It contains up to eight processors, each with its own local memory connected via a private high speed bus. Some clusters include disk controllers for secondary storage and Ethernet interface boards. All boards within the cluster use a common Multibus. Each cluster contains a global memory interface board. Each cluster also contains DLA and SySM instrumentation hardware. A typical Concert cluster is shown in Figure 1.

The MIT Concert uses a dynamically segmented RingBus to interconnect eight clusters of four processors each. Each cluster holds 1 Mbyte of the 8 Mbyte global memory. Processors accessing global memory *within* their cluster use the cluster's internal bus; memory in other clusters is accessed via the RingBus. A central arbiter processes requests for RingBus access from the clusters, establishing non-overlapping paths between the requesting clusters and the desired global memory segments. A cluster is attached to the RingBus by means of the RIB (RingBus Interface Board) which also contains the relevant subset of the system global registers. The system level architecture of the MIT Concert multiprocessor is shown in Figure 2a.

The Harris Concert employs a conventional crossbar switch to connect its eight clusters of eight processors each to the system's global memory and registers. The memory is organized in a 16-way interleaved structure of 512 Kbyte blocks. This interleaving reduces memory contention by distributing memory references across the blocks. An additional block contains the global system registers. The architecture of the Harris Concert multiprocessor is shown in Figure 2b.

2.2. Performance Degradation

The performance of a multiprocessor is intimately coupled to the factors which contribute to performance degradation. These factors cause the actual performance of a multiprocessor to deviate from the ideal case. By quantifying and reducing these losses, overall performance can be improved. There are four general sources of loss which must be observed to characterize a multiprocessor performance. These are:

Starvation the time a processor is idle due to inadequate parallelism in the application program.

Contention the delay experienced by a processor attempting to obtain exclusive access to a shared resource already in use. Two processors accessing the same queue, for example, must serialize their accesses in order to retain queue integrity.

Overhead the work that must be performed by a processor to manage the application's parallelism. This work would not be performed by a uniprocessor executing the same application. Overhead includes synchronization, task creation, and task scheduling.

Latency the time required to access distant memory objects in systems with distributed communication and memory. This includes the impact of cache misses due to frequent context switching.

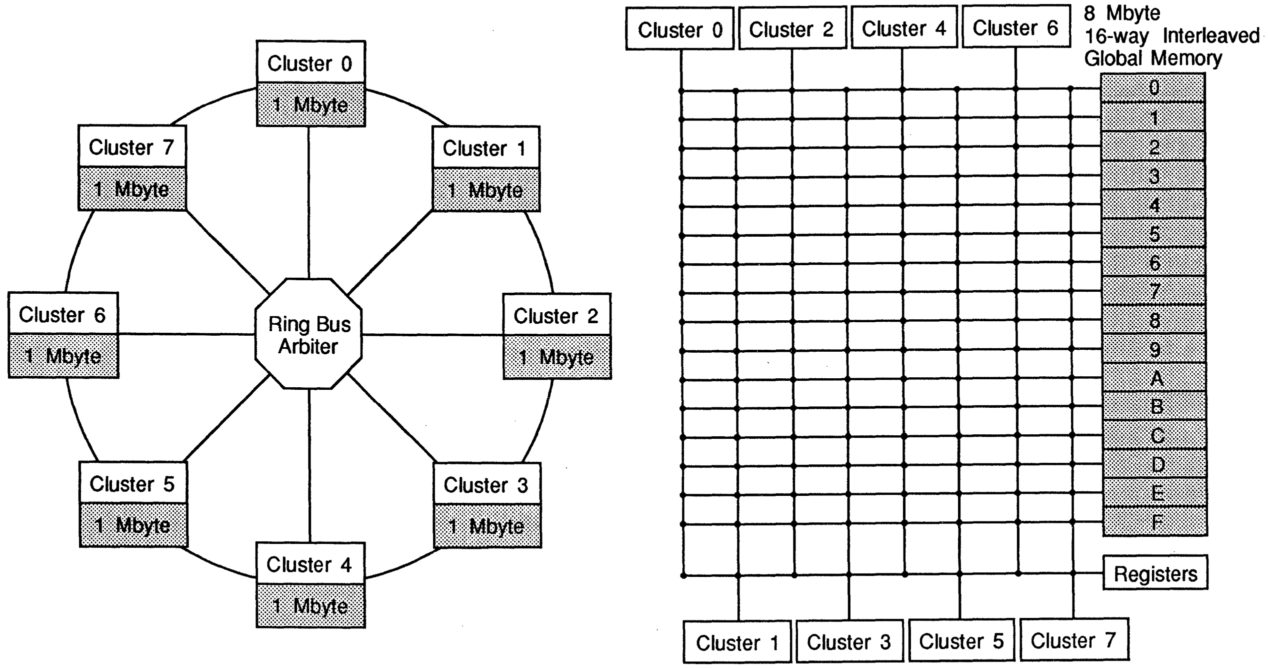
Measuring each of these losses permits the characterization of the multiprocessor's performance model.

3. Embedded Instrumentation

Hardware support for efficient monitoring of multiprocessor system behavior has been realized in two embedded instruments within the Concert Multiprocessor testbed. Two classes of behavior are examined: contention and latency at the hardware level, and starvation and overhead at the software level.

3.1. DLA: Monitoring Contention and Latency

Performance loss due to hardware operation includes contention for access to shared physical resources and latency of access to nonlocal objects. Contention in Concert occurs when multiple processors require access to the same Multibus, RingBus segment, or global memory block simultaneously. Losses resulting from latency occur when processors must access data in global memory, which has a slower cycle time than local memory. Neither of



(a) The MIT Machine **(b) The Harris Machine**

Figure 2. The Concert Multiprocessors

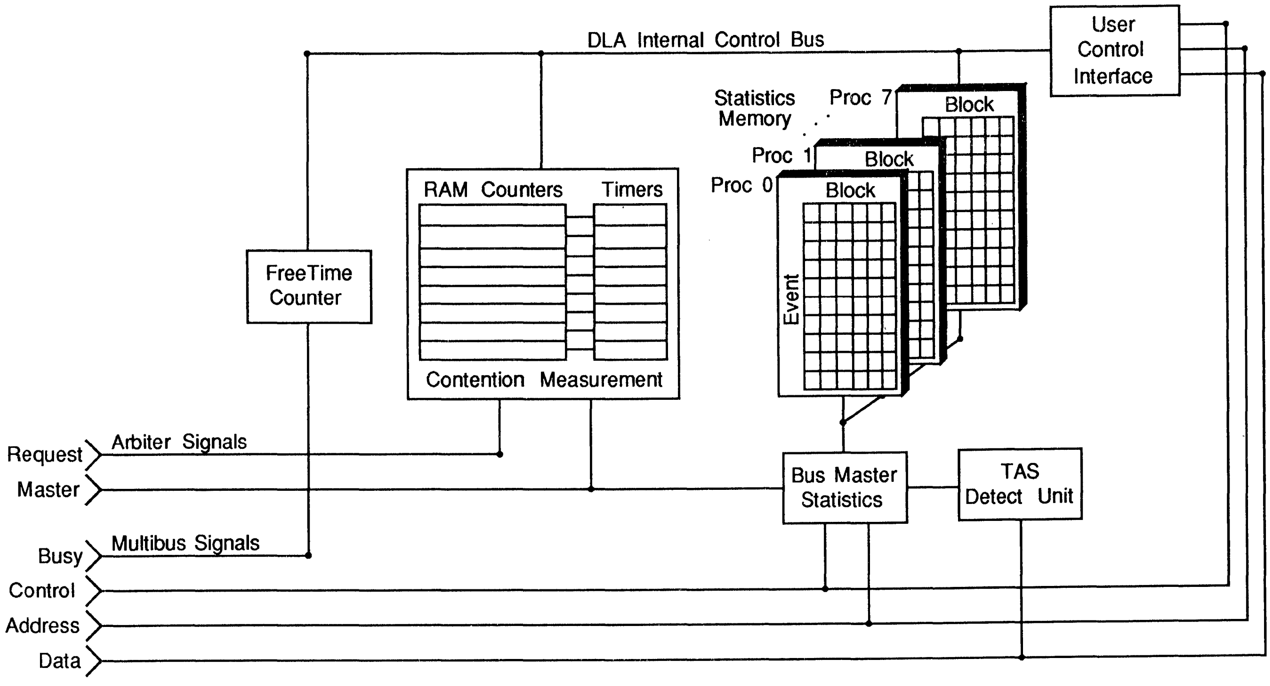


Figure 3. The DLA Block Diagram

these losses occurs in a uniprocessor; such degradation can be attributed to parallel processing. The DLA (Degradation due to Latency and Arbitration) hardware installed in each cluster of Concert measures the time each processor spends waiting for and using the Multibus, and accessing memory.

DLA measures several aspects of Multibus activity, including:

Free Time the time in which the Multibus is idle.

Wait Time time a processor spends contending for use of the Multibus.

Access Time time the processor spends using the bus. DLA also counts read and write operations within each block of global memory.

Global memory latency is the global memory cycle time, plus global communication arbitration time, plus contention for an individual global memory block. On Concert, the loss resulting from memory block contention can be determined by measuring the uncontended cycle time and subtracting this from the total memory access time.

Another source of loss is contention for shared data structures protected by some mutual exclusion discipline, such as atomic test-and-set operations. Although this loss is not a type of hardware degradation, DLA can be used to estimate it. DLA counts the number of successful and unsuccessful TAS operations for each processor. The ratio of failed to successful TAS operations gives an indication of the amount of performance loss caused by contention for shared data structures.

The functional block diagram for DLA is shown in Figure 3. Information is acquired from two external sources: the Multibus and its arbitration unit. DLA has four parts: free time measurement, bus contention measurement, bus cycle statistics, and DLA control. These are depicted with their external and primary internal connections.

The free time module monitors the busy signal from the Multibus and determines the amount of time the bus is not in use. The contention module maintains separate timers dedicated to each bus master to measure the amount of time lost by each processor waiting for the bus. An extended measurement range is obtained by incrementing counters in RAM when a particular timer overflows. A timer is active when its respective processor has requested the bus but has not been granted master status by the arbiter. Any number of timers can be active simultaneously depending on the traffic density.

The majority of measurements come from the bus master statistics module. Only one processor is involved at a time since only one can be master of the bus at a time. The measurements to be updated in the statistics memory are determined by the current bus master, the memory block accessed, and the type of access. Both the event counter and the accumulated time for the specific event type are modified. In addition, the TAS detect module senses if a compound test-and-set operation is being performed and updates the appropriate counter based upon its result.

The user control and interface module initializes the DLA, starts and stops each experiment, and collects the

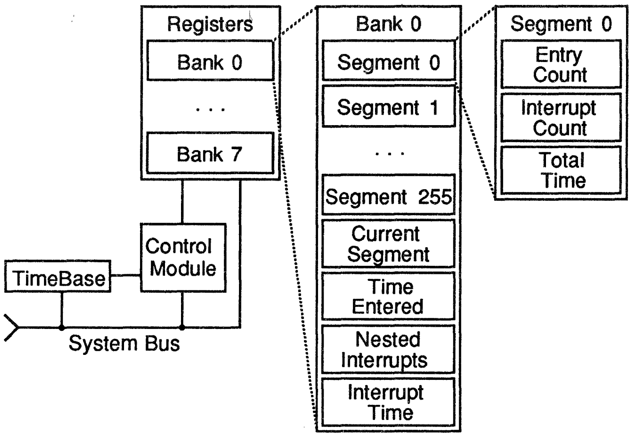


Figure 4. The SySM Block Diagram

results. Measurement intervals can be started and stopped independently on a per processor basis. Between successive measurement intervals, statistics values can be reset or allowed to accumulate.

3.2. SySM: Monitoring Software Behavior

SySM (System Software Monitor) is an instrument for monitoring the behavior of software running on individual processors within a tightly coupled multiprocessor. While the DLA can be used with no software modifications, the SySM requires a small amount of interaction with the application code. A program is divided into a set of mutually exclusive segments. At any point in time, the program is in one of these segments. When the program passes into a new segment, an instruction included by the programmer sends a single word message to SySM identifying the new segment being entered. SySM tracks the amount of time each processor spends in each segment and the number of times each segment is entered. The result is a profile of the computation in terms of the user defined segments.

A SySM board is installed in each cluster of the Concert multiprocessor, monitoring the software behavior of processors in the cluster. SySM consists of eight banks of segment registers, a control module, a time base, and external interfaces, as shown in Figure 4. Each bank of segment registers is dedicated to a specific processor in the cluster. SySM interfaces to the cluster's Multibus and the bus arbiter. The control module accepts state transition commands from the Multibus and modifies the appropriate register bank accordingly, using the current time base value.

A bank of segment registers is dedicated to each processor. Each bank is divided into 255 segments, each segment containing three registers. These registers include number of entries, time in segment, and number of interrupts. The *number of entries* register indicates the number of times the segment was entered. The *time in segment* register is the total time spent in the segment by the processor. The *number of interrupts* register shows the number of times the processor was interrupted while in that program segment.

There are four additional registers in each bank. These are the *current segment* register, the *time entered* register, the *number of nested interrupts* register, and the *time in interrupt* register. The current segment register indicates the program segment that is being executed. The time entered register contains the value of the time base at the time of the most recent segment change. The number of nested interrupts register indicates the current interrupt nesting level. The time in interrupt register indicates the total time spent servicing interrupts.

Unlike other profiling methods that produce stochastic measurements[13], the data obtained from SySM is deterministic. SySM permits this data to be collected in an almost non-intrusive manner, causing little perturbation to system behavior. SySM provides measurements with one microsecond resolution; the intrusion of each SySM access is 3.2 microseconds. Segment as small as 20 microseconds can be measured accurately. The user defines and can easily alter the definition of segments in order to narrow down the source of a particular behavior characteristic. This is useful in determining the amount of overhead incurred by the executing program. Overhead can be isolated from useful work in order to measure its impact on overall performance.

4. Examples In Parallel Processing

SySM and DLA have been used in a variety of parallel applications. These applications range from implementations of high-level parallel applications to a low-level simulation of a static data flow architecture. The differences in these applications serve to illustrate the range of applicability the SySM and DLA provide.

4.1. SpC

The SpC project is the implementation of the Simultaneous Pascal programming language[14][15] on the Concert Multiprocessor. The project encompasses the design and implementation of several significant software components, including compilers and language tools, user interface software for the Concert host, and machine dependent runtime support software which runs on the Concert hardware. Programs written in Simultaneous Pascal are compiled on the host machine, downloaded to Concert, and executed. Statistics are gathered and displayed by the runtime software.

Simultaneous Pascal is an extension of standard Pascal[16], providing the programmer with a set of explicit parallel control structures. These parallel statements include `forall`, allowing homogeneous parallelism, `fork`, allowing heterogeneous parallelism, and `traverse`, allowing parallel access to dynamic data structures. In addition, fine grained scoping and parallel expression evaluation are supported. The threads created by these parallel constructs are dynamically scheduled by the underlying runtime support software.

The compiler translates Simultaneous Pascal programs into MC68000 object code, intermixed with calls to library routines which create, schedule, synchronize, and destroy threads. These library routines are the heart of the SpoC runtime system, and map the virtual Simultaneous Pascal machine onto the physical Concert multiprocessor hardware. The effective performance and scalability of Simultaneous Pascal applications depends upon the efficient implementation of these routines.

SySM was instrumental in analyzing and tuning the SpoC runtime library. A goal of the SpoC development team was to reduce the overhead per thread to under 100 microseconds. Using SySM, the time spent in each portion of the runtime library was determined. In some cases, each routine was then subdivided into smaller parts, as short as ten microseconds, and SySM was used to measure the time each processor spent in each routine segment. The developers could then focus on those segments which were executed most often, and would yield the biggest payoff in performance tuning. The average segment times for the runtime package shown in Figure 5 indicate that this tuning operation was a success. The two critical routines, `task fetch` and `do join`, together execute in just 88 microseconds.

Application programmers can take advantage of SySM instrumentation via compiler directives.[17] The compiler generates instructions which access SySM during execution, and the runtime software gathers and displays the desired statistics when the application has completed. Such profiling provides detailed insight into how parallel applications perform, allowing application programmers to evaluate algorithms without worrying about the machine level details of SySM access. For example, a parallel application involving digital image processing was thought to be constrained by serialized access to global memory, and SySM was used to time the portions of the application which accessed the image data. A second version of the application was coded, exploiting locality by copying portions of the image into memory local to each processor. SySM data showed that the global version of the application ran faster, since the cost of copying the image exceeded the time saved by the local accesses. Without SySM instrumentation to analyze individual statement execution times, such insight could only be guessed at, rather than determined empirically.[18] The execution times of a typical instrumented application are shown in the second portion of the table in Figure 5. In particular, this data shows how programmer defined segments appear within the display of SySM statistics.

The SpoC system also uses SySM to derive computing profiles of executing applications.[19] A computing profile allows the programmer to determine how many processors are active at any point during execution, and to determine what each processor is doing at any time. In order to derive a computing profile, each processor records the time (provided by SySM) each state transition occurs. After execution, the runtime software collects this data and passes it to the Concert host machine. Subsequent processing by host based tools yields the computing profile and processor activity chart shown in Figure 6. The computing profile relates processor utilization to time, and the activity graph uses various shadings to indicate which state of a processor at each moment. These graphs are invaluable for analyzing the performance and behavior of parallel applications, and allow the programmer to see which parts of his application are serialized, reducing effective scalability.

```
Processor 07: (all times in microseconds)
State              Entered      Time    Average    Percent
System
  Awaiting Work        358     11257      31.44       0.17
  Task Fetch           358     11677      32.62       0.18
  Executing            372    781204    2100.01      11.97
  Do Fork                0         0       0.00
  Do Forall              7     30560    4365.71       0.47
  Do Traverse            0         0       0.00
  Do Join              364     20242      55.61       0.31
Application
  Thin Pixel           364   2717114    7464.60      41.62
  Neighbors           4150    866067     208.69      13.27
  Pattern             4150   1526713     367.88      23.39
  Condition C         4150    279607      67.38       4.28
  Condition D         4150    283728      68.37       4.35
** Total **                  6528169
```

Figure 5. SpoC Execution Statistics Generated by SySM

The DLA hardware allows the SpoC implementors to determine the effect of various queuing strategies on the global memory access time. The scalability of Simultaneous Pascal applications depends upon the efficiency of the runtime software, which can degrade due to excessive memory contention. Different queuing strategies (varying the number and location of queues), coupled with techniques which reduce the number of memory accesses (such as exponential falloff and interprocessor interrupts) will alter memory access patterns and affect contention. The DLA provides immediate feedback about changes in contention and access times, and allows the implementors to obtain quantitative results which document the effects of each runtime system modification.

Application programmers can obtain DLA statistics from the runtime package, and can use the resulting data to determine how object distribution in Concert's hierarchical memory affects application performance. DLA provides information about activity within each global memory bank, giving insight into how well data objects were distributed throughout global memory. Often, parallel algorithms rely upon locality of reference to exploit the available parallelism, and DLA provides insight as to how effectively the programmer (and the language tools) have exploited the locality available in Concert.

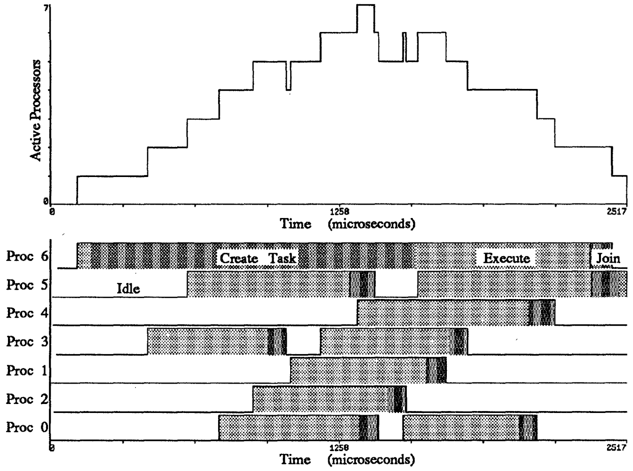


Figure 6. Computing Profile (top) and Activity Graph generated by SPoC using SySM

4.2. Multilisp

Multilisp is an extended version of the Scheme programming language [20] with explicit parallel constructs. The principal such construct is the *future*. `(future x)` creates a task to evaluate `x` and returns a placeholder—a future—for the result of `x`. When this result is computed, it replaces the placeholder; the future is then said to be determined. Meanwhile, the original task may continue execution. If a task attempts to perform a strict operation—one which requires a value, not a placeholder—on an undetermined future, the task is suspended and placed on a queue of tasks awaiting determination of the value. These tasks are activated when the future becomes determined.

Multilisp programs are compiled to a stack oriented machine language called MCODE which is then executed by an interpreter written in C and assembly language. Each processor runs an identical copy of the interpreter code. Further information is available in [21]-[24].

The MCODE interpreter running on Concert has been instrumented with SySM to determine the overhead associated with futures. The simple expression `(touch (future nil))` was used as a basis for data collection. `touch` is a strict identity operator: it returns the result of evaluating its argument. If the result is an undetermined future the task is suspended until the future is determined. The average time for various future operations computed from SySM data collected while evaluating `(touch (future nil))` is shown in Figure 7.

The data collected by DLA for one processor in cluster five during the execution of a Multilisp application is shown in Figure 8. Several aspects of Multilisp behavior are revealed. First, the large fraction of accesses to cluster five's global memory indicates that the Multilisp implementation possesses a fair degree of locality. The associated access times are relatively small because the processors in a cluster have direct access to the cluster's global memory via the Multibus. Second, the large number of accesses to cluster seven is due primarily to MCODE instruction fetches. MCODE instructions are stored in the heap in global memory. The location of these instructions varies due to garbage collection activity. Consequently, Multilisp performance varies unpredictably, depending on the cluster in which the instructions reside. A "hot spot" caused by large number of accesses to cluster seven leads to the large access time for the downstream clusters (clusters zero, one, and two). Third, many of the accesses to cluster zero (particularly the TASes) are for global Multilisp and Concert system information. Finally, the access time tends to increase with the number of RingBus segments required for the access. In this case, the cluster seven hot spot obscures this trend.

The ratio of state transitions for the `touch` and `ffib` examples discussed earlier is about 2500 and 8700 transitions per second, per processor, respectively. This yields an average time penalty of 3% (for touch) and 7% (for ffib) due to SySM segment transition commands.

The SySM and DLA have allowed concrete identification of the time inefficiencies in the Multilisp implementation and have been instrumental in speeding the execution of Multilisp programs by a factor of two to three. For example, an earlier version of the implementation suffered from frequent accesses to global information in cluster zero. A significant improvement in performance was achieved by minimizing the global information stored in cluster zero. The severity of the RingBus contention resulting from the original centralization of this information was not realized until DLA data was available.

4.3. YARC

The YARC project encompasses the development of a practical static dataflow system[25][26]. Given the small number of physical implementations of dataflow architectures and the resulting inadequate understanding of their behavior, it would be useful to estimate the performance and identify the bottlenecks of proposed systems. The Concert multiprocessor was used to simulate one such system.

The purpose of the simulator is twofold: to provide an accurate emulation of the proposed machine architecture that is fast enough to run significant programs and, more importantly, to analyze the dynamics of the proposed archi-

Operation	**Average Time**
create a future object	1.00 msec
determine a future	
0 tasks queued on future	0.43 msec
1 task queued on future	1.1 msec
touch a future	
future determined	0.24 msec
future undetermined	
until task enqueued on future	0.86 msec
total, excluding time to find task	2.6 msec
start new task (once a task found)	1.2 msec

Figure 7. SySM Timings for Various Future Operations

```
                        /--------------- Cluster Number ---------------\
                           0     1     2     3     4      5     6      7   SySM
RingBus Distance           3     4     5     4     3    n/a     1      2    n/a
Number Reads            8592  1086   460   342  3206 108833  5070  53729      0
Avg Read Time (usec)     4.9   6.4   6.5   3.5   3.5    0.8   3.4    5.0    0.0
Number Writes            626   100    59     0     0  48428   537    276  43553
Avg Write Time (usec)    6.5   6.6   7.1   0.0   0.0    0.9   3.9    6.0    0.8
Number TAS Successes     568    48    25     0     0   5039   253    142      0
Number TAS Failures        8    17     0     0     0    252    27     17      0
```

Figure 8. Multibus Traffic Generated by a Processor in Cluster Five While Executing `(ffib 20)`

tecture. In order to reveal the characteristics of an architecture, programs of significant size must be run. Without running sufficiently large programs, the true dynamics of the system cannot be observed.

Modeling a parallel computer is a job particularly well suited to parallel computers. The problem of simulating a physical system naturally decomposes into parallel subproblems because of its inherently distributed nature. The Concert multiprocessor is general enough to support simulation of the YARC static dataflow architecture, and fast enough to allow large programs to be executed. The simulator uses the hardware instrumentation in Concert to measure the pertinent parameters of execution and to separate the details of the Concert system from the those of the YARC architecture.

The YARC system simulated on Concert is a token-based static dataflow system. A collection of template storage modules/token processors is associated with an arithmetic function unit to form a processing ensemble. Processing ensembles are connected by a routing unit in a ring or toroidal configuration.

This target architecture is mapped directly onto Concert. The similarities between the organization of Concert and YARC permit each processing ensemble to be mapped onto a Concert cluster. The inter-cluster communication directly reflects the communication between the processing ensembles in the YARC system. In a similar manner, the individual template stores are mapped onto Concert processors. All template storage module communication takes place in Concert global memory.

This direct mapping permits the execution of the simulator on Concert to approximate the actions of the YARC architecture. SySM and DLA are used to account for (and factor out) the anomalous effects of the Concert architecture as well as monitor the execution of YARC. This is a crucial aspect of the simulator system. The static mapping of simulated entities to specific processors permits the instrumentation, which is processor-oriented, to be brought directly to bear.

As an example of usefulness of the SySM data, see Figure 9. Many of the entries are an exact enumeration of YARC's activity. This data, along with performance models of various potential physical YARC implementations, can be used to predict the performance of those systems.

Actions such as queueing or dequeueing are made necessary only by the structure of the simulator. The time it takes on Concert is unimportant to the extent that it does not affect the rest of the target system by serializing operations. The average time for each entry represents the token activity in YARC, and may be used to better understand or model its activity. Any variation from the average represents contention for global memory, which provides an approximation of the contention YARC would experience. While the time data produced by SySM is not directly useful, the variation among different executions represents differing utilization of the target system.

A substantial portion of the total token traffic is acknowledgments (ACK tokens) to the data tokens, as shown in Figure 9. Acknowledgment tokens constitute synchronization overhead in the static dataflow model. There are techniques to reduce such overhead at the expense of reducing parallelism.

The lack of parallelism in this problem, due to the statically distributed nature of the model, is indicated by the large value of the `Queue wait` entry. This processor spent about 33% of its time waiting for incoming data. In contrast, the processor that contained the critical path waited only twice, and other processors in the system spent up to 63% of their time waiting. This imbalance shows that the simple allocation scheme used needs much improvement.

```
Processor 62: (all times are in microseconds)
State              Entered     Time    Average   Percent
INITIALIZE               1     1956                 0.05
Total tokens          4128   237562      57.55      6.27
Memory tokens            0        0                 0.00
Data tokens           1744    58314      33.44      1.54
ACK tokens            2384    39254      16.47      1.04
Complete check        4128   367502      89.03      9.70
Template firings      1288   151090     117.31      3.99
Inspect Token         4528   253876      56.07      6.70
Reset template        1288   124893      96.97      3.30
Tokens enqueued       4144   151223      36.49      3.99
Queue wait             294  1253528    4263.70     33.09
Dequeue token         4128   482214     116.82     12.73
Local enqueue            0        0                 0.00
Foreign enqueue       4144   624905     150.80     16.49
Termination chk        472    42232      89.47      1.11
Termination inc          0        0                 0.00
** Total **                 3788549
```

Figure 9. YARC Simulation Data Obtained by SySM

5. Conclusions

Special purpose instrumentation can be an effective tool for evaluating the performance of a multiprocessor. Two devices were developed and used in a 64 processor shared memory multiprocessor. DLA measures the delay processors experience contending for shared physical resources and the losses attributed to memory access latency. SySM provides execution profiling statistics with a minimum of intrusion. The functionality and operation of both instruments were described. Examples of their application to parallel processing research were presented, along with data reflecting performance losses within each application. The SySM and DLA provide a mechanism for observing detailed characteristics of system operation, providing quantitative feedback to support systematic research.

SySM and DLA can be used to provide performance data in a variety of parallel applications. The YARC emulator uses the highest level tool, SySM, as an intrinsic and explicitly coded part of the simulator, but does not use the low-level measurement capabilities of DLA. *Sp*oC uses both tools as part of the run-time system, and provides optional user coded SySM states. Multilisp uses SySM to measure a small but important sections of the run-time system and, in contrast to YARC, makes heavy use of DLA to estimate communication and locality.

There are several advantages in the current SySM and DLA hardware. The small amount of intrusion in SySM, and lack thereof in DLA, minimize the impact of measurement on application performance. In addition, the small intrusion allows accurate, fine-grained measurements to be made. SySM and DLA operate in real time; the data they derive represent real times, not synthetic values produced from simulation. The ability to integrate both devices into significant parallel applications with a minimum of overhead is an important feature for systems developers.

The use of SySM and DLA has revealed limitations that should be considered in the design of more advanced instrumentation. SySM and DLA were developed separately; each performs its monitoring functions independently of the other. An unfortunate consequence is that data from each SySM segment cannot be directly correlated with contention and latency losses derived from DLA. Integration of the two systems would permit easy determination of hardware performance losses within each segment. Currently, this can only be estimated from the averages supplied by DLA or by starting and stopping DLA measurements on the boundaries of a specific segment.

While SySM is only slightly intrusive, there is a lower bound on the effective segment length. Each segment transition requires a single instruction cycle, obscuring the measurement of short instruction sequences. This problem can be alleviated by providing an associative buffer that stores the segment transition addresses. The buffer would monitor the address bus, looking for matches. When a match occurs, the registers of the relevant segment would be updated. This mechanism performs the same function as the current SySM but in a totally nonintrusive manner.

The associative buffer technique could also be used to control experiment interval windows. DLA and SySM could be turned on and off at selected points during program execution to acquire measurements of just part of the program. The size of the time interval that can be effectively windowed is currently constrained by the intrusive nature of the start and stop commands that must be provided explicitly in the code. The use of an associative buffer would eliminate this source of intrusion, permitting fine grained window intervals for experiments.

Parallel programs that are dynamically scheduled by underlying run time software do not reside on any one processor but migrate throughout the system. SySM measurements are processor oriented, unable to track specific tasks as they move among processors. This processor orientation can make it difficult to study dynamically scheduled applications. More sophisticated instrumentation is needed to support this type of analysis. SySM and DLA compute the average execution time of a specific event, but do not provide information about events of variable duration. Two solutions to this problem have been implemented in other experimental instruments. The System Activity Monitor[27] calculates the sum of squares in real time, yielding the variance of the parameters being measured. The Spectron[28] generates histograms of the parameters, rather than averages.

Although SySM can be used to acquire traces of processor activity versus time, the cost in processor overhead and memory utilization is prohibitive. Future instrumentation hardware should migrate this functionality from the processor into the monitoring hardware. In conjunction with this problem, synchronizing multiple SySM time bases in different clusters is difficult. Future versions of SySM should be less cluster chauvinistic, and support system wide communication.

Finally, performance losses are not the only way to characterize system behavior. While loss measurement focuses on the temporal resources of a parallel computing system, alternate measurements could examine system resources such as memory usage. This can be particularly important when studying cache demands in multitasking and virtual memory systems, areas in which the described work does not readily apply.

Acknowledgments

The authors wish to express their appreciation to their many colleagues at Harris and MIT who have contributed to the development of the Concert research environment. In particular, we wish to recognize Steve Mitchell, Maria Laprade, and Marcus Neal for their critical contributions to the design and implementation of DLA and SySM, as well as Ken Laprade and Mike Noakes for their

valuable efforts in helping to apply these tools to parallel processing research. In addition, Juan Loaiza provided significant assistance in helping to understand the MIT Multilisp implementation. Lastly, we wish to thank Ellery Chan for his invaluable assistance in the composition of this paper.

References

[1] Denning, P. J. and J. P. Buzen, *The Operational Analysis of Queuing Network Models*, ACM Computing Surveys 10:3, September, 1978.

[2] Heidelberger, P. and Lavenberg, *Computer Performance Evaluation Methodology*, IEEE Transactions on Computers C-33:12, pp. 1195-1220, December, 1984.

[3] Law, A. and W. Kelton, *Simulation Modeling and Analysis,* McGraw-Hill, 1982.

[4] Swan, R. J., S. H. Fuller and D. P. Siewiorck, *CM* – A Modular, Multi-Microprocessor*, AFIPS 46, NCC 1977, pp. 637-644.

[5] Hillis, W. D., *The Connection Machine*, MIT Press, 1985.

[6] Halstead, R., T. Anderson, R. Osborne, and T. Sterling, *Concert: Design of a Multiprocessor Development System*, 13th Annual Symposium on Computer Architecture, Tokyo, June 1986, p. 40-48.

[7] Sterling, T. L., and Becker, D. J., *System Software Monitor (SySM) Overview*, Harris GSS internal note, October 21, 1985.

[8] Laprade, M., *Degradation from Latency and Arbitration (DLA)*, Harris GSS internal note, June 29, 1987.

[9] Sterling, T. L., A. J. Musciano, E. Y. Chan, D. A. Thomae, *SpoC: An Effective Implementation of a Parallel Language on a Multiprocessor*, IEEE Micro, December, 1987.

[10] Halstead, R., *An Assessment of Multilisp: Lessons From Experience*, International Journal of Parallel Programming 15:6, December, 1986, Plenum Press, New York

[11] Becker, D. J., *A Multiprocessor Emulation of a Static Data Flow Parallel Architecture*, Proceedings of the 25th Annual Southeast Regional Conference, April 1, 1987, pp. 465-472.

[12] Motorola Corporation, *MC68000 16/32-bit Microprocessor Programmer's Reference Manual*, Prentice-Hall, Inc., 1984.

[13] Graham, S. L., P. B. Kessler, M. K. McKusick, *gprof: A Call Graph Execution Profiler*, Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices 17:6, pp. 120-126, June, 1982.

[14] Musciano, A. J., *The Simultaneous Pascal Language Reference Manual*, Harris GSS internal report, September, 1987.

[15] Sterling, T. L., *Parallel Control Flow Mechanisms for Dynamic Scheduling of Tightly Coupled Multiprocessors*, MIT/EECS Ph.D Thesis, May, 1984.

[16] Cooper, Doug, *Standard Pascal User Reference Manual*, W. W. Norton and Company, 1983.

[17] Musciano, A. J., *SpoC Note 24: Incorporating SySM into SpoC*, Harris GSS internal note, June 4, 1987.

[18] Musciano, A. J., *SpoC Note 23: The Results of Locality Exploitation in SpoC*, Harris GSS internal note, May 13, 1987.

[19] Musciano A. J., *SpoC Note 26: Deriving Computing Profiles in SpoC*, Harris GSS internal note, September 10, 1987.

[20] Abelson, H. and G. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA., 1984.

[21] Halstead, R., *Multilisp: A Language for Concurrent Symbolic Computation*, ACM Trans. on Prog. Languages and Systems, October, p. 501-538.

[22] Halstead, R., J. Loaiza, and M. Ma, *The Multilisp Manual*, Parallel Processing Group, MIT Laboratory for Computer Science, June 1986.

[23] Halstead, R., *Parallel Symbolic Computing*, IEEE Computer 19:8, August 1986, p. 35-43.

[24] Halstead, R., *Parallel Computing Using Multilisp*, J. Kowalik, ed., Parallel Computation and Computers for Artificial Intelligence, Kluwer Academic Publishers, 1987.

[25] Rumbaugh, J., *A Data Flow Multiprocessor*, IEEE Transactions on Computers C-26:2, pp. 138-146, February, 1977.

[26] Watson, Ian and John Gurd, *A Practical Data Flow Computer*, Computer, pp. 51-57, February, 1982.

[27] Chan, E. Y., *System Activity Monitor (SAM) Overview*, Harris GSS internal note, October 27, 1985.

[28] Chan, E. Y., *The Spectron*, Harris GSS internal note, September, 1984.

BLOCKING FOR PARALLEL SPARSE LINEAR SYSTEM SOLVERS

Santosh G. Abraham
Dept. of Electrical Engineering and Computer Science
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109

Timothy A. Davis
Center for Supercomputing Research and Development
University of Illinois
104 S. Wright Street
Urbana, IL 61801

ABSTRACT

We consider the parallel solution of sparse systems of linear equations. In such systems, parallelism and communication patterns are dependent on the nature of sparsity in the input matrix system. A new algorithm, Block Solve, in which processors access blocks of rows from shared memory, is described. Experiments were carried out on the eight-processor Alliant FX/8 to determine the effectiveness of various blocking strategies in reducing execution times. An average block size of between four and eight minimized execution times. The Alliant FX/8 was used to emulate the execution of Block Solve on a shared memory multiprocessor with private memories.

1. INTRODUCTION

In shared-memory multiprocessor architectures, communication and synchronization overhead can significantly affect performance. Synchronization overhead is incurred when serialized access to shared variables must be enforced thus resulting in contention on shared lock variables. In shared-memory multiprocessors where each processor has a local memory or private cache [1], access to variables in the shared global memory is slower than accesses to variables in the private cache. For such multiprocessor systems, the *communication delay* is the difference between the delay in a write followed by a read on a shared variable from shared memory and the delay in a write followed by a read on a private variable from local memory. In many algorithms and problems, increasing the task granularity decreases the communication and synchronization overhead. However, allocation of large tasks to processors increases the likelihood of some processors being forced to idle, i.e., worsens load balancing. In problems and algorithms, where communication and parallelism are relatively independent of the input data, analytic models and deterministic schedules are applicable, e.g., FFT algorithm. However, such models are not usually applicable where the communication patterns are not uniform and regular.

In this paper, we consider the solution of sparse systems of linear equations, in which parallelism and communication patterns are dependent on the nature of sparsity in the input matrix system. An asynchronous parallel algorithm is developed to solve the matrix system. Dynamic techniques are used in this algorithm to estimate the current parallelism, i.e., the number of row operations that can be performed simultaneously. The estimate of the current parallelism is used to continuously balance parallelism and communication requirements. When a large amount of parallelism is available, individual processors are assigned large tasks. However, when the parallelism decreases, for instance, towards the end of the computation, the task granularity is decreased to improve the utilization of the processors.

A new algorithm, Block Solve, in which processors access blocks of rows from shared memory, is described. This algorithm is a generalization of Gaussian Elimination with pairwise pivoting, in which processors only access pairs of rows from shared memory. In the Block Solve algorithm, processors access blocks of rows, where the *block size*, (the number of rows brought into local memory) need not necessarily be two as in pairwise pivoting. Various blocking strategies that control the block size, i.e., the number of rows accessed by each processor on each access to shared memory, are described. Blocking is an attempt to parameterize the behavior of the linear system solver with respect to interprocessor communication. The right choice of block size balances communication requirements and parallelism and optimizes performance.

Experiments were carried out on the eight-processor Alliant FX/8 to determine the effectiveness of these blocking strategies in reducing execution times for various linear systems. The execution of the Block Solve algorithm on a multiprocessor system with private caches for each processor and a slower shared global memory is emulated on the Alliant FX/8. The interrelationship between blocking and global communication delay was examined by measuring performance for different types of matrix systems using a range of block sizes and global delays.

Increasing the block size reduces communication and synchronization overhead and thereby reduces completion times. However, a large block size increases idle time of processors. Measurements indicate that a moderate block size of between four and eight balances these conflicting requirements and minimizes

This research is supported in part by DOE grant DE-FG02-85ER25001, NSF grants DCR-84-10110 and DCR-85-09970, a donation from IBM, and by an American Electronics Association fellowship for Mr. Davis (with funds from Digital Equipment Corporation).

execution times on the Alliant FX/8 for a range of matrix systems. In a multiprocessor system with a shared memory and local memories for each processor, the effect on performance of large shared global memory delays relative to local memory delays is reduced if appropriate block sizes are chosen through the blocking strategies presented in this paper.

The number of processors in shared-memory multiprocessor systems continues to increase. In future systems, the shared global memory will be much slower than the private local memories of individual processors. The algorithm described here attempts to reduce the number of accesses to the relatively slow global memory and therefore is likely to be superior to existing direct solvers for such multiprocessor systems. Furthermore, the technique of modifying task granularity based on a current parallelism estimate, may be applicable to other important numerical algorithms.

In Section 1, the problem domain and the algorithm that was implemented are described. In Section 2, the tradeoff between synchronization and computation is examined. In Section 3, the results obtained by varying the block size in a parallel linear system solver running on the Alliant FX/8 are presented. In Section 4, the Alliant FX/8 is used to emulate a machine with private memories and a single shared global memory. In each of the above two sections, details of the machine and the algorithm are followed by the results of experiments performed by running the algorithm.

2. THE SOLUTION OF SPARSE LINEAR SYSTEMS

The problem domain is the solution of a sparse system of n simultaneous linear equations, represented as $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a (possibly unsymmetric) sparse $N{\times}N$ coefficient matrix, and $\mathbf{x}$ and $\mathbf{b}$ are N-vectors. $\mathbf{A}$ and $\mathbf{b}$ are known and it is necessary to determine the N-vector $\mathbf{x}$. In a general sparse system, there are relatively few nonzero elements in A, but the distribution of the nonzero elements does not fall into any regular pattern. A large number of computationally expensive scientific and engineering applications, e.g., structural analysis, fluid dynamics, aerodynamics, computer-aided design, and circuit simulation, are based on the solution of large sparse systems of linear equations [2]. It is therefore important to develop good parallel algorithms for solving sparse linear systems.

LU decomposition is a direct method for solving linear systems [3]. It involves a forward reduction phase that obtains lower and upper triangular matrices, $\mathbf{L}$ and $\mathbf{U}$, where $\mathbf{A} = \mathbf{LU}$, and a back substitution phase to get the solution vector $\mathbf{x}$. Only performance of the forward reduction phase is analyzed in this paper since it is computationally more expensive than the back substitution phase. Parallel algorithms for solving $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ when $\mathbf{A}$ is dense (i.e., when most coefficients are non-zero) employ schedules where the actions of each of the processors are predetermined before run time [4]. These algorithms are not efficient for general sparse systems.

2.1. Pairwise Solve

Pairwise Solve, or PSolve, is an asynchronous, nondeterministic, parallel algorithm based on pairwise pivoting [5, 6]. Consider two rows of the A matrix whose leading (or leftmost) nonzero elements lie in the same column. If one row (called the *pivot row*) is multiplied by an appropriate factor and added to the second row, the leading nonzero of the second row can be reduced to zero, thus simplifying the equation corresponding to the reduced row. In *pairwise pivoting* then, elementary 2×2 stabilized eliminators S are constructed and the pair of rows is premultiplied by S to create a zero (Figure 1) [7]. The column index of the leading nonzero element in a row is referred to as the *column index* of the row.

The algorithm uses the data structures shown in Figure 2 to detect parallelism efficiently. A *column list*, col_j, associated with

$$\begin{bmatrix}1 & 0\\ \alpha & 1\end{bmatrix}\begin{bmatrix}a_{i1} & \cdots & a_{in}\\ a_{j1} & \cdots & a_{jn}\end{bmatrix}=\begin{bmatrix}a_{i1} & \cdots & a_{in}\\ 0 & \cdots & \tilde{a}_{jn}\end{bmatrix}$$

$$\alpha=-a_{j1}/a_{i1}$$

$$\begin{bmatrix}c & s\\ -s & c\end{bmatrix}\begin{bmatrix}a_{i1} & \cdots & a_{in}\\ a_{j1} & \cdots & a_{jn}\end{bmatrix}=\begin{bmatrix}\tilde{a}_{i1} & \tilde{a}_{i2} & \cdots & \tilde{a}_{in}\\ 0 & \tilde{a}_{j2} & \cdots & \tilde{a}_{jn}\end{bmatrix}$$

$$c=\frac{a_{i1}}{\sqrt{a_{i1}^2+a_{j1}^2}}$$

$$s=\frac{a_{j1}}{\sqrt{a_{i1}^2+a_{j1}^2}}$$

Fig. 1. Pairwise reduction and Givens' reduction

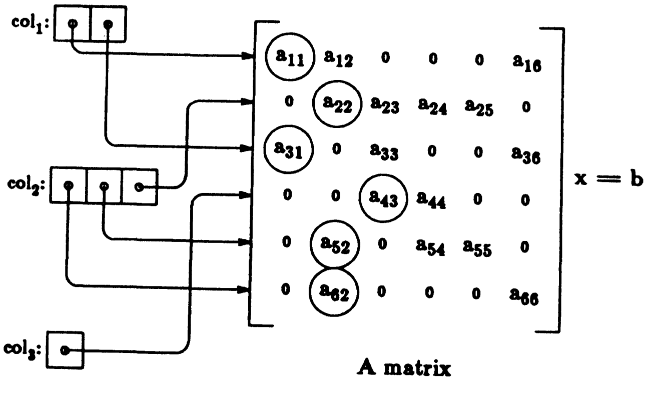


Fig. 2. Sparse matrix data structure

each column j, is a list of rows that have a *column index* of j. Rows with fewer nonzeros are kept to the front of the column lists to reduce fill-in, i.e., the extent to which zero-valued elements in the original A matrix are converted to nonzero elements by the reduction process. Only the nonzero values of the rows are actually stored and operated on.

To find work, a processor scans the column lists for a list with two or more rows. It removes the first two rows in the list, reads the pivot row and returns it to the original list unchanged, and reduces the second row, thereby increasing its column index by at least one. The second row is put in a new column list corresponding to its new column index. The algorithm completes when the A matrix is upper triangular and each column list contains exactly one row. The advantages of PSolve have been demonstrated by measuring execution times on the Alliant FX/8 for 38 test matrices from the Harwell/Boeing sparse matrix collection [6].

2.2. Characterization of parallelism in Psolve

The amount of parallelism, i.e., the number of row reductions that can be executed simultaneously in this context, is a function of the input matrix structure, specifically the pattern of zero and nonzero elements. Parallelism also varies during the execution of the algorithm. The amount of parallelism determines the number of processors that can be used effectively and the following attempts to characterize the parallelism in Pairwise Solve.

Let n_j be the number of rows in column list j. Let N_1 and N_z be the number of column lists with n_j equal to one and zero, respectively. Assume that, during each row reduction, only the row being reduced is locked for exclusive access by a processor and a pivot row is simply read and released. An upper bound on the maximum number of simultaneous row reductions and hence the currently available parallelism is $N - N_1 - 1$, since the N_1 rows belonging to the N_1 column lists with n_j=1 cannot take part in row reductions.

Lemma 1. The pairwise algorithm terminates within $N(N-1)/2$ reduction steps, if the $N \times N$ matrix, A, is nonsingular.

Each reduction step creates a new zero below the diagonal. Such zeros are not converted back to nonzero elements. Since the initial maximum number of nonzero elements below the diagonal is $N(N-1)/2$, the pairwise algorithm terminates within $N(N-1)/2$ reduction steps. □

Lemma 2. The currently available parallelism is N_z.

Proof: Column lists, j, with $n_j \geq 2$ contain rows on which further reductions can be currently performed. In such columns lists, n_j-1 rows can be simultaneously reduced by n_j-1 processors using the remaining n_jth row. Thus, the parallelism for Pairwise Solve, P_2, is

$$P_2 = \sum_{\substack{j=1\\ n_j>0}}^{N} (n_j-1) \tag{1}$$

Since each of the N rows is associated with exactly one column list, $\sum_{j=1}^{N} n_j = N$. Expanding (1) gives

$$P_2 = \sum_{j=1}^{N}(n_j-1) - \sum_{\substack{j=1\\ n_j=0}}^{N}(n_j-1)$$

$$= \sum_{j=1}^{N} n_j - \sum_{i=1}^{N} 1 + \sum_{\substack{j=1\\ n_j=0}}^{N} 1 = N_z \tag{2}$$

Thus, the parallelism, P_2, is N_z, the number of empty column lists. □

Lemma 3. The currently available parallelism, P_k, is a monotonically nonincreasing function of time during the execution of the algorithm.

Proof: No row reduction can eliminate the last row from a column list. Hence, once a column list, j, has $n_j \geq 1$, it will continue to have $n_j > 0$. Therefore, N_z and hence P_2 are monotonically nonincreasing. □

2.3. Block Solve

Block Solve is a new algorithm developed in this paper to explore blocking. It is based on the Pairwise Solve algorithm. In Block Solve, a processor accesses a block of k rows from a set of contiguous column lists. The row containing the leading element that has the largest absolute value among the rows with the least column index is not reduced. All other rows may be reduced and are locked for exclusive access. Pairwise row reduction steps are performed on rows within the block until no two rows share the same column index. At this stage, no further row reductions can be performed locally within the block. The processor releases the block of rows and accesses a fresh set of rows. This is a generalization of Pairwise Solve in the sense that Block Solve with a block size of two is identical to Pairwise Solve, where *block size*, k, is the number of rows accessed. The available parallelism with a block size of k, P_k, is the number of size k blocks that can be reduced simultaneously. Since each processor reserves $(k-1)$ rows for exclusive access, the maximum parallelism in Block Solve is $(N - 1)/(k - 1)$, as compared to $N-1$ in Pairwise Solve. As the block size, k, increases, the parallelism decreases. If the number of processors, p, is constant, some processors must idle when P_k falls below p.

Despite this drawback, a block size, k, greater than two results in better performance when interprocessor communication is expensive. In a multiprocessor system with private caches and a shared global memory, a block size of k>2 reduces the communication time of a processor, i.e., the time spent accessing data from global memory. In the ideal case, the communication time is reduced by a factor of $(k-1)/2$ as indicated in the following.

A row i is *k–solid* if the $k-1$ elements following the leading nonzero element are also nonzero, i.e., $\{a_{i,1}, \cdots a_{i,j-1}\}$ are zero and $\{a_{i,j}, \cdots a_{i,j+k-1}\}$ are nonzero. A block of k rows is *solid* if all rows are k-solid and have the same column index. Since $k(k-1)/2$ elements in the block must be reduced to zero before releasing the block and since each row reduction step reduces one element to zero, a processor performs $k(k-1)/2$ reductions after accessing a solid block of k rows. Therefore, the compute-to-communication ratio is $(k-1)/2$ in units of "row reductions per globally accessed row." In rare instances, a row reduction step may reduce two or more elements to zero, and consequently, a processor may perform fewer than $k(k-1)/2$ reductions. In such cases, the compute-to-communication ratio approaches, but is less than, $(k-1)/2$.

A solid block represents the ideal case. In general, a processor may not find k k-solid rows with the same column index and, in this case, the compute-to-communication ratio is less than $(k-1)/2$. Each processor scans successive columns for a column list, j, with $n_j \geq 2$. The first set of rows is dequeued from this column list. This criterion is used because if the first row is from a column list with n_j=1, then this row cannot be used in any row reduction step. A processor then scans additional columns, if needed, until it acquires a total of k rows. Single rows may be acquired from succeeding column lists, i.e., rows from column lists j with n_j=1, because row reductions can usually be performed on such rows. For instance, if two rows are acquired from column 5 and a single row is acquired from column 6, a row reduction involving the two rows from column 5 usually results in a row with a leading nonzero element in column 6. A row reduction step is then performed using the row from column 6 and the reduced row obtained from the first row reduction.

The following *early-quit* criterion may terminate a scan before k rows are acquired. A scan is obviously terminated once column N has been scanned. The scan is also abandoned when the difference between the current column and the column from which the first row was acquired is greater than the number of rows already acquired. This early-quit criterion abandons a scan if it is expected that acquiring an additional row will not reduce communication time. If this condition holds, rows from the current column list cannot be reduced using the rows already acquired. Therefore, accessing additional rows reduces parallelism but does not reduce communication time. For instance, if a processor acquires the first two rows from column five, reserves a total of four rows, and is scanning column nine, then the scan is ended and the processor attempts to reduce the four rows acquired. Acquiring additional rows from column nine does not reduce communication, since the four rows acquired cannot be used to reduce rows acquired from column nine. However, acquiring additional rows would reduce the number of rows available to the remaining processors.

2.4. Givens' reduction

Another basic difference between the Pairwise Solve algorithm [5] and the Block Solve algorithm presented here is in the strategy used to create zeros to transform the matrix A into upper triangular form. While pairwise pivoting was used in PSolve, Givens' reduction is used here. In *Givens' reduction*, the premultiplication is performed by the 2 × 2 matrix shown in Figure 1. The computation required in Givens' reduction to create a zero is twice that in pairwise pivoting. Furthermore, in a parallel implementation, both rows involved in a row reduction step must be locked, as both rows are modified. Since the earlier results on parallelism assumed that only one of the rows is locked, those results must be modified appropriately. The major advantage of Givens' reduction is its increased numerical stability.

2.5. Matrix systems

In this paper, results obtained on two types of sparse matrices are presented. *Synthetic sparse* matrices are parameterized by the *size* of the matrix and *width*, *ewidth* and *scatter*. An element, a_{ij}, is nonzero if $|i-j| \le width$ and has a probability, *scatter*, of being nonzero if $width < |i-j| \le ewidth$; it is always zero if $|i-j| > ewidth$. Two types of synthetic sparse matrices were chosen with the parameters, (*width, ewidth, scatter*) chosen to be (3, 60, 0.01) and (10, 100, 0.01), respectively. Six *real sparse* matrices were chosen from the Harwell/Boeing sparse matrix collection [8].

3. BLOCKING ON THE ALLIANT FX/8

Execution times were determined on the Alliant FX/8 for a range of matrices and block sizes. The Alliant FX/8 is a shared-memory multiprocessor in which all eight processors share a common cache. Therefore, access to shared and nonshared variables takes the same time, and communication delay is zero. However, a large block size reduces the number of mutually exclusive accesses to column lists. As a result, a large block size reduces the degree of contention for shared variables and consequently the synchronization time.

The PSolve algorithm was observed to have a speedup of between five and seven on an eight-processor system for a wide range of matrices from the Harwell/Boeing collection. The Block Solve algorithm improves on this speedup by varying block size. Since the PSolve algorithm achieves appreciable speedup, the improvement that can be obtained using Block Solve on an eight-processor system is limited. However, in larger multiprocessor systems, the degree of contention and hence the synchronization overhead increases. In such systems, the blocking techniques in Block Solve that reduce synchronization overhead are more effective.

3.1. Constant and variable blocking

The *maximum block size* is the maximum number of rows that a processor can acquire on each access. If one of the early-quit criteria is applicable, a processor may terminate a block access and perform a block reduction on a block size smaller than the maximum block size. In *constant blocking*, the maximum block size is fixed throughout program execution. In *variable blocking*, the maximum block size is based on an estimate of the currently available parallelism. Variable blocking is superior because parallelism decreases during program execution (Lemma 3) and the choice of block size should attempt to rebalance communication and parallelism continuously in order to optimize performance.

Two strategies for obtaining an estimate of the available parallelism are described in the following. In the C_1 method, a count, c_1, of the number of elements in the set, C_1, of consecutive column lists $\{1 \cdots c_1\}$ with n_j=1 is maintained. Since no reductions can be performed on the set of rows with column indices in the set C_1, $(N-c_1)$ is an estimate (rather optimistic) of the available parallelism. In the N_z method, a count of the number of columns with n_j=0, N_z is maintained. The implementation of this method uses the fact that once n_j has a nonzero value it will never be zero again. Before program execution, N_z is initialized to its correct value. When a reduced row is returned to shared storage by a processor, N_z is decremented if no other row has the same column index.

The N_z method of estimating parallelism is more accurate than the C_1 method. For instance, consider the solution of a tridiagonal system using the Block Solve algorithm. In a *tridiagonal matrix*, element a_{ij} is nonzero if and only if $|i-j| \le 1$. Exactly one row reduction can be performed at each step and the parallelism is one. However, the amount of parallelism indicated by the first method is $N-c_1$, where c_1 is initialized to zero and is incremented after each reduction. The N_z method accurately indicates a parallelism of exactly one throughout the reduction phase, since only one column, column N, has n_j=0. When the last reduced row is returned, n_N=1, the available parallelism is zero and the reduction phase is complete. However, the C_1 method is sufficiently accurate for dense matrices. Furthermore, the count, c_1, is maintained by one processor. In contrast, since the count, N_z, is decremented by all processors, contention on the corresponding variable can degrade performance in a large system with several processors.

In a variable blocking scheme, an estimate of the parallelism is multiplied by some factor in order to obtain the current maximum block size. Since the C_1 method overestimates parallelism, this factor compensates in part for the overestimation. Even in the accurate N_z method, performance may be improved by varying the multiplicative factor. Two methods of specifying the multiplicative factor are outlined. The first specifies the maximum block size, $k_{\max}$, when there is full parallelism. Thus, the current maximum block size is chosen to be $max(2,k_{\max}.(N-c_1)/N)$ or $max(2,k_{\max}.(N_z/N))$. The choice of $k_{\max}$ should be based on the ratio of actual to estimated parallelism and on the number of processors in the system. The second method specifies blocking independently of the number of processors as a parallelism factor. Given a parallelism factor, p_{fac}, the block size is chosen to be $max(2,(N-c_1)/(p.p_{fac}))$ or $max(2,N_z/(p.p_{fac}))$. The $k_{\max}$ method limits the maximum number of rows that are brought into local memory. This restriction is useful, for instance, if the size of a processor's local memory poses a limitation on block size. In the p_{fac} method, the specification of the parameter p_{fac} is independent of the number of processors. Thus, the optimum p_{fac} on a particular multiprocessor system might be expected to give good performance on a system with a different number of processors.

3.2. Data structures in the implementation of Block Solve

The real-valued arrays a and b contain the nonzero array elements of **A** and the vector **b**. The $N \times N$ array c contains the column index of the corresponding element in a. For instance, if c(i,j) contains k, then the element a(i,j) is the value of A(i,k). Thus, only the nonzero elements of **A** are stored explicitly. The integer array e of length N contains the number of nonzero elements in each row of the matrix. On each row access, the corresponding element in the array e is used to determine the number of elements that should be accessed from a and c. In addition, the array row_next of length N is used to maintain column lists. Since a row belongs to one column list at any instant, all the N column lists are maintained using row_next. Each column list has a head pointer that points to the first row in the list. The row_next pointer of the first row points to the next row in the list and so on. The row_next pointer of the last row in the list is set to NULL.

The data structures also maintain column-based information. A column is locked by a test-and-set instruction on the corresponding element in col_lock. The jth entry in the array col_elements contains n_j, the number of rows with column indices equal to j, minus any rows currently reserved from column list j by processors. The entries in the array col_missing contain the number of rows reserved for exclusive access by processors. During a block reduction, when no further reductions can be performed with a row, it is transferred back to shared storage and the corresponding col_elements entry is incremented. At the end of a block reduction, a processor subtracts its contribution from the array col_missing. The jth element in the pointer array col_head points to the first row in the list of rows with column index j.

The following data structures are used in Block Solve but not in PSolve. A scalar integer head_col that points to the first column with $n_j>1$ is maintained. The value of head_col is equal to (c_1+1) and hence head_col is useful in obtaining an estimate of the current parallelism in the C_1 method. Since columns j with $j \leq c_1$ have $n_j = 1$, no further reductions are possible on rows contained in columns $\{1, \cdots, c_1\}$. Therefore, processors use head_col to skip over the first c_1 columns. Thirdly, when head_col is N, the matrix is in upper triangular form and the reduction phase is complete. Therefore, the processors check head_col and terminate execution when head_col is equal to N.

The head_col pointer is maintained to improve performance; an up-to-date value of head_col is not necessary for the correctness of the algorithm. The head_col pointer is initialized to 1, and thereafter, in order to simplify implementation and reduce synchronization, only processor 1 advances head_col. To the other processors, head_col is a read-only variable. Processor 1 increments head_col if, during a block access, the column indexed by head_col has col_elements equal to 1 and col_missing equal to 0.

Lemma 4. The procedure for advancing head_col guarantees that all columns j with $j <$ head_col always have n_j=1.

Proof: Reduction steps performed by the processors never decrease the column index of a row. Therefore, if a row is read with column index l and is written back with a column index m, then $m \geq l$. When head_col is incremented, the column previously indexed by head_col has only one row associated with it. Ensuring that col_missing is zero guarantees that rows accessed from these columns have been returned before head_col is advanced. Rows accessed from column lists j with $j \geq$ head_col will be returned to column lists with indices no less than head_col. Therefore, all column lists j with $j <$ head_col will continue to contain exactly one row. □

3.3. Implementation of the algorithm

A processor scans column lists beginning from head_col. It precedes accesses to column data structures by a lock operation on the corresponding element of col_lock and releases the lock on completing the data access. On encountering a column list with col_elements greater than 1, the processor dequeues all rows up to a maximum of the current maximum block size. If the number of rows acquired is less than the maximum block size, the processor scans additional columns accessing one or more rows. A scan may be terminated by the early-quit criteria.

Once a set of rows has been reserved exclusively for a processor, Givens' reduction is used to reduce the set of rows until no two rows share the same column index. A local list of rows in the current block is maintained in sorted order using the respective column indices as the first key and the number of nonzero entries in a row as the second key. The second key is chosen to reduce fill-in. If the two rows at the head of the local list have different column indices, the row at the head is written back to shared storage since it cannot be used for performing further reductions. Otherwise, the rows are dequeued, one of the rows is reduced and the two rows are inserted into their correct positions in the local list. Ultimately, when there is just one row left, it too is written into shared storage. At this stage, since all rows have been returned, the col_missing array is updated. The above is repeated until head_col is advanced to N.

3.4. Measurements and analysis

This section describes, presents and analyzes the measurements that were taken to explore blocking on the Alliant FX/8. The four blocking schemes that were implemented and the current maximum block size, max_blk, for each of the four schemes are

(1) const_blk blocking uses constant blocking where the maximum block size is fixed throughout program execution and is one of the command-line parameters.

(2) hd_var blocking uses variable blocking where an estimate of the parallelism from the head_col variable is scaled by a command-line parameter, $k_{\max}$, which is the maximum blocking size with full parallelism.

$$max\_blk = \max \left[2, \frac{k_{\max} \cdot ((N+1) - head\_col)}{N} \right] \quad (3)$$

(3) pf_var blocking is similar to hd_var except that the parallelism estimate is scaled by the command-line parameter, p_{fac}, and the number of processors in the system.

$$max\_blk = \max \left[2, \frac{((N+1) - head\_col)}{p_{fac} \cdot p \cdot N} \right] \quad (4)$$

(4) nz_var blocking is similar to pf_var but uses the number of columns with n_j=0, N_z, to estimate parallelism. The same value of the command-line parameter, $p_{fac,nz}$, might be expected to give similar performance for a range of matrices, since the program dynamically adjusts to the available parallelism.

$$max\_blk = \max \left[2, \frac{N_z}{p_{fac,nz} \cdot p} \right] \quad (5)$$

Experiments were conducted by running Block Solve on the Alliant FX/8 using various maximum block sizes and parallelism factors, for each of the matrices and blocking schemes described above. In each case, the execution time, and the average block size were measured and tabulated. The *average block size* is the average number of rows that a processor acquires on each access to the global queue. This number is never greater than the maximum specified indirectly through the command-line parameters, $k_{\max}$ or p_{fac}. Other measurements that were carried out for each run of the program include the error norm, the density of nonzeros in the upper triangular matrix, the number of floating-point operations, the number of row reductions, the number of accesses, the number of columns scanned on each access (*skips*), and the megaflops rating attained.

Each of the four different blocking schemes that were implemented restricts the number of rows that a processor can acquire on each access. This restriction implicitly determines the average number of rows that a processor acquires on each access and thereby affects performance. For each blocking scheme, different ranges of the command-line parameters exhibit near-optimal performance. The execution times are therefore plotted versus the average block size, rather than versus the value of the command-line parameter.

In Figure 3, the execution times are plotted versus the average block size for each blocking scheme for a type2 matrix of size 1000. The points correspond to observed values of execution time and average block size, for each value of $k_{\max}$ (or p_{fac}). A line joins each point to the point corresponding to the next higher value of $k_{\max}$. A higher value of $k_{\max}$ does not necessarily result in a higher average block size because there may be fewer rows for the other processors if one processor accesses a larger block. As a result, the lines occasionally move to the left. In general, the execution time decreases slightly with increasing block size for small block sizes, but then increases for larger block sizes. A block size of four, in general, gives better performance than a block size of two. Command-line parameter choices that result in average block sizes of between three and four are optimal for the variable blocking schemes, whereas a maximum block size specification of eight (with resulting average block size of 7.4) is optimum for the const blocking scheme. The more sophisticated blocking schemes are more robust in the sense that they give near-optimal performance over a wider range of the command-line parameters, $k_{\max}$ and p_{fac}. In particular, the execution time with nz_var blocking remains close to the minimum of 10.50 seconds for any choice of the command-line parameter, p_{fac}, in the range [0.3-6.0], whereas the execution time with const blocking remains

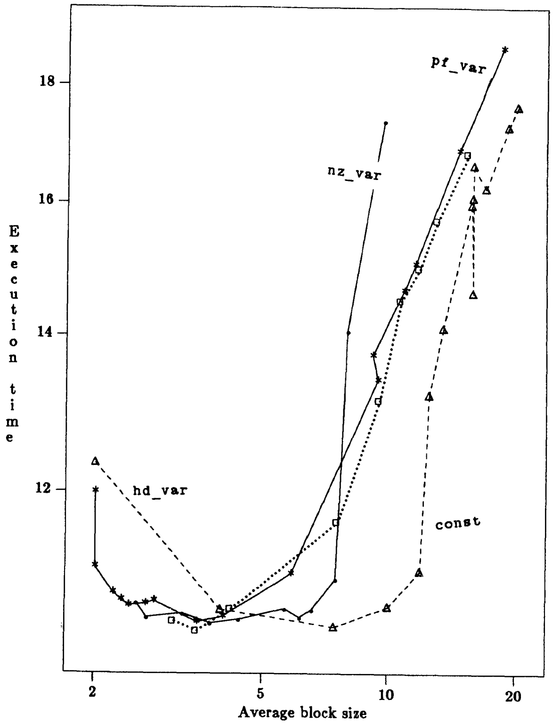


Fig. 3. Execution time (seconds) vs. average block size (for various blocking schemes on a type2 size 1000 matrix)

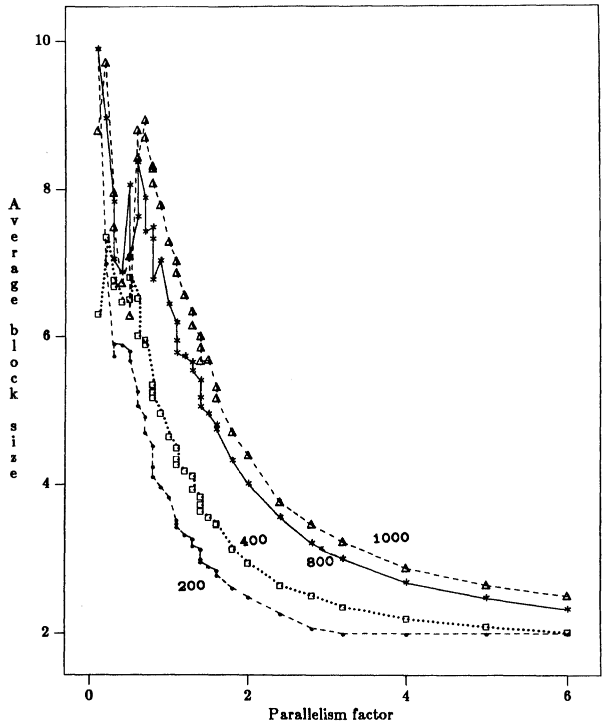


Fig. 4. Average block size achieved vs. specified parallelism factor, p_{fac} (for type2 matrices with sizes 200, 400, 800, 1000)

close to the minimum only for a choice of the command-line parameter, maximum block size, in the range [4-10].

For the rest of this section, runs using nz_var blocking on type2 matrices are examined further. Figure 4 shows the relationship between the command-line parameter, p_{fac}, and the resulting average block size. When $p_{fac}>1$, the number of rows available for reduction exceeds the sum of the sizes of blocks in individual processors. The sizes of the acquired blocks are indeed limited by the maximum block size. Since the maximum block size varies inversely as p_{fac}, the average block size also varies inversely as p_{fac}. However, when $p_{fac}<1$, the block sizes are limited by the parallelism in the matrix system. Thus, each processor attempts to acquire blocks of size greater than N_z/p, but since there are only N_z rows available, the processors, in general, acquire fewer rows than specified by the maximum block size. Therefore, for $p_{fac}<1$, the average block size is only weakly dependent on p_{fac}.

In this section, various blocking schemes were analyzed. Blocking can improve performance to some extent even in multiprocessor systems that have a shared cache and effectively zero communication delay because of a reduction in synchronization overhead. An average block size between three and four leads to minimum execution times for variable blocking schemes. Execution time curves are fairly flat over block size ranges near their optimum value. The performance of all four blocking schemes with the corresponding optimal choices of command-line parameters is comparable. However, a choice of maximum block size outside the near-optimum range can greatly increase execution time for the const blocking scheme. In contrast, the sophisticated blocking schemes, pf_var and nz_var, are more robust because a nonoptimal choice of p_{fac} does not significantly affect performance.

4. EMULATION OF GLOBAL DELAYS

An eight-processor system where each processor has its own private memory and all the processors share a common global memory was emulated using the Alliant FX/8. The emulation was accomplished by identifying shared variables in the program and inserting additional accesses for each shared variable access to model the shared memory access time. The interrelationship between blocking and global communication delay was examined by measuring performance for different types of systems using a range of parallelism factors and global delays.

4.1. Global delays

In order to emulate a shared global memory multiprocessor with private memories for each processor, occurrences of variables shared between processors are identified. In the linear system application, the following are shared and therefore in global memory: a, b, c, e, row_next, col_elements, col_lock, col_missing. Thus, the entire sparse matrix data structure as represented in Figure 2 resides in global memory. A block of rows is accessed from global memory and stored in local memory. Subsequent accesses to a row contained in the block are satisfied by the local memory.

To emulate the additional time that it would take to complete accesses on shared variables from global memory, *g_dly - 1* additional accesses are made to locations in an otherwise unused array where *g_dly* is the factor by which the global memory is slower than local memory. Additional vector and scalar accesses are introduced for vector and scalar accesses respectively in the original program. Provided that all accesses on the host multiprocessor on which the emulation runs (the Alliant FX/8, in this case) take identical time, this strategy should emulate the global delay accurately.

However, the Alliant FX/8 has a hierarchical memory structure with a virtual memory and a cache which pose special problems in performance evaluation [9]. Therefore, the additional accesses used to emulate global delay were structured so that they do not benefit from the cache prefetching effect any more than the normal accesses (i.e., they have similar spatial locality). For sufficiently large problems, cache hits are primarily due to prefetching an entire line (spatial locality). When the problem size is large, cached data tends to be replaced before it is reused. Therefore, the temporal locality is small and does not contribute significantly to cache hit ratio. With care taken to preserve the amount of spatial locality, the cache miss ratio is expected to approximate the miss ratio in the original program. Furthermore, the size of the dummy arrays used to emulate global accesses was chosen to be an order of magnitude smaller than the largest array in the program. This reduced the likelihood of thrashing on the disk. Thus, the paging activity in the emulation is expected to approximate that in the original program. Further study is necessary to verify that these steps are sufficient for accurate emulation.

4.2. Measurements and analysis

The program was run on synthetic and real sparse systems. For each matrix system, the program was executed for choices of p_{fac} in the range 0.1 through 5.0 and global delays of 1 through 10 times the private memory access delays.

In Figure 5, the execution times for the matrices, bp_1000 from the Harwell-Boeing collection, synthetic sparse matrix of size 1000 of type2, and synthetic sparse matrix of size 800 of type1, are plotted against parallelism factor, p_{fac}. The multiprocessor system that is emulated has a global memory that is five times slower than the individual local memories. A parallelism factor slightly less than one is found to give the minimum execution time for the synthetic sparse matrix systems, whereas a parallelism factor around 0.5 is ideal for the bp_1000 matrix. Because of the early-quit criteria, the average block size is less than the maximum block size, even if sufficient parallelism is present. A parallelism factor of 0.5, therefore, does not imply that only half the processors are busy.

The execution times for fs_541_2 are examined in Figure 6. The global delay is chosen to be 3, 5, and 9 times the local delay and execution times are plotted against the parallelism factor. Parallelism factors at either ends of the range result in large execution times, either because of a lack of parallelism or because of large communication delays. Furthermore, in the optimum range of p_{fac}, the effect of global communication delays is masked and execution times for global delays of 3, 5, and 9 are nearly identical. This masking occurs because although changes in global delays do affect communication time, communication time is not a significant part of execution time in the optimum range of p_{fac}. A choice of parallelism factor around 0.5 minimizes execution times for all three global delays. An average block size of approximately 13 is achieved with p_{fac}=0.5 which implies that on the average only about one-fifth of the 541 rows in fs_541_2 are typically reserved by processors for exclusive access.

5. CONCLUSIONS

This paper examined blocking in the context of linear system solvers. A large block size decreases communication and synchronization at the expense of reduced parallelism. Four different implementations of blocking were examined by solving various linear systems on the Alliant FX/8. For the type2-1000 matrix running on eight processors, minimum execution times were obtained with an average block size of approximately four for variable blocking schemes and an average block size of 7.4 for the const blocking scheme. The more sophisticated blocking schemes that adjust the block size to match the currently available parallelism give good performance even if the user-specified parameters do not match the problem and the multiprocessor.

The emulation of the execution of the Block Solve algorithm on a multiprocessor system with global memory that is slower than the private memories of the processors was described. Blocking is more useful in such a system because the communication delays (as well as synchronization overhead) are reduced with larger block sizes. Parallelism factors are used to define task granularity for multitasking large problems; very small or very large factors result in large execution times either because of a lack of parallelism or because of large communication delays, respectively. A parallelism factor of approximately 0.5 minimizes execution time for a range of global delays. (Note that because of the early-quit criteria, a parallelism factor less than one does not necessarily imply that some processors are idle.) Furthermore, as global delay is increased, the execution time does not increase rapidly with this choice of parallelism factor. With an average block size of 13, communication time is not significant compared to computation time; hence, execution time is not affected significantly by changes in communication delay.

This paper presents a new algorithm, Block Solve, for solving sparse systems of linear equations that is a generalization of the Psolve algorithm discussed in [6]. For most test matrices, the Psolve algorithm was found to run faster on the Alliant FX/8 multiprocessor than Gaussian Elimination which does not exploit sparsity and the Yale Sparse Matrix Package which does not exploit parallelism. In this paper, the performance of Block Solve with moderate block sizes is found to be superior to a block size of two, which is required by Psolve. The algorithm presented here is likely to find application in shared-memory multiprocessors that have private local memories that are significantly faster than the shared global memory. The Block Solve algorithm also introduces techniques to estimate parallelism during program execution and continuously balance communication requirements and parallelism based on the parallelism estimate. These techniques may be useful in other asynchronous algorithms.

REFERENCES

[1] G. F. Pfister, W. C. Brantley, D. A. George, S. I. Harvey, W. J. Kleineder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture," *Proc. Int. Conf. Parallel Processing*, pp. 764-771, 1985.

[2] H. F. Jordan, "Experience with pipelined multiple instruction streams," *Proc. IEEE*, vol. 72, pp. 113-123, 1984.

[3] G. Dahlquist and A. Bjork, *Numerical Methods*. London: Prentice Hall, 1974.

[4] Y. Saad, "Communication complexity of the Gaussian Elimination algorithm on multiprocessors," Dept. of Computer Science, Yale University, Research Report No. YALEU/DCS/RR-348, pp. 1-18, July 1985.

[5] T. A. Davis, "Psolve: A concurrent algorithm for solving sparse systems of linear equations," Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, CSRD Report No. 612, pp. 1-56, Dec. 1986.

[6] T. A. Davis and E. S. Davidson, "PSOLVE: A concurrent algorithm for solving sparse systems of linear equations," *Proc. Int. Conf. Parallel Processing*, pp. 483-490, 1987.

[7] D. C. Sorensen, "Analysis of pairwise pivoting in Gaussian Elimination," *IEEE Trans. Comput.*, vol. C-34, pp. 274-278, Mar. 1985.

[8] Harwell/Boeing Computer Services Sparse Matrix Test Collection.

[9] W. Jalby and U. Meier, "Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system," *Proc. Int. Conf. Parallel Processing*, pp. 429-432, 1986.

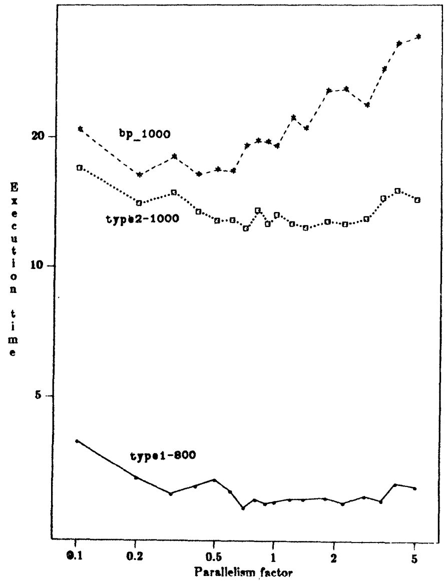


Fig. 5. Emulated execution time (seconds) vs. parallelism factor (for various matrices on an eight-processor system with global delay five times local delay)

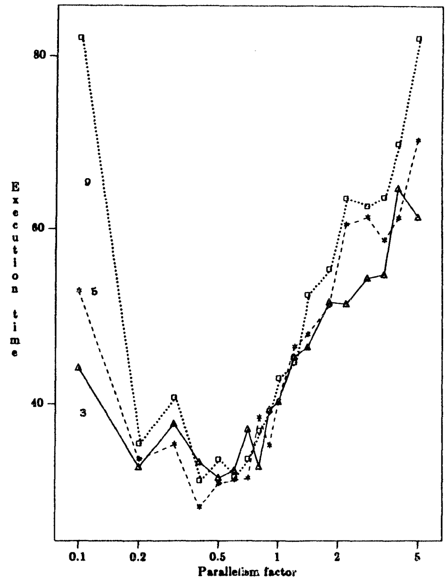


Fig. 6. Emulated execution time (seconds) vs. parallelism factor (for fs_541_2 matrix on a eight processor system with global delays of 3, 5, 9 times local delay)

PERFORMANCE ANALYSIS OF A SHARED MEMORY MULTIPROCESSOR: CASE STUDY

R. T. Dimpsey and R. K. Iyer
Coordinated Science Laboratory, Computer Systems Group
University of Illinios at Urbana-Champaign
1101 W. Springfield Ave.
Urbana, Illinois, 61801

Abstract -- This paper presents an analysis of an Alliant FX/8 system running Xylem (Cedar's operating system) at the University of Illinois Center for Supercomputing Research and Development. Results for two distinct, real, scientific workloads executing on an Alliant FX/8 are discussed. A combination of user concurrency and system overhead measurements were taken for both workloads. Statistical cluster analysis is used to extract a state transition model to jointly characterize user concurrency and system overhead. A skewness factor, is introduced and used to bring out the effects of unbalanced clustering when determining states with significant transitions.

1. INTRODUCTION

The evaluation of a parallel processor often consists of determining numerical performance indices, such as MFLOPS, for the machine using standard benchmarks. Although these indices are useful in detecting global weaknesses of the system, they are unable to provide detailed insight into system behavior.. It is important to have methods which provide information about the system's performance under a certain workload, along with insight into how the workload and system interact. With such methods, the system can be more easily tuned for specific applications and vice versa.

This paper presents an analysis of an Alliant FX/8 system running the Cedar[1] operating system, Xylem, at the University of Illinois Center for Supercomputing Research and Development (CSRD). Results for two distinct, real, scientific workload samples executing on an Alliant FX/8 are presented. In the analysis, a combination of user concurrency and system overhead measurements are employed. Statistical clustering is performed on these measurements to identify commonly recurring patterns of resource usage. State transition models are extracted and interpreted for both sampled workloads to obtain practical insight into the system behavior. Skewness factors are then calculated for each interstate transition in the identified model and used to determine significant transitional relationships among the states of the machine.

The results show that during the collection of the first sample, the system was operating in states of high user concurrency approximately 79% of the time. The second sample, on the other hand, captures a system operating in states of high user concurrency only 26% of the time. In addition, the analysis shows that high system overhead is usually accompanied by low user concurrency. The analysis also indicates that for both workloads, the state of the system was highly predictable. This predictability was largely due to slow changes in system states. In particular, states with extremely high values of paging or user concurrency are usually preceded by states with less paging and user concurrency, much like stair climbing. A stepping down effect is observed when the machine leaves these extreme states.

1.1 Related Research

There have been several studies which analyze the performance of multiprocessor systems. Most of these employ simulation or analytical-based techniques [3], [4], [5]. Few have investigated the effect of a real workload on system performance. In McGuire and Iyer [6] concurrency of real workloads executing on an Alliant is monitored and analyzed. The rest of the performance related work on the Alliant FX/8 has dealt mainly with the use of tools for evaluation or determination of performance indices [7], [8], and [9].

The current study not only analyzes real performance and resource usage data but also extracts transition models to represent the measured workload environment. The models are interpreted to gain insight into the interaction of the workload and system and to determine the amount of concurrency in the workloads.

A major step in obtaining the workload models is statistical clustering. In recent years, this approach has found many uses in the field of computer evaluation [10]. Devarakonda and Iyer [11] use clustering as a step in creating transition models which are then used to predict resource usage. Hsueh et al. [12] use similar techniques to create performability models for a multiprocessor system. Ferrari [13], on the other hand, uses clustering in the creation of artificial workloads.

The next section contains a discussion of the measured environment. Section 3 introduces the measurements used in this study. A number of preliminary results for the two samples are presented in Section 4. Section 5 describes the modeling techniques and presents the cluster and transition models obtained for the two samples. Section 6 summarizes the major results and suggests possibilities for future work.

[1] The Cedar project is a parallel supercomputing experiment which consists of interconnecting Alliant FX/8's to a large shared global memory [1] and [2]. Each Alliant is known as a cluster of the Cedar machine.

This research was supported by the National Aeronautical and Space Administration under NASA grant NAG-1-613.

Measure	Description
CONCUSER	% of time CEs clustered and running user code
clsyst	% of cluster time spent running system code
cluset	% of cluster time spent running user code
CLUSTIM	% of time spent in the cluster configuration
ipsyst	% of time IPs spent running system code
CEUT	Utilization of entire CE complex
IPUT	Utilization of entire IP complex

Table 1
Measurement Descriptions

2. THE MEASUREMENT ENVIRONMENT

The measurements for this study were taken from real, scientific workloads being executed by an Alliant FX/8 on weekday afternoons. The FX/8 is a multiprocessor mini-supercomputer with a 32 Megabyte shared global memory [14]. It can best be understood as two complexes or clusters[2] of processors. The main complex, the Computational Element (CE) cluster, consists of eight processors. These either work concurrently in the "clustered" configuration or separately in the detached configuration. When the CEs are detached, they can be used as eight separate processors working on different jobs, or groups of them can be used to multiprocess the same job. When in the clustered configuration, the concurrency control bus synchronizes the eight CEs to concurrently process a single job.

The second complex of processors on the measured Alliant consists of three Motorola MC68012 microprocessors called the Interactive Processors (IPs). In the measured system, the IPs handle all accesses to secondary memory and interactive user work such as editing jobs. It is important to note that the operating system on the measured machine is Xylem, which was specifically designed for the Cedar supercomputer, and not Concentrix, Alliant's operating system. For this reason, this paper is more an analysis of a single cluster Cedar supercomputer, and less of an analysis of the Alliant FX/8.

The measured FX/8 is used for application and algorithm development at (CSRD). This diverse environment is representative of many scientific, parallel program developmental situations. The measured programs include those specifically designed to optimize the concurrency allowed by the Alliant's architecture along with jobs that were suboptimal.

3. MEASUREMENTS

Two software facilities developed at UICSRD were used to measure system behavior. The facilities monitored the system concurrently so both types of measurements were collected approximately simultaneously.

The first facility was used to measure the amount of concurrency in the workload. It used a high resolution (10 microsecond) timer to measure the amount of time each processor was executing system and user code, as well as the amount of time each processor was idle. These measurements were taken separately for the two CE configurations (i.e., detached and clustered). The percentage of time the CEs were clustered and executing user code (CONCUSER) was then determined. The CONCUSER parameter thus measures user concurrency in the workload and should be high for observations with well-tuned applications running.

The second software facility measured the overhead associated with virtual memory and system operations such as paging, swapping, system calls, context switches, and file searches. Of approximately 150 meters available, those presented in this paper are context switches, page-ins, and page-outs. Page-ins are defined as the number of disk accesses to bring pages into main memory. Correspondingly, page-outs are the number of separate disk accesses for writing back to disk. It should be noted that the O/S facility does not provide separate measurements for each processor, but running totals for all the processors combined.

All measurements discussed above were sampled approximately simultaneously every 45 seconds. In addition to these measurements, the parameters summarized in Table 1 were calculated. Notice that some of the percentages in the table are calculated over the entire 45-second period, and others are calculated just over the time spent in a specific configuration.

Each 45-second period is one observation of the system, and the measurements collected during that period depict the state of the system for that observation. The length of the observation was experimentally determined and chosen so that it would best correspond to the length of an actual, physical state of the machine.

Several workload samples were collected for this study. In this paper, two markedly different workload samples are presented. The first sample was taken over a 138-minute period. The second sample, on the other hand, is 168 minutes long. To provide a broad understanding of the two workloads and their interactions with the system, some preliminary statistical analysis is presented in the next section.

[2] The use of the word cluster is admittedly overused in this paper. The Alliant FX/8s are clusters of the Cedar, while the FX/8's have their own clusters. Later, cluster models will be introduced. This confusion was inevitable, in order to maintain consistency with the results in the other literature on these subjects.

Parameter	Sample One		Sample Two	
	mean	std. dev.	mean	std. dev.
context switches	1782.508	508.201	1503.382	665.230
device interrupts	20389.339	5976.630	18459.958	11929.022
page-ins	0.109	1.016	24.747	66.278
page-outs	1.869	14.630	18.116	40.957
CE utilization	0.723	0.077	0.393	0.246
IP utilization	0.304	0.078	0.271	0.101
CLUSTIM	71.632	10.472	63.920	14.971
cluset	90.165	4.946	39.879	34.000
clsyst	9.835	4.946	21.727	15.159
ipsyst	23.716	5.325	17.231	4.146
CONCUSER	64.625	10.470	27.028	26.028

Table 2
Measurement Means and Standard Deviations

4. PRELIMINARY ANALYSIS

4.1 Means and Standard Deviations

Table 2 summarizes the means and standard deviations of each parameter studied. Sample One is characterized by high user concurrency and CE utilization. The relatively small standard deviations for the parameters indicate stable activity during the collection of the sample. Sample Two, on the other hand, is characterized by low user concurrency and CE utilization. The standard deviations for this sample are high (e.g. see CONCUSER) indicating that the sample captured a workload consisting of bursts of work surrounded by idleness.

The table also shows an imbalance between the IP and CE utilizations for both samples. This imbalance, especially for Sample One, may be partially attributed to the low paging activity. (All accesses to disk must be made through IP0, thus when the paging is low the IP utilization tends to be low.) Another cause of this imbalance is Xylem's scheduling policy. Whenever possible jobs are scheduled on the CEs because they are much faster than the IPs.

Table 2 also highlights the paging differences between the two sampled workloads. Sample One contains very little paging, while Sample Two has a substantial amount of paging. The standard deviations for the paging activities of both samples are quite high, suggesting intervals of high paging activity interspersed with periods of little or no paging. The periods of little paging activity are easily explained by the large 32-MB physical memory found in the Alliant.

4.2 Individual System/User/Idle Times

In this section the behavior of the individual processors is studied. Figures 1 and 2 show the percentage of time each processor spends executing system and user code, along with the percentage of time the processors are idle. The bars shown for the individual CEs, CE0-CE7, pertain to the time spent in detached configuration. The cluster bar (CL) shows the breakdown for the CEs' utilizations while in the clustered configuration (only one bar is needed because all CEs work on the same job in this configuration). It is important to realize that these percentages are not calculated over the whole period, but only the period in which the CEs are in the specified configuration. For example, Figure 1 shows that while detached, CE7 is idle 45% of the time, executing system code 30% of the time, and executing user code 25% of the time.

Figures 1 and 2 confirm the low utilization of the IPs. They also show that the work done on the IPs is evenly balanced. Note also the low utilization of the lowered numbered CEs while in the detached mode. The majority of the work in the detached mode is done by CE6 and CE7. These results suggest that a better design may be to allow the four lower CEs to form their own cluster. Thus, when the detached mode is needed, the upper four processors can break free and handle the work. Meanwhile, the lower four stay in clustered configuration and continue to service the jobs waiting on the cluster queue.

In summary, the preliminary analysis shows that Sample One captured a system with high, steady CE utilization, little paging, and a high degree of user concurrency. This is the result of a relatively stable workload. Sample Two, on the other hand, is made up of observations with high variability in their CE utilization, and the amount of paging they capture. On average, the sample also shows very little user concurrency. This is the result of a generally light workload with bursts of high activity. In addition, for both samples, the lower numbered CEs in the detached configuration and all three IPs showed low utilization.

5. MODEL EXTRACTION

In this section we extract state transition models to quantify the variation in system activity for each workload. Four parameters were selected to jointly characterize user concurrency and system overhead. These were IPUT, context switches, CONCUSER, and pagact (pagact = page-ins + page-outs, the total number of accesses to disk). Each observation is treated as a point in four-dimensional space. Statistical clustering analysis is used to identify similar classes (clusters) in this space. Each cluster is then defined as a system state, and a state transition model (consisting of intercluster transition probabilities) is developed.

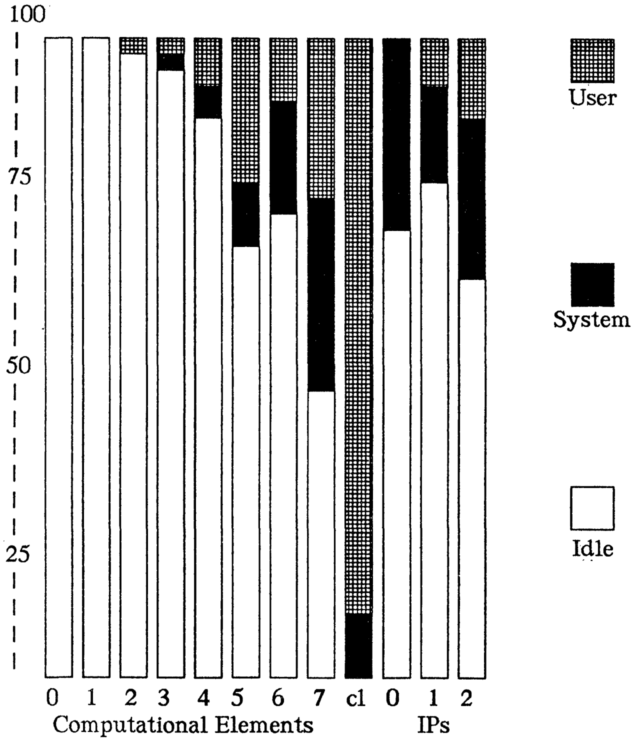


Figure 1
User/System/Idle times

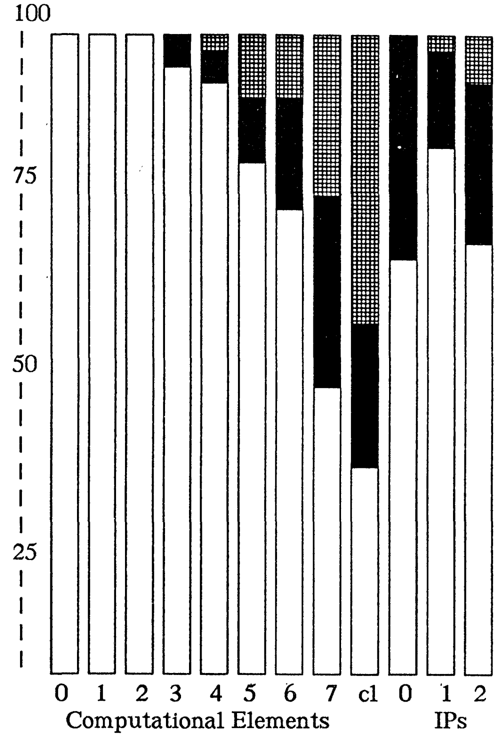


Figure 2
User/System/Idle times

These transition probabilities may be used to predict forthcoming states of the machine. They also provide a solid understanding of the relationships between states.

Next the extracted cluster model is interpreted by computing "skewness factors" for each transition. The skewness factors quantify the degree to which transitional relations between states were caused by random transitions. More specifically, a skewness factor determines the skewness of a transition probability with respect to the transition probability that would be obtained if each inter-observation transition was equally likely. The skewness factor (S_{ij}) of a transition from state i to state j is defined as

$$S_{ij} = \frac{\textit{observed number of transitions from state i to state j}}{\textit{probable* number of transitions from state i to state j}}$$

*Assuming that the transition to any observation is equally likely regardless of the cluster it is in.

The skewness factors bring out the effect of unbalanced clusters and quantify significant transitions between clusters. A significant transition is one that may an have underlying system-related cause, and is not just the result of random action. In some cases, small transition probabilities can mask these significant transitions. In other cases, the skewness factor may show that transitions which appear to be significant (because of high transition probabilities) may actually be explained by random transitions among states. A skewness factor near unity indicates that there is probably not a significant transition between states.

Following this, skewness factors are calculated and used to detect significant transitional relationships between the states of the system.

5.1 Clustering, Transition Models, Skewness Factors

The cluster models were obtained using the FASTCLUS procedure from the SAS software package [15]. This procedure uses a K-means clustering method, grouping observations into clusters that minimize the intracluster distances between points, while maximizing the intercluster distances. All distances are Euclidean.

The cluster models obtained are studied from three different perspectives, each providing different types of results. At the most basic level, the clusterings of observations are studied verbatim to determine the characteristics of the different states in which the machine is found. By the number of observations in each cluster, the percentage of time the machine is in each of these states may be determined. From this, the efficiency of the machine may be ascertained.

The second form of analysis requires the creation of a state transition model, which consists of the probabilities for each intercluster (interstate) transition. These probabilities are easily estimated from the collected data with the following formula (P_{ij} - probability of transition from state i to state j):

$$P_{ij} = \frac{\textit{observed number of transitions from state i to state j}}{\textit{observed number of transitions from state i}}$$

concuser % of time all 8 CES
are working together.

cluster number	% of obs.	context switches	CONCUSER	IPUT	pagact
one	6.01	2978	52.97	0.4668	0.000
two	12.02	1965	51.21	0.4176	0.000
three	2.73	2124	51.95	0.3441	65.86
four	26.78	1963	57.13	0.3136	0.3795
five	30.05	1716	71.16	0.2787	0.000
six	22.40	1195	76.69	0.2169	0.3354

Table 3
Centroids of Clusters: Sample One

The most desirable state, i.e., high user concurrency, is captured by the observations found in clusters five and six. Cluster six contains observations with higher user concurrency, lower IP utilization, and fewer context switches than the observations in cluster five. It is interesting to note that the high user concurrency captured by observations in these clusters is accompanied by relatively low IP utilization and few context switches.

The system is in a state of high user concurrency approximately 52% of the time (clusters five and six), with less, but still impressive amounts of concurrency being seen about 27% of the time (cluster four). The undesirable states (one and two) account for only 18% of the sample.

The transition model extracted for Sample One is shown in Figure 3. The high self-loop transition probabilities suggest that for all states (except state three, where the self-loop probability is only 0.2), there is a good chance the machine will operate in the same state during the following observation. The skewness factors confirm this relationship, and show that state three also has an affinity to return to itself.

In summary, the skewness factor may be viewed as a validity test which provides a measure of credibility for the state transition model. In other words, it indicates whether there is any 'real' information in the transition model, or whether it just captured random activity.

5.2 Cluster Analysis for Sample One

The cluster model extracted from Sample One is summarized in Table 3. Cluster one, depicts a system with a high context switch rate, relatively high IP utilization, and low user concurrency. Cluster two has similar characteristics, except they are not quite as extreme. The observations in these clusters most likely reflect a high degree of multiprogramming which may have reduced the concurrency exploitation.

The third cluster, which only accounts for 2.73% of the sample, contains observations with considerable paging activity. As expected, the paging activity is accompanied by above-average values for both IP utilization and context switching. These observations also show a lower than average user concurrency.

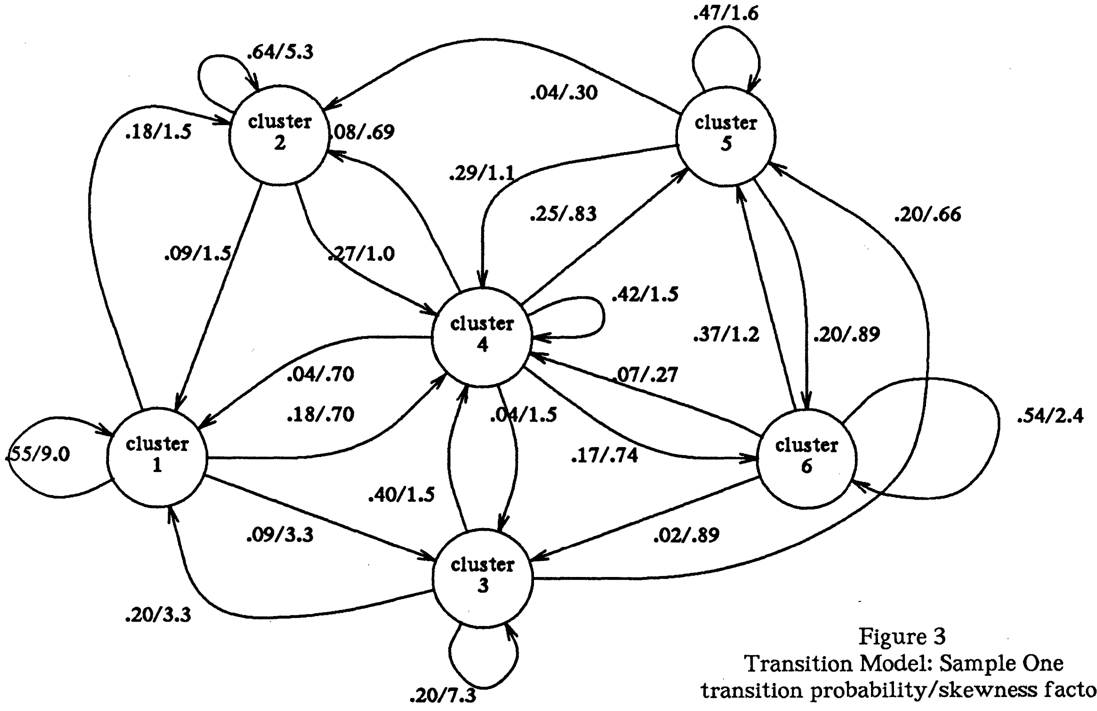


Figure 3
Transition Model: Sample One
transition probability/skewness factor

cluster number	% of obs.	context switches	CONCUSER	IPUT	pagact
one	7.11	2391	10.61	0.2985	8.972
two	49.78	1174	6.362	0.1932	9.127
three	14.2	1032	68.91	0.2585	11.46
four	12.00	2362	41.76	0.4376	164.9
five	12.44	1563	55.24	0.3564	24.24
six	4.44	2797	31.84	0.4485	298.2

Table 4
Centroids of Clusters: Sample Two

This is a good example of a skewness factor identifying a significant transition that the transition probability by itself would have masked.

An interesting phenomenon brought out by the transition model is the lack of interaction between the high and low concurrency states. The only observed transitions into the high user concurrency state (six) were from states four, five, or six, which are other states depicting substantial user concurrency. This phenomenon is also seen for transitions into state five, the state depicting the second highest degree of concurrency in this model. Conversely, there are few observed transitions from the two high concurrency states (five and six) to the low concurrency states (one and two). Thus, it can be concluded that the machine does not experience sudden jumps from high user concurrency to low user concurrency, or vice versa. Transitions from these extremes are made by stepping through intermediate states, such as state four.

The near unity skewness factors for all six transitions from state four indicate that the transitions from this state were almost uniformly distributed among the observations, regardless of the clusters obtained. Obviously, the behavior of the machine after being in this state would be the most difficult to predict. As hinted at above, state four acts as the dispenser, or lowest step, to the extreme states of the system.

A final point of interest is the relationship between state one and state three as indicated by the skewness factors (but masked by the transition probabilities). The transition probabilities between these states are not very high, but the skewness factors are both 3.3. Recall that both states depict a system of low user concurrency, with state three also corresponding to high paging activity, and state one corresponding to high IP utilization. Thus the skewness factor is able to bring out an underlying system related dependency between paging and IP utilization.

5.3 Cluster Analysis for Sample Two

A summary of the cluster model extracted for Sample Two is presented in Table 4. The dominant cluster in the model is cluster two which accounts for almost half of the observations. Although the cluster depicts a near idle system, it should not be regarded as a weakness of the machine, but a consequence of monitoring real workloads. (Long periods of time passed with an extremely light workload while this sample was taken.) In the analysis, cluster two is ignored, when possible, because it reveals little about the system's behavior under a substantial workload. A more revealing cluster is cluster one. The observations in this cluster show very little user concurrency, relatively high IP utilization, and a large number of context switches.

The desirable state, high user concurrency, is captured by the observations in clusters three and five. Cluster three contains observations with high user concurrency low IP utilization, little paging activity, and few context switches. Cluster five contains observations with similar, but less extreme characteristics.

The paging activity that was first observed in the preliminary analysis is captured by the observations in clusters four and six. Of the two, cluster six contains the observations with the higher paging activity. The high paging is accompanied by high IP utilization, and a large number of context switches. It should also be pointed out that both paging clusters contain observations having low user concurrency, with cluster six (extreme paging observations) showing less concurrency than cluster four (medium paging observations). Note that for both samples, paging is seen to adversely affect the amount of user concurrency exploited.

If we work under the assumption that cluster two contains only observations of the system under a light workload, we can discard these values for a quick analysis of the efficiency of the system under substantial workload. With the cluster two observations discarded, the percentage of observations for the other clusters is doubled. This puts the system in the desirable clusters (three and five) about 52% of the time, which is similar to Sample One. In addition, we find the system is in the paging clusters about 32% of the time, and in the undesirable cluster (one) about 14% of the time. In summary, the analysis shows that while the machine was under a substantial workload, which was only about half the time, user concurrency was high at times, but not consistently high.

The transition model for Sample Two is presented in Figure 4. As in Sample One, the transition probabilities and skewness factors are largest for self-loops. This indicates that the state of the system is fairly stable.

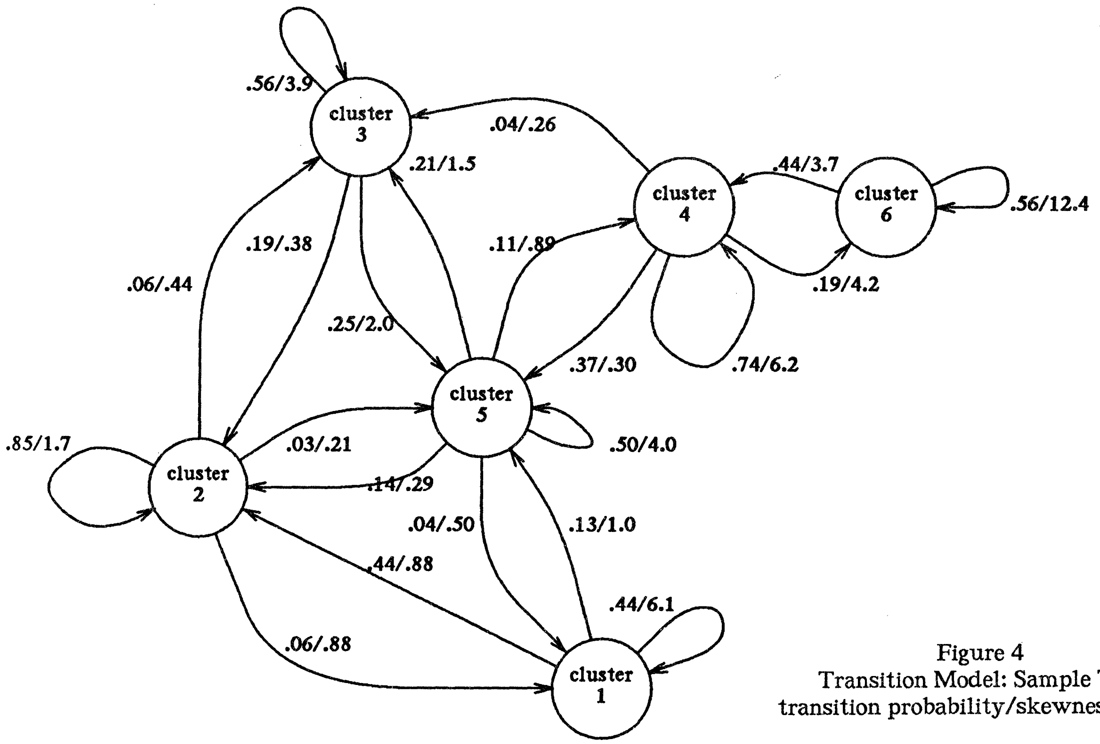


Figure 4
Transition Model: Sample Two
transition probability/skewness factor

The skewness factors for the two paging clusters (four and six) are especially interesting because there are few nonzero values. The only way to get to state six (high paging) is through state four (medium paging), and the only way to leave it is again through state four. This stepping-stone effect goes even further. The only way to get to state four (beside itself or six) is through state five, the third highest paging state. Therefore, the system gradually builds up to high levels of paging and then gradually dissipates back down to nothing. In addition, as in Sample One, this stepping-stone effect is also seen for the user concurrency measurement.

6. Conclusions

In this paper we have presented an analysis of an Alliant FX/8 system running Xylem (Cedar's operating system) at the University of Illinois Center for Supercomputing Research and Development. Preliminary analysis showed that the first workload sample was comprised of consistently high user concurrency, low system overhead, and little paging. The second sample captured much less user concurrency, but had significant paging and system overhead. In addition, it was seen that both the IPs and the four lowered numbered CEs, while detached, were underutilized.

The results from the statistical models showed that during the collection of the first sample, the system was operating in states of high user concurrency approximately 79% of the time. The second workload sample captured the system in high user concurrency states only 26% of the time. In addition, it was discovered that high system overhead was usually accompanied by low user concurrency. The analysis also showed a high predictability of system behavior, for both workloads. This predictability was largely due to slow changes in system states. In particular, states with extremely high values of paging or user concurrency are usually preceded by states with less paging and user concurrency, much like stair climbing. A stepping down effect was observed when the machine left these extreme states.

Future research will include cluster analysis of individual programs and benchmarks to determine their behavior on the system, and to further evaluate the techniques developed. Similar studies on other multiprocessor environments are also in the planning stages.

Acknowledgments

The authors would like to thank the researchers at CSRD for their assistance during the course of this work. In particular thanks are due to Richard Barton and Bob McGrath for use of their software and to Allen Maloney for many useful discussions. Thanks are also due to Sujatha Subramani for her assistance with this work and to Janet Adams for her careful proofreading.

References

[1] D. J. Kuck, E. S. Davidson, D. H. Lawrie and A. S. Sameh, *Parallel Supercomputing Today and the Cedar Approach*, CSRD Report No. 652, University of Illinois at Urbana-Champaign, June 1987.

[2] P. Yew, *Architecture of the Cedar Supercomputer*, **Proc. IBM Institute of Europe,** pp. 8-12, August 1986.

[3] P. Heidelberger, and K. Trivedi, *Queueing Network Models for Parallel Processing with Asynchronous Tasks*, **IEEE Trans. Comp.,** vol. C-31, pp. 1099-1108, 1982.

[4] P. Heidelberger, and K. Trivedi, *Analytic Queueing Models for Programs with Internal Concurrency*, **IEEE Trans. Comp.,** vol. C-32, pp. 73-82, January 1983.

[5] U. Herzog, W. Hoffman, and W. Kleinoder, *Performance Modeling and Evaluation for Hierarchically Organized Multiprocessor Computer Systems*, **Proc. 1979 Int'l. Conf. on Parallel Processing,** pp. 103-114.

[6] McGuire, P.J., Iyer, R.K., *A Measurement-Based Study of Concurrency in a Multiprocessor*, **Proc. of the 1987 Int. Conf. on Parallel Proc.** August, 1987.

[7] W. Abu-Sufah and A. Kwok, *Performance Prediction Tools for Cedar: A Multiprocessor Supercomputer*, **Proc. 12th Int'l. Symp. Computer Architecture,** pp. 406-413, 1985.

[8] A. Malony, *Cedar Performance Measurements*, CSRD Report No. 579, University of Illinois at Urbana-Champaign, June 1986.

[9] A. Malony, *Cedar Performance Evaluation Tools: A Status Report*, CSRD Report No. 582, University of Illinois at Urbana-Champaign, July 1986.

[10] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems*, Englewood Cliffs, NJ:Prentice-Hall, Inc., 1981.

[11] M. Devarakonda, and R. K. Iyer, *Predictability of Process Resource Usage: A Measurement-Based Study of Unix*, **Ph.D. dissertation, University of Illinois at Urbana-Champaign,** October 1987.

[12] M. C. Hsueh, R. K. Iyer, and K. Trivedi, *A Measurement-Based Performability Model for a Multiprocessor System*, **IEEE Transactions on Computers,** pp. 478-484, April 1988.

[13] Ferrari, D., *On the Foundations of Artificial Workload Design*, **ACM,** 1984.

[14] Alliant Computer System Corp., *FX/Series Product Summary*, June 1985.

[15] H. Artis, *Workload Characterization using SAS PROC FASTCLUS*, **Proc. Int'l. Workshop Workload Characterization Computer Systems and Computer Networks,** October 1985.

PERFORMANCE COMPARISON OF TWO MULTIPROCESSOR B-LINK TREE IMPLEMENTATIONS

Ravi Mukkamala

Department of Computer Science
Old Dominion University
Norfolk, VA 23529.

Roger K. Shultz

Rockwell-Collins International
Cedar Rapids, IA 52498.

Abstract

In this paper, we investigate the interaction of concurrent database algorithms with the underlying multiprocessor computer architectures. We implement an optimistic concurrent *B-link* tree access algorithm on two *simulated* multiple processor computer architectures: a shared secondary storage system, and a processor-per-secondary storage system. It has been observed that the average degree of concurrency and the transaction throughput of the processor-per-secondary storage system are much greater than those of the shared secondary storage system.

1. Introduction

The performance of a distributed system is influenced both by the underlying architecture and the algorithms that control the execution of the system. Among the several architectural factors that influence the performance of a multiprocessor based distributed system, the interconnection of processors and secondary memories is an important consideration. Similarly, the concurrency control algorithm employed by a distributed system greatly influences the availability of the system to its users. Currently, we are investigating the interaction between concurrency control algorithms and the underlying computer system components in a distributed system.

We study two configurations of processors and memories, each communicating on a shared interconnection network. Both systems contain processors that execute search and insert operations on a shared file indexed by a B-tree based structure. Due to its efficient sequential and random access mechanism, we chose the the *B-link tree* of Lehman and Yao [5].

Previous experiments in this area investigated a *centrally-accessed* data object (a B-link tree) shared between a set of processors [3]. The experiments presented here investigate the performance characteristics of a *distributed* B-link tree. The addition of these results allows us to compare two different system architectures executing the same concurrency control methods.

This paper is organized as follows. Section 2 describes concurrent access algorithms for B-tree systems. Section 3 briefly describes an evaluation scheme for the B-tree concurrency algorithms. Section 4 describes the proposed implementation of the B-link tree system. Section 5 briefly describes the simulation parameters. Section 6 describes the evaluation metrics adopted in this paper. Section 7 presents the results obtained from the current experiments and compares it with the earlier results. Finally, Section 8 has some concluding remarks.

2. A Concurrent B-tree System

Our focus in this paper is on evaluation of the optimistic B-tree system proposed by Lehman and Yao [5]. This system uses a modified form of B-tree, called a *B-link tree*, which provides multiple access paths to terminal data nodes. Users are provided with a high degree of concurrent access to the shared tree through a limited set of operations, which include *Search*, *Insert*, and a simplified *Delete* (this Delete is a logical Delete – it removes key values but does not cause tree restructuring). More complete discussions of B-tree algorithms can be found in [3,5].

The key performance parameters in the algorithm are: frequency of interference, cost of validation (a static quantity), and cost of recovery (a dynamic quantity). In a previous study, interference and recovery of multiple processors accessing a shared B-link tree were measured [3]. Average number of interferences per processor were found to approach a constant as the number of processors increased. Also, it was found that an average of between one and two links were traversed during each recovery [3].

3. Concurrent System Evaluation

Performance evaluation of concurrent tree algorithms have typically involved modeling and analysis, with high-level simulation used to provide additional support for analytical results. Analytical techniques are primarily used to predict (bound) certain aspects of concurrent algorithm performance. Markov chain models have been used to derive throughput and response time of static locking on a centralized database system [8]. Probabilistic analysis has proved useful in estimating the expected number of waiting updaters (processes), waiting readers, and the number of locks held by the processes [1]. Kung and Robinson employ probabilistic analysis to compute the number of conflicts between transactions that occur for locking protocols [4]. Because of the complexity of concurrent transaction systems and the simplifying assumptions necessary for using analytical models, we concentrate on simulation experiments to measure system performance.

Evaluation of a binary search tree shared among transaction and maintenance processors using a two-phase locking protocol has been performed in [6]. In this study, each processor has its own local memory and shares a common global memory. The average ratio of processor waiting time to total processor execution time is determined as a function of the number of operations per transaction [6]. Locking protocols for AVL-trees, 2-3 trees, and linear hashing [2] have been proposed and evaluated through analysis and high-level simulation; these evaluations have the *average number of concurrently busy transactions* as the metric of interest.

4. User Transaction Managers and Shared Resource Managers

A range of possible multiprocessor architectures may support a concurrent B-link system. We have narrowed this investigation to systems composed of user transaction managers (UTMs), shared resource managers (SRMs), and an interconnection network.

A *user transaction manager* executes a B-link tree access algorithm on behalf of user transactions. There is one user transaction manager per processor to manage the transaction requests received at that processor (or network node). Each user transaction manager executes the same access method and uses the system interconnection network to access the nodes of the B-link tree(s). Each UTM coordinates the processing of a transaction by sending the access requests to shared resource managers, and processing the received B-link tree node data.

A *shared resource manager* maintains and controls the acccess to the B-link trees (or their partitions). The B-link tree is itself stored on a secondary storage system. The shared resource manager communicates with the secondary storage device to execute the requests received from the different user transaction managers. An SRM grants locks on disk pages, unlocks pages, transfers pages to-and-from the secondary store, and communicates with the user transaction managers through the interconnection network. All low-level hardware related secondary storage functions such as disk allocation and physical data format are hidden from the user transaction managers.

In this paper, we discuss two alternative computer architectures to interconnect URMs and SRMs. These are shown in Figures 1 and 2.

System 1 consists of multiple user transaction managers each communicating with a *single* shared resource manager. Communications between the managers is achieved over a shared network. In Figure 1, P_{utm} represents a processor executing one of the user transaction managers. Similarly, P_{srm} represents a processor executing the one and only shared resource manager. S_{nw} represents the shared bus system. Finally, M_{ss} represents the secondary storage system that stores the B-link tree and the corresponding data associated with the keys in this tree [3].

In *System 2*, each processor in the muliprocessor system contains a user transaction manager and a shared resource manager. Each processor is associated with a dedicated secondary storage device (M_{ss}). The two separate functions of *System 1*, user transaction manager and shared resource manager, are *multiprogrammed* on the same processor in *System 2*. Thus P_{utm} and P_{srm} are logical processors corresponding to one physical processor. The processors are interconnected on a shared communication network (S_{nw}).

5. Simulation Parameters

The performance of the application system is governed by the operation workload and the execution characteristics of each of the following components: the processors, the user transaction managers, secondary storage access, shared bus, and the shared resource managers. Figure 3 describes one cycle of user transaction execution. This figure describes a non-local access to a B-tree. It consists of three parts: processing by UTM, transmission of requests and replies through the shared bus, and the processing at the SRM.

We consider two different workloads to represent different kinds of B-tree applications: Search-intensive (70% Searches and 30% Inserts or 70/30) and Insert-intensive (30% Searches and 70% Inserts, or 30/70). The previous investigations revealed that it is sufficient to have 25 transactions for each experiment. We decided to have 100 transactions from each processor in the system per experiment. Each experiment was repeated four time to provide a 95% confidence level.

Specific event timings in the simulator are based on published component speeds in a Hewlett-Packard HP Series 200 SRM configuration. In order to relate processing and data access costs, the processing speeds of each statement of user transaction manager's code are expressed in terms of the time required to write one word to disk. This correlates with other experiments performed with the simulation system [7].

6. Performance Measures

In order to compare the performance of *System 1* and *System 2*, we have selected the following metrics: cyclic processing power, system throughput, degree of concurrency, average waiting time for the bus, and the average waiting time for SRM. We have also measured the number of interferences, the average time for a search transaction, and the average time for an insert operation. Due to space limitation, all these measurements are not presented in this paper. Except for cyclic processing power, the rest of the metrics assume the usual meaning and hence are not explained here.

6.1. Cyclic Processing Power

In order to provide an algorithm-independent measure of the effectiveness of the concurrent algorithm in a particular application, we use the concept of *cyclic processing power* (CPP) described by Vrasalovic, et al. [9]. Intutively, cyclic processing power measures the percentage time a processor actively executes its user program, as opposed to the time it waits for service from a system resource.

Let us consider the processing cycle in Figure 3. The local processing (in each cycle) that a user request initially receives at a processor is denoted by t_p. After the initial processing, if the initiating processor decides to access a non-local shared resource manager, then the corresponding user transaction manager attempts to access the shared bus. The waiting time to access the shared bus is indicated by t_{wbus}. The time to completely transmit the request and release the bus is indicated by t_a. The time for the remote processor (or SRM) to execute this request and send the reply is indicated by t_{wsrm}. As shown in Figure 3, t_{wsrm} consists of four components of time: wait time in the SRM's queue prior to processing, time to process the request, wait time before accessing a bus to send the reply, and time to transmit the reply. The total waiting time per processing cycle is defined as $t_w = t_{wbus} + t_{wsrm}$.

In case the initiator decides to execute the request locally, there is no need to access the bus. Thus the time spent by the initiator to acess and transmit a request (t_{wbus}) and the time spent by the SRM to access the bus and transmit the reply are avoided.

If we have n processors (or UTMs) in the system, then the average cyclic processing power per processor may be defined as :

$$CPP = \frac{\sum_{i=1}^{n} \frac{t_{a_i}+t_{p_i}}{t_{a_i}+t_{p_i}+t_{w_i}}}{n} \qquad (1)$$

where the subscript i indicates the measurements corresponding to the processing cycle at the i^{th} processor.

7. Simulation Results

Tables 1-4 summarize the measurements of access time (t_a), processing time (t_p), waiting time for the shared bus (t_{wbus}), the waiting time for the SRM, and the cyclic processing power (CPP). These timings represent measurements averaged over all processors and over all the transactions for a given number of processors (n). CPP is computed from the other average values using Equation (1).

1. *Degree of Concurrency:* As shown in Tables 3 and 4, processors of both systems show a decrease in degree of concurrency with an increase in the number of processors. From Tables 1-4 it may be observed that System 1's concurrency is reduced primarily due to the time spent waiting for the SRM to send the results of a command. In System 2, however, concurrency decreases with the addition of processors (and their transactions) due to competition for access to the shared network.

2. *Throughput:* Tables 3 and 4 summarize the throughput measurements for Systems 1 and 2. As the number of processors increase, the parallel execution of SRM operations boosts the throughput of System 2 higher than System 1. For System 1, the throughput decreases with the number of processors. This is attributed to the contention at the single SRM. For System 2, however, the throughput increases with the number of processors. The throughput of System 2 should keep increasing until enough processors and transactions are added to cause high contention on the shared network for remote SRM requests. At this point the migration of operations to data rather than the movement of data to operations may be a good approach.

3. *Shared Bus Access:* From Table 1 it may be observed that t_{wbus} is much higher in System 2 as compared to System 1. This is attributed to the increased processing activity at the processors in System 2 due to the distribution of the B-Tree. In System 1, the processors spend most of the time waiting for the SRM's reply, and hence there is less contention on the bus. The t_{wbus}, however, does not linearly increase with the number of processors. This is also clear from the throughput statistics in Tables 1 and 2.

4. *SRM Access:* From Table 1 it is clear that the shared resource manager is the bottleneck in System 1. The waiting time for a user transaction manager to await a reply from the central SRM continues to grow with the number of processors (or requestors). This waiting time also increases with the number of insert transactions in the input mix. The possible node splits and the resulting increase in read/write actions on B-tree nodes can explain this phenonmenon. Due to the distributed nature of the B-tree processing, shared resource manager is no longer the bottleneck in System 2.

5. *Cyclic Processing Power:* The cyclic processing power (in both Systems 1 and 2) decreases with the increase in the number of processors. This is due to the idle time of the processors due to either waiting for the reply from SRM (in System 1) or due to contention for the bus (in System 2). Over all, the cyclic processing power of System 2 appears to be slightly higher than that of System 1. So by speeding up the shared bus, we can reduce the waiting time for bus access, and in turn increase the cyclic processing power of System 2. No such improvements are possible for System 1.

8. Conclusion

In this paper, we described the results that we obtained during the implementation of a B-link tree system on a multiprocessor system with two different architectures. *System 1* implements the B-tree on a single processor with secondary memory. *System 2* implements a distributed version of the B-tree.

The current investigations have lead to several interesting (some may be obvious) observations. Without much additional hardware or software cost, performance of concurrent B-link tree operations can be improved dramatically. System 2 requires additional secondary storage devices (with divided capacity). This cost is more than balanced by the increase in performance (throughput and transaction response time). Since each node in System 2 contains a copy of SRM and UTM software (the same as in System 1), costs for software maintenance should be similar in both systems. Thus, we conclude that parallel disk access and multiprogramming of UTM and SRM functions make System 2 far superior.

References

[1] Bayer, R, and M Scholnick, "Concurrency of Operations on B-Trees," Acta Informatica, Vol. 9, pp. 1-21, 1977.

[2] Ellis, C S, "Concurrency and Linear Hashing," Proceedings of the 1985 Symposium on Principles of Database Systems, Portland, Oregon, pp. 1-7, April 1985.

[3] Ford, R, M Jipping, and R K Shultz, "On the Performance of an Optimistic Concurrent Tree Algorithm," Department of Computer Science TR # 85-07, September 1985.

[4] Kung, H T, and J T Robinson, "On Optimistic Methods for Concurrency Control," ACM Transactions on Database Systems, Vol. 6, No. 6, pp. 213-226, June 1981.

[5] Lehman, P J, and S B Yao, "Efficient Locking for Concurrent Operations on B-Trees," ACM Transactions on Database Systems, Vol. 6, No. 4, pp. 650-670, Decemeber 1981.

[6] Manber, U, "Concurrent Maintenance of Binary Search Trees," IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, pp. 777-784, November 1984.

[7] Shultz, R K, "Simulation of Multiprocessor Computer Architectures using ACL," Technical Report, TR 83-11, Department of Computer Science, Universityof Iowa, 1983.

[8] Thomasian, A, "Performance Evaluation of Centralized Databases with Static Locking," IEEE Transactions on Software Engineering, Vol. SE-11, No. 4, pp. 346-355, April 1985.

[9] Vrsalovic, D, D Siewiorek, Z Segall, and E Gehringer, "Performance Prediction for Multiprocessor Systems," Proceedings of the 1984 International Conference on Parallel Processing, Michigan, August 1984.

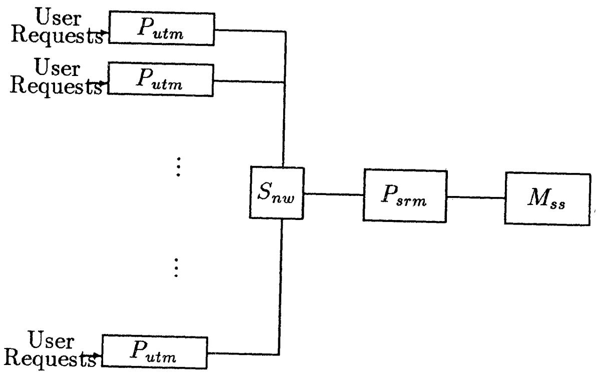


Figure 1: System 1 Architecture

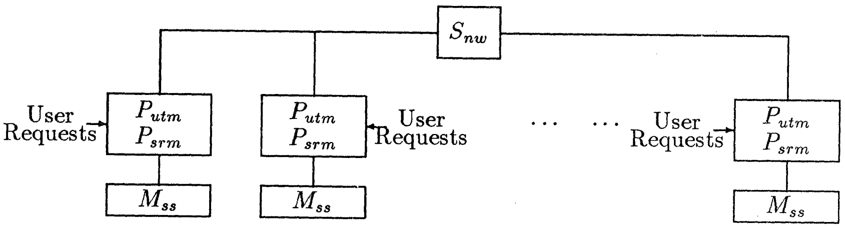


Figure 2: System 2 Architecture

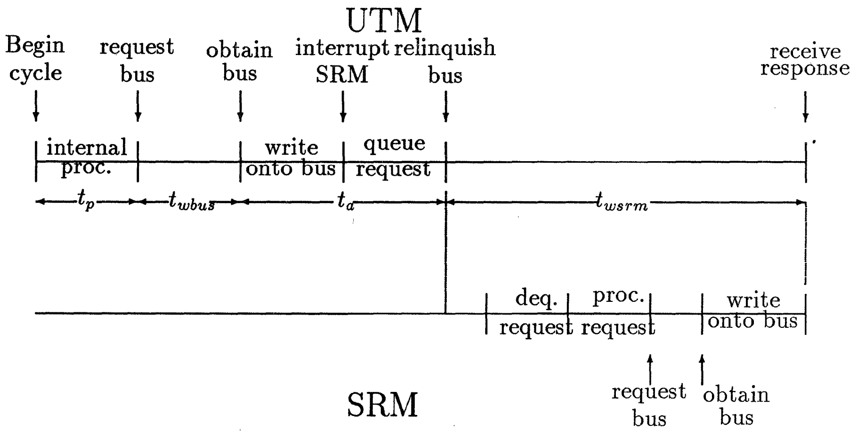


Figure 3: Breakdown of a Single UTM Cycle for a Remote SRM Access

# of	*System 1*					*System 2*				
Proc.	t_a	t_p	t_{wbus}	t_{swrm}	CPP	t_a	t_p	t_{wbus}	t_{swrm}	CPP
1	0.068	0.202	0.000	0.694	0.280	0.094	0.159	0.000	0.222	0.533
4	0.066	0.186	0.377	4.692	0.047	0.124	0.204	8.306	1.303	0.033
7	0.070	0.202	0.371	9.772	0.026	0.112	0.186	8.188	1.825	0.025
10	0.071	0.202	0.383	17.054	0.015	0.108	0.182	9.255	1.976	0.025
13	0.072	0.211	0.395	17.634	0.016	0.106	0.179	11.237	2.534	0.020
16	0.073	0.216	0.401	20.900	0.013	0.106	0.177	12.674	2.879	0.018
19	0.074	0.217	0.411	27.480	0.010	0.103	0.173	14.545	3.085	0.013
25	-	-	-	-	-	0.104	0.170	17.145	3.255	0.013

Table 1: Simulation Results for Systems 1 and 2: 70/30 Mix

# of	*System 1*					*System 2*				
Proc.	t_a	t_p	t_{wbus}	t_{swrm}	CPP	t_a	t_p	t_{wbus}	t_{swrm}	CPP
1	0.080	0.219	0.000	0.666	0.310	0.112	0.144	0.000	0.207	0.553
4	0.074	0.185	0.258	12.677	0.020	0.140	0.192	22.332	6.223	0.012
7	0.074	0.177	0.258	27.426	0.009	0.129	0.186	27.230	2.345	0.011
10	0.078	0.200	0.281	22.389	0.012	0.125	0.175	23.267	1.825	0.012
13	0.077	0.193	0.282	34.202	0.008	0.119	0.169	23.289	8.134	0.009
16	0.078	0.197	0.287	39.636	0.007	0.108	0.160	24.378	10.030	0.008
19	0.079	0.196	0.296	55.242	0.005	0.104	0.154	24.145	10.457	0.007
25	-	-	-	-	-	0.100	0.145	24.235	11.178	0.007

Table 2: Simulation Results for Systems 1 and 2: 30/70 Mix

#of	*System 1*		*System 2*	
Procs.	Av. Conc.	Av. Thro.	Av. Conc	Av. Thro.
n	Dc_1	Th_1	Dc_2	Th_2
1	0.35	0.2	1.0	0.3
4	0.11	0.25	0.69	0.35
7	0.07	0.25	0.63	0.49
10	0.05	0.21	0.63	0.62
13	0.05	0.19	0.63	0.71
16	0.05	0.18	0.63	0.80
19	0.05	0.18	0.63	0.89
25	-	-	0.63	0.91

Table 3: Average Degree of Concurrency and Throughput for 70/30 Mix

#of	*System 1*		*System 2*	
Procs.	Av. Conc.	Av. Thro.	Av. Conc	Av. Conc.
n	Dc_1	Th_1	Dc_2	Th_2
1	0.35	0.14	1.0	0.2
4	0.11	0.18	0.62	0.25
7	0.07	0.18	0.61	0.38
10	0.05	0.18	0.58	0.45
13	0.05	0.18	0.58	0.53
16	0.05	0.18	0.58	0.60
19	0.05	0.18	0.58	0.68
25	-	-	0.58	0.72

Table 4: Average Degree of Concurrency and Throughput for 30/70 Mix

INDEPENDENT CONNECTIONS : AN EASY CHARACTERIZATION OF BASELINE-EQUIVALENT MULTISTAGE INTERCONNECTION NETWORKS

J.C. BERMOND *
J. M. FOURNEAU **

* L.R.I., UA 410 CNRS, bat 490, Université Paris Sud Orsay, France
** I.S.E.M., bat 490, Université Paris Sud Orsay, France

Abstract : we study topological properties of multistage interconnection networks. We state a graph characterization of all the networks topologically equivalent to the Baseline networks and we explain why networks defined by some types of permutations are equivalent. Independent Connections are the link between graph theory and network definition using a numerical characterization for adjacency relationship. We establish that Banyan networks built with independent connections are topologically equivalent. We also consider the PIPID field, an useful set of permutations, which allow the construction of the usual multistage interconnection networks, and which are easily modeled by independent connections.

1. Introduction

Several multistage interconnection networks have been proposed for communication in parallel architectures. They are typically designed using at least $n=log_2(N)$ stages of $N/2$ 2×2 switching cells to connect N inputs to N outputs [8]. Topological properties of these networks have been extensively studied, as only few parameters (number of stages, type and number of cells, connections between stages) may drastically change their functionalities. Topological equivalence between the "classical" networks (Omega [11] , Flip [3] , Indirect Binary Cube [14] , Modified Data Manipulator [6] , Baseline and Reverse Baseline (see Fig. 1) [7]) has been proved by Wu and Feng [7] who have exhibited one to one mappings of the nodes between each network and the Baseline network.

Another approach consists in modeling the networks by graphs or directed graphs. Such an approach was considered by Agrawal in [2] (see also [1]). He proposed a characterization of this class by "Buddy Properties"; unfortunately, the assertion of Theorem 1 of [2] is not sufficient to prove equivalence as it has been stated in [5].

Kruskal and Snir [10] within the graph theory framework, used a labeling schemes to describe routing in the network. They defined a network isomorphism as a graph isomorphism, which furthermore preserves the vertices labels. They obtained a sufficient condition, called bidelta property, to insure that a network is isomorphic, in their sense, to the classical ones.

Extending Agrawal's property, we obtain a graph theoretical characterization of topologically equivalent networks [4] using connected components of families of subgraphs which is unfortunately difficult to apply to the networks definition. The aim of this paper is to show the relation between our graph characterization and the usual definitions of Multistage Interconnection Networks using a set of permutations.

In section II, we introduce the notations, and we state the characterization in terms of graph theory. Section III is devoted to the study of Independent Connection : our link between graph theory and networks definition using permutations. In section IV we consider the PIPID field, an useful set of permutations which allow the construction of the six "classical" networks. PIPID permutations on N symbols are defined by a permutation of the index digit of the binary representation of these symbols. We show that PIPID permutations used to built Banyan networks may easily be modeled by independent connections. The main result of this paper is to establish that banyan networks built with PIPID permutations are topologically equivalent to the Baseline network. Note that the six networks studied by Wu and Feng [7] are designed using a subset of PIPID and that such a design allow an efficient bit directed routing.

2. A Graph Characterization

Interconnection networks may easily be modeled by directed graphs (digraphs) in which nodes represent the switching cells and arcs the communication links. We do not add extra nodes for the inputs and the outputs of the network as they do not play any role in the graph isomorphism.

Let C be a set of nodes, we will denote by $\Gamma^+(C)$ the set of children of nodes in C, and by $\Gamma^-(C)$ the set of parents of nodes in C.

A multistage interconnection digraph (MI-digraph) with n stages is a digraph whose nodes are partitioned into n ordered stages. We denote by V_i the nodes of the i^{th} stage. There are arcs only from nodes of the i^{th} stage to nodes of the $(i+1)^{th}$ stage (i.e. from V_i to V_{i+1}). The nodes are of indegree 2 and outdegree 2 except the nodes from the first and the last stage.

With this definition, we say that two multistage interconnection networks are topologically equivalent if and only if their MI-digraphs are isomorphic. Two digraphs are isomorphic if and only if there exists a bijection from the nodes of the first digraph into the nodes of the second digraph, which preserves the relationship of adjacency.

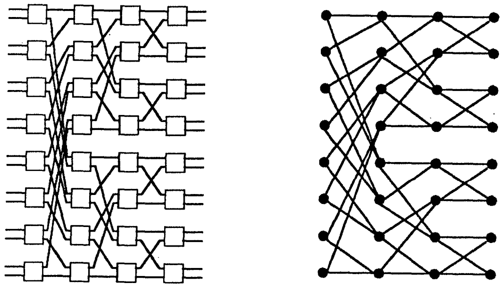


Fig. 1 : Baseline Network and Baseline MI-digraph

Remark : In all figures, the arcs are directed from the left to the right.

Banyan Property Definition : One minimal requirement is to allow a connection between any pair of input and output nodes. We say that a network has the Banyan Property if and only if for any input and any output there exists a unique path connecting them.

Definition : The connected components of an MI-digraph are those of the undirected underlying graph, obtained from the digraph by deleting the arcs orientation.

Definition : We denote by $(G)_{i,j}$ the subgraph of G that contains the vertices of the stages from i to j : $V_i \cup V_{i+1} \cup \cdots \cup V_j$

P(i,j) Property Definition : We say that an MI-digraph with n stages satisfies the P(i,j) property for $1 \le i \le j \le n$ if and only if the subdigraph $(G)_{i,j}$ has exactly $2^{n-1-(j-i)}$ connected components. And we say that an MI-digraph satisfies property P(1,*) if and only if it satisfies P(1,j) for every j such that $1 \le j \le n$. Similarly it satisfies property P(*,n) if and only if satisfies P(i,n) for every i.

This research is supported by a C.N.R.S. C^3 grant for the REGAL project.

Using this notations, the next theorem states the weakest condition of topological equivalence for multistage interconnection networks.

Theorem : All the MI-digraphs with n stages satisfying the Banyan property P(*,n) and P(1,*) are isomorphic. Although these properties are easy to check, the proof is too long to be included here; it will appear in [4]. The proof is done by induction, using the left and right recursive construction of the Baseline to design the isomorphism.

The assumptions of the theorem are very easy to check numerically using breadth first search algorithm to compute the number of connected components and the number of nodes at distance k. Unfortunately, these conditions are hardly related to numerical definitions of multistage interconnection networks (i.e. the permutations realized at each stage). For instance, the Omega network is defined by n perfect shuffles, and it is not obvious to understand why this type of definition implies the P(1,*) and P(*,n) topological properties.

In the next section, we define independent connections as a pair of mappings satisfying numerical constraints. We prove, using the former theorem, that banyan graphs built with these connections are isomorphic to the Baseline MI-digraph. Furthermore, we show that the set of permutations on N symbols, defined by a permutation of the binary digits of the symbol representation, may easily be associated to independent connections. Like the perfect shuffle, permutations used to design multistage interconnection networks often exhibit this property, and the equivalence relationship between "classical" networks becomes obvious.

3. Independent Connection

As we consider networks defined in term of permutations on N symbols, we add a labeling of the nodes in the graph. At each stage, nodes are labeled from 0 to $N-1=2^{n-1}-1$, following the natural order of the drawing (Fig. 2). The label of a node is a $n-1$ tuple $(x_{n-1},..,x_1)$, in base 2, so $(x_{n-1},..,x_1) \in [Z_{/2Z}]^{n-1}$. We consider the usual addition in the field $([Z_{/2Z}]^{n-1},+)$.

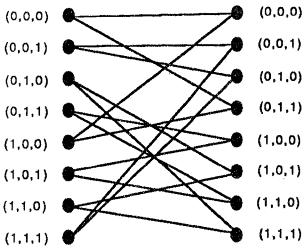


Fig.2 : Labeling of an MI-digraph

Now, we define connections and independent connections. The major result of this section is that banyan networks built with independent connections are topologically equivalent.

Consider a MI-digraph G and its subgraph $(G)_{i,i+1}$. Recall that this subgraph is a bipartite graph consisting in two consecutive set, V_i, V_{i+1} of nodes labeled in $[Z_{/2Z}]^{n-1}$, and a set of arcs from V_i to V_{i+1}.

Definition of a Connection : For all $i \neq n$, a connection (f, g) from the i-th stage of the MI-digraph G is a pair of functions f and g defined on $[Z_{/2Z}]^{n-1}$ such that, if x is a node of the i-th stage of G (i.e. V_i) then the two children of x in the i+1-th stage (i.e. V_{i+1}) are $f(x)$ and $g(x)$ (i.e. $\Gamma^+(x)=\{f(x),g(x)\}$).

Such a decomposition of the adjacency relationship exists as the outdegree of a node is always two, except in the last stage.

Definition of an Independent Connection : a connection (f,g) is independent if and only if

$$\left[\begin{array}{l} \forall\, \alpha \in Z2Z,\ \alpha \neq (0,..,0),\ \exists\, \beta \in [Z_{/2Z}]^{n-1} \text{ such that} \\ \forall\, x \in [Z_{/2Z}]^{n-1}, \\ f(x+\alpha) = \beta + f(x) \text{ and } g(x+\alpha) = \beta + g(x) \end{array}\right.$$

We exhibit in section IV some examples of independent connections.

Proposition 1 : A banyan graph built with independent connections satisfies the buddy property [2] (i.e. the interconnection pattern between nodes of consecutive stages is the $K_{2,2}$ graph).

Proof : Let x and y be two nodes in V_i, such that x and y share one neighbour in V_{i+1}. As connection (f,g) between V_i and V_{i+1} is independent we have

$$h_1(x) = h_2(y).$$

where h_1 and h_2 are either f or g. Indeed x and y share one neighbour, but we do not know if this child is obtained by function f or g. Let us denote by α the difference between y and x, then there exists β

$$\left[\begin{array}{l} h_1(x+\alpha) = h_1(y) = h_1(x) + \beta \\ h_2(x+\alpha) = h_2(y) = h_2(x) + \beta \end{array}\right.$$

Then $h_2(x) = h_2(y) - \beta = h_1(x) - \beta = h_1(y)$.
Hence x and y share two neighbours in row V_{i+1}, and the MI-digraph satisfies the Buddy Property. Following Agrawal in [2], we define x and y as buddy nodes. Note that $f(x)$ and $f(y)$ are buddy nodes two. Indeed buddy nodes share two children or two parents.

We give now a definition and a technical lemma that help us to prove one of the assumptions of theorem 1 : the P(1,*) property.

Definition of a translated set : Let A be a subset of V_i, and v a vector in $[Z_{/2Z}]^{n-1}$, we call the v-translated set of A, the set of nodes $\{a_i + v\}$ when a_i takes all values in A.

Lemma 2 : Consider an independent connection (f,g). Let A be a subset of V_i such that the number of nodes in both A and $\Gamma^-(A)$ is 2^k. Let v be an arbitrary vector of $[Z_{/2Z}]^{n-1}$. If B is a v-translated set of A, then $\Gamma^-(B)$ has 2^k nodes and is a translated set of $\Gamma^-(A)$.

Proof :

As A and $\Gamma^-(A)$ have the same number of nodes, all nodes of A are buddy nodes. Similarly all nodes in $\Gamma^-(A)$ are buddy nodes. Let a^1 and a^2 be two buddy nodes in A, and b^1 and b^2 their v-translated nodes. We first show that b^1 and b^2 are buddy nodes too. We have,

$$\left[\begin{array}{ll} f(b^i) = f(a^i + v) = f(a^i) + w & \text{for } i{=}1{,}2 \\ g(b^i) = g(a^i + v) = g(a^i) + w & \text{for } i{=}1{,}2 \end{array}\right.$$

as a^1 and a^2 are buddy, we have

$$h_1(a^1) = h_2(a^2)$$

where both h_1 and h_2 are either function f or function g. Therefore :

$$h_1(b^1) = h_1(a^1) + w = h_2(a^2) + w = h_2(b^2)$$

Hence b^1 and b^2 are buddy nodes.

Now, we terminate the proof by showing that $\Gamma^-(B)$ is a translated set of $\Gamma^-(A)$. Let x be a node in A, let y be the v-translated of x, let w (resp. z) be a parent of x (resp. y). We have :

$$h_1(z) + v = h_2(w)$$

where both h_1 and h_2 are function f or g.

Then, let u be an arbitrary point of $\Gamma^-(B)$, and $\alpha = u - w$. We prove that node $u + z - w$ is in $\Gamma^-(A)$.

$$h_2(u) = h_2(w + \alpha) = h_2(w) + \beta(\alpha) = h_1(z) + v + \beta(\alpha)$$

According to the definition of an independent connection we have :

$$h_1(z + \alpha) = \beta(\alpha) + h_1(z)$$

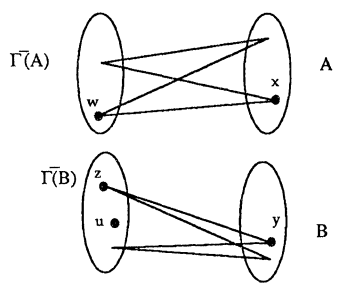


Fig. 3 : Construction of the sets

Therefore,

$$h_1(z + \alpha) = h_2(u) - v$$

For any node u in $\Gamma^-(B)$, nodes $\{\Gamma(u) - v\}$ are in A. Therefore, node $z + \alpha$ is a node in $\Gamma^-(A)$ and node u is $w-z$-translated node in $\Gamma^-(A)$. This is true for any node u, so $\Gamma^-(B)$ is a $(w-z)$-translated set of $\Gamma^-(A)$.

■

Lemma 3 : A banyan MI-digraph built with independent connections satisfies the P(1,*) property.

Proof : the proof proceeds by induction.

- Lemma 1 proves that such an MI-digraph satisfies property P(1,2). Indeed, Buddy Property and P(1,2) property are equivalent. We prove in the following that $P(1,j)$ implies $P(1,j+1)$ under the assumptions of Lemma 3.
- Let x be a node of V_1 and K be the connected component of $(G)_{1,j}$ containing x. Let Z be the set of children of x in V_{j+1}, and A_i be the intersection of K and V_i, for all i, $1 \le i \le j$. According to the induction hypothesis, the number of nodes in A_i is 2^{j-1}. As the graph is banyan, each node of Z has only one parent in A_j and each node of A_i has two children in Z.

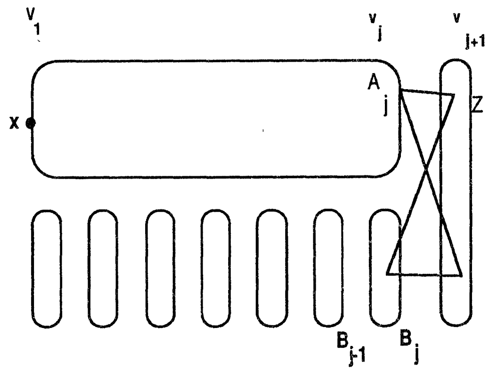


Fig. 4 : Construction of G

Let B_j be the set of buddy nodes of A_j. We will prove that B_j is a translated set of A_j. Let a^1 and b^1 be two arbitrary buddy nodes in $A_j \times B_j$ and let y be an arbitrary node in A_j.

$$\left[\begin{array}{l} \alpha = y - a^1 \\ f(a^1 + \alpha) = f(a^1) + \beta \\ g(a^1 + \alpha) = g(a^1) + \beta \end{array}\right.$$

As a^1 and b^1 are buddy nodes, we have

$$h_1(a^1) = h_2(b^1)$$

where both h_1 and h_2 are function f or g. Furthermore

$$h_2(b^1 + \alpha) = h_2(b^1) + \beta = h_1(a^1 + \alpha)$$

Hence $b^1 + \alpha$ and $a^1 + \alpha$ are buddy nodes, and B_j is a $(b^1 - a^1)$-translated set of A_j.

Then, we apply Lemma 2 twice on A_j, and B_j. Indeed, according to the buddy property, the sets A_j and A_{j-1} have the same number of nodes, and according to the banyan property we have :

$$\Gamma^-(A_j) = A_{j-1}$$

As we denote by B_{j-1} the set $\Gamma^-(B_j)$, Lemma 2 implies that this set is a translated set of A_{j-1} and has the same number of nodes than A_{j-1} (i.e. 2^j). We define B_k as the set $\Gamma^-(B_{k+1})$. We can now apply Lemma 2 on A_{j-1} and B_{j-1} to show that the set B_{j-2} has the right number of nodes. By induction we prove this property on every set B_k.

- As the connected component of $(G)_{1,j+1}$ containing Z is exactly Z and K and the union of B_k sets, we have shown that $(G)_{1,j+1}$ has 2^{j+1} nodes per stage. Hence the graph G satisfies the P(1,j+1) property.

■

Similarly we prove by induction the following lemma :

Lemma 4 : A banyan MI-digraph G built with independent connections satisfies the P(*,n) property.

Indeed Lemma 1 proves that G satisfies property P(n-1,n). We prove that the property P(j,n) implies P(j-1,n) under the assumption of the Lemma. We apply the same technique than in Lemma 3 : we decompose a connected component of $(G)_{j-1,n}$ in a connected component of $(G)_{j,n}$ called K, the parents of K in V_{j-1} called Z, and the others children of Z in V_j. Then, we prove that at each stage, this set of children has the right number of nodes. The proof is too long to be included here.

■

According to our graph characterization, we can state the announced result whose corollary are developed in the next section. We consider in the following section some connections defined by a permutation of the digital representation (i.e. the tuple $(x_{n-1},..,x_1)$) and we show the relations between these connections and the PIPID set of permutations. Fortunately enough, these connections are independent connections, allowing us to use our main theorem :

Theorem 5 : A banyan MI-digraph built with independent connections is topologically equivalent to Baseline MI-digraph.

■

4. Pipid Permutations

Consider now a labeling of the links of the network at the inputs and outputs of all cells following the natural order of the drawing. A label is a number between 0 and $N-1$ whose binary representation is denoted by $(x_{n-1},..,x_1,x_0)$. Each link is defined by two labels and each stage is defined by a permutation of these N labels.

Multistage interconnection networks have been often defined using these permutations and functional properties have been derived from this model [13]. For instance, the Omega network is defined as n stages of perfect shuffle. A perfect shuffle σ may be defined as a circular left shift of the binary representation of the operand. Similarly, the k-subshuffle σ_k, the k-butterfly, β_k, and the bit reversal, ρ, are easily defined by permutations on the bits of the number representation (see [9] for more definitions). These permutations have been used to design the six networks studied by Wu and Feng and one may ask if this scheme of construction is the reason of the networks topological equivalence.

Consider numbers from 0 to $N-1$ and their binary representation $(x_{n-1},..,x_1,x_0)$. Following Lenfant [12], we define PIPID permutations on these numbers, by a permutation on the index of the representation.

$$\left[\begin{array}{l} \lambda \in PIPID\,(N=2^n) <\!\!-\!\!> \exists\ \theta \text{ permutation on n symbols such that} \\ \lambda(x_{n-1},..,x_1,x_0) = (x_{\theta(n-1)},..,x_{\theta(1)},x_{\theta(0)}) \end{array}\right.$$

Perfect shuffle, bit reversal and butterfly are examples of Permutations Induced by a Permutation on the Index Digits (PIPID). We prove in the following that these permutations are also associated to a family of very simple independent connections.

Compare now the label of the node or a cell used in section III and the labels of the links connected to the outputs of this cell as stated in the beginning of this section. One can obviously remark that the $n-1$ first bits of a link label are exactly the binary representation of the incident node label.

Let λ be an arbitrary permutation of PIPID used to design a stage of a network, and let θ be the associated permutation of the index. Let x be a node or cell label, $x = (x_{n-1},..,x_1)$. The links connected to this cell are labeled :

$$\left[\begin{array}{l} y^0 = (x_{n-1},..,x_1,0) \\ y^1 = (x_{n-1},..,x_1,1) \end{array}\right.$$

Applying permutation λ on these two labels give the two labels of the links (z^0,z^1) in the next stage.

$$\left[\begin{array}{l} z^0 = \lambda(y_0) \\ z^1 = \lambda(y^1) \end{array}\right.$$

Let $k = \theta^{-1}(0)$ and $m = \theta(0)$. We have

$$\left[\begin{array}{l} z^0 = (x_{\theta(n-1)},..,x_{\theta(k+1)},0,x_{\theta(k-1)},..,x_{\theta(1)},x_{\theta(0)}) \\ z^1 = (x_{\theta(n-1)},..,x_{\theta(k+1)},1,x_{\theta(k-1)},..,x_{\theta(1)},x_{\theta(0)}) \end{array}\right.$$

And, if we consider only the $(n-1)$ first digits, we obtain the labels of the cells connected to cell x :

$$\left[\begin{array}{l} (x_{\theta(n-1)},..,x_{\theta(k+1)},0,x_{\theta(k-1)},..,x_{\theta(1)}) \\ (x_{\theta(n-1)},..,x_{\theta(k+1)},1,x_{\theta(k-1)},..,x_{\theta(1)}) \end{array}\right.$$

Now, we have to identify the two mappings f and g and to check that the connection (f,g) satisfy the independence property.

Note that we had supposed in the former equations that k is not zero. Indeed, we can give up this particular case as such permutations are not useful to build banyan networks. If k is zero, then there are two links between the cells, and the graph do not obviously satisfy the banyan property.

Let us suppose that $k \neq 0$ and let ϕ denote the following permutation on $[1..n-1]$.

$$\left[\begin{array}{l} \forall\, i \neq m \quad \phi(i) = \theta(i) \\ \phi(m) = k \end{array}\right.$$

To compute the labels of the cells connected to cell x, one just have to apply the permutation ϕ on the index of the binary representation of x and force to 0 or 1 the k^{th} bit. We suggest to use the following two functions f and g to design an independent connection which realize this operation. We define the two functions by their projections f_i and g_i, for all i, $1 \le i \le n-1$.

$$\left[\begin{array}{l} f_i(x) = x_{\phi(i)} \quad \forall\ \phi(i) \neq m \\ g_i(x) = x_{\phi(i)} \quad \forall\ \phi(i) \neq m \\ f_k(x) = 1+x_m \\ g_k(x) = x_m \end{array}\right.$$

Therefore, the connection (f,g) is independent. Let α be an arbitrary non zero vector, we obtain the vector β by applying the permutation ϕ on the index of the binary representation of α.

So, we can associate independent connections to the PIPID permutations used to build banyan networks. We have now an easy to check sufficient condition of equivalence with the Baseline network : all banyan multistage networks built with PIPID permutations are topologically equivalent to the Baseline network. As Omega, Baseline, Reverse Baseline, Flip, Indirect Binary Cube and Modified Data Manipulator networks are designed using PIPID permutations, they are topologically equivalent.

5. Conclusion

We stated a characterization of Baseline equivalent networks using a graph model of multistage interconnection networks. As this characterization is difficult to apply to networks defined by permutations, we design a new tool, independent connections. The independence property is a numerical constraint on the adjacency relationship. We derived from this constraint a characterization of permutations (PIPID) which can be used to build equivalent networks. As these permutations are associated to a very simple bit directed routing, they have been used to design most of the multistage interconnection networks presented in the literature. Note that the results obtained here apply only to networks built with 2×2 switching cells, whereas our graph characterization have been generalized to arbitrary size of cells. Finally, we hope that this approach will be useful to study others topological or functional properties of multistage interconnection networks.

References

1. Agrawal, D.P. and Kim, S.C., "On non-equivalent multistage interconnection networks," *Proc. Int. Conf. Parallel Processing*, pp. 234-237, 1981.
2. Agrawal, D.P., "Graph theoretical analysis and design of multistage interconnection networks," *IEEE Trans. Computers*, vol. C32, pp. 637-648, Jul. 1983.
3. Batcher, K.E., "The flip network in Staran," *Proc. Int. Conf. Parallel Processing*, pp. 65-71, Aug 1976.
4. Bermond, J.C., Fourneau, J.M., and Jean-Marie, A., "A graph theoretical approach to equivalence of multistage interconnection networks," *Rapport LRI 242*, Nov 1985. to appear in Discrete Applied Math.
5. Bermond, J.C., Fourneau, J.M., and Jean-Marie, A., "Equivalence of multistage interconnection networks," *Inf. Proc. Let.*, vol. 26, pp. 45-50, Sept. 1987. Rapport LRI 217
6. Feng, T., "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Computers*, vol. C23, pp. 309-318, Mar. 1974.
7. Feng, T. and Wu, C., "On a class of multistage interconnection networks," *IEEE Trans. Computers*, vol. C29, pp. 694-702, Aug. 1980.
8. Feng, T. and Wu, C., *Tutorial : Interconnection networks for parallel and distributed processing*, IEEE Publications, 1984.
9. Hockney, R.W. and Jesshope, C.R., *Parallel Computers*, Adam Hilger Ltd, 1981.
10. Kruskal, C. P. and Snir, M., "A unified theory of interconnection network structure," *Th. Comp. Sci.*, vol. 48, no. 1, pp. 75-94, 1986.
11. Lawrie, D.H., "Access and alignment of data in an A.P.," *IEEE Trans. Computers*, vol. C24, pp. 1145-1155, Dec. 1975.
12. Lenfant, J. and Tahe, S., "Permuting data with the omega network," *Acta Informatica*, vol. 21, pp. 629-641, 1985.
13. Parker, D.S., "Notes on shuffle/exchange type networks," *IEEE Trans. Computers*, vol. C29, pp. 213-222, Mar. 1980.
14. Pease, M.C., "The indirect binary cube microprocessors array," *IEEE Trans. Computers*, vol. C26, pp. 458-473, May 1977.

NONUNIFORM TRAFFIC SPOTS (NUTS) IN MULTISTAGE INTERCONNECTION NETWORKS

Tomas Lang and Lance Kurisaki
Computer Science Department
University of California, Los Angeles

Abstract

The performance of multistage interconnection networks with blocking switches is degraded when the traffic pattern produces nonuniform congestion in the switches, that is, when there exist nonuniform traffic spots (NUTS). For some specific patterns we evaluate this degradation in performance and propose modifications to the network organization and operation to reduce the degradation. Successful modifications are the use of diverting switches and the extension of the network to include alternate paths. The use of these modifications to the basic blocking policy for control of contention makes the network more effective for a larger variety of traffic patterns.

1. Introduction

Multistage interconnection networks (MIN) are used in multiprocessor systems to connect processors with other processors or with memory modules. These networks provide a compromise between networks of low latency and high cost, such as the crossbar, and networks of high latency and low cost, such as the shared bus. Moreover, MINs can be pipelined to provide a bandwidth comparable to that of the crossbar for suitable traffic patterns. In addition, the control of routing is simple. A large body of work has been done on the structure, operation, and performance of these networks; a comprehensive reference is [1]. These networks were initially introduced for use in array computers of the SIMD type; in this context the interconnection networks are sometimes called permutation networks. More recently, they are being proposed and used in multiprocessors of the MIMD type, especially of the shared-memory variety [2]. In this paper we are concerned with this second type of use.

MINs, in their basic form, provide a unique path between any source-destination pair. However, the paths for different pairs are not disjoint and, therefore, conflicts might occur when simultaneous communication is established between several source-destination pairs. The basic method used to handle this problem is to use a packet-switched type of operation and to buffer the packets in the switches. Blocking occurs whenever the buffers become full.

It has been shown that the performance of these networks is satisfactory for uniform traffic [3]. More recently, several studies [4] have indicated that the performance of the network is degraded significantly when the traffic includes hot-spot traffic, that is, when each source generates a larger fraction of the traffic to one particular destination. This type of traffic occurs because of access to shared variables, such as semaphores. To overcome this degradation, a network with combining switches has been proposed.

The topic of this paper is a more general type of nonuniform traffic, in which there is no concentration of the traffic to one destination, but the traffic is not uniformly distributed among the switches, producing nonuniform traffic spots (NUTS). We illustrate some typical cases of this type of traffic and show the degradation in network performance produced by them. We then explore solutions to reduce this performance degradation.

Of course, in this case the use of combining switches is not a solution since the contention packets do not necessarily have the same destination. We show that randomization of the traffic, proposed for reducing contention in multicomputers [5], is not suitable either. As positive alternatives to improve the performance, we consider the use of diverting switches, with several diverting policies, and networks with alternate paths. Because of the reduction in degradation produced, the proposed modifications to the basic network with blocking make the multistage network suitable for a larger variety of multiprocessor applications.

The performance of the proposed solutions is evaluated by simulation. The objective of this evaluation is to show that, under reasonable conditions, performance of the original network with blocking switches is badly degraded by the presence of NUTS and that the modifications proposed significantly reduce this degradation. On the other hand, it is not our objective to give an extensive set of graphs from which the performance of particular networks with specific traffic patterns can be determined. Consequently, we select a set of reasonable network parameters and traffic patterns and use these for the simulation. More detail can be found in [6].

2. Multistage Network Structure and Operation

We now give a brief description of the structure and operation of the multistage network, emphasising the assumptions we make. A more detailed discussion can be found in [1]. The type of multistage interconnection network we are considering has $N = 2^n$ inputs (sources) and outputs (destinations). It consists of n stages of $N/2$ 2×2 switches, as shown in Figure 1. The outputs of stage-i switches are connected to the inputs of stage-$(i-1)$ switches, with the network inputs going to stage-$(n-1)$ switches and the network outputs coming from the stage-0 switches.

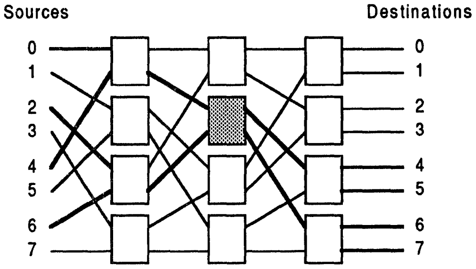


Figure 1. An 8×8 Omega Network

Several specific multistage networks have been proposed, differing in the interconnection pattern between stages. Since the characteristics, in terms of type of operation and performance, are similar for all these different topologies, we consider here the Omega network [7], which has been extensively studied [8] and is being used in several multiprocessor systems.

The routing of packets in the network is unique since there is a single path from a specific source to a specific destination. The control of routing is done using a destination tag associated with the message as part of each packet.

Since each output can send only one message per cycle, (the network is synchronous and pipelined) there is a conflict when both packets entering a switch in a cycle have to be routed to the same output. One solution to this conflict is to have a buffer for each output and to store the additional packet in such a buffer. Of course, these buffers are finite so it is necessary to have an operation policy when the buffer is full. The basic scheme used is a

blocking policy in which the predecessor switches do not send packets to a full buffer. To support this policy it is necessary to have signals from a switch to its predecessors indicating that the corresponding buffer(s) is full (Figure 2). Note that since both predecessors can send messages to the same buffer, it is necessary to establish a policy also for the case in which there is just one space in the buffer. In such a case, we select alternatively the predecessor that is blocked.

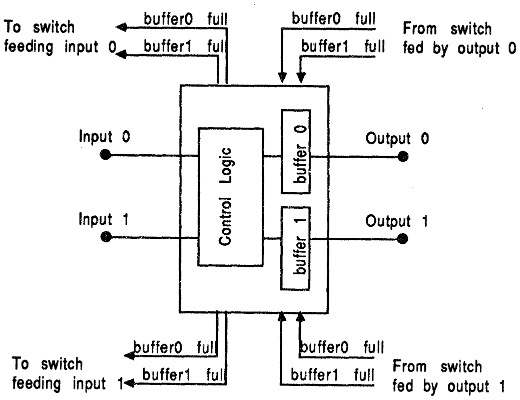


Figure 2. Blocking Switch With *FULL* Control Signals

In this paper we do not evaluate the different buffering organizations and policies. The degradation produced by NUTS is inherent to the blocking operation of the network, which is present for any of the buffer organizations and policies. Consequently, we perform our analysis using output buffers with FIFO policy. but the

When processors send request packets to remote memory modules, traffic in the opposite direction is also generated. These return packets must traverse an analogous network to reach the processors. The analysis of this type of traffic is similar to the request traffic, and is not considered here.

3. Performance evaluation by simulation

We now describe the measures that we will use to evaluate the performance of the network. We also indicate the types of traffic and network parameters considered. As discussed in the introduction, we select a reasonable set of parameters and perform simulations to compare the performance for the original network and for the modifications proposed.

Of importance in our study are the different **traffic patterns** used, since the degradation due to nonuniform traffic spots (NUTS) and the applicable solutions depend on the traffic patterns considered. In the next section we present the patterns used.

In addition to the traffic pattern, the **traffic load** is of importance. We distinguish two types of systems: open and closed systems. In an **open** system, each processor generates a packet each r cycles, so that the load is specified by the fraction $1/r$. In a **closed** system, on the other hand, the load is defined by the maximum number of outstanding packets. We have found that the results are qualitatively similar for both cases for the same total throughput. Consequently, to concentrate on significant parameters, we only report on results for open systems.

We evaluate the **steady-state** behavior of the system, that is, we assume that the traffic pattern under consideration remains for a period long enough to achieve this steady state.

The fundamental parameters for the network are its size and the size of the queues. We have found that the relative performance of the network remains essentially the same for different values of these parameters. Consequently, we report our results for a network of size 64 and queues of size 2, which are also convenient because they produce a relatively small delay, except for the blocking case where, because of the way the full signal is generated, this size of queue is not adequate. In this latter case, we use a queue of size 4.

The main performance measures of interest are the **throughput** of the network in packets/cycle, and the **average delay** of the packets. The maximum throughput is of N packets per cycle and the minimum delay is of n cycles. This performance is obtained when there are no conflicts, that is, when in all cycles all switches receive two packets and route one to each output. For other cases, the performance is shown by the function **delay vs. throughput.**

In a multiprocessor system all processors cooperate in the execution of a task and have to synchronize periodically. Consequently, it is convenient for all processors to advance at a uniform pace, so that processors do not have to wait unnecessarily for slower processors. The measure we use to evaluate the relative advance of the processors is the **distribution of throughput.**

For the simulations we built a network simulator using as a basis SIMON, a general-purpose multiprocessor simulator developed at the University of Utah [9].

Several studies have been reported on the performance of multistage interconnection networks with **uniform traffic** [3]. The results of our simulations for uniform traffic confirm what previous studies have indicated.

4. Traffic patterns producing NUTS

The evaluation studies that have been made for the "hot spot" problem point to a more general situation with nonuniform traffic. The same type of degradation should occur whenever the traffic is such that one or more switches carry a larger fraction of the total traffic than its share. This degradation is due to the same "tree saturation" effect observed in the hot-spot case. In the context of the Omega network, switch i of stage j carries the traffic going from a specific subset of 2^j sources to a specific subset of destinations (Figure 1). Consequently, switch congestion occurs whenever this traffic is excessive. This can occur even in situations in which the fraction of traffic going to each destination is the same. The main objective of this research is to identify the traffic patterns that produce non-uniform traffic spots (NUTS), to evaluate the degradation in performance, and to propose and evaluate solutions to this problem.

To study the influence of NUTS on performance we have considered two types of traffic as follows. These types are just examples to illustrate the problem and evaluate the solutions; they correspond to situations that could occur, but are not specific practical patterns.

Traffic of Type I.

In the first type, each source issues all its requests to one destination and no pair of sources sends to the same destination. In the shared memory case, this type of pattern models a system in which each processor has a preferred memory module that contains both the code and the data for that processor. It might be argued that in such a case it would be better to assign to each processor a local memory module with direct access without going through the network (this is the scheme used, for example, in the BBN Butterfly). However, the use of the network to have a uniform access time from any processor to any memory module, permits a flexible dynamic scheduling approach that is not possible in the local scheme. In this dynamic scheduling model, a processor can have its code/data in any memory module, and this module can vary with time. This type of traffic would model also situations in which the communication is among pairs of processors.

Some specific instances of this type of traffic patterns produce significant NUTS while others do not. We use two different instances for our study. In instance 1, we use a **bit-reversal** permutation which is known to produce a large contention in the

Omega network as evident from the switch positions shown in Figure 3. This is an extreme case, it shows a lower bound on the improvement that can be achieved with the techniques used. To model a more typical situation, as instance 2 we generated an **arbitrary** permutation.

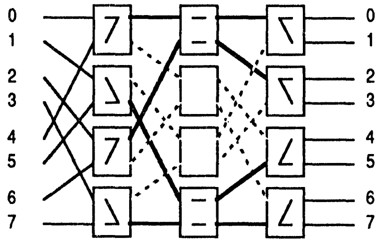
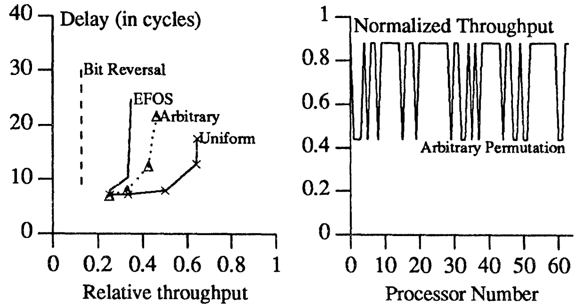


Figure 3. Switch Positions for the Bit Reversal Permutation

The throughput-delay for these patterns is shown in Figure 4. As can be seen, there is significant degradation in performance, as compared with the uniform traffic case.



(4) Blocking switches (5) Throughput Distribution

Figures 4 and 5. Simulation Results for Blocking Switches

Moreover, in case of the arbitrary permutation there is a large variation between the throughput of the different processors (Figure 5). As mentioned before, this is not desirable when the processors are cooperating in a single task.

Traffic of Type II.

The second traffic pattern we consider consists of requests going from even numbered sources to destinations in the first half and from odd numbered sources to destinations in the second half (EFOS). This pattern serves to illustrate a case in which each source accesses a subset of destinations. In this case, there are also NUTS. The performance of the net is shown in Figure 4.

We conclude from these simulations that the performance of the network is badly degraded by the NUTS, with respect to the performance for uniform traffic. We now explore ways to reduce this degradation.

5. Unsuccessful solutions: randomization and discarding

We now report on randomization and discarding, two approaches to reduce the degradation due to NUTS which turned out to be unsuccessful.

Randomization

As a first solution to the degradation due to NUTS, we consider the use of *randomization* of the traffic. In this approach, proposed previously to handle load imbalances in routing of multicomputers [5, 10], packets are first sent to random destinations and then rerouted to their final destinations. This scheme has the effect of making the traffic pattern uniform and, therefore, of eliminating the added congestion of nonuniform traffic.

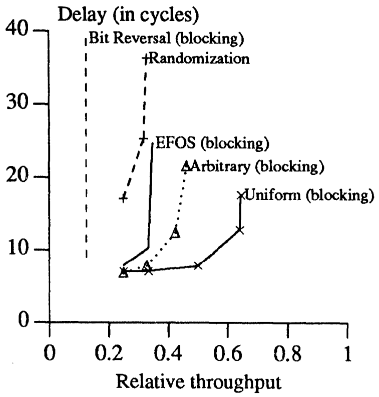


Figure 6. Randomization

In the context of multistage networks, the use of this scheme implies that all messages make two passes through the network. This has the two negative effects of doubling the minimum delay and reducing the effective throughput to half, because of the additional traffic through the net produced by rerouting. The results of simulations for the two types of traffic described in the previous section are shown in Figure 6, which exhibits the expected throughput and delay. As seen from there, randomization produces a relatively small improvement for the extreme bit-reversal case, while it is detrimental for the others.

Discarding Switch

Another solution we considered was the use of discarding switches. Switches of this type resolve congestion by discarding overflow packets. The original source of the packet is made aware of the status of the packet, through an explicit signal from the switch or a timeout mechanism, and retransmits it. Note that this requires the source to buffer all outstanding packets until an acknowledgement is received from the destination. The switches also require the ability to signal the appropriate source that a particular packet was discarded. This requires additional interconnect and more complex control. This type of switch is used in the Butterfly Parallel Processor to deal with contention in the network and avoid "tree saturation" [11].

The simulations show that, for the traffic patterns considered, there is no improvement with respect to the network with blocking switch. This can be explained by the fact that the discarded traffic is reissued by the same processor as the first time and, therefore, follows the same path leading to the NUTS.

6. Diverting Switch

In a *diverting switch* the messages in front of the buffers are always sent to the successors, irrespective of whether there is space for them in the corresponding destination buffers. If both messages that arrive to a switch go to the same output buffer and there is no space for both, then one of the messages is diverted to the other buffer of the switch (Figure 7). Note that there is always at least space for one message in each buffer since one message departs from each buffer in every cycle. Of course, the diverted message will go to a wrong destination (since there is just one path in the network for each source/destination pair); therefore, the message will have to be resent into the network to the correct destination. Consequently, this mode of operation requires a connection between each network output and corresponding input (a wrapped-around organization).

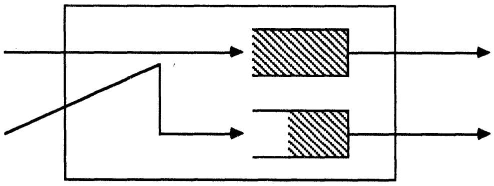


Figure 7. Behavior of the Diverting Switch

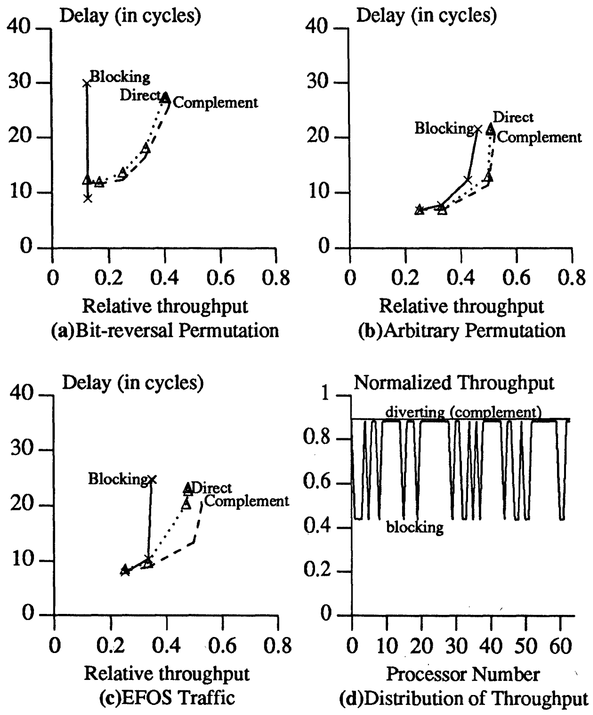
Diversion has potentially a better performance than discarding because the packets are rerouted from a source that is **different** from the original source. This makes it possible for the message to avoid the NUTS in the second pass.

Since to obtain a good performance it is convenient to reduce the number of packets that are diverted, whenever a conflict occurs and one of the packets in the conflict has already been diverted (in that pass through the network) we give preference to the nondiverted packet (to go to the correct destination).

Because of the diversions, a message might traverse the network several times before getting to its destination. A possible problem with this form of operation is that it is not possible to assure that a particular message will have a bounded delay. To avoid this and make the delay more uniform, we give *preference to older packets.*

Diverting policies

Once a packet is diverted in the network, it cannot reach its desired destination during that pass; it has to go through the network again. This means that we now have a great deal of freedom in deciding where to route these diverted packets. The main goal is to route the diverted packet to an interim destination that has a "clear" path to the true destination, so that it will not be diverted again.



(a)Bit-reversal Permutation

(b)Arbitrary Permutation

(c)EFOS Traffic

(d)Distribution of Throughput

Figure 8. Simulation Results for Diverting Switches

We have experimented with several diverting policies. We present here the results of two of them, to show that diverting produces an improvement in the performance and that the specific diverting policy has an impact.

The first diverting policy we call *direct diverting.* In it the routing of the diverted message continues using the destination tag. That is, each time the message is diverted, the actual destination is wrong in the corresponding bit. As shown in Figure 8 the performance is significantly better than with the blocking policy.

The second diverting policy we call *complement diverting.* In this case, once a message is diverted, instead of using the destination tag for routing, it is routed using a tag corresponding to the complement of the source. On its next pass through the network, the original destination tag is again used. This policy has the advantage that it assures that the rerouted message will avoid the NUTS where it was diverted in the first pass. Of course, it can pass through some other NUTS.

Figures 8(a-c) show the corresponding performance for the various traffic patterns. We see that this policy produces a somewhat better performance than the direct policy.

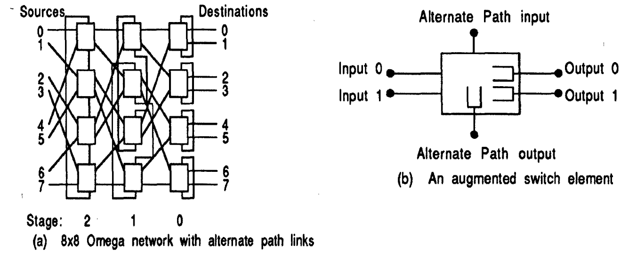
These simulation results indicate that the use of diverting switch improves the throughput-delay characteristic of the network when the traffic produces NUTS. Moreover, the use of diverting switches makes the distribution of throughput more uniform, as shown in Figure 8(d).

7. Network with alternate paths

The MIN's previously considered have the characteristic of a unique path between each source-destination pair. Several reports [12, 13, 14] have described adding redundant paths to MIN's to improve fault tolerance characteristics. These alternate paths can also improve the performance of a fully functional network.

In particular [13] and [14] propose the addition of links to connect switches in the same stage into rings so that from any switch in a particular ring packets can reach the same subset of destinations. The application of this technique to the OMEGA network is illustrated in Figure 9(a). If a packet entering a switch finds the desired output queue full, it can be re-routed to another switch in the same group via the alternate path link, and still be able to reach its true destination directly without a second pass through the network.

This modified OMEGA network requires augmented switches acting as a 3×3 crossbar, as shown in Figure 9(b). The routing control is somewhat more complex than for the original 2×2 switch. We still use a diverting policy, because this has given better performance for the original network and this policy is also simpler to control since no "full signals" are needed. Each cycle up to three packets enter the switch. They are placed in the output queues giving priority to the older packets that have not been diverted (in that pass). The highest-priority packet is always placed in the correct queue, since there is always at least one space in each queue (because one packet leaves each queue each cycle). The next packet is placed in the correct queue, if there is space, or in the alternate queue. Finally, the least-priority packet is placed in the correct queue, in the alternate queue, or in the wrong queue (diverted).



(a) 8x8 Omega network with alternate path links

(b) An augmented switch element

Figure 9. An Alternate Path Network and Switch Element

Figure 10 shows the performance of the network with alternate paths for two of the traffic patterns. We can see that the introduction of alternate paths produces a significant reduction in the degradation due to NUTS.

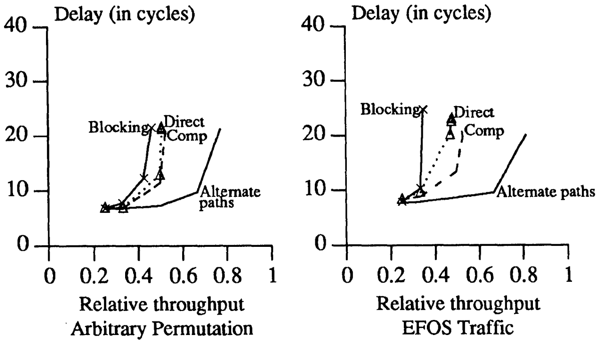


Figure 10. Throughput-Delay Graph (alternate paths)

8. Conclusions

We have shown several traffic patterns that produce NUTS in multistage interconnection networks and therefore result in a degradation of performance. The randomization technique, proposed for eliminating imbalances of loading in multicomputers, is not appropriate in this case because it increases the delay of each packet and the real traffic through the network. The use of discarding switches is not advantageous either because the discarded traffic has to be resent through the same congested path.

As positive solutions, we have shown that diverting switches produce a significant reduction in the degradation. Moreover, the control of congestion is simpler than that for blocking switches because no "full signals" are needed. However, to implement this policy, it is necessary to have a network with wrap-around connections.

The performance is much better using networks with alternate paths. However, this network require 3×3 switches instead of the basic 2×2, which complicates the implementation.

The use of these modifications to the basic blocking policy in the control of contention in multistage interconnection networks makes it possible to use the network effectively for a larger variety of traffic patterns.

Acknowledgement

This research was sponsored in part by the State of California MICRO program and Hughes Aircraft Co.

9. References

[1] H. J. Siegel, "Interconnection Networks for Large-Scale Parallel Processing, Theory and Case Studies", Lexington Books, 1985.

[2] K. Hwang and F. Briggs, "Computer Architecture and Parallel Processing", McGraw Hill, 1984.

[3] D. Dias and R. Jump, "Packet Switching Interconnection Networks for Modular Systems", Computer, December 1981, pp. 43-54.

[4] G. F. Pfister and V. A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks", IEEE Transactions on Computers, vol. C-34, No. 10, October 1985, pp. 943-948.

[5] L.G. Valiant, "A Scheme for Fast Parallel Communication", SIAM J. Comput. Vol. 11, No. 2, May 1982, pp. 350-361.

[6] T. Lang and L. Kurisaki, "Nonuniform Traffic Spots (NUTS) In Multistage Interconnection Networks", UCLA Technical Report CSD-880001, January 1988.

[7] D.H. Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Transactions on Computers, Vol. C-24, No. 12, Dec. 1975, pp. 1145-1155.

[8] P. Chen, D. Lawrie, and P. Yew, "Interconnection Networks Using Shuffles", Computer, December 1981, pp. 55-64.

[9] R.M. Fujimoto, "The CSIMON Interface", Computer Science Department, University of Utah, 1986.

[10] D. Mitra, "Randomized Parallel Communications", Proc. of the 1986 Int. Conf. on Parallel Processing, pp. 224-230.

[11] R. H. Thomas, "Behavior of the Butterfly Parallel Processor in the Presence of Memory Hot Spots", IEEE Parallel Processing, 1986.

[12] G.B. Adams and H.J. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems", IEEE Transactions on Computers, Vol. C-31, No. 5, May 1982, pp. 443-454.

[13] V. P. Kumar and S. M. Reddy, "Design and Analysis of Fault-Tolerant Multistage Interconnection Networks With Low Link Complexity", 12th Annual Symposium on Computer Architecture, pp. 376-386 (June 1985).

[14] N. Tzeng, P. Yew, and C. Zhu, "A Fault-Tolerant Scheme for Multistage Interconnection Networks", 12th Annual Symposium on Computer Architecture, pp. 368-375 (June 1985).

[15] W. C. Brantley, K. P. McAuliffe, J. Weiss, "RP3 Processor-Memory Element", Proceedings of the 1985 International Conference in Parallel Processing, August 1985, pp. 782-789.

[16] A. Gottlieb, et al. "The NYU Ultracomputer--Designing an MIMD Shared Memory Parallel Computer", IEEE Transactions on Computers, Vol. C-32, No. 2, pp. 175-189.

[17] M. Kumar and G. Pfister, "The Onset of Hot Spot Contention", Proceedings of the 1986 International Conference on Parallel Processing, August 1986.

[18] C.P. Krustal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors", IEEE Transactions on Computers, Vol. 32, No. 12, December 1983, pp. 1091-1098.

[19] R. Lee, "On Hot Spot Contention", Computer Architecture News, Vol. 13, No. 5, December 1985, pp. 15-20.

[20] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", Proceedings of the 1985 International Conference in Parallel Processing, August 1985, pp.764-771.

[21] D. S. Rosenblum and E. W. Mayr, "Simulation of an Ultracomputer with Several 'Hot Spots' ", Stanford Technical Report STAN-CS-86-1119, June 1986.

On Self Routing in Beneš and Shuffle Exchange Networks*

Rajendra Boppana C. S. Raghavendra
Dept. of Electrical Engineering–Systems
University of Southern California
Los Angeles, CA 90089

Abstract

A self routing algorithm for passing Linear class of permutations in Beneš, π and $(2n - 1)$–stage shuffle exchange networks of $N = 2^n$ inputs/outputs is presented. In these networks, switches in the first $(n - 1)$ stages are set by comparing the destination tags of the inputs to the switch; switches in the remaining stages are set by the self routing Ω algorithm. Thus, the total time required for routing any Linear permutation is $O(n)$, same as the network delay time. The algorithm also routes Ω^{-1} permutations in Beneš and Ω permutations in π network trivially. The class of permutations that are routable by the algorithm is much richer than the class of Linear permutations. This algorithm routes all possible permutations for 4 input/output Beneš network $\mathcal{B}(2)$ (same as 3–stage shuffle exchange network) and π–network, since all the permutations are in the Linear Class.

1 Introduction

Typically, a parallel computer consists of a number of processors and an interconnection network for exchange of information between them as well as with memory modules. Considering a processor/memory network model, any processor should be able to communicate with any memory module which is called full access. To support SIMD type computations, ideally we would like the network to be able to perform all the permutations that allow simultaneous use of the memory modules. Such capabilities exist in crossbar networks and networks that are rearrangeable, for example the Beneš network.

We view parallel computing as computation steps—during which time some or all of the processors are busy computing, and communication steps—at which some permutation function is set up by the network to allow data exchanges. If the underlying network can not support a required permutation function then it has to be realized in multiple steps. The advantage with a rearrangeable network is that any permutation can be realized in one communication step. Further, if they are built using smaller switches such as 2 × 2, then they are relatively cheaper than crossbar networks. Therefore rearrangeable networks are used in some parallel computer implementations (e.g. GF–11 [1]).

A well known rearrangeable network is the Beneš network [2] which is built in a recursive manner using 2 × 2 switches, and is shown in figure 1. In such networks, it takes some time to set up the switches to realize a given arbitrary permutation. For an $N = 2^n$ inputs and outputs Beneš network, determining the switch settings to realize an arbitrary permutation takes $O(N \log N)$ time on a uniprocessor computer[7]. If the required permutations change frequently while computing a problem, the communication time may become a bottleneck. An approach to solve this problem is to compute the switch settings for a given permutation using a parallel computer with N PE's. A separate network with static links between the PE's in the parallel computer under consideration could be used for this computation as suggested by Nassimi and Sahni[6]. Alternatively, the Beneš network itself can be set to realize perfect shuffle permutation easily, to convert the parallel computer under consideration to a perfect shuffle computer and determine the switch settings in $O(n^5)$ time using the algorithm proposed by Nassimi and Sahni[5]. However, it still takes considerable amount of time to realize a permutaion compared to the propogation delay $O(n)$.

We are interested in developing fast self–routing algorithms for many useful permutations required in parallel processing, if not for all the $N!$ permutations. Due to the nature of techniques used in developing parallel algorithms, the permutaitons required are generally nice and regular and can be expressed as algebraic functions. Some work was done on developing self–routing algorithms for classes of permutations, in particular Bit-Permute-complement ($\mathcal{BPC}$) by Nassimi and Sahni[6]. They also prove that their algorithm routes the Lenfant's FUB families[3].

In this paper we develop self–routing algorithms for the Linear Class ($\mathcal{L}$) of permutations. The algorithm is very simple and routes many other classes of permutations as well. We consider Beneš network as well as the π network of Yew and Lawrie[8] and $(2n - 1)$–stage shuffle exchange network. The results include simple routing algorithms for the classes $\mathcal{L}$ (we extend this class with complements of bits), Ω, and Ω^{-1} on all these networks. For other permutations one can use a general looping type algorithm or break it into multiple simpler permutations.

2 Routing in Beneš Network

We will use I to represent any source and O to represent its destination tag. All binary additions in this paper are modulo 2.

*This research is supported by the NSF Presidential Young Investigator Award No. MIP 8452003, DARPA/ARO Contract No. DAAG 29–84–k–0066, ONR Contract No. N00014–86–k–0602.

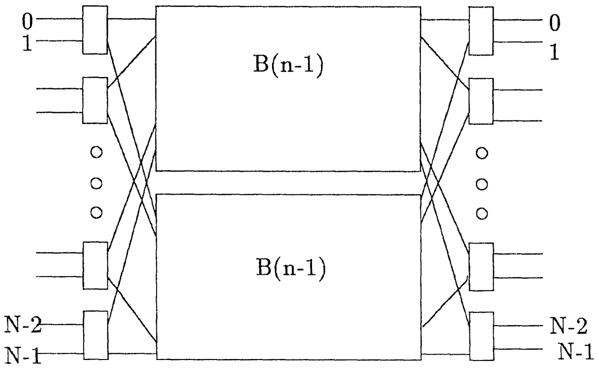


Figure 1: 2^n input/output Beneš network $\mathcal{B}(n)$.

Definition 1 *A permutation is said to be a linear permutation*[4] *if for all input I (whose binary representation is $(I_n\ I_{n-1}\ \dots\ I_1)$) and output O (whose binary representation is $(O_n\ O_{n-1}\ \dots\ O_1)$) pair there exists a non singular binary matrix $Q_{n\times n}$ that satisfies 1.*

$$O^T = Q \times I^T \tag{1}$$

Definition 2 *Let $I' = (I_n\ I_{n-1}\ \dots\ I_1, 1)$. A permutation is a Linear-Complement ($\mathcal{LC}$) permutation if there exists a binary matrix $P_{n\times n+1}$ where the submatrix of P formed by taking first n columns is non singular, such that every (I,O) pair satisfies the equation 2.*

$$O^T = P \times I'^T \tag{2}$$

With the definition given above, $\mathcal{LC}$ contains $\mathcal{BPC}$. Throughout this paper we will assume that the number of inputs/outputs to the interconnection network is $N = 2^n$. We will denote linear–complement, omega and inverse omega permutations on N inputs in compact form as $\mathcal{LC}(n)$, $\Omega(n)$ and $\Omega^{-1}(n)$ respectively. And $\mathcal{B}(n)$ denotes Beneš network with N inputs/outputs.

2.1 Routing Algorithm

Let the output lines of a switch be numbered as '0' and '1' for upper and lower outputs respectively. Each input line to a switch will have a routing bit. An input line to a switch is connected to the output line of the switch indicated by its routing bit. If the bit is '1' then that input is connected to the lower output of the switch otherwise, it is connected to the upper output of the switch. Routing of $\mathcal{LC}$ permutations in Beneš network is given by the following algorithm.

Algorithm 1 For the first $(n-1)$ stages, an input line to a switch in stage $i, 1 \leq i \leq (n-1)$ will have i–th bit of its destination tag as its routing bit. For the next n stages, an input line to a switch in stage $j, n \leq j \leq (2n-1)$ will have $(2n-j)$–th bit of its destination tag as its routing bit. For the first $(n-1)$ stages, switches are set up such that input line with smaller destination tag value is routed according to its routing bit. For the next n stages switches, are set up such that both the inputs are routed according to their routing bits. ∎

In first $(n-1)$ stages, conflicts are resolved by giving priority to one of the input lines. This algorithm is different from that of Nassimi and Sahni's[6] since in case of conflict in setting up a switch, their algorithm gives priority to the top input line, whereas our algorithm gives priority to the input line with smaller destination tag value. Consider figure 2(a) with destination tags for its inputs as shown. Let the bit indicated by the arrow be the routing bit. In this case, routing bit for both the inputs is '1' so there is a conflict. This is resolved by comparing the destination tags and giving priority for the input with smaller destination tag value, which in this case is the lower input. The other input line is automatically routed to the remaining output line. In figure 2(b) routing bits for both inputs are different so they get what they want and the switch is set as shown.

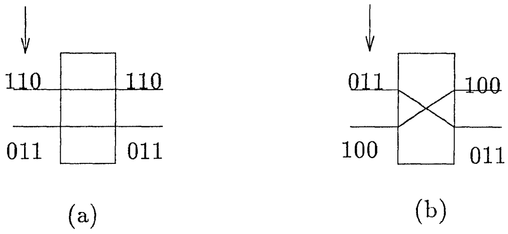


Figure 2: An example showing switch settings done by the algorithm.

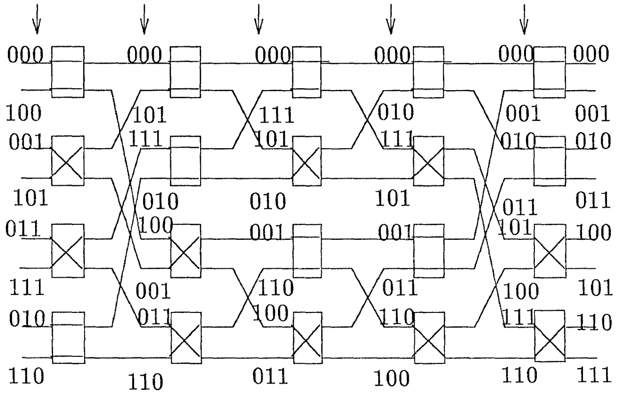


Figure 3: Routing a $\mathcal{LC}$ permutation in Beneš using the algorithm proposed

A complete example of this routing scheme is given in figure 3. Destination tags for each input line to a switch are given in the binary form. Routing bit for each stage is indicated by an arrow. This permutation is not routable by Nassimi and Sahni's (see figure 4) algorithm. The $\mathcal{LC}$ permutation given in the figures 3 and 4 has the functional form given below.

$$O_3 = I_1;\ O_2 = I_3;\ O_1 = I_3 + I_2$$

In the first stage (figure 3), routing bit is same for both the inputs to a switch. Hence switches in the first stage are set up such that input with smaller destination tag is routed correctly, which in this case are top input lines. After the first stage of routing, there exists $\mathcal{LC}$ permutation between O_3, O_2 of destination tag and I_2, I_1 of input line, for both top 4×4 Beneš network $\mathcal{B}(2)$ given as,

$$O_3 = I_2 + I_1; \; O_2 = I_2$$

and bottom $\mathcal{B}(2)$ given as,

$$O_3 = 1 + I_2 + I_1; \; O_2 = I_2$$

There exists conflict in setting up switches in the second stage of the network as well. For top most and bottom most switches in the second stage top input line has a smaller destination tag value, so these switches are set to route top input line correctly. For the other two switches bottom input lines have smaller destination tag value, hence, those switches are set to route bottom input lines correctly. Conflict exists only in the first two stages of the network. Last 3 stages are routed without any conflicts as given by the algorithm.

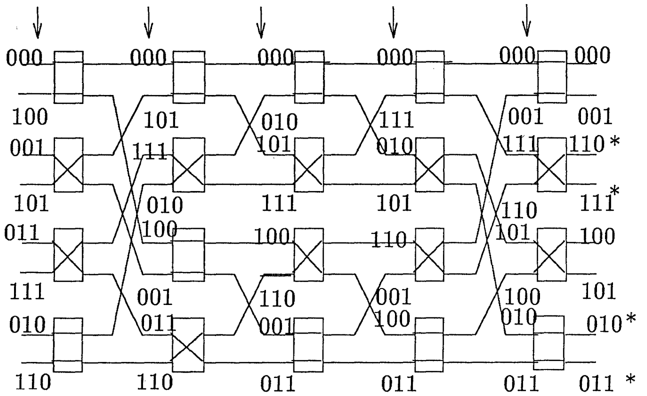


Figure 4: Routing an $\mathcal{LC}$ in Beneš using Nassimi and Sahni's algorithm fails. Incorrectly routed inputs are indicated by an asterisk.

2.2 Proof of Correctness

Theorem 1 *Any $\mathcal{LC}(n)$ permutation is routable by the routing algorithm 1, in $\mathcal{B}(n)$.*

Proof: We will use the fact that stages $2, \ldots, 2n-2$ of $\mathcal{B}(n)$ are just two $\mathcal{B}(n-1)$ networks, to prove the theorem by induction. To do this we need to show that after first stage of routing, the resulting permutation between most significant $(n-1)$ bits of the destination tag to an input of $\mathcal{B}(n-1)$ is still an $\mathcal{LC}(n-1)$ permutation.

More formally, this is true for $n = 1$. Let it be true for all $m < n$. Now consider the following lemma.

Lemma 1 *After one stage of routing of an $\mathcal{LC}(n)$ permutation using the algorithm 1, for any input-output pair I and O, the permutation between $(O_n, \ldots, O_2)$ and $(I_{n-1}, \ldots, I_1)$ for the top and bottom Beneš networks for 2^{n-1} inputs/outputs belong to $\mathcal{LC}$(n-1).*

Proof for the lemma: Inputs to a switch differ only in bit I_1. So depending on whether the equation for routing bit O_1 contains I_1 or not, the routing tags of the inputs to a switch are different or are same. Consider the first case; the equation for O_1 will be of the form $O_1 = I_1 + LF_1$, where LF_1 is independent of I_1. Since each input is routed according to its routing bit because there are no conflicts, the equation for O_1 after exchange is given as $O_1 = I_1$. So the effect of exchange is like substituting $I_1 + LF_1$ in all occurrences of I_1 in the equations for $O_n, \ldots, O_1$. Since an inverse shuffle is performed after exchange, all the top outputs of the switches go to the top Beneš network $\mathcal{B}(n-1)$ and all bottom outputs of the switches go to bottom Beneš network $\mathcal{B}(n-1)$. So substituting $I_1 = 0(1)$ in the equations for bits $O_n, \ldots, O_2$ of the routing tags of the inputs routed to top(bottom) $\mathcal{B}(n-1)$ we get $\mathcal{LC}(n-1)$ permutation as desired. In the second case, the equation for O_1 will be of the form, $O_1 = LF_1$, where LF_1 is independent of I_1. Let k be the most significant bit in which two destinations differ. Then the equation for O_k contains I_1 and is given as $O_k = I_1 + LF_k$, LF_k is independent of I_1. The algorithm routes inputs such that input with $O_1 = O_k$ is routed to top ouput line of the switch and the other input to the bottm ouput line of the switch. So after the exchange operation $O_1 = I_1 + O_k$. So the net effect is equivalent to substituting $I_1 + LF_1 + LF_k$ in all the occurrences of equations for $O_n, \ldots, O_1$. Since an inverse shuffle is performed after exchange, as in the previous case we get $\mathcal{LC}(n-1)$ permutation between $O_n, \ldots, O_2$ bits of the routing tags and inputs $I_{n-1}, \ldots, I_1$ of the top and bottom $\mathcal{B}(n-1)$ networks. ∎

From the above lemma $\mathcal{LC}$ (n) is routed in the first stage of $\mathcal{B}(n)$ such that there exists $\mathcal{LC}(n-1)$ permutation between $O_n, \cdots, O_2$ and the inputs of $\mathcal{B}(n-1)$. Since this is correctly routed by induction hypothesis, after $(2n-2)$ stages all the outputs are in the correct place as far as first $n-1$ bits are concerned. This means two destinations which differ only in the last bit of their destination do not exist in the same $\mathcal{B}(n-1)$. A shuffle and exchange will route these inputs to the correct places. ∎

3 Routing in Shuffle Exchange Networks

We will modify the routing algorithm to route $\mathcal{LC}$ permutations in π–network. A π–network is a cascade of two Ω networks [8].

3.1 Routing Algorithm

Algorithm 2 For the first n stages of the *pi*–network, an input to a switch in stage $i, 1 \leq i \leq n$, will have $(n-i+1)$–th bit of its destination tag as the routing bit. Routing is done as follows. First the destination tags are bit reversed and then compared. The smaller one will be routed according to its routing bit as before. For the next n stages of the network we use the standard Ω self–routing algorithm. ∎

A complete example is given in figure 5. Routing bit in each stage is indicated by an arrow. This permutation is not routable by the self routing algorithm given in [8].

Consider second switch from top in stage 1 of figure 5. Both inputs have the same routing bit '1'. But upper input has destination tag with smaller value when compared to that of lower one after bit reversal of the destination tags. Hence upper input is routed to the lower output of the switch. But in the case of bottom most switch in the first stage lower input has smaller destination tag value after bit reversal. So, that switch is set such that lower input is routed according to its routing bit which is lower output of the switch.

3.2 Proof of Correctness

We need the following lemmas, to prove that the algorithm works correctly.

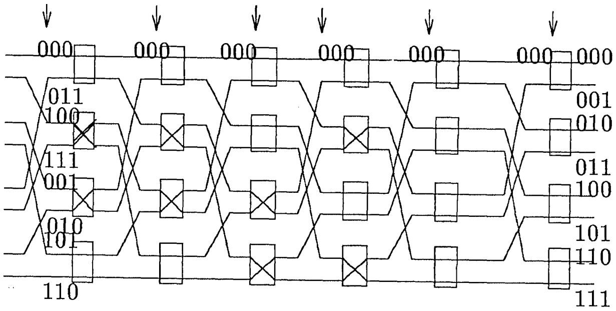


Figure 5: Routing an $\mathcal{LC}$ permutation in π–network using the algorithm proposed

Lemma 2 *If a permutation is $\mathcal{LC}$ permutation then after a shuffle on the input bits, the resulting permutation is still $\mathcal{LC}$.*

Proof is obvious, hence omitted. ∎

Lemma 3 *If a permutation is $\mathcal{LC}$ permutation then after performing an exchange operation on the inputs using the algorithm 2 the resulting permutation is still $\mathcal{LC}$.*

Proof for this lemma follows very closely that of lemma 1. Crucial part of the proof is showing that for the first n stages the algorithm performs exchange operation such that routing bit O_i is set to I_{n-i+1} if the equation for O_i contains I_{n-i+1}, otherwise to $I_{n-i+1} + O_j$ for some $j < i$ as specified by the algorithm. ∎

In the case of Beneš network we noted that two input lines to a switch in stage 1 differ only in I_1. However, in the case of the π network we shall take into account the fact that a shuffle was performed before the exchange operation hence I_n becomes I_1. So the two input lines to a switch in the first stage of a π network differ only in I_n. In the first stage, algorithm 2 will route using O_n as the routing bit. So there will not be any conflicts if O_n contains I_n. Proceeding in this manner it is easy to see that for the first n stages there will not be any conflicts if the routing bit O_{n-i+1} contains I_{n-i+1}.

Lemma 4 *Routing an $\mathcal{LC}$ permutation using algorithm 2 will always assure that at any stage i, $1 \leq i \leq n$, the destination tags for the inputs of a switch will differ atleast in one of the bits O_{n-i+1} through O_1.*

Proof: This is true for stage 1 since $\mathcal{LC}$ is a bijection. After $(m-1)$ stages of shuffle–exchange I will be of the form $(I_{n-m+1}, \ldots, I_1, \tilde{I}_n, \ldots, \tilde{I}_{n-m+2})$. $\tilde{I}_i$ means either I_i or $\bar{I}_i$, complement of I_i. After a shuffle I will be of the form $(I_{n-m}, \ldots, I_1, \tilde{I}_n, \ldots, \tilde{I}_{n-m+2}, I_{n-m+1})$. So destination tags for the two inputs of a switch will not differ in any of the bits $O_{n-m+1}, \ldots, O_1$ iff none of the equations for $O_{n-m+1}, \ldots, O_1$ contain I_{n-m+1}. From lemma 3 we know that after an exchange operation at stage i, O_{n-i+1} is set to I_{n-i+1} or to $I_{n-i+1} + O_j$, $j < i$, whereupon O_j is set to $I_{n-i+1} + O_{n-i+1}$. So the equations for $O_n, \ldots, O_{n-m+2}$ are either independent of I_{n-m+1} or if they contain the term I_{n-m+1} then the equation for some O_j, $1 \leq j < (n-m+2)$ will also contain that term. ∎

Theorem 2 *$\mathcal{LC}$ permutations are routable using the algorithm 2 in π-network.*

Proof: From lemmas 2 and 3 it follows that after routing one stage of shuffle and exchange the resulting permutation is still an $\mathcal{LC}$ permutation but it could be different form the earlier one. So to distinguish this, we use superscript for the matrix $P_{n\times(n+1)}$. So the P matrix in the $\mathcal{LC}$ permutation for stage i is indicated as P^i. The P matrix for the first stage denoted as P^1 is same as the P matrix in the original $\mathcal{LC}$ permutation.

Consider an input $I = (I_n, \ldots, I_1)$ with destination $O = (O_n, \ldots, O_1)$. Let I be of the form $D = (D_n, \ldots, D_1)$ after routing first n stages using the algorithm. From lemma 3 and the discussion following the lemma, at any stage i, $1 \leq i \leq n$, D_{n-i+1} is same as O_{n-i+1} if the destinations have different $(n-i+1)$–bit, which is true only when $P^i_{i1} \neq 0$. Otherwise, $D_{n-i+1} = O_{n-i+1} + O_j$, $j < (n-i+1)$. Therefore we have,

$$D_{n-i+1} = O_{n-i+1} + \bar{P}^i_{i1}\, O_j,\ 1 \leq i \leq n \qquad (3)$$

Thus the equations for $D_n, \ldots, D_1$ will be of the form given below.

$$\begin{aligned} D_n &= O_n + \bar{P}^1_{11}\, O_j, && \text{for some } j < n \\ D_{n-1} &= O_{n-1} + \bar{P}^2_{21}\, O_j, && \text{for some } j < n-1 \\ &\vdots \\ D_1 &= O_1 \end{aligned}$$

We can rewrite these equations such that they are in the Ω characteristic equation form. For example we can rewrite the equation for D_n as given below.

$$O_n = D_n + \bar{P}^1_{11}\, O_j, \quad \text{for some } j < n$$

But, O_j can again be rewritten in the form given above. In general we can substitute $D_k + \bar{P}^k_{k1}\, O_j$, for some $j < k$, for O_k. Proceeding in this manner we obtain the equation for O_n only in terms of D's. In the same manner we can rearrange equations to get equations for all O's as given below.

$$O_i = D_i + F_i(D_{i-1}, \ldots, D_1),\ 1 \leq i \leq n \qquad (4)$$

Clearly these equations are in Ω form hence routable by the last n stages of the π–network. ∎

As an example consider the $\mathcal{LC}$ permutation in figure 5 characterized by the following set of equations.

$$O_3 = I_1;\;\; O_2 = I_3;\;\; O_1 = I_3 + I_2$$

Here O_3 does not contain I_3 but O_1 does. Hence substituting $I_3 + I_1 (= LF_3) + I_2 (= LF_1)$ in all the occurrences of I_3 and performing a shuffle on input bits we get the following set of equations which characterize $\mathcal{LC}$ permutation for the second stage.

$$O_3 = I_2;\;\; O_2 = I_3 + I_2 + I_1;\;\; O_1 = I_2 + I_1$$

One can verify that these equations hold after the first stage of switches. D_3 is given by the following equation.

$$D_3 = O_3 + O_1$$

In a similar manner we can obtain equations for D_2 and D_1 as given below.

$$D_2 = O_2;\;\; D_1 = O_1$$

Rewriting these equations we get,

$$O_3 = D_3 + D_1;\;\; O_2 = D_2;\;\; O_1 = D_1$$

which are in characteristic Ω form, hence routable in the last three stages of the network (same as 8 input/output Ω network). ∎

3.3 $(2n - 1)$–stage Shuffle Exchange Network

In the proof given above, we showed that $D_1 = O_1$. This implies that all the switches in the last stage are set straight. Hence, we can eliminate all the switches in the last stage. So, we need only $(2n - 1)$ stages of shuffle exchange and a perfect shuffle. However, we can eliminate this shuffle at the output as follows.

We change the algorithm to treat destination tags as if a shuffle was performed on them. i.e., O_i is treated as $O_{(i+1)\, mod\, n}$. Let the given permutation be denoted as Π. With the modification the algorithm treats as if a shuffle was performed on Π. Hence in effect it routes $\Pi' = (\sigma\Pi)$. After routing for $(2n-1)$ stages a σ is required to route Π' correctly. So, after $(2n-1)$ stages we have routed $(\sigma^{-1}\Pi')$ correctly. But, $\sigma^{-1}\Pi' = \sigma^{-1}\sigma\Pi = \Pi$. Hence,

Theorem 3 *$\mathcal{LC}$ is routable in $(2n - 1)$–stage shuffle exchange using the modified algorithm described above.*

4 Conclusions

In this paper we have presented algorithms to route $\mathcal{LC}$ permutations on Beneš, π and $(2n - 1)$–stage shuffle exchange networks. Since there will not be any conflicts in the first n stages of the Beneš network if the permutation is in Ω^{-1}, this algorithm routes Ω^{-1} permutations as well. In fact the class of permutations routable using the algorithm given in this paper is much larger than $\mathcal{LC}$ class. With a similar argument any Ω permutation is routable using the algorithm in π network. It is interesting to note that it routes all permutations in $\mathcal{B}(2)$ for 4 input/output Beneš network. However this algorithm does not route all Ω permutations in Beneš networks, with $N > 4$. If the permutation is known to be Ω then it can be routed by setting the first $(n - 1)$ stages of the Beneš network straight as suggested by Nassimi and Sahni [6].

References

[1] J. Beetem, M. Denneau, and D. Weingarten. The GF11 Supercomputer. In *Int'l Symp. on Comput. Arch.*, pages 108–115, 1985.

[2] V. E. Beneš. On rearrangeable three–stage connecting networks. *The Bell System Technical Journal*, XLI(5):1481—1492, 1962.

[3] J. Lenfant. Parallel permutations of data: a Beneš network control algorithm for frequently used permutations. *IEEE Trans. on Computers*, c–27(7), 1978.

[4] M. C. Pease, III. The indirect binary n–cube microprocessor array. *IEEE Trans. on Computers*, c–26(5), 1977.

[5] D. Nassimi and S. Sahni. Parallel permutation algorithms to set up the Beneš permutation network. *IEEE Trans. on Computers*, c–31(2):148—154, 1982.

[6] D. Nassimi and S. Sahni. A self–routing Beneš network and parallel permutation algorithms. *IEEE Trans. on Computers*, c–30(5), 1981.

[7] A. Waksman. A permutation network. *J. Assoc. Comput. for Mach.*, 15(1), 1968.

[8] P. Yew and D. H. Lawrie. An easily controlled network for frequently used permutations. *IEEE Trans. on Computers*, c–30(4), 1981.

DESIGN AND ANALYSIS OF A FAULT-TOLERANT MULTISTAGE INTERCONNECTION NETWORK FOR LARGE-SCALE SHARED MEMORY PARALLEL COMPUTERS

Gyungho Lee and *Sizheng Wei*

The Center for Advanced Computer Studies
University of Southwestern Louisiana
P.O. Box 44330
Lafayette, Louisiana 70504-4330

Abstract

This paper introduces a class of multiple path multistage interconnection networks termed Reduced Size Interconnection (RSI). A RSI-m network of size N is designed to have m unique path multistage networks of size N/m. This approach of designing the network allows to construct a fault-tolerant network at the same cost for a unique path multistage network of the same size. We have considered the cost-effectiveness of RSI networks for reliability and for performance. RSI networks are shown to be compared favorably with some well-known multistage networks.

1. Introduction

Multistage interconnection networks such as Omega networks [6] and Delta networks [10], have been favored for processor-memory connection in "large-scale" shared memory machines because of its cost-effectiveness. However, as is well known, the multistage network lacks fault-tolerant capability because of its basic property that there is only a single path between any source-destination pair (*unique path network*). This lack of fault-tolerant capability has received considerable attention, and many ways of providing fault-tolerance to the network have been proposed.

The basic idea of fault-tolerant network is to provide multiple paths for a source-destination pair so that alternate paths can be used in case of faults in a path. Providing multiple paths can be done in various ways. The methods include increasing the number of stages [2], using multiple links between switches [3, 7, 9], increasing the size of switches [8], partitioning a unique path network into several subnetworks [5, 12], and incorporating multiple copies of a unique path multistage network [4, 11]. Compared to the unique path networks, these multiple path networks certainly have higher reliability but with increased hardware complexity, which not only increases cost but also puts some wrinkle on the claim of enhanced reliability.

In this paper, a class of multiple path multistage interconnection networks, dubbed as "Reduced Size Interconnection", is proposed to provide a cost-effective fault-tolerant network for large-scale shared memory parallel computers. The network is designed to provide fault-tolerant capability without increasing hardware complexity and cost over a unique path multistage network.

2. Reduced Size Interconnection Network

Suppose, for a network of *size* N, i.e., with N sources and N destinations, m disjoint partitions of N/m sources and N/m destinations are formed first, where m (≥ 2) and N ($> m$) are the powers of 2.[1] In a *Reduced Size Interconnection (RSI)* network of size N, m unique path multistage networks of size N/m are provided, i.e. one for each partition, and each source and destination are linked to all the m unique path networks via $m \times 1$ multiplexers and $1 \times m$ demultiplexers, respectively (the rules for the connection come shortly). Thus, a RSI-m network of size N has one $m \times 1$ multiplexer stage (*input stage*), $\log_2 \frac{N}{m}$ stages (*intermediate stages*) of 2×2 crossbar switches, and one $1 \times m$ demultiplexer stage (*output stage*). There are $N/2$ switches in each of the intermediate stages, N multiplexers in the input stage, and N demultiplexers in the output stage. Each unique path network of size N/m plus its associated multiplexers and demultiplexers will be called a *subnetwork*. These subnetworks will be denoted by G^0, G^1, $\cdots$, G^{m-1}. Although it is possible to have different types of unique path networks, we assume that all the m unique path networks are of identical type. The type of unique path network taken is called *base network*.

Let $S_{i,j}$ and $D_{i,j}$ denote the source j and the destination j, respectively, which are associated with a subnetwork G^i based on the partition ($0 \leq i \leq m-1$ and $0 \leq j \leq N/m-1$). Also, let $MUX_{l,k}$ and $DEMUX_{l,k}$ represent the multiplexer k and demultiplexer k in G^l, respectively, where $0 \leq l \leq m-1$ and $0 \leq k \leq N/m-1$. Then, the sources and destinations are connected to each subnetwork as follows:

i) Each $S_{i,j}$ is connected to every i^{th} input port of the m multiplexers from $MUX_{0,j}$ to $MUX_{m-1,j}$.

ii) Every i^{th} output port of the m demultiplexers from $DEMUX_{0,j}$ to $DEMUX_{m-1,j}$ is connected to each $D_{i,j}$.

An example of RSI-2 network of size 8, in which Omega network is used as the base network, is illustrated in Figure 1.

Routing in a RSI network can be divided into three steps, assuming that the selection of a path out of the m disjoint paths is already done at a source. At the input stage, i.e. the stage of multiplexers, a request with a routing tag just passes a multiplexer in a subnetwork which is chosen by the source. None of the tag is consumed because a multiplexer is always with only one output port. After that, the routing of the request in the intermediate stages is the routing in the base network of size N/m by using $\log_2 \frac{N}{m}$ bits.

[1] Each partition may have different sizes. Also, the number of partitions may vary. However, in this paper, we consider only the equal partitioning and the "small" value of m (2 or 4). Also, for the sake of simplicity, our discussion is restricted to "rectangular" networks, i.e. having the same number of inputs and outputs, of 2×2 switches.

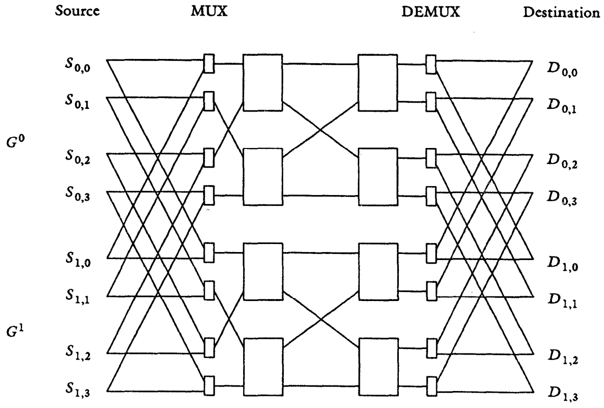


Figure 1. A RSI-2 network of size 8 with 2×2 switches.

Finally, the request arriving at the output stage, i.e. the stage of demultiplexers, is routed to the proper destination by using $\log_2 m$ bits, the rest part of the tag. Notice that selecting any particular subnetwork does not change the routing in a RSI network: the routing algorithm is always the same regardless of the subnetwork selected by a source.

3. Reliability of RSI Network

We consider the reliability of RSI networks under the "full access" criterion, and measure their reliability in terms of "Mean Time to Failure" (MTTF). We assume that any of the switching components — crossbar switches, multiplexers, and demultiplexers — in a RSI network can fail. Based on the recent survey by Adams, Agrawal, and Siegel [1], the fault model and the fault-tolerance criterion applied in our analysis are common and strict.

Before we are involved in the reliability analysis in terms of MTTF, the number of faults that RSI networks can tolerate is worth to be mentioned. Since a RSI-m network provides m disjoint paths for each source-destination pair through the m independent subnetworks, it is $(m-1)$-fault tolerant and is robust in the presence of more than $m-1$ faults, up to $\frac{m-1}{m}(\frac{N}{2}\log_2\frac{N}{m}+2N)$ faults in the network.

To make the analysis of MTTF tractable, we use assumptions similar to the ones that have been made previously in other studies of fault-tolerant networks. Each component has an independent, Poisson distribution of failures with a constant failure rate. The failure rate is assumed to be proportional to the gate complexity of a switching component. The complexity of a component is considered in terms of a "crosspoint." Thus, we assume the failure rate for a $m \times 1$ multiplexer or an $1 \times m$ demultiplexer $\lambda' \simeq (m-1) \times \lambda/4$, if the failure rate for a 2×2 crossbar switch is λ, because an $m \times 1$ multiplexer or an $1 \times m$ demultiplexer has $m-1$ crosspoints while a 2×2 crossbar switch has 4 crosspoints.

By considering each stage of the network separately, we have an optimistic probability that a RSI-m network is not faulty for a time period $(0, t)$:

$$R_{RSI-m}(t) = [1-(1-e^{-\lambda t})^m]^{M_1} \times [1-(1-e^{-(m-1)\lambda t/4})^m]^{M_2},$$

where $M_1 = \frac{N}{2m} \times (\log_2 \frac{N}{m})$ and $M_2 = 2N/m$. So, we have the upper bound

$$MTTF^U_{RSI-m} = \int_0^\infty R_{RSI-m}(t)\,dt \ .$$

Since a sufficient condition for the network to be operative is that at least one of the subnetworks is fault-free, the lower bound is

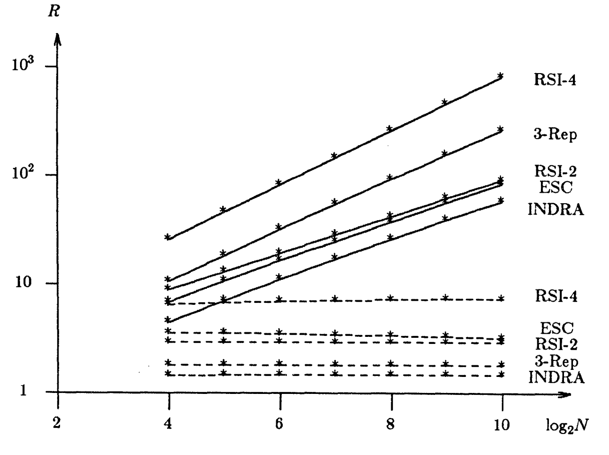
$$MTTF^L_{RSI-m} = \int_0^\infty [1-(1-e^{-(M_1+M_2 \times \frac{(m-1)}{4})\lambda t})^m]\,dt \ ,$$

where $M_1 = \frac{N}{2m} \times (\log_2 \frac{N}{m})$ and $M_2 = 2N/m$.

For comparison purpose, we obtained the MTTFs of Omega network and some other fault-tolerant networks (ESC [2], 3-replicated [4], and INDRA with $R=2$ [11]) in a similar way. Although Omega network is not a fault-tolerant network, we use its reliability as a yardstick to measure the improved reliability of RSI networks. The ratios of the bounds on MTTF of the fault-tolerant networks to that of Omega network are shown in Figure 2, in which the network size N varies from 16 to 1024; one can easily see that RSI networks have significant improvement on the reliability compared to Omega network.

4. Performance of RSI Networks

The important performance measure for unbuffered interconnection networks is "bandwidth (BW)" or "probability of acceptance (PA)". The performance analysis of unbuffered RSI networks is based on the assumptions made for the usual "uniform traffic model" [4, 5, 10, 11]. Concerning the selection of a particular path out of the m disjoint paths in the network, we assume "uniform random" selection at each processor; each path is selected randomly with equal probability. We also assume that destinations are able to accept more than one requests simultaneously, i.e., the memory modules are considered as multi-ported memory units which can be accessed through more than one ports at the same time.



N : network size

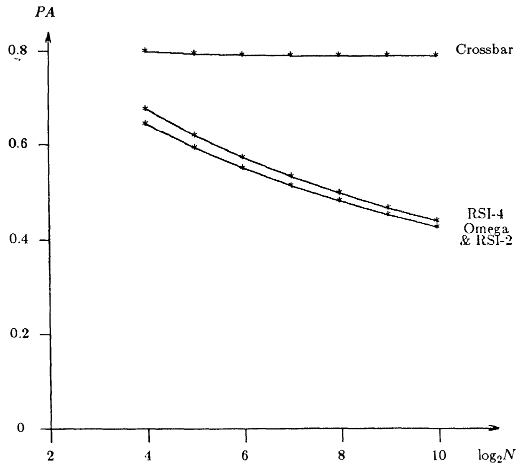
R = MTTF of a fault tolerant network/MTTF of Omega network

--- : for lower bound

— : for upper bound

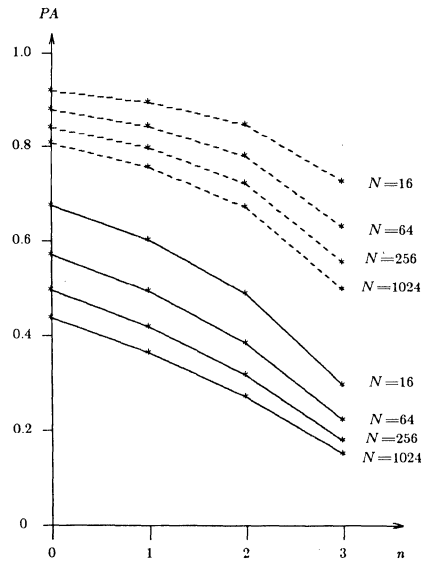
All the networks are of 2×2 switches.

Figure 2. Ratios of MTTF for some fault-tolerant networks with respect to that of Omega network



N : network size
PA : probability of acceptance
Request generation rate p = 0.5
All the multistage networks are of 2×2 switches.

Figure 3. Comparison of PA of Unbuffered Networks.



N : network size
n : the number of the faulty subnetworks
PA : probability of acceptance
--- : request generation rate p = 0.1
— : request generation rate p = 0.5
The network is of 2×2 switches.

Figure 4. Probability of Acceptance of an Unbuffered RSI-4 with Faulty Subnetworks.

Following the analysis of Delta network by Patel [10], the PA's of RSI-2 network and RSI-4 network of size N from 16 to 1024 were computed. A typical comparison of PA's of RSI networks with those of Omega network and crossbar network is shown in Figure 3. We can find that with an increased value of m, a RSI network may provide higher bandwidth and probability of acceptance than those of an Omega network of the same size, which is the result of having multiple ports at each memory module. As one of the paths becomes faulty, the number of paths a processor can utilize decreases. Assuming that a faulty subnetwork of the RSI network is totally unusable, we considered the performance degradation with at most $m-1$ faulty subnetworks. Figure 4 shows the probability of acceptance of an unbuffered RSI-4 network with the number of faulty subnetworks from 0 to 3, when the request generation rate p equals 0.5 and 0.1, respectively. From this Figure, we can see that RSI networks can achieve graceful degradation of the performance and that the performance degradation will not be significant when the traffic is "light." The performance study of buffered RSI networks has also been carried out, and the results are similar to the case of unbuffered RSI networks.

5. Cost-Effectiveness

For an interconnection network designed for large-scale general-purpose parallel computers, one of the important considerations is its cost-effectiveness. If the high performance and reliability of a network comes at the expense of too high cost, it may have little value in practice.

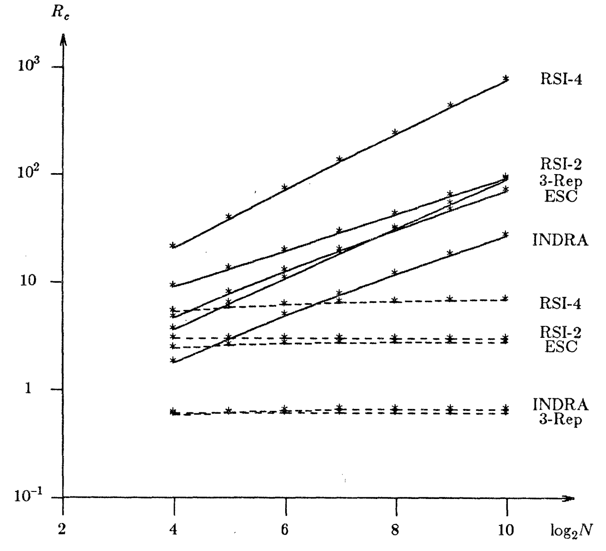
For the cost-effectiveness, we first need to figure out the cost of the networks. To estimate the cost of a network of size N, one common method is to calculate the switch complexity with an assumption that the cost of a switch is proportional to the number of gates involved, which is roughly proportional to the number of "crosspoints" within a switch [10, 13]. For example, a 2×2 switch has 4 units of hardware cost whereas a $m \times 1$ multiplexer has $m-1$ units. In this way, Omega, 3-replicated, INDRA ($R=2$) networks of 2×2 switchs and crossbar network have costs of $2N\log_2 N$, $6N\log_2 N$, $4N(\log_2 N+1)$ and N^2, respectively. Also, a RSI-m network has the cost of $2N(\log_2\frac{N}{m}+m-1)$, and an ESC network has the cost of $2N(\log_2 N+2)$.

Now, a simple measure of the cost-effectiveness for reliability can be given by comparing MTTFs of the networks with respect to the cost. Let the cost-effectiveness, η, of a network for reliability be the ratio of its MTTF to its cost. The cost-effectiveness η of some fault-tolerant networks relative to that of Omega network (for both upper bounds and lower bounds) are shown in Figure 5. In the same way, a simple measure of the cost-effectiveness for performance can be given by comparing the probability of acceptance of the networks with respect to the costs. Figure 6 shows the cost-effectiveness of RSI networks and crossbar networks for performance relative to that of Omega network. We can see that many advantages of RSI networks in reliability and in performance comes at a modest cost.

6. Conclusion

We proposed and analyzed a class of fault tolerant multistage interconnection networks, named Reduced Size Interconnection (RSI). By providing m identical subnetworks of size N/m for a RSI-m network of size N, we can achieve significant reliability gain and good performance at the same cost for constructing a unique path multistage network of the same size.

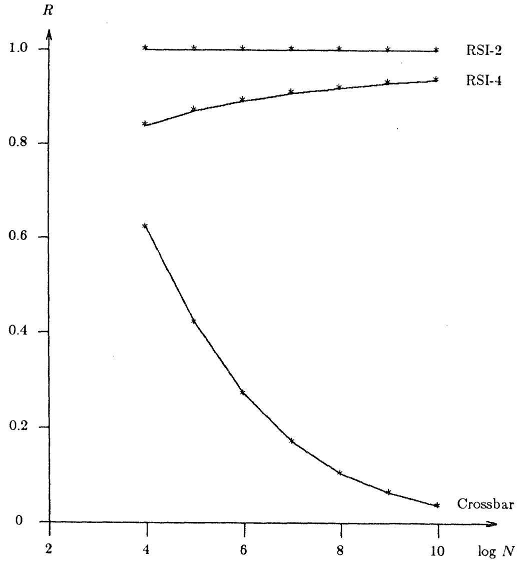
The performance analyses of unbuffered and buffered RSI networks have been carried out. We considered the performance in the fault-free situation and in the presence of faults. The results showed that compared to a unique path



N : network size
R_c $= \eta$ of a fault tolerant network/η of Omega network, where
$\eta =$ MTTF of a network/Cost of the network
--- : for lower bound
— : for upper bound

All the networks are of 2×2 switches.

Figure 5. Ratios of MTTF/Cost for some fault-tolerant networks with respect to that of Omega network



N: network size
$R = \eta$ of a network/η of Omega network, where
$\eta =$ probability of acceptance of a network/Cost of the network
Request generation rate $p = 0.5$

All the multistage networks are of 2 × 2 switches.

Figure 6. Ratios of PA/Cost for some unbuffered networks with respect to that of Omega network

multistage network, a RSI network improved reliability without decreasing the performance. Also, the performance degradation due to faulty subnetworks can be insignificant for "light" traffic.

To show the cost-effectiveness of RSI networks, the comparison with other networks for reliability and for performance was made. The results indicated that RSI networks are compared favorably with other fault-tolerant networks such as ESC, INDRA (R =2), and 3-replicated network.

References

[1] G. B. Adams III, D. P. Agrawal, H. J. Siegel, "A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks," *Computer,* Vol. 20, No. 6, June 1987, pp. 14-27.

[2] G. B. Adams III, H. J. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Trans. Comp.,* Vol. 31, May 1982, pp. 443-454.

[3] L. Ciminiera, A. Serra, "A Connecting Network with Fault Tolerance Capabilities," *IEEE Trans. Comp.,* Vol. 35, June 1986, pp. 578-580.

[4] C. P. Kruskal, M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors," *IEEE Trans. Comp.,* Vol. 32, Dec. 1983, pp. 1091-1098.

[5] V. P. Kumar, S. M. Reddy, "Design and Analysis of Fault-Tolerant Multistage Interconnection Networks with Low Link Complexity," *12th International Symposium on Computer Architecture,* June 1985, pp. 376-386.

[6] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Comp.,* Vol. 24, Dec. 1975, pp. 99-109.

[7] R. J. McMillen, H. J. Siegel, "Performance and Fault-Tolerance Improvements in the Inverse Augmented Data Manipulator Network," *Proc. 9th Annual Symposium on Computer Architecture,* June 1982, pp. 63-72.

[8] K. Padmanabhan, D. H. Lawrie, "A Class of Redundant Path Multistage Interconnection Networks," *IEEE Trans. Comp.,* Vol. 32, Dec. 1983, pp. 1099-1108.

[9] D. S. Parker, C. S. Raghavenda, "The Gamma Network: A Multiprocessor Interconnection Network with Redundant Paths," *Proc. 9th Annual Symposium on Computer Architecture,* June 1982, pp. 73-80.

[10] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Trans. Comp.,* Vol. 30, Oct. 1981, pp. 771-780.

[11] C. S. Raghavendra, A. Varma, "INDRA: A Class of Interconnection Networks with Redundant Paths," *1984 Real-Time Systems Symp.,* Computer Society Press, Silver Spring, Md., 1984, pp. 153-164.

[12] S. M. Reddy, V. P. Kumar, "On Fault-Tolerant Multistage Interconnection Networks," *1984 Int'l Conf. Parallel Processing,* Computer Society Press, Silver Spring, Md., 1984, pp. 155-164.

[13] N. Tzeng, P. Yew, C. Zhu, "A Fault-Tolerant Scheme for Multistage Interconnection Networks," *12th International Symposium on Computer Architecture,* June 1985, pp. 368-375.

Data Movement Operations and Applications on Reconfigurable VLSI Arrays

Russ Miller
Dept. of Comp. Sci.
SUNY-Buffalo
Buffalo, NY 14260

V. K. Prasanna Kumar
Dept. of EE-Systems
USC
Los Angeles, CA 90089

Dionisios I. Reisis
Dept. of EE-Systems
USC
Los Angeles, CA 90089

Quentin F. Stout
Dept. of EECS
Univ. Michigan
Ann Arbor, MI, 48109

Abstract

This paper considers a *mesh with reconfigurable bus (reconfigurable mesh)*, that consists of a VLSI array of processors connected to a reconfigurable bus system. The N PEs are laid out as a square mesh in $O(N)$ VLSI area. The reconfiguration scheme can be used to dynamically obtain various interconnection patterns between the PEs. In fact, the array can be used as a universal chip capable of simulating any $O(N)$ area organization with a planar wiring layout without loss in time. The reconfiguration scheme also supports several parallel techniques developed for the CRCW PRAM. In this paper, we develop fundamental data movement operations for the reconfigurable mesh. These operations are used to give efficient solutions to a variety of problems involving graphs and digitized pictures. The running times of these algorithms are asymptotically superior to those developed for the mesh with multiple broadcasting, the mesh with multiple buses, the mesh-of-trees, and the pyramid computer.

1 Introduction

In this paper, we consider a reconfigurable VLSI array of processing elements that combines the advantages of a number of architectures including the mesh, pyramid, mesh-of-trees, and meshes with broadcast buses. Due to page limitations, this paper will only summarize a subset of the results that we have obtained for the reconfigurable mesh. The reader is referred to [7] for discussions and algorithms associated with the results given in this paper.

The *mesh with reconfigurable bus (reconfigurable mesh) of size N* consists of an $N^{1/2} \times N^{1/2}$ array of processors connected to a grid-shaped reconfigurable broadcast bus, where each processor has four locally controllable bus *switches*, as shown in Figure 1. Other than the buses and switches, the reconfigurable mesh is similar to the standard mesh in that it operates in SIMD mode and has $O(N)$ area, under the assumption that processors, switches, and individual links have constant size. In one unit of time each processor can perform standard arithmetic and boolean operations on its own data, can set any of its four switches, and can send and receive a piece of data from the bus.

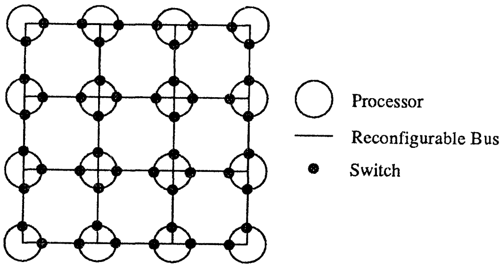


Figure 1: A reconfigurable mesh of size 16.

In each subbus shared by multiple processors, at any given time we assume that at most one processor may use the bus to broadcast a value, where a value consists of $O(\log N)$ bits. Notice that by setting the switches properly, sub-row (column) buses can be created within each row (column), sub-meshes with reconfigurable buses can be created, a global broadcast bus can be created, distinct buses can be created within distinct sets of contiguously labeled processors, and so forth.

Major advantages of the reconfigurable mesh are as follows.

1. Buses can be used to speed up parallel arithmetic and logic operations among data stored in different processors. The reconfiguration scheme supports several CRCW PRAM techniques. In fact, for some problems the reconfigurable mesh is superior to the PRAM.
2. The reconfigurable mesh provides an environment for efficient sparse data movement operations.
3. A significant asymptotic improvement can be achieved in the running times of algorithms that solve several problems on the reconfigurable mesh compared to efficient algorithms for the mesh-of-trees, pyramid, and mesh with static broadcast buses.
4. The reconfigurable mesh can act as a universal chip in that VLSI organizations with equivalent area and a planar wiring layout can be simulated without loss in time.

Many of the algorithms for the reconfigurable mesh (c.f., [7]) will continually reconfigure the system by setting the switches to give the desired substructures.

2 Related Architectures

It should be noted that although there are similarities, the reconfigurable mesh is very different from the CHiP project [15], the mesh augmented with broadcast buses[1, 3, 12, 16], and the *bus automaton* [4]. However, the reconfigurable mesh is similar to the *polymorphic-torus network* [6], with the major difference being that in the polymorphic-torus network there is an arbitrary crossbar in each processor to control connections between the north, south, east, and west bus ports. Finally, the reconfigurable mesh appears to be almost identical to the latest version of the *Content Addressable Array Parallel Processor (CAAPP)* [18], which was developed independently of the reconfigurable mesh.

3 Data Movement Operations

Data movement operations form the foundation of numerous algorithms for machines constructed as an interconnection of processors.

Proposition 3.1 *Given a set $S = \{a_i\}$ of N values, distributed one per processor on a reconfigurable mesh of size N so that processor P_i contains a_i, $0 \leq i \leq N-1$, and a unit-time binary associative operation $\otimes$, in $\Theta(\log N)$ time the parallel prefix problem can be solved so that each processor P_i knows $a_0 \otimes a_1 \otimes \ldots \otimes a_i$.*

We introduce a technique called *bus splitting*, in which processors exploit the ability to locally control the effective size of subbuses, to obtain the following Proposition.

Proposition 3.2 *Given a reconfigurable mesh of size N, in which each processor stores a bit of data, the logical OR of the data in each row (column), or the entire reconfigurable mesh, can be determined in $\Theta(1)$ time.*

The reconfigurable mesh can be superior to other parallel models for performing some computations. Consider, for example, computing the exclusive OR (EXOR) function of $N^{1/2}$ values stored in a row of the mesh. Note that [5] has shown that the exclusive OR function cannot be computed in $O(1)$ time on a PRAM using a polynomial number of processors. However, by exploiting the reconfigurability available, the EXOR function can be computed in $\Theta(1)$ time on the reconfigurable mesh.

Proposition 3.3 *Given a reconfigurable mesh of size N, in which each processor stores a bit of data, the exclusive OR (EXOR) of the $N^{1/2}$ data stored in a row (column) can be computed in $\Theta(1)$ time.*

Lemma 3.4 *Given a reconfigurable mesh of size N, suppose each processor in a row (column) stores a bit of data d_j, $0 \leq j \leq N^{1/2}-1$. Then, the computation of $\mathcal{F}_i$ given by $\mathcal{F}_i = \sum_{i=0}^{j} d_i$, $0 \leq j \leq N^{1/2}-1$, can be performed in $\Theta(1)$ time.*

Many parallel algorithms are designed to reduce data at intermediate stages of the algorithm. It is, therefore, often useful to be able to efficiently perform fundamental operations on reduced sets of data.

Proposition 3.5 *Given a reconfigurable mesh of size N, in which no more than one processor in each column stores a data value, the minimum (maximum) of these $O(N^{1/2})$ data items can be determined in $\Theta(1)$ time.*

By a somewhat more complicated sequence, Valiant's PRAM algorithm for finding the maximum [17] can be simulated on a reconfigurable mesh to find the maximum of all N values, assuming they are stored one value per processor.

Proposition 3.6 *Given a set of data items S of size N stored one per processor on a reconfigurable mesh of size N, the maximum value of S can be determined in $O(\log\log N)$ time.*

Proposition 3.7 *Given a set of bits S of size N stored one per processor on a reconfigurable mesh of size N, the EXOR of all items in S can be computed in $O(\log\log N)$ time.*

Proposition 3.8 *Suppose on a reconfigurable mesh of size N each processor has a label from a set of k distinct labels, $1 \leq k \leq N$. Further, suppose processors having the same label form arbitrary contiguous regions on the mesh. Then, each processor can know whether there is at least one tagged processor with its label in $\Theta(1)$ time. Also, given that there is at least one tagged processor for each label and all tagged processors with the same label store identical data, the tagged data can be broadcast to all other processors with the same label in the above time, for all labels in parallel.*

It is often desirable to model PRAM algorithms on other machines. In order to efficiently simulate the CRCW PRAM, one must be able to efficiently simulate the concurrent read and concurrent write properties. Define a *Random Access Read (RAR)* to be a data movement operation that models a concurrent read, in which each processor knows the index of another processor from which it wants to read data [11]. Similarly, a *Random Access Write (RAW)* will model a concurrent write in that each processor knows the index of a processor that it wishes to write to [11]. In case of multiple writes to the same processor, a tie-breaking scheme is used, such as minimum or maximum data value, or arbitrarily letting one value succeed.

Proposition 3.9 *Given a reconfigurable mesh of size N, in $O(k^{1/2}+\log N)$ time k data items may be moved in a RAR or RAW, where $k \leq N$.*

In fact, more efficient data movement can be performed if the distribution of the source processors, i.e., those processors sending data, as well as the destination processors, i.e., those processors receiving data, is uniform over the reconfigurable mesh.

Proposition 3.10 *Given a reconfigurable mesh of size N, if the number of source and destination processors within any block of size k^2 is $O(k)$, $1 \leq k \leq N^{1/2}$ then RAR and RAW can be performed in $O(\log N)$ time.*

Another fundamental operation that involves data movement is data reduction. Assume that each processor has at most one record having a *key* field and a *data* field. *Data reduction* will perform an associative binary operation on the data of records having the same key. At the end of the data reduction operation, each processor with key k will have the result of the binary operation performed over all data with key k.

Proposition 3.11 *Given a binary associative operator $\otimes$, data reduction can be performed on k distinct keys in $O(k^{1/2}+\log N)$ time on a reconfigurable mesh of size N, so that each processor knows the result of applying $\otimes$ over all data items with its key.*

Lemma 3.12 *Given a reconfigurable mesh of size N with k distinct keys randomly distributed one key per processor, the number of distinct keys can be determined in $O(k^{1/2}+\log N)$ time.*

4 Applications

In this section, we illustrate the performance of the reconfigurable mesh by giving simulations of other low wire area organizations, such as the mesh-of-trees and pyramid, discussing the use of the reconfigurable mesh as a universal chip, and by giving efficient parallel algorithms to solve problems involving graphs and images.

4.1 Simulations

Well known organizations such the mesh-of-trees and pyramid computer can be efficiently simulated by the reconfigurable mesh due to the numerous communications patterns that the reconfigurable mesh provides. In the first part of this section, we consider step by step simulation of the mesh-of-trees and pyramid.

A *mesh-of-trees (MOT) of base size N*, where N is an integral power of 4, has a total of $3N-2N^{1/2}$ processors. N of these are base processors arranged as a mesh of size N. Above each row and above each column of the mesh is a perfect binary tree of processors. Each row (column) tree has as its leaves an entire row (column) of base processors. All row trees are disjoint, as are all column trees. Every row has exactly one leaf processor in common with each column tree. Each base processor is connected to 6 other processors (assuming they exist): 4 neighbors in the base, a parent in its row tree, and a parent in its column tree. Each processor in a row or column tree that is neither a leaf nor a root is connected to exactly 3 other processors in its tree: a parent and 2 children. Each root in a row or column tree is connected to its 2 children. Notice that in the MOT the processors in each row and in each column can be looked upon as placed at levels $0, 1, \ldots, k$ where $N^{1/2} = 2^k$.

Define a c-embedding of a hierarchical organization onto the reconfigurable mesh to have the following properties.

1. A constant number of processors of the hierarchical organization are mapped to each processor of the reconfigurable mesh.
2. The number of communication links between levels l and $l+1$, $0 \leq l \leq k-1$, incident on any row or column bus segment is $\leq c$.

Define a class of algorithms on a hierarchical organization to be *normalized algorithms* if the following hold.

1. During a computation step of a hierarchical algorithm, all data operated on are located at the same level of the hierarchical organization.
2. During a communication step of a hierarchical algorithm, communication is performed between at most two adjacent levels of the hierarchical organization.

[9] shows how to embed the mesh-of-trees into a mesh. This embedding is used to embed the mesh-of-trees into the reconfigurable mesh and obtain the following two propositions.

Proposition 4.1 *Any normalized algorithm running in $T(N)$ time on a mesh-of-trees of base size N can be simulated on a reconfigurable mesh of size N to finish in $O(T(N))$ time.*

Proposition 4.2 *Any algorithm running in $T(N)$ time on a mesh-of-trees of base size N, can be simulated on a reconfigurable mesh of size N to finish in $O(T(N)\log N)$ time. Further this time is optimal.*

We now turn our attention to the simulation of the reconfigurable mesh by the mesh-of-trees. During the execution of an algorithm on the reconfigurable mesh the buses are continuously configured. A configuration of the bus corresponds to partitioning the mesh into disjoint sets of contiguous processors. Reconfiguration of the bus can be simulated on the mesh-of-trees by identifying contiguous processors in the mesh. This reduces to the problem of identifying connected 1's in an $N^{1/2} \times N^{1/2}$ digitized image (see Section 5.2 for more details). On the mesh-of-trees this can be done in $O(\frac{\log^3 N}{\log\log N})$ time [9, 10].

Proposition 4.3 *A mesh-of-trees of base size N can simulate a reconfigurable mesh of size N in $O(\frac{T(N)\log^3 N}{\log\log N})$ time if the switch settings are dynamic, or $O(\frac{T(N)\log^2 N}{\log\log N} + \log^2 N)$ if the switch settings are static.*

We now turn our attention to relationships between the reconfigurable mesh and the pyramid computer. A *pyramid computer (pyramid) of size N* is a machine that can be viewed as a full, rooted, 4-ary tree of height $\log_4 N$, with additional horizontal links so that each horizontal *level* is a mesh. It is often convenient to view the pyramid as a tapering array of meshes. A pyramid of size N has at its base a mesh of size N, and a total of $\frac{4}{3}N - \frac{1}{3}$ processors. The levels are numbered so that the base is level 0 and the apex is level $\log_4 N$. A processor at level i is connected via bidirectional unit-time communication links to its 9 neighbors (assuming they exist): 4 siblings at level i, 4 children at level $i-1$, and a parent at level $i+1$.

An embedding of the pyramid into the reconfigurable mesh, similar to the mesh-of-trees embedding used in Propositions 4.1 and 4.2, is used to give the following.

Proposition 4.4 *Any algorithm running in time $T(N)$ on a pyramid of size N can be simulated on a reconfigurable mesh of size N in $O(T(N))$ time.*

The $\Theta(N^{1/4})$ time solution to the connected 1's problem on a pyramid of size N [8] is used to give the following.

Proposition 4.5 *Any algorithm running on the reconfigurable mesh of size N in time $T(N)$ can be simulated on a pyramid of size N in $O(T(N)N^{1/4})$ time. This simulation is optimal.*

Theorem 4.1 *Any architecture that can be laid out in an $N^{1/2} \times N^{1/2}$ grid and use a planar wiring (assuming wires have unit width) can be simulated by the reconfigurable mesh in constant time per unit time of the target architecture.*

4.2 Graph Problems

The first problem considered in this section is that of computing the connected components of an undirected graph with $N^{1/2}$ vertices, given as an adjacency matrix. The $(i,j)^{th}$ entry of the adjacency matrix of the graph is initially stored in processor $P_{i,j}$ of the reconfigurable mesh. The algorithm that we use is based on the $O(\log N)$ time algorithm presented for the CRCW PRAM [14].

Theorem 4.2 *Given the adjacency matrix of an undirected graph with $N^{1/2}$ vertices distributed so that the $(i,j)^{th}$ element of the matrix is stored in processor $P_{i,j}$ of a reconfigurable mesh of size N, the connected components of the graph can be determined in $O(\log N)$ time.*

The reconfigurable mesh can also be used to provide efficient solutions to some graph problems that assume unordered edges as input.

Theorem 4.3 *The connected components of a V vertex graph given in unordered edge input format, can be computed in $O(V^{1/2})$ time on the reconfigurable mesh of size N, where $N^{1/2} \leq V \leq N$.*

Corollary 4.6 *A minimal spanning forest of a V vertex graph given in unordered edge input format, can be computed in $O(V^{1/2})$ time on the reconfigurable mesh of size N, where $N^{1/2} \leq V \leq N$.*

Several graph properties can be deduced once a spanning tree of the graph is determined [2]. Using Corollary 4.6 and the data movement operations presented in Section 3, the following results can be obtained.

Corollary 4.7 *Given N edges of a graph G with V vertices distributed one vertex per processor in a reconfigurable mesh of size N, $N^{1/2} \leq V \leq N$, in $O(V^{1/2})$ time, one can*

a) check if G is bipartite,
b) compute the cyclic index of G, and
c) compute the articulation points of G.

4.3 Image Problems

Many problems involving digitized images can be solved efficiently on the reconfigurable mesh. The input to these problems is an $N^{1/2} \times N^{1/2}$ digitized image distributed one pixel per processor on a reconfigurable mesh of size N so that processor $P_{i,j}$ has pixel (i,j). The problems that we examine focus on labeling *figures (connected components)* and determining properties of the figures. The reconfigurable bus is used to isolate individual figures so as to be able to efficiently extract information concerning multiple figures in a digitized image. A subbus is created and dedicated to keep track of all deliberations with respect to each figure.

Theorem 4.4 *Given an $N^{1/2} \times N^{1/2}$ digitized image mapped one pixel per processor onto the processors of a reconfigurable mesh of size N in a natural fashion, in $O(\log N)$ time the figures (connected components) can be labeled.*

Theorem 4.5 *Given an $N^{1/2} \times N^{1/2}$ digitized image mapped one pixel per processor onto the processors of a reconfigurable mesh of size N in a natural fashion, in $O(\log N)$ time a closest figure to each figure can be determined.*

Theorem 4.6 *Given an $N^{1/2} \times N^{1/2}$ digitized image mapped one pixel per processor onto the processors of a reconfigurable mesh of size N in a natural fashion, in $O(\log^2 N)$ time the extreme points of the convex hull can be enumerated for every figure.*

Theorem 4.7 *Given an $N^{1/2} \times N^{1/2}$ digitized image mapped one pixel per processor onto the processors of a reconfigurable mesh of size N in a natural fashion, in $\Theta(1)$ time several geometric properties of a set S of pixels can be determined. These properties include marking and enumerating the extreme points of the convex hull of the points, determining the diameter of the points, determining a smallest enclosing box of the points, and determining a smallest enclosing circle of the points.*

5 Conclusion

This paper considers the reconfigurable mesh as a viable alternative to a variety of processor organizations. We have presented efficient implementations of fundamental data movement operations for the reconfigurable mesh and have shown that it can be used as a universal chip, in that the reconfigurable mesh is capable of simulating any organization of processors occupying the same area and using a planar wiring layout without loss of time. We have also presented algorithms that show how the reconfigurable mesh can efficiently solve a number of graph and image problems using the fundamental data movement operations. The running times of these algorithms are asymptotically superior to running times of solutions for the mesh with multiple broadcasting, the mesh with multiple buses, the mesh-of-trees, and the pyramid computer. Further, we have shown that there are problems for which solutions on the reconfigurable mesh are more efficient than those possible for a PRAM.

6 Acknowledgments

The work of R. Miller was supported in part by NSF grant DCR-8608640. The work of V.K. Prasanna-Kumar was supported in part by NSF grant IRI-8710863. The work of D.I. Reisis was supported in part by DARPA contract F 33615-84-K-1404 monitored by the Air Force Wright Aeronautical Laboratory. The work of Q.F. Stout was supported in part by NSF grant DCR-8507851, and by an Incentives for Excellence Award from Digital Equipment Corporation.

References

[1] Alok Aggarwal, *Optimal Bounds for Finding Maximum on Array of Processors with k Global Buses*, IEEE Transactions on Computers, vol. C-35, no 1, pp 62-64, Jan 1986.

[2] M. Atallah and R. Kosaraju, *Graph problems on a mesh connected processor array*, JACM, 1983.

[3] S. H. Bokhari, *Finding Maximum on an Array Processor with a Global Bus*, IEEE Transactions on Computers, Vol. C-33, No. 2, February 1984, pp 133-139.

[4] D. M. Champion and J. Rothstein, *Immediate parallel solution of the longest common subsequence problem*, 1987 International Conference on Parallel Processing, 70-77.

[5] M. Furst, J. Saxe and M. Sipser, *Parity, Circuits and Polynomial Time Hierarchy*, Proc. IEEE Foundations on Computer Science, pp. 260-270, 1981.

[6] H. Li and M. Maresca, *Polymorphic-Torus Network*, Proc. International Conference on Parallel Processing, 1987.

[7] R. Miller, V.K. Prasanna Kumar, D. Reisis, and Q.F. Stout, *Parallel computations on reconfigurable meshes*, Tech. Rept. 229, Dept. of EE-Systems and IRIS, USC, March, 1988.

[8] R. Miller and Q. F. Stout, *Data Movement Techniques for the Pyramid Computer*, SIAM Journal on Computing, Vol. 16, No. 1, pp. 38-60, February 1987.

[9] R. Miller and Q. F. Stout, *Some graph and image processing algorithms for the hypercube*, Hypercube Multiprocessors 1987, SIAM, pp. 418-425, 1987.

[10] R. Miller and Q. F. Stout, *Parallel Algorithms for Regular Architectures*, The MIT Press, 1988.

[11] D. Nassimi and S. Sahni, *Data Broadcasting in SIMD Computers*, IEEE Transactions on Computers 1981.

[12] V. K. Prasanna Kumar and C. S. Raghavendra, *Array Processor with Multiple Broadcasting*, Proceedings of the 1985 Annual Symposium on Computer Architecture, June 1985.

[13] V. K. Prasanna Kumar and M. Eshaghian, *Parallel Geometric algorithms for Digitized pictures on Mesh of Trees organization*, International Conference on Parallel Processing, 1986.

[14] Y. Shiloach and U. Vishkin, *A $O(\log N)$ Parallel Connectivity Algorithm*, Journal of Algorithms 3, 1982.

[15] L. Snyder, *Introduction to the Configurable, Highly Parallel Computer*, Computer 1S(1):47-56, January, 1982.

[16] Q. F. Stout, *Mesh Connected Computers with Broadcasting*, IEEE Trans. on Computers C-32, pp. 826-830, 1983.

[17] L. G. Valiant, *Parallelism in comparison problems*, SIAM J. on Computing 3, 1975.

[18] C.C. Weems, S.P. Levitan, A.R. Hanson, E.M. Riseman, J.G. Nash, D.B. Shu, *The image understanding architecture*, COINS Tech. Rept. 87-76, University of Massachusetts at Amherst.

A Pipelined Dataflow Processor Architecture Based on a Variable Length Token Concept

Kaoru UCHIDA and *Tsutomu TEMMA*

C&C Information Technology Research Laboratories
NEC Corporation
Miyamae, Kawasaki, 213 Japan

Abstract

A dataflow processor architecture is presented which enables achieving high speed processing for vector data through a pipelining technique. A new dataflow concept, "Variable Length Token", is proposed for enhancing data processing capability and flexibility.

A *variable length token*—VLT—, is a token set consisting of a specifiable quantity of fixed size tokens. Multiple tokens to be processed together form a VLT and flow so as to maintain their consecutivity. In the proposed processor, a VLT is taken as a unit of processing both in firing control and in operations. This technique reduces the inter-token synchronization and communication overhead common to conventional dataflow machines, and it facilitates the handling of composite data, such as multiple precision data, vector data, and structured data, in a static dataflow model.

Also discussed is a system architecture with the multiple processor elements able to operate in parallel. System performance analysis results show that the system is particularly well suited to pattern processing.

1. Introduction

A data-driven computation model, in which the activation of an instruction execution is determined by the availability of its operands, can efficiently extract and exploit the concurrency inherent in computation [1]-[3]. Since the model was proposed, several computers with dataflow architecture have been proposed, designed and became operational in various fields [4]-[6].

Dynamic architecture is adopted in several machines, including, for example, the University of Manchester's machine [7], the machine by MIT's Arvind and his group [5], and the SIGMA-1 by the Electrotechnical Laboratory in Japan [8]. They are intended to cover rather large scale computation with big hardware--and sometimes multiprocessor—system.

NEC's TIP (*Template-controlled Image Processor*) project [9][10] has designed and developed a static dataflow VLSI "ImPP" (μPD7281), which can achieve high speed processing on a sequence of data through a pipeline approach and thus is especially well-suited to image processing applications [11]. Instruction level parallelism is achieved by pipelining the execution of operations in ImPP. Accordingly, this fine-grain data-driven parallelism enables efficient utilization of the processing (functional) unit, which leads to increased performance, for "irregular" data as well as regular vector data. In an ImPP multiprocessor system, moreover, individual ImPPs, sequentially connected in an array, can execute programs in parallel. The system has consequently been able to prove its high speed processing capability on a large amount of data in such applications as image processing, etc. [12][13].

Pattern processing, on the other hand, deals with a huge amount of two dimensional data and thus requires high speed computation capability, which has so far only been met with special and expensive hardware. Nevertheless, the practical systems to cover it have to be handy and with high cost-performance. The computations involved in such applications, furthermore, may demand extreme flexibility and a high level programmability for the processors.

Pipelining technique, which permits concurrent operations, is an effective approach to meet those demands and is widely used to attain high performance with comparatively small hardware. To assure that the pipeline hardware would be flexible and optimally utilized, data-driven control is an appropriate measure.

In a conventional dataflow approach, however, and especially in static models, the unit of computation is always a single data packet—known as a token—; at the most only two tokens can be processed at one time. To handle composite data, such as multiple precision data, vector-type data, or structured data, synchronization and communication among those composite data tokens have to be programmed by combinations of dyadic operations, overhead of which leads to a large amount of token flow traffic and thus degrades the performance.

This paper describes a dataflow pipeline processor architecture employing the authors' newly proposed "Variable Length Token" technique to overcome the problems described above. A *variable length token* (VLT) is a token set consisting of a specifiable quantity of fixed size tokens; the proposed processor "*V-TIP*" (*Template-controlled Image Processor with VLT*) deals with it as a unit of computation. Tokens in a VLT always flow consecutively in the V-TIP processor element and in the V-TIP multiprocessor system, which enables composite operations.

In Section 2, the V-TIP processor architecture is explained in detail; Section 3 covers the concept, implementation, and resulting usefulness of the variable length token. The architecture for a dataflow processing system with this proposed processor is presented in Section 4, which further explains how the parallel processing is implemented. Finally, the V-TIP system performance is discussed in Section 5.

2. Advanced Dataflow Processor V-TIP

Pipeline and Dataflow

In pattern processing, such as image processing and pattern recognition, and in general purpose numerical computation, such as numerical simulations and solution of large systems of equations, vector data is the major element in the computations. To handle efficiently a large amount of uniform and sequential data like vectors, a pipeline technique is very convenient and is widely used in all kinds of high speed processors. Pipeline techniques may be classified into two categories: one is instruction level pipelining, in which data items flow through functioning elements that are lined up in sequence and operate in parallel; the other is more fine-grained pipelining, in which each instruction execution is partitioned into multiple stages of subfunctions that operate in parallel on each data item in a pipeline manner.

With both of these approaches, however, though designs can be optimized for high level performance in specific applications, the number and allocation of stages are fixed, which considerably limits flexibility.

In the system proposed in this paper, in order to make pipelining more flexible and programmable, dataflow architecture has been adopted in the processor. In the dataflow approach, a tag to identify the data accompanies the data itself through the pipeline. A single data item with its tag is called a *token*, the unit to be handled and processed in dataflow computers. As Fig. 1(a) illustrates, a token in our architecture, when in the V-TIP processor element, comprises a *Link Table Address*, a flag called *control flag*, and a data value. A *Module Number* for specifying the processor of destination is affixed while the token is on the outer system bus (*outer TIP bus)* outside the processor elements (Fig. 1(b)).

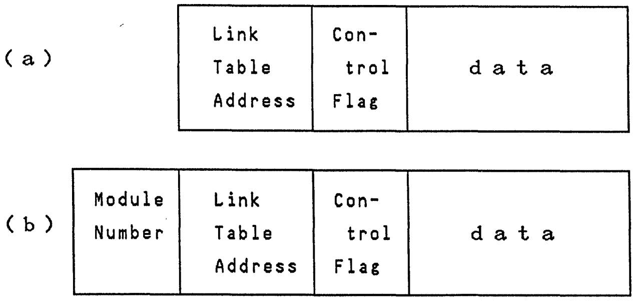


Figure 1. V-TIP token format.
(Control Flag field contains a "VLT Flag".)
(a): when in a V-TIP processor element
(b): when on the outer TIP bus

The Link Table Address is an address for accessing the program table—LT(*Link Table*)—in the processor element. The control flag specifies the class of the token—whether it is a program load token, status dump token or an executable token —, and gives information about VLT structure.

With this dataflow pipeline architecture, there is no need for the processor to "fetch" instructions between carrying out them, allowing all the pipeline stages to operate fully. The processor will maintain maximum performance, so long as data is constantly fed to the pipeline. Thus, vector data processing is especially efficient, since its flow is uniform and consecutive. This makes this particular architecture especially well-suited to pattern processing and scientific numerical computations that must handle large amounts of vector and matrix data.

Overall Architecture

Figure 2 shows the architecture of the proposed processor, V-TIP (*Template-controlled Image Processor with Variable length token*).

As Fig. 2 shows, V-TIP is composed of multiple functional modules which operate upon the flowing tokens completely in parallel. Some of the modules, moreover, are partitioned into several stages to achieve more concurrency and to speed up the pipeline cycles. A token flows along the outer TIP bus and goes into the processor through a Bus Input Controller (BIC). Tokens on the outer TIP bus each contain data to be processed, a Module Number (MN) to identify their processor of destination, a Link Table Address (LTA) to refer to the inner program table in the destination processor, and a control flag (CTLF). Part of the control flag—the "VLT flag field"— is also used to identity the tokens in VLT from others tokens. VLT implementation and usage will be described later.

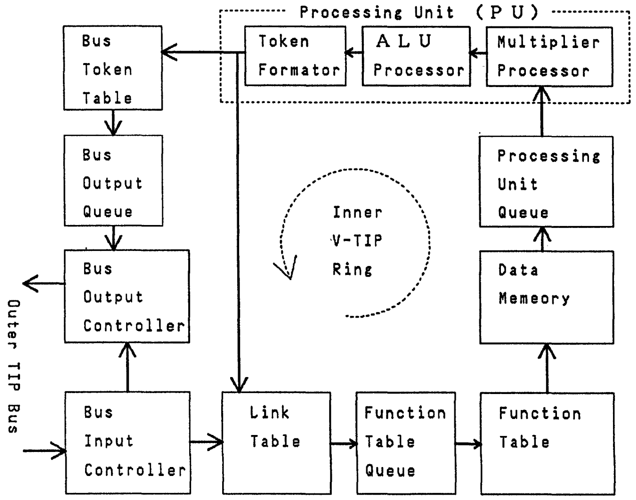


Figure 2. V-TIP architecture.

If the destination of the token, indicated by its MN field, matches the MN assigned to the V-TIP beforehand, the token is sent into its Link Table (LT). If the destination is not for this V-TIP, on the other hand, the token is sent directly back to the outer TIP bus through the Bus Output Controller (BOC) as a "pass token". The Link Table maps the Link Table Address (LTA) field of the incoming token to a next LTA and to a Function Table (FT) access address. The token is sent to the FT through the Function Table Queue.

Firing Control and Operand Fetch

In the Function Table (FT), the incoming token undergoes firing control —wherein it is checked to determine if the two operands needed for the dyadic operation are both available—, and the waiting token, if it exists, is extracted from the Data Memory (DM). The V-TIP employs "queued architecture"" and thus multiple tokens —or multiple VLTs—can wait in a first-in-first-out (FIFO) buffer area allocated in the DM.

The firing control at the Function Table and Data Memory enables both inter-token synchronization and inter-VLT synchronization. A VLT can wait for the corresponding VLT and, when they match, they are sent to the Processing Unit together. By use of this matching mechanism, two VLTs, of length M and of N, can be synchronized and concatenated to construct a VLT of length $M+N$. (Here, "A VLT of length L" means, naturally, "a VLT consisting of L tokens.")

Token generation is needed in the Function Table, in case a longer VLT than the input token is needed as a result of firing control, as in the above-mentioned VLT concatenation for example. While the generation is carried out in the Function Table, a busy signal from the Function Table keeps the Function Table Queue from sending successive tokens to the Function

Table.

When the token pair is fired in the Function Table, it then fetches a PU instruction code there and is sent to the Processin Unit Queue (PUQ). The PUQ serves as a token buffer and has tokens wait till the Processing Unit accepts the next token input. This buffering is indispensable to smooth irregularities of processing and thus to attain effective utilization of the processing power.

Processing Unit Architecture

The Processing Unit (PU) consists of a Multiplier Processor (MLP), an ALU Processor (ALUP) and a Token Formator (TF), that are connected sequentially and operate in a pipeline manner.

The Multiplier Processor (MLP) contains a multiplier and executes one-word by one-word multiplication, bit shift and bit rotate operations on fixed point format data. An adder for the exponent parts of floating point format data is also provided to carry out floating point data multiplication.

The ALU Processor (ALUP) performs arithmetic and logical operations. The ALUP has registers and carrys out status dependent processings using registers. The register is used, for example, to accumulate values or to detect the minimum value in the sequence of tokens.

The Token Formator (TF) makes up a result token data from the ALUP output by normalizing floating point data format. The LTA field for the result token which indicates its destination, can be modified according to the result of the PU operation, indicated by the status flags of the ALU, causing the token to branch conditionally to other destinations.

The Processing Unit can also be used to generate multiple tokens out of one given token. It is used mainly to:

1) Copy and distribute result data to different destinations.
2) Make a sequence of tokens (like tokens with data '0', '1', '2', .. , '15'), when given the initial data 0, the difference 1, and the length of the sequence 16.
3) Generate VLTs out of given tokens, which is accomplished by modifying the VLT-flag field for each token.

When multiple tokens are being generated, the Processing Unit sends a busy signal to the Processing Unit Queue; accordingly, the Processing Unit Queue stops its output to the Processing Unit.

After a set of operations is applied to a token in the Processing Unit, the PU output token is sent either to the Link Table or to the Bus Token Table, depending on the code. If more operations are to be performed on the token in this processor, it is accepted by the LT, goes on to the next operation, and keeps on going around the Inner V-TIP Ring (circular pipeline), which consists of the LT, FTQ, FT, DM, PUQ and PU, until finally outputting to the outer TIP bus.

Token Output

If the token from the Processing Unit is to be sent to the outer TIP bus, it is passed on to the Bus Token Table (BTT). In the BTT, the token accesses the table and fetches a set of identifiers, a Module Number (MN), and an Link Table address (LTA) that are needed for tokens on the outer bus. The token is then sent to the Bus Output Queue (BOQ), where it is kept waiting while the outer TIP bus is busy, and to the outer TIP bus through the Bus Output Controller (BOC). In the BOC, the pass token flow from the Bus Input Controller and the output token flow from the Bus Output Queue are controlled and merged.

3. Variable Length Token

VLT Concept

In conventional static dataflow machines, the unit of computation is always one token. Therefore, to handle composite data, such as multiple precision data, vector data and structured data, it is necessary to program explicitly the synchronization and communication among those data tokens by combinations of dyadic operations. The reduction in the effective utilization of processing power, due to this synchronization overhead, is one of the biggest problems in fine-grained dataflow machines.

When adding two double precision values with fixed point format using conventional dataflow machines like the μPD7281, for example, a flow graph, such as that illustrated in Fig. 3, is needed to describe the program. This flow graph shows that the token has to go around the inner ring of the processor sequentially twice. Moreover, the lower word processing and higher word processing should be programmed separately, considering synchronization between them.

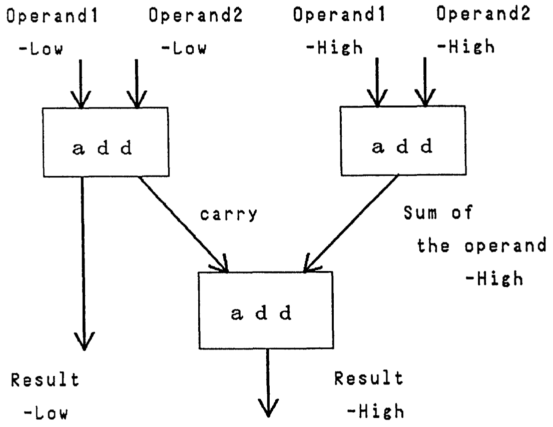


Figure 3. Dataflow graph for double-precision data addition in a conventional dataflow machine

The *variable length token* technique, newly proposed here, provides a solution to these problems. A variable length token (VLT) is a token set consisting of a specifiable quantity of fixed size tokens. The tokens in a VLT always flow consecutively in the V-TIP processor element and in the system using the processor. The VLT is taken as a unit of processing both in firing control at the Function Table and in operations at the Processing Unit just as a single token is considered in conventional dataflow machines.

The VLT technique enables flexible flow control and enables high speed processing of composite data structures in a static dataflow model. It provides the following three major advantages, details of which will be explained in the next section:

a. Overhead reduction
To reduce the communication overhead between tokens, the VLT technique provides a way to increase data granularity by concatenating relevant data set together.

b. Functionality in vector operations
The functionality of vector operations can be assured by the consecutivity of the data items involved, which are chained in a VLT. Vector data can, therefore, be efficiently handled with the help of registers, in case of data sequence accumulation. for example.

c. Affixation of control information
Appending an index or a relevant address by means of VLT allows the control information to go with the data token, when the token changes its path depending on its data value. This technique facilitates the re-ordering for token streams when they are to be merged after branching.

VLT Implementation

A token has a VLT flag for VLT identification, which is a part of the control flag field, besides a data value and a Link Table Address. Tokens in a VLT, though they may have different VLT flags, have the same Link Table Address and are destined to the same node in the dataflow graph. The VLT flag of a token indicates whether:

a) It is the last (tail) token for a VLT or it is a single-token-VLT.

b) It is in the midst of a VLT, so that the next tokens must be handled consecutively in regard to the previous token.

c) The token has a special meaning, such as an "index" token, which will be explained later.

VLT consecutivity is assured by controlling the merging of the two token streams using the information included in the VLT flag of the tokens involved. For example, in the Link Table (LT), where tokens from the Bus Input Controller (BIC) and from the Processing Unit (PU) merge, the LT checks the VLT flag of the token currently being sent to the Function Table Queue, then determines from which direction it should accept the next token. If the flag indicates the token is not the tail of the VLT, the next token must be accepted from the same direction as the previous one, since the two belong to the same VLT and the sequence may not be broken. If it is the tail token, on the other hand, then the direction of the next token may be determined according to the priority rule, as there are no other restrictions. The same control scheme is employed at the Bus Output Controller, which accepts tokens from Bus Output Queue and Bus Input Controller.

In the Function Table (FT) and Data Memory (DM), as has been explained in Section 2, the firing (enabling) control for tokens is carried out. In token synchronizations by firing control, VLTs are regarded as one token. When two VLTs are to be synchronized, the one that comes to the FT first waits in DM till the other comes. The two VLTs then match each other in the FT and go to the Processing Unit together. As is shown in Fig. 4, the `Queue-Concatenate` operation allows two VLTs, of length M and length N, to synchronize, and then to construct a VLT of length $M+N$ by concatenation.

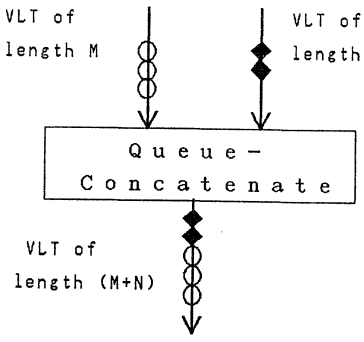


Figure 4. Synchronization and concatenation of two VLTs.

VLT: Use and Effectiveness

The following presents some examples of the use and effectiveness of the VLT technique in handling composite data in various processings, in comparison with conventional techniques.

i) Multiple Precision Data Multiple precision data, like double precision or quad precision data, can be represented using VLTs. In the case of double precision data, a VLT consisting of two tokens, one token with the lower word data at the front and one with the higher word data in the back, represents a data item (See Fig. 5 (a)). When two double precision data are to be added, for example, two operand VLTs meet in corresponding order in the Function Table and the Data Memory. (In Fig. 5, an operand VLT, with data `opd2 L` and `opd2 H` —the operand2 low word and high word, respectively—, had been waiting in the Data Memory.) In FT, at the same time, the tokens fetch an PU instruction code `add-multiple` (Fig. 5 (b)).

The token with lower operands goes into the Processing Unit (PU) first, two operands are added there, and the carry of the addition is kept in its carry register. The tail token with the higher word operands then goes into the PU, where another addition is performed. But this time, the carry from the lower words addition is added at the same time. The resultant data, as in Fig. 5 (c), is sent out from the Processing Unit in the same form as in the start of the previous operation: the two token VLT, with the lower word in the front and the higher word in the back.

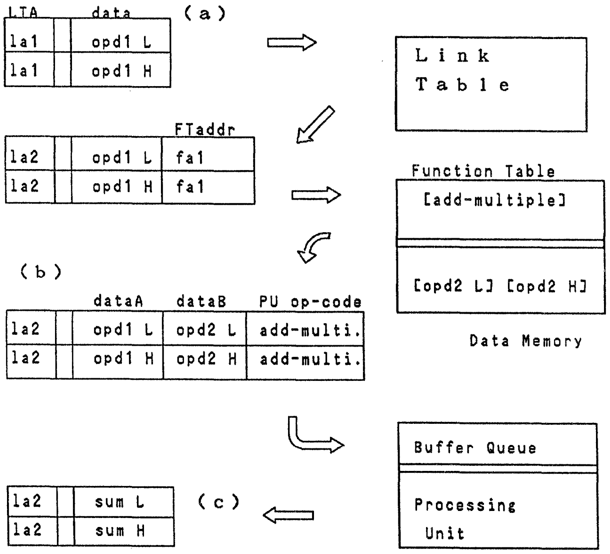


Figure 5. Token format transition in double-precision data addition with VLT.

With this technique, a data with multiple precision can be handled in the same manner as a single precision ordinary data. The processing cost for synchronization between lower and higher tokens with a conventional dataflow approach can be eliminated, as may be seen in Fig. 6, a dataflow graph illustrating a program for double precision data addition. This figure, for a processor using the VLT technique, may be compared to the graph in Fig. 3, for a conventional case. The time lag between the start of and end of an operation (latency) may, moreover, be reduced by half.

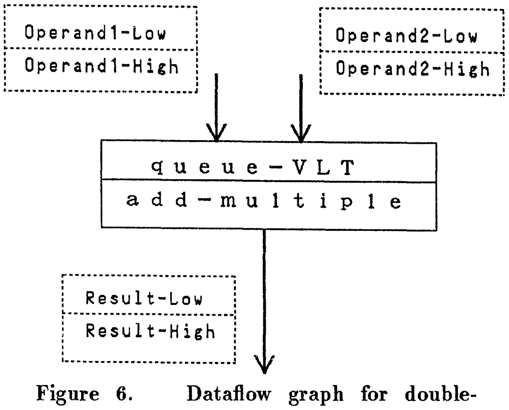


Figure 6. Dataflow graph for double-precision data addition in the proposed V-TIP.

This approach can be applied to multiple (longer than two) word data addition. A similar technique, moreover, can cover multiplication of a multiple word data $(A_1, A_2, \ldots, A_n)$ with a single word data B. In the latter case, each word of the 'A' data, A_k, is multiplied by B in the Multiplier Processor, and then, in the ALU Processor, the lower part of the product $P^{(low)}{}_k$, the higher part of the previous product $P^{(high)}{}_{k-1}$, and the carry of the previous addition C_{k-1} are added. In order to multiply two multiple word values, moreover, the enabling section (the Function Table and the Data Memory) provides facilities to decompose one VLT to single component tokens before multiplications and add the products with word shifts afterwards.

ii) Vector Accumulation A set of vector data can be represented by a VLT, and by using registers in the Processing Unit, the summation of the data in a vector can be obtained. One practical example is the calculation of the inner product of two vectors, which is often used in matrix multiplications and convolutions in spatial filters.

To calculate the inner product s_N of vector a_i and vector b_i, for i=1 to N, one must calculate sequentially, with conventional machines such as the μPD7281, as follows:

$$\begin{array}{ll} s_1 = a_1 b_1 & t_2 = a_2 b_2 \\ s_2 = t_2 + s_1 & t_3 = a_3 b_3 \\ \vdots & \vdots \\ s_{N-1} = t_{N-1} + s_{N-2} & t_N = a_N b_N \\ s_N = t_N + s_{N-1}. & \end{array}$$

In this case, the whole operation is carried out using combinations of dyadic operations and takes $2N-1$ processing clocks, and the latency is $N-1$ times the number of steps required for the token to go around along the inner loop of the processor (with μPD7281, it is 7 steps).

In the V-TIP proposed dataflow processor, a vector is represented by a VLT with the same word length. In this case, a VLT consisting of N tokens with data $a_1, \ldots, a_N$ comes into the processor from the outer TIP bus, while the coefficients $b_1, \ldots, b_N$ reside in the Data Memory. A token with data a_k in the incoming VLT matches(and fetches) the corresponding coefficient b_k in the Data Memory, and the two operands go to the Processing Unit.

In the Multiplier Processor of the PU, data a_k and b_k are multiplied. Immediately after the multiplication, the product is added to the value in the register of the ALU Processor, whose initial value is 0. Accordingly, after the multiplication of the last *N-th* vector components, the product $a_N b_N$ is added to the value in the register, which is the *(N-1)th* partial sum s_{N-1}, and the inner product s_N, is obtained. The token with the total sum is then sent out and the register in the ALUP used in the operation is cleared to zero for the next use. Consequently, calculation of the inner product of two vectors of length N can be accomplished in N processing clocks. Moreover, the latency is only N clocks.

This improvement has been made possible by the fact that the Processing Unit has a multiplier (in the Multiplier Processor) and an adder with a register (accumulator, in the ALU Processor) in sequential order, both operating in parallel. Thus, the two component operations, multiplication and addition, can be executed in a pipeline manner.

One thing that should be noted here is the use of a register in dataflow machines. One of the advantages of the dataflow computation model is the referential transparency ensured by the functionality of operations. The use of registers in dataflow machines is, therefore, generally considered harmful, as it causes side effects and detracts from the above advantage, although the introduction of registers does serve to speed up the processing. In the proposed V-TIP architecture, however, the status dependencies are confined in VLTs, as the data on the registers is set, referenced and modified only by the tokens in the same VLT—which always flow consecutively—, and is cleared by its tail token. Thus register use has no side effect on tokens, other than those in the involved VLT. Consequently, users will be able to benefit from the high speed computation enabled by the registers, without having to give consideration to the exclusive register utilization or to program specifically the management of inter-token synchronizations, in order to avoid side effects.

iii) Indexing In this V-TIP, as in μPD7281, the sequence of tokens flowing on the same arc in a fixed order and carrying out same instructions is called a *"stream data."* Data in this stream will be processed as intended, if and only if the order of tokens in the stream is preserved. Data in a stream are, for example, retrieved from the memory, undergo several instructions and are finally stored in the memory in the correct place, because the final order of the data tokens in the stream is the same as that of the prepared address tokens.

However, when some of the tokens in the stream conditionally switch —as a result of a `conditional-branch` instruction—, depending on the token data value and take a different dataflow path from that of others, then the token sequence from different paths cannot satisfy the above condition after the paths merge. This is because the steps, which the tokens on different paths require to run from the branch point to the merging point, differ depending on the path involved.

In this case, it is possible to make a VLT of a token pair, a data token and the address token with which the data is finally to be stored in the memory. (Here, the second token carrys control additional information for the first data token, so can be called an *"index token"*.) This technique causes the index token with the address to "accompany" the data token along the same path, and makes it possible to ensure the correctness of memory storage in whatever order the data is written into the memory, when this write action takes place after the processing is over.

A token with a number, instead of a memory address, to indicate the original position of the data token in the stream, can also be appended to it as an index token in the form of a VLT. This number can be used to re-order the tokens using a temporary buffer.

Figure 7 shows how this is done. An index token with "Context-k" ($k = 1, 2, \ldots, N$), which denotes the original position of the "Data-k" token in the sequence, is affixed to the data token using VLT. It always goes with the data token but passes through the Processing Unit with no-operation being carried out. When branching depending on the data value occurs the Link Table Address field —which serves as a token identifier— of the index token is modified accordingly, and keeps on taking the same route as the data token. Finally when the VLTs from different paths are to be merged, they come into the same node and there the data tokens are written into the memory at the address indicated by the "context" in the index token. After all the tokens arrive and are stored into the memory, the original data stream, Data-1, Data-2, ..., Data-N, can be obtained by reading it sequentially.

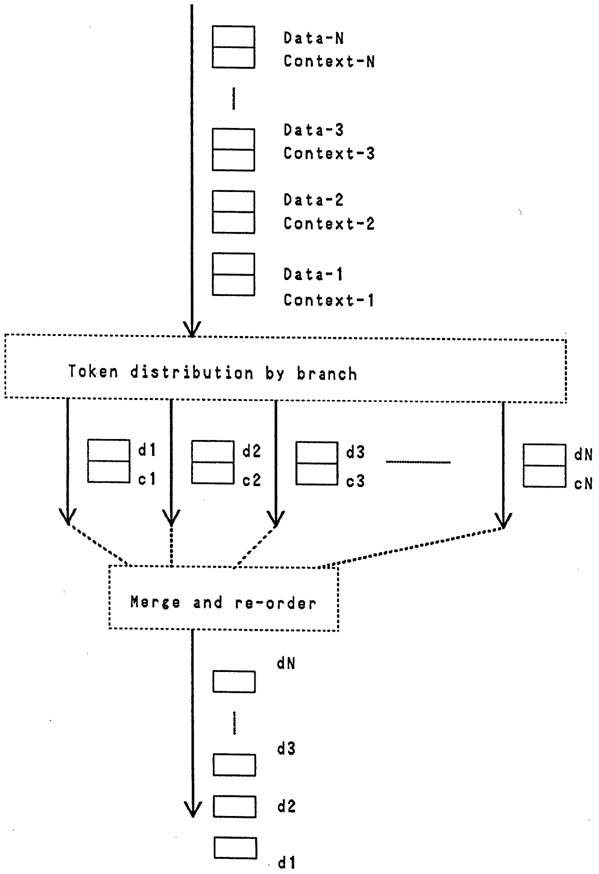


Figure 7. Context addition to data tokens using VLT. Tokens from different paths can be re-ordered using the indexes appended to the data tokens.

This technique facilitates the handling of stream data with less consideration for execution timing and provides an advantageous way to high level language programming.

In another example, when subroutines are to be called from different places, the information about the calling sites has to be preserved to distinguish the invocations and to return to the original calling sites. The user, in that case, can append context information, an identifier for the invocation, to the data token as an index token by means of VLT. (It should be noted here that the tagged-token approach used in dynamic architecture machines, which is common and effective in handling reentrancy, requires complicated hardware and is not appropriate for implementation on simple VLSI oriented machines like V-TIP [6].)

As has been shown, any additional control information associated with the data can be affixed to the data token by an index token using VLT. While these index tokens pass the Processing Unit with no-operations carried out and the indexes untouched, these no-operations result in waste of the PU computation power. Therefore, during the operations that do not change the order of the stream data—which is often the case—, the index token can be removed from the data token and can wait in the memory. When it is necessary, the data and index tokens are synchronized and concatenated again.

4. V-TIP Multiprocessor System

The authors propose here a system with V-TIP dataflow processors for practical implementation of the VLT technique.

The basic system, shown in Fig. 8, consists of multiple data flow processor elements (V-TIPs) and an *"Interface Processor"* (IFP), connected by a ring-shaped outer TIP bus. The Interface Processor, as well as the V-TIP, is data-driven and deals with the interfaces among the V-TIP, the *Memory Unit* and the host computer. Individual V-TIPs operate concurrently and independently, depending on the programs loaded into them. They access data in the Memory Unit by sending read and write request tokens to the IFP.

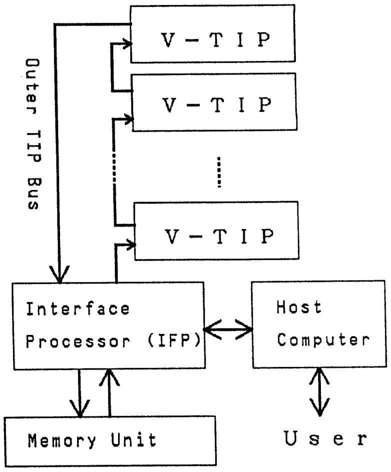


Figure 8. V-TIP Multiprocessor System.

At the very beginning, the V-TIP program is stored in the Memory Unit in the form of program load tokens. When the user gives a program load command through the host computer, the IFP begins reading program loading tokens in the Memory Unit and sending them to individual processors. Tokens reach the V-TIP, which is denoted by its Module Number field, and the contents of the tokens are set to the local memories in the processor. i.e. to the Link Table, the Function Table, the Data Memory, and the Bus Token Table.

When the user commands the processing start by sending tokens to V-TIPs. V-TIPs begin operating by sending data read request tokens to the IFP; the IFP then fetches the data from the Memory Unit, constructs tokens and sends the data sequences (*streams*) to the requesting V-TIPs. The tokens flow into a V-TIP, circulate along the inner ring bus, undergo the pertinent programmed operations, and are sent out to the IFP as data write tokens when the processing is over. The IFP automatically generates sequences of write addresses, and the arriving stream data are stored in the Memory Unit in the appropriate positions.

Generally the clock rate with which the processor element operates and the token transmission rate on the outer TIP bus (50 nsec to 100 nsec) is much faster than the memory access rate at the Memory Unit, i.e. around 400 nsec to 600 nsec with Dynamic RAM. (With Static RAM, of course, the access time is comparable to the clock cycle. However, for achieving high cost-performance in a compact system with a large amount of memory, Dynamic RAM is more realistic.) Thus, a technique to enhance the total memory access rate is needed to make the most of the processing speed available when using the V-TIPs.

In the proposed Interface Processor for the V-TIP system, simultaneous memory access by interleaved memory is used to permit rapid access to the Memory Unit. The Memory Unit for the system comprises 16 memory modules and supports parallel access to any consecutive addresses, in both horizontal and vertical sequence of 16 points on a 2-dimensional area. When the sequential memory read is requested by a V-TIP, for example, the Interface Processor generates access addresses, sets them in the registers each corresponding to the 16 memory modules, and gives the read signal. The retrieved data are copied into the data registers. Each register set has two groups of registers; there are 16 plus 16 read address registers, for instance. Those two groups are used like swinging (double) buffers.

By connecting multiple V-TIPs on the ring bus, as with this system, the system performance is enhanced in proportion to the number of V-TIPs, since the processors can work independently and concurrently. In such a case, of course, performance degradation due to a bus bottleneck and shared-memory access contention has to be considered. Moreover, when memory access is so frequent that it imposes the limit of system performance (for example, the interleaved memory cannot provide increased speed for random access to the Memory Unit), multiple Memory Units have to be and are able to be connected to the outer TIP bus. In this case, each Memory Unit is connected to the outer TIP bus through an IFP, and is assigned a different module number for accessing.

As has been explained, the V-TIP system has concurrency in two levels; individual instructions are partitioned into pipeline stages that function in parallel, and processors in the system function in parallel as tokens flow.

To meet larger-scale computation requirements, a more highly-parallel system can be built by connecting a number of the above TIP ring units (multiple V-TIP elements and an Interface Processor connected by the outer TIp bus) together. The Interface Processor (IFP) has communication ports that exchange communication packets in a byte-serial manner with IFPs in the neighboring TIP ring units. Thus, this architecture permits construction of a massively parallel system, suited to the needs of the user.

5. System Performance

The V-TIP system performance was evaluated by analyzing and simulating basic image processing and numerical computation application programs.

Evaluation

Evaluation was begun by making a dataflow graph for an application computation, the equivalent of a flow chart and a program description for control-flow machines, and then summing up the clock cycles needed for the Processing Unit to cover the whole process. Underutilization is analyzed by estimating the number of idle steps during the execution. The number of idle steps can be approximated by calculating the quantity of token flow on arcs and analyzing the dependency among the instruction nodes in the dataflow graph. This is because if only one token flows between two nodes and if there is no other parallelism, for example, the instructions on those nodes are executed sequentially and cannot be pipelined, thus leading to an idling of the Processing Unit between two instruction executions.

Assumed Parameters

In the performance analysis, the pipeline cycle for the dataflow processor V-TIP was assumed to be 40 nsec, token transmission rate on the outer TIP bus to be 80 nsec/token, and memory access speed to be 400 nsec/access. (Remember that the memory unit is 16-way interleaved, so it can both read and write data in 50 nsec, if the access is sequential.) This means that a V-TIP can attain a maximum performance of 25 MOPS if the stream data are fed to the processor element at a sufficient rate and its Processing Unit functions fully.

Applications and Speeds

a) Spatial Filter In a spatial filter operation with a 3 × 3 kernel on a 256 × 256 pixel image, three VLTs containing 3 pairs of data —for 3 neighboring points—and the coefficients effectively utilize the pipeline hardware. The nine-point convolution operation for one output pixel requires 23 clocks of the Processing Unit, i.e. 920 nsec/pixel ($40nsec \times 23$). The number of memory access requests from the V-TIP to the Interface Processor (IFP) can be reduced to only a pair of read and write requests per unit convolution by storing three lines of the original image in the Data Memory. In other words, a unit convolution occupies 160 nsec in the bus transmission and 100 nsec in the memory access. This means that this computation is processing-bound, not memory access bound or outer TIP bus transmission bound, and can be speeded up by using more V-TIPs in parallel. Since the token transmission rate is 160 nsec/unit (One read-request token and one write-data token go from a V-TIP to the IFP and this becomes the bottleneck), maximum overall performance can be attained when 6 processors are used in parallel, this being 160 nsec/unit, i.e. 10.5 msec ($160nsec \times 256 \times 256$) for a 256 × 256 image.

b) FFT A 2-dimensional complex Fast Fourier Transform was implemented with constant geometry algorithm on a 512 × 512 image of floating point format data. One butterfly requires 4 input tokens and 4 output tokens, taking 320 nsec ($80nsec \times 4$) for token transmission. One butterfly calculation consumes 18 PU clock cycles, that is, $40nsec \times 18 = 720nsec$. Accordingly, a two V-TIP system can complete this computation twice as fast as one V-TIP system, namely 360 nsec/butterfly. Thus, using this two V-TIP system, the whole operation takes $360ns \times 128 \times 256(lines) \times 9(stages) \times 2(directions) = 0.21$ sec.

c) Character Recognition To demonstrate the suitability of the V-TIP system for pattern processing, character recognition processing performance was studied. This method for recognizing printed alphanumeric characters, based on multiple discriminant analysis, includes the following procedures [14]:

i. Resample of the input image by a 5×5 spatial filter, thus composing a 50-dimension feature vector.
ii. Reduction of the dimension of the feature vector by multiplication with a 50×48 matrix.
iii. Calculation of the distances for each of the dictionary vectors.
iv. Detection of the nearest matching vector in the given dictionary by comparison of the distances.

The first and second steps each take 0.2 msec. In the third and fourth step, tokens for dictionary vector elements flow into the processor one after another. Differences from the corresponding elements are accumulated and then compared to the existing minimum. Each distance calculation takes 120 PU cycles and this is repeated 91 times, the number of dictionary vectors; thus, the last two steps calls for 0.44 msec.

The above evaluation shows, therefore, that a system with one operating V-TIP can perform printed character recognition at the rate of $(0.2 + 0.2 + 0.44 =)$ 0.84 msec/char with a 91-character dictionary of 48 dimensions. This performance, 1190 characters/sec, is about 240 times faster than that of a 16 bit micro-computer system implementing the same algorithm[14].

6. Conclusion

The proposed processor, *V-TIP*, consists of multiple functional modules operating in parallel. The inherent concurrency in computations is extracted and effective utilization of the Processing Unit is possible by employing dataflow scheme computation. The pipelining technique enables high speed processing, especially of vector data.

A new dataflow concept, the *Variable Length Token* (VLT) technique, is introduced to enhance data processing capability and flexibility, in which multiple tokens are controlled so that they flow consecutively throughout the system and are processed together. This technique, by providing a means to increase data granularity, reduces the inter-token synchronization and communication overhead for conventional dataflow machines. Thus, it facilitates high speed processing of composite data structures in a static dataflow model. Its effectiveness has been shown in multi-precision data computation and re-ordering of tokens.

The enabling section of the V-TIP offers measures to control VLTs, such as allowing two VLTs to synchronize, concatenating them and decomposing one into single component tokens; the Processing Unit operates on VLTs.

The architecture of a system with multiple V-TIP processing elements, an Interface Processor (IFP), and a Memory Unit has been explained. V-TIPs in the system can operate concurrently and independently, exchanging tokens with each other and with the IFP. The IFP supports interleaved parallel access to the Memory Unit so that the V-TIPs are supplied tokens at a sufficient rate.

V-TIP system performance has been estimated by analyzing application program executions. Results indicate that the system with the VLT technique produces high performance for vector-type data. The proposed architecture appears especially suitable for pattern processing.

Acknowledgments

The authors would like to thank Masanori Mizoguchi for his immense contribution to the work. They are also grateful to Ko Asai, Masao Iwashita and other laboratory members for their encouragement and advice.

References

[1] J.B. Dennis et al, "Data flow schemas," *Int. Symp. on Theoretical Programming, Lecture Notes in Computer Science*, Vol.5, Springer-Verlag, 1974, 187–216.

[2] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Dataflow Processor," *Proc. 2nd Ann. Int. Symp. Computer Architecture*, IEEE, 1975, 126–132.

[3] J.B. Dennis, "The Varieties of Data Flow Computers," *Proc. 1st Int. Conf. Distributed Computing Systems*, 1979, 430–439.

[4] P.C. Treleaven et al, "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Surveys*, vol.14, No.1, Mar. 1982, 93–143.

[5] Arvind and D.E. Culler, "Dataflow Architectures," *Annual Review of Computer Science*, vol.1, Annual Reviews Inc. , 1986, 225–253.

[6] A.H. Veen, "Dataflow Machine Architecture," *ACM Computing Surveys*, vol.18, No.4, Dec. 1986, 365–396.

[7] J.R. .Gurd et al, "The Manchester Prototype Dataflow Computer," *Commun. ACM*, vol. 28, No. 1, Jan. 1985, 34–52.

[8] K. Hiraki et al, "Maintenance Architecture and Its LSI Implementation of a Dataflow Computer with a Large Number of Processors," *Proc. of 1986 Int. Conf. on Parallel Processing*, 1986, 584–591.

[9] T. Temma et al, "Template-Controlled Image Processor TIP-1 Performance Evaluation," *Proc. of IEEE CVPR*, 1983, 468–473.

[10] M. Iwashita, T. Temma et al, "Modular Data Flow Image Processor," *Proc. IEEE COMPCON Spring '83*, 1983, 464–467.

[11] T. Temma et al, "Data Flow Processor Chip for Image Processing," *IEEE Trans. Electron Devices*, vol. ED–32, Sep. 1985, 1784-1791.

[12] T. Temma et al, "Chip-Oriented Data-Flow Image Processor TIP-3," *Proc. IEEE COMPCON Fall '84*, 1984, 245–254.

[13] M. Iwashita and T. Temma, "Data Flow Chip ImPP and Its System for Image Processing," *Proc. IEEE ICASSP '86*, Tokyo, Apr. 1986, 785–788.

[14] H. Kami et al, "Character Recognition by Two Stage Discriminant Analysis," *NEC Research & Development*, No.87, Oct. 1987, 20–25.

A DYNAMIC DATAFLOW ARCHITECTURE FOR IMAGE GENERATION

Philip C. Chao and Ming Y. Chern

AT&T Bell Laboratories
Naperville, IL 60566

Abstract -- In computer graphics, the ray tracing algorithm can generate highly realistic images. However, it has a major disadvantage : the high computational expense associated with generation of an image. To increase the image generation speed, our approach is to map the computation processes of ray tracing into a specialized dynamic data flow architecture for parallel processing. To support the computation further, we used a spatial-information hierarchy. A model for ray tracing computation based on probability is used to analyze the load. The architecture is modeled with a closed queueing network. Through the analytical models, we have studied the relative performance of the architecture under various load conditions.

1. Introduction

Dataflow architecture is an alternative to Von Neumann architecture and is capable of efficiently exploiting a massive amount of parallelism inherent in many types of computation. A dynamic dataflow architecture uses tagged tokens to unfold iterative computations so that a high degree of parallelism can be achieved [2] [13] [16]. This paper proposes a specialized dynamic dataflow architecture for the ray tracing image generation and investigates its performance for the task.

Ray tracing is a technique for generating images of three dimensional objects with a computer. Programs using ray tracing can simulate the effects of reflection, refraction and shadows to produce computer images that possess a strikingly high degree of realism. The technique was originally suggested by Appel [1] and later enhanced by Whitted and others to generate images according to the laws of optics [18] [12] [4].

In this scheme, a ray is fired from the viewer through the pixel into the world. The intersection between this ray and objects in the world determines the visible surface. Shadows are determined by firing rays from the intersection point toward the light sources. Two additional rays may also be fired from the intersection point depending on the surface characteristics, one along the reflected direction and the other along the direction of transmission. Figure 1 shows the ray tracing terminology used in this paper.

In the rendering process, the color of a pixel is determined by an intersection tree. The pixel is the root node of the tree. All other nodes in the tree represent ray-surface intersection points. Each arc in the tree represents a ray used to determine the color of the root pixel. A leaf node of the tree corresponds to an intersection point between the ray and either a non-reflecting, non-transparent surface, or the boundary of the modeled world. Figure 2 shows an example of an intersection tree. In the case of surfaces aligned in such a way that a branch of the tree is very deep (for example, two reflective surfaces in parallel can cause a tree to have infinite depth), the branch may be truncated at a predefined depth. This is reasonable since the truncated portion contributes very little to the color of the pixel. After the tree is completely grown, the colors of all nodes in the tree are computed and are used to find the color of their root pixel.

Using ray tracing, we may model accurately the distortions of reflecting and refracting surfaces, thus producing highly realistic images. However, there is a major drawback to ray tracing : very high computational expense. To determine the color of each pixel requires one to compute the intersection points between every ray in the tree and the surfaces in the scene. One way to reduce the amount of computation needed to produce an image is to divide the modeled space into subvolumes and to keep a note of the surfaces in each subvolume [3] [7] [10].. As a ray propagates from one subvolume to the next, the surfaces in each subvolume become candidates for ray intersection. Thus the nearest objects are the first candidates for intersection, leading to a quick determination of the closest object. One way to partition the space is to divide it uniformly. In this scheme, the traversing algorithm which traces only the relevant subvolumes is based on a three-dimensional scan conversion algorithm. Therefore, the time to find a small set of surfaces that potentially intersect the ray is O(M), where M is the number of subdivisions on each axis, and it is independent of the number of objects in the scene.

Ullner proposed a 2-D array of processor elements (or PEs) for ray tracing based on the 3D world space division parallelism model [17]. Each PE contains a general purpose processor, an intersection processor, and a memory module. Only those surface models intersecting a subvolume are kept in the corresponding PE that covers the subvolume. The two axes of the 3D space are directly mapped onto the processor array. The third dimension of the partitioning grid must be simulated within each processor in the array. As a ray travels across the space, the ray message travels across the corresponding processors. This approach has several disadvantages : (1) high storage requirement due to the need to store copies of the same object model in multiple PEs; (2) as the number of PEs in the system increases, the time for the ray to transverse in space increases; (3) difficulty in balancing the load due to its rigid mapping between subvolume and processor. Dippe and Swensen [6] relaxed the mapping function in order to balance the load and proposed a 3-D array of processor elements to perform ray tracing. However, the approach does not eliminate the other shortcomings and the mapping function is more difficult to implement.

In the paper [5] and [14], a different parallel processing scheme for ray tracing was proposed. In this scheme, the screen is partitioned into subscreens and each subscreen is processed by a PE in a multimicrocomputer environment. Object models are stored in each PE if their projection intersects the subscreen of the PE (called Y-clipping [5]). Additional object models may be fetched by a PE if there is a need due to the computation for shadow, reflection, or transmission. There are several disadvantages to this scheme: (1) it explores parallelism only at image level (i.e., screen); (2) as the area of each subscreen decreases, the number of copies of each object model in the system increases, and the benefit of using Y-clipping diminishes.

In viewing the shortcomings of the above architectures, we list the desirable features for a parallel ray tracing architecture as follows:

- Allowing parallelism among rays and among different computation processes for each ray.
- Keeping only one copy of each object model in the system regardless of the number of PEs used. This reduces the memory requirement of the system and improves the updatability for the object models in the scene.
- Allowing the empty-space transversal time for each ray to be virtually independent of the number of PEs in the system.

To achieve these goals, we proposed a dynamic dataflow architecture with a hierarchy of spatial information (locations of objects in space) and specialized execution modules for ray tracing image generation.

The next section describes the proposed architecture. A model for the ray tracing image generation computation based on probability is described in section 3. The queuing network of the proposed architecture is also described in section 3. The results are discussed in section 4. Section 5 concludes the paper.

2. The Architecture

To explore the parallelism in ray tracing image generation, a dataflow graph for ray tracing based on space subdivision is presented in figure 3a. The data tokens for the operators are shown in figure 3b. Basically, the diagrams indicate how to grow an intersection tree from a root pixel. The color of a pixel is obtained by accumulating all the color components inside the corresponding intersection tree.

In this paper, we propose to use a specialized dynamic dataflow architecture for ray tracing image generation. The processes in figure 3a map directly onto the modules in the dynamic dataflow architecture shown in figure 4. This architecture contains six types of modules : Ray Generator (RG), Empty Space Processor (ESP), Fetch Unit (FU), Intersection Processor (IP), Intersection Result Buffer (IRB) and Color Accumulator (CA). The modules are connected in a circular-pipelined fashion. Because all intersection trees are mutually independent, many intersections may be processed simultaneously. To allow a high degree of parallelism, multiple copies of each type of module may be used. In each module, execution starts once the "operand token" is available. If more than one "operand token" enters a processor, the processor places them in a queue, and executes them one at a time. In the system, the "root pixel" is used as the tag for most tokens for the purpose of accumulating color for each pixel. Tokens from P4 to P6 in figure 3a have longer tags due to the unfolding of iterative computation P5.

The ray tracing system keeps a hierarchy of subvolume and surface information. This "spatial-information" hierarchy includes an Object Model Storage (OMS), Subvolume Surface Lists (SSL), and an Empty Subvolume Map (ESM). (See Figures 3 and 4) The ESM stores empty/non-empty bits for all the subvolumes in the scene. This map is used to determine the first non-empty subvolume encountered as a ray travels in space. It allows rays to bypass empty subvolumes quickly. The SSL is a list of surfaces intersecting a given subvolume. Each entry in the SSL contains a surface ID and a pointer pointing to the beginning of the surface in the OMS. The SSL, however, adds a level of indirection for reading surface models; instead of having multiple copies of a surface in several subvolumes, now, in a multiple IP-OMS system, we need only pointers in multiple SSLs, one for each intersecting subvolume. Therefore, we need only one copy of each surface description in a multiple IP-OMS system.

The following describes the function of each module and how the architecture works.

The architecture acts as a display system attached to a host computer. After first initializing the system, the host computer must program the system with parameters. (For example, how to partition the world into subvolumes, the position and color of every light source, and parameters of the viewing pyramid.) Then, the host computer transfers all the object models and texture maps in the scene to the display system. This information is distributively stored in the OMS to allow parallel accessing.

The spatial information hierarchy is created by performing clipping on all the surfaces in the scene with each subvolume. In this clipping operation, the majority of the subvolumes are trivially rejected and only a small set of subvolumes becomes candidates for a more detailed clipping test. Such a test can be performed either by the IP or by different dedicated hardware. The results of the clipping operations for each subvolume are stored in the SSL for the subvolume, and the empty/non-empty conditions for all subvolumes are stored in the ESM. Another alternative to the detailed clipping test is not to perform the detailed clipping test at all. In this case, surfaces are always included in the SSL of their candidate subvolumes. The load for the intersection processor becomes heavier due to the longer SSL lists. To avoid counting any intersection point multiple times during image generation, the IP must compute and output only those intersection points that reside inside the intended subvolume.

As the number of subvolumes increases, the average length of the SSL becomes shorter, and the ray-surface intersection computation becomes faster. Thus, the proposed spatial-information hierarchy can reduce both the object storage requirement and the ray tracing time.

When the initial transferring of the system parameters and object models is complete, the RG begins to generate primary rays and sends these rays to the ESP to find the first non-empty subvolume on the path of the ray. Once the ESP finds the first non-empty subvolume, the ray with the subvolume number is transferred to the FU. The FU identifies all surfaces contained in the subvolume and sends a sequence of "ray-surface ID" pairs to the IP. The IP retrieves the surfaces from the OMS and computes the intersection point between the pair. The IRB then compares the intersection points from the output of IP and keeps only the intersection point closest to the origin of the ray in its register.

If there is no intersection in a subvolume, the ESP is so informed. The ESP finds the next non-empty subvolume on the path of the ray. Once the intersection point closest to the origin of the ray is found, shadow rays are fired from this point toward all the light sources in the scene. These shadow rays are sent to the ESP. If the intersected surface is reflective or transparent, a reflection ray or a transmission ray is fired and sent to the ESP. The color of each intersection point is determined by either the IP or the IRB, depending upon the type of ray. The color of each pixel is determined in the CA by accumulating the color of all nodes in an intersection tree.

3. Analytical Models

3.1 Load Model

The amount of system load for ray tracing an image is modeled stocastically. First, we find the amount of computation needed to trace a ray. Then, we determine the total number of rays needed to be traced for an image. A numerical example is given to illustrate our model. All mathematical symbols used in the model are listed in table 1.

Assume that the 3-dimensional modeled scene is uniformly divided into M^3 subvolumes and that there are N surfaces in the scene.

Let S = average number of subvolumes that a surface occupies. Then, S is a function of the ratio between the average object surface size and the subvolume size. Let's assume that S is proportional to M^2, and call the proportional constant β, the average surface size coefficient. Then, $S=\beta \cdot M^2$. β is related to the ratio between the average surface area and the area covered by one side of the scene.

Let p be the probability of a surface in a subvolume. Since a surface on the average occupies S subvolumes, and there are a total of M^3 subvolumes,

$$p=\frac{S}{M^3}=\frac{\beta \cdot M^2}{M^3}=\frac{\beta}{M}$$

Let f(x) be the probability of having x surfaces in a subvolume. Assuming the surfaces in the scene are similar in sizes, and are placed randomly, the probability of having a set of x surfaces in a subvolume is $p^x(1-p)^{N-x}$. However, this is merely one way of having x surfaces in a subvolume. There are a total of $\binom{N}{x}$ different ways of selecting x surfaces in a subvolume. Therefore, f(x) follows a binomial distribution and can be expressed as:

$$f(x) = \binom{N}{x} p^x(1-p)^{N-x}$$

Let n be the average number of surfaces in a subvolume. According to the binomial distribution,

$$n = \frac{N \cdot S}{M^3} = N \cdot p = \frac{N \cdot \beta}{M}$$

The probability that a subvolume is empty is f(0).

$$f(0) = \binom{N}{0} p^0(1-p)^{N-0} = (1-p)^N$$

This is true because f(0) is the probability that not all N surface are in the subvolume.

Let q be the probability of a ray intersecting any surface.

$$q= 1-f^M(0)$$

In our numerical example we will assume $\beta = 0.001$ and $M = 200$, then $p = 5.0 \text{ x } 10^{-6}$, $n = 0.005$ and $q = 0.632$. Figure 6 shows q as a function of β and N.

If we assume that there are 1,000 surfaces in the scene, then $f(0) = 0.995$.

Let g(x) be the probability that a ray travels x-1 empty subvolumes before reaching a non-empty subvolume.

$$g(1)=1-f(0)$$
$$g(2)=f(0) \cdot [1-f(0)]$$
$$\cdots\cdot$$
$$g(x)=[f(0)]^{x-1} \cdot [1-f(0)]$$

Assume that we can always find a ray-surface intersection once a non-empty subvolume is reached. In other words, the IP never rejects a subvolume which is given by the ESP because it could not find a surface to intersect.

Let D = average number of empty subvolumes traveled by a ray before reaching a non-empty subvolume. Then:

$$D=M \cdot (1-\sum_{x=1}^{M} g(x))+\sum_{x=1}^{M} x \cdot g(x)$$

$$=M \cdot f^M(0)+\sum_{x=1}^{M} x \cdot [f(0)]^{x-1} \cdot [1-f(0)]$$

Using the parameters already assumed in our numerical example, D = 127.

Let t_g = average ray generation time (for any type of ray)
t_e = average computation time to by-pass an empty subvolume
t_f = average time to fetch a Subvolume Surface List
t_o = average time to fetch a surface from OMS
t_b = average time to compare intersection points
t_i = average computation time for each ray-surface intersection
t_c = average time to determine the color value of the intersection point
t_s = average color summation time
Then,
T_R = average computation time per ray in a serial computer.
= average ray generation time (for any type of ray)
+ average time to travel across the empty space
+ average time to find intersection point in a subvolume
+ average time to accumulate color for the pixel
$= t_g + D \cdot t_e + t_f + n \cdot (t_o + t_i + t_{b)} + t_c + t_s$

Let's call u and w the average percentages of reflective and transparent surfaces in all subvolumes respectively. Therefore, $1-u-w$ is the percentage of opaque surfaces. Let l be the number of light sources in the scene and R be the ray tracing resolution of the image. In our example, assuming the screen resolution is 512×512 pixels and each pixel has 4 supersamples for anti-aliasing, then $R = 1024$. Next, we will determine the average number of nodes in an intersection tree and the average number of rays traced for an intersection tree.

The root node of each tree represents a pixel point and, by definition, it has exactly one primary ray to be traced for it. Its probability of intersecting a surface is q. Therefore, on the average, it has q child nodes. Among them, $q \cdot u$ nodes are created by intersecting a reflective surface and $q \cdot w$ nodes are created by intersecting a transparent surface. As a result, a total of $q \cdot (u+w)$ rays is traced for the next level of the tree. In addition, to determine whether the intersection points are directly illuminated by any light source, $q \cdot l$ shadow rays need to be traced. The same reasoning is used to determine the total number of rays traced and the total number of child nodes for the entire intersection tree. They are listed as follows:

Total number of nodes per tree
$$= 1 + \sum_{x=0}^{\infty} q^{x+1}(u+w)^x = 1 + \frac{q}{1-q(u+w)}$$
Total number of vision rays traced per tree
$$= \sum_{x=0}^{\infty} q^x(u+w)^x = \frac{1}{1-q(u+w)}$$
Total number of shadow rays traced per tree
= (total number of node per tree − 1)·l
$$= \frac{q \cdot l}{1-q(u+w)}$$
Total number of rays traced per tree
= total number of vision rays
+ total number of shadow rays in the tree
$$= \frac{1 + q \cdot l}{1-q(u+w)}$$

Let B be the total number of rays for the final image.

B=(the total number of intersection trees)
· (the total number of rays traced per tree)
$$= R^2 \cdot \frac{1 + q \cdot l}{1-q(u+w)}$$

where the ratio between the total number of vision rays and shadow rays is equal to $1 : q \cdot l$. Figure 7 shows B as a function of the number of light sources in the scene (l) and q.

Therefore, the estimated total ray tracing time for generating an image based on space subdivision on a serial processor is Φ, where

$$\Phi = T_R \cdot B$$
$$= [t_g + t_e \cdot (M \cdot f^M(0) + \sum_{x=1}^{M} x \cdot [f(0)]^{x-1} \cdot [1-f(0)])$$
$$+ t_f + \frac{N \cdot \beta}{M} \cdot (t_o + t_i + t_{b)} + t_c + t_s] \cdot R^2 \cdot \frac{1+q \cdot l}{1-q \cdot (u+w)}$$

The time used for clipping objects against each subvolume can be substantial on a serial processor. However, other algorithms to reduce the need for intersection tests between every ray and every surface in the scene have high overhead too [11]. The modeling of the clipping overhead for a serial processor is beyond the scope of this paper.

In our example, we will further assume that $t_i = 50$ micro-seconds and all other time measurements in the T_R equation are 2 micro-seconds. Also assume the number of light sources $l = 4$ and $u + w = 0.15$. Then, the total number of rays $B = 4.57 \text{ x } 10^6$, the average ray tracing time per ray $T_R = 2.62 \text{ x } 10^{-4}$ seconds. In a serial processor, the estimated total ray tracing time for an image is 1198 seconds. The performance of our parallel architecture is estimated by developing and analyzing its queueing model.

3.2 Performance Model

Our goal of modeling is to obtain relative performance measures of the proposed architecture under various loading conditions. Although queueing models for dynamic dataflow architectures have been studied recently [8] [9], they are not suitable for our proposed architecture because of the specialized modules used in the architecture. Figure 5 shows our architecture as a network of queues. Service centers are the RG, ESP, FU, OMS, IP, and IRB. Due to different routings, we need to distinguish two types of jobs in the queueing network. They are jobs associated with vision rays (i.e., primary ray, reflective ray, and transmission ray) and jobs associated with shadow rays. The following describes the sources and sinks in the model.

Sink 1 : When a vision ray travels in space, there is a chance that it does not intersect any surface before exiting the modeled space. If this happens, no further processing is needed for the ray, and the background color should be returned to the Color Accumulator. According to our load model, the probability of this is 1-q.

Sink 2 : When a shadow ray travels in space, there is a chance that it does not intersect any surface before reaching a light source. If this happens, the color of the light source should be used to compute the color of the ray-surface intersection point and return the color of the intersection point to the Color Accumulator. According to our load model, the probability of this is also 1-q.

Source 1 : The intersection computation between a vision ray and the multiple surfaces inside a subvolume is modeled by the feedback route. The feedback probability depends upon the average length of a Subvolume Surface List. For a vision ray, in order to determine the closest intersection point to the origin of the ray, the IP must perform intersection computation between the ray and every surface in the SSL. Due to the nature of the computation, only the vision ray enters the IRB.

Source 2 : Intersection computations between a shadow ray and the multiple surfaces inside a subvolume are modeled by the feedback route. The feedback probability also depends upon the average length of a Subvolume Surface List. However, for a shadow ray, only one ray-surface intersection is required to determine that a light source does not directly illuminate a point. Therefore, on the average, half of the SSL is tested for intersection with a shadow ray before finding a blocking surface.

Because of the large degree of parallelism available in ray tracing image generation, we assume that the system is overloaded most of the time. A throttling mechanism is assumed to be used among the modules to limit the number of jobs in the system below a threshold. This threshold is used as the job size in the closed queueing network. Other assumptions made for the performance model are listed as follows:

Assumption 1 : In the model, the IP and the OMS are inside a feedback loop. They are assumed to be the system's bottleneck. Multiple OMSs and IPs are used to increase the system throughput. A close-coupled IP-OMS arrangement is used to reduce the interconnection overhead between them.

Assumption 2 : A ray will always find an intersection point within a non-empty subvolume. This is not true in general. However, it is true if the number of subvolumes approaches infinity.

Assumption 3 : The service rate for each service center is an exponentially distributed random variable. In addition, the queueing discipline at each service center follows a first come first served (FCFS) policy.

4. Results and Discussions

The main intention of this paper is to show a new parallel architecture for ray tracing and to determine the relative performance of the proposed architecture under different load conditions based on the analytical model. For this purpose, we have chosen some of the system parameters based on the current hardware technology. The service rates of the service centers in the queueing model are as follows. Each IP has an average service rate of 0.1 million ray-surface intersections per second. This is derived from the data provided in [17] assuming all surfaces are convex polygons and specialized parallel hardware is used in the IP. Also, based on the above, we assume that the OMS can fetch 0.25 million surfaces per second. The IRB has an execution rate of 5 million comparisons per second. We also assume the ESP can retrieve empty bits at a rate of 10 million retrievals per second, the average ray generation rate is 1 million rays per second, and the SSL retrieval rate is 1 million SSLs per second. To estimate the performance of the system, the load model of the image generation task and the queueing model of the system are combined and analyzed by the PANACEA [15]. PANACEA is a software package for analyzing multiple job class Markovian queueing network.

Figures 6 and 7 show how image complexity (parameterized by q) relates to the composition of the image. Figure 8 shows the average time to travel across empty space in the scene and the average time to find an intersection point inside a subvolume as a function of the number of subvolumes on each axis (M). In this figure, we have normalized the time scale to the empty-bit retrieval time and assumed that the ray-surface intersection time is equal to 100. Three images with a different number of surfaces are plotted in figure 8, which shows that for a simple scene, a low value of M is a better choice because the Empty Space Processor is the system's bottleneck. However, for a complex scene, a large number of subvolumes substantially improves the performance of the system. This is due to the fact that for a complex scene, the Intersection Processor becomes the system's bottleneck. In this case, the higher M reduces the average length of Subvolume Surface List, consequently reducing the time to find an intersection point inside a subvolume. An optimal M which gives a minimum computation time exists for each image.

Table 2 lists the queue length in each server and the system's processing time for a vision ray and a shadow ray as functions of q. The results indicate that there are two ways that image complexity affects the performance of the system. A higher q increases the number of rays that need to be traced. Also, it increases the percentage of rays entering a non-empty subvolume, consequently increasing the number of intersection computations required. This causes a higher queue length at the IP-OMS, as shown in table 2.

Table 3 shows the queue length of the service centers against the variation of the number of IPs. The shifting of the bottleneck from the IP-OMS to the ESP and the FU in table 3 is similar to the bottleneck shifting from the execution unit to the match unit in a dynamic dataflow computer. To relax the potential of being the bottleneck of the system, additional ESPs or FUs may be added in parallel. In this case, an interconnection structure must be used between ESPs and FUs and between FUs and IP-OMSs.

Figure 9 shows the intersection test processing power (total number of busy IPs) against the number of IPs in the system for two different degrees of image complexity. It shows that the processing power approaches a constant when the number of IPs is large. However, more intersection test processing power is usable if the work load to the IPs increases. Figure 10 shows the time to generate an image frame against the number of IPs. The result shows that there is an optimal number of IPs which require minimum time to generate an image frame. This optimal number of IPs increases as the image complexity increases.

In summary, image complexity affects the system load in several ways and can cause the system's bottleneck to shift. When the image complexity is high, our architecture allows several ways to improve its performance:

1. It uses the spatial-information hierarchy for faster empty space transversal.
2. It allows one to increase the subvolume resolution (M), consequently reducing the total number of intersection computations needed to produce an image.
3. It uses multiple IPs and OMSs in a close-coupled arrangement for a higher intersection-test rate. If the bottleneck shifts to the ESP and FU, one may also add more ESPs and FUs in parallel to relax the bottleneck.

5. CONCLUSION

In this paper, we have proposed a specialized dynamic dataflow architecture for ray tracing image generation. Our architecture reduces the object storage requirement and increases the image generation rate especially when the complexity of the image is high. We have developed a load model for ray tracing computation based on probability. This parallel architecture is modeled with a closed queuing network. The results of the load model provide some parameters for the queueing model. Through these analytical models we have learned the relative performance of the architecture under various load conditions.

6. References

[1] A. Appel, "Some Techniques for Shading Machine Renderings of Solids," Proc. AFIPS JSCC, Vol.32, 1968, pp.37-45.

[2] Arvind and K. P. Gostelow, "The U-Interpreter," IEEE, Computer, Vol. 15, No. 2, Feb 1982. pp.42-50.

[3] S. Coquillart, "An Improvement of the Ray-Tracing Algorithm," Eurographics'85. pp.77-88.

[4] R. L. Cook, T. Porter, L. Carpenter, "Distributed Ray Tracing," Computer Graphics (Proc. SIGGRAPH 84), vol.18 No.3, July 1984. pp. 137-145.

[5] H. Deguchi, et al., "A Parallel Processing Scheme for Three-Dimensional Image Generation," Proc. International Symposium on Circuits and Systems, 1984. pp.1285-1288.

[6] M. Dippe, J. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," Computer Graphics Volume 18, Number 3, July 1984. pp.149-158.

[7] A. Fujimoto, K. Iwata, "Accelerated Ray Tracing," Proceedings of Computer Graphics, Tokyo 1985. pp. 41-66.

[8] D. Ghosal, L. N. Bhuyan, "Analytical Modeling and Architectural Modifications of a Dataflow Computer," Proceeding of International Symposium on Computer Architecture, 1987, pp.81-89.

[9] D. Ghosal, L. N. Bhuyan, "Performance Analysis of the MIT Tagged Token Dataflow Architecture," Proceeding of International Conference on Parallel Processing, 1987, pp.680-683.

[10] A. S. Glassner, "Space Subdivision for Fast Ray Tracing," IEEE CG&A, October 1984. pp. 15-22.

[11] J. Goldsmith, J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," IEEE Computer Graphics and Applications, May 1987. pp. 14.

[12] R. A. Hall, D. P. Greenberg, "A Testbed for Realistic Image Synthesis," IEEE Computer Graphics and Applications, November, 1983. pp. 10-20.

[13] K. Hwang, F. A. Briggs, "Computer Architecture and Parallel Processing," McGraw-Hill, 1984.

[14] H. Nishimura, et al., "LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation," Proc. 10th Symposium on Computer Architecture, 1983. pp.387-394.

[15] K. G. Ramakrishnan, D. Mitra, "An Overview of PANACEA, A Software Package for Analyzing Markovian Queueing Networks," Bell System Technical Journal, Vol.61, No.10, Dec.1982, pp.2849-2872.

[16] V. P. Srini "An Architectural Comparison of Dataflow System," IEEE Computer, Vol. 19, no.3, March 1986. pp.68-87.

[17] M. K. Ullner, "Parallel Machines for Computer Graphics," PhD Dissertation, California Institute of Technology, 1983.

[18] T. Whitted, "An Improved Illumination Model for Shaded Display," Communications of the ACM, Volume 23, Number 6, June 1980. pp.343-349.

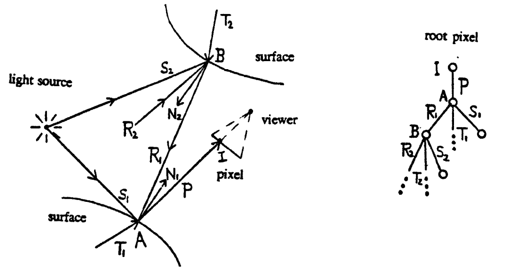


Figure 2. INTERSECTION TREE FOR A PIXEL

In ray tracing scheme, the color of a pixel is determined by an intersection tree. The root node of the tree corresponds to a pixel in the image. All other nodes in the intersection tree corresponds to the intersection points. And, arcs correspond to rays. In this figure, I, P, R, T, S and N indicate pixel, primary ray, reflection rays, transmission rays, shadow rays, and surface normals respectively.

R= the ray tracing resolution of the image.
N= the number of surfaces in the scene.
M= the number of subdivisions in each axis.
l= the number of light sources in the scene.
u= the percentage of reflective surfaces in all subvolumes.
w= the percentage of transparent surfaces in all subvolumes.
S= the average number of subvolumes that a surface occupies.
β= the average surface Size coefficient.
i.e. the proportional constant between S and M^2
p= the probability of a surface in a subvolume.
f(x)= the probability of having x surfaces in a subvolume.
n= the average number of surfaces in a subvolume.
q= the probability of a ray intersecting any surface.
g(x)= the probability of a ray traveling x−1 Empty subvolumes before reaching a non−Empty subvolume.
D= the average number of Empty subvolumes traveled by a ray before reaching a non−Empty subvolume.
t_g= the average ray generation time (for any type of ray).
t_e= the average computation time To bypass an Empty subvolume.
t_f= average time To fetch a Subvolume Surface List.
t_o= average time To fetch a surface OMS.
t_b= average time To compare intersection points.
t_i= the average computation time for each ray−surface intersection.
t_c= the average time To determine the color value of the intersection point.
t_s= the average color summation time.
T_R= the average computation time per ray.
B= the total number of rays traced for generating an image.
Φ= the total ray tracing time for generating an image on a serial processor.

Table 1. MATHEMATIC SYMBOLS USED IN THIS PAPER

q	PROCESS TIME		QUEUE LENGTH			
	V.RAY	S.RAY	EP	FU	IP-OMS	IRB
0.1	40	35	29.5	0.11	1.01	0.06
0.2	54	31	8.0	0.20	3.70	0.07
0.3	81	42	2.3	0.27	4.46	0.055
0.4	109	56	1.2	0.29	4.60	0.046
0.5	136	69	0.86	0.30	4.66	0.040
0.6	164	83	0.66	0.31	4.69	0.033
0.7	191	97	0.53	0.32	4.71	0.030
0.8	219	111	0.45	0.33	4.72	0.027
0.9	247	125	0.39	0.33	4.73	0.024

V.RAY = VISION RAY
S.RAY = SHADOW RAY

NUMBER OF IP-OMS = 8
TOTAL POPULATION = 40

Table 2. QUEUE LENGTH AND RAY PROCESSING TIME (in micro-second) vs THE PROBABILITY OF A RAY INTERSECTING ANY SURFACE (q)

NUMBER OF IP-OMS	QUEUE LENGTH			
	EP	FU	IP-OMS	IRB
1	0.05	0.04	38.9	0.004
2	0.09	0.08	19.4	0.008
4	0.21	0.16	9.6	0.014
8	0.45	0.33	4.7	0.027
12	0.73	0.51	3.1	0.037
16	1.05	0.70	2.3	0.046
24	1.8	1.09	1.5	0.061
32	2.7	1.5	1.05	0.072
48	4.5	2.1	0.65	0.086
64	6.0	2.4	0.46	0.093

q = 0.8
TOTAL POPULATION = 40

Table 3. QUEUE LENGTH vs THE NUMBER OF IP-OMS IN THE SYSTEM

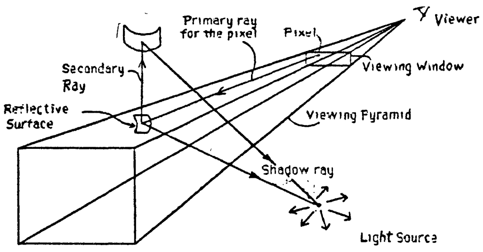


Figure 1. RAY TRACING TERMINOLOGY USED IN THE PAPER

Primary ray : The first ray traced for a pixel. Primary rays are created by projecting pixels in the viewing direction.

Secondary ray : all reflected rays and transmitted rays.

Vision ray : all primary rays , and secondary rays.

Shadow ray : the ray fired from intersection point toward a light source.

A vision ray may be terminated in the following three ways :

1. intersecting a non-reflective and non-transparent surface, (shown in this figure).
2. exiting the world been modeled.
3. terminated at predefined tree depth.

A shadow ray terminates after reaching a surface or a light source.

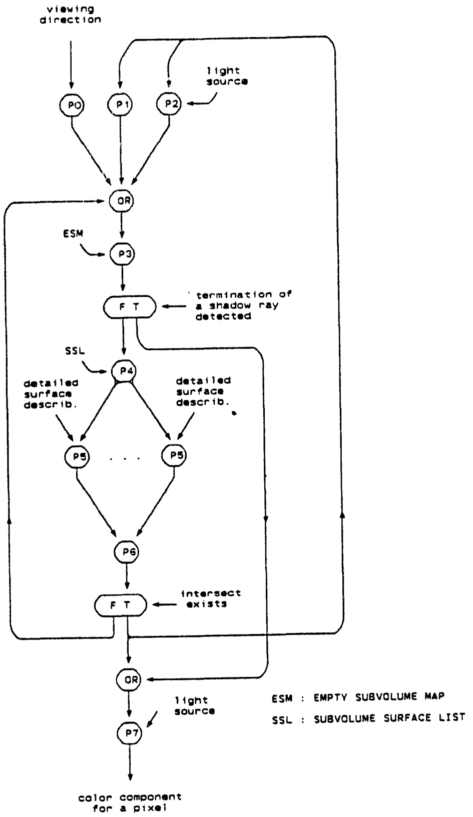
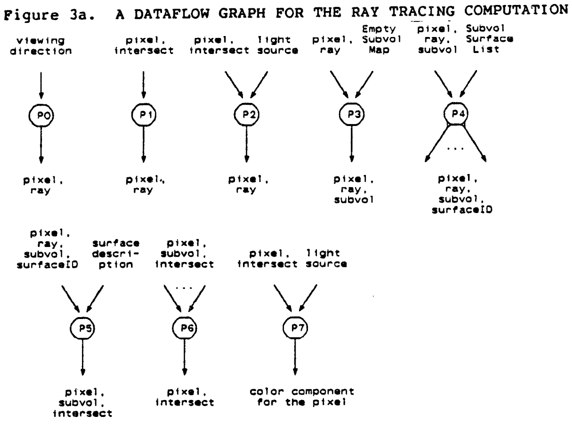

viewing direction
light source
P0
P1
P2
OR
ESM
P3
F T
termination of a shadow ray detected
SSL
P4
detailed surface describ.
detailed surface describ.
P5
. . .
P5
P6
F T
intersect exists
OR
light source
P7
color component for a pixel
ESM : EMPTY SUBVOLUME MAP
SSL : SUBVOLUME SURFACE LIST

Figure 3a. A DATAFLOW GRAPH FOR THE RAY TRACING COMPUTATION



P0 : Generate primary ray
P1 : Generate secondary ray
P2 : Generate shadow ray
P3 : Transverse empty space
P4 : Retrieve surface-ID inside a non-empty subvolume
P5 : Compute the ray-surface intersection
P6 : Determine the intersection point closest to the origin of the ray
P7 : Calculate color component for the pixel based on the type of the ray and intersecting surface

Figure 3b. OPERATORS USED IN THE RAY TRACING COMPUTATION

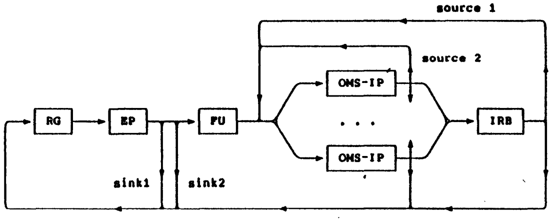


Figure 5. PERFORMANCE MODEL OF THE ARCHITECTURE

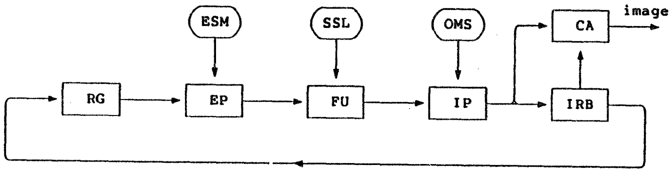


Figure 4. PARALLEL ARCHITECTURE FOR RAY TRACING IMAGE GENERATION

(additional modules of each type may be added in parallel)

6 typies of module :

RG	:	Ray Generator
EP	:	Empty Space Processor
FU	:	Fetch Unit
IP	:	Intersection Processor
IRB	:	Intersection Result Buffer
CA	:	Color Accumulator

spatial-information hierarchy :

ESM	:	Empty Subvolume Map
SSL	:	Subvolume Surface List
OMS	:	Object Model Storage

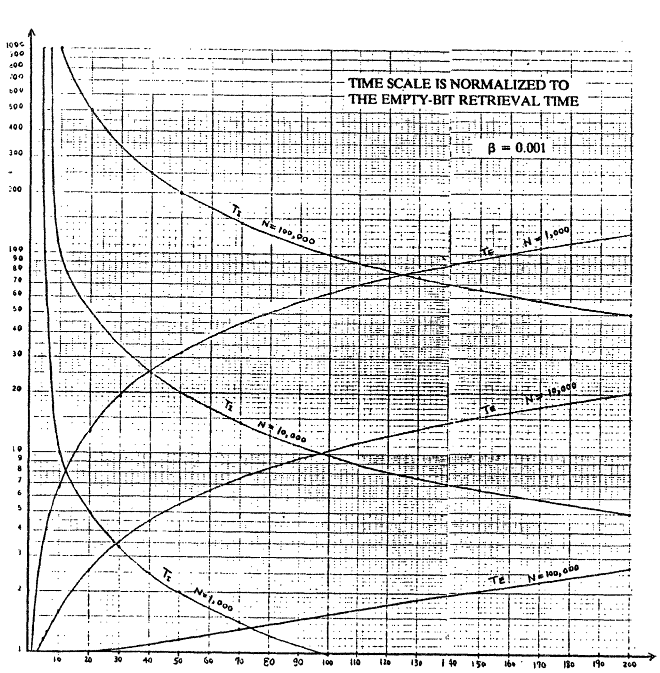


Figure 8. TIME TO TRAVEL ACROSS EMPTY SPACE (T_E) and TIME TO FIND INTERSECTION INSIDE A SUBVOLUME (T_I) vs THE NUMBER OF SUBVOLUME ON EACH AXIS (M)

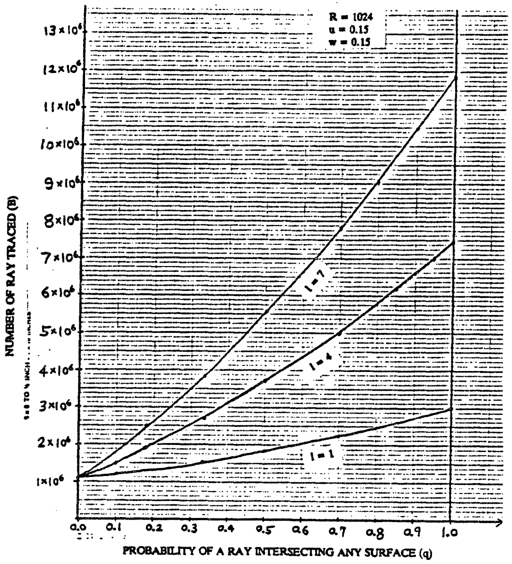


Figure 7. NUMBER OF RAY TRACED FOR AN IMAGE vs PROBABILITY OF A RAY INTERSECTING ANY SURFACE

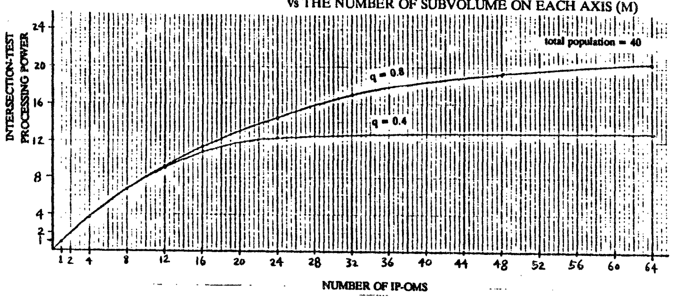


Figure 9. INTERSECTION-TEST PROCESSING POWER vs NUMBER OF IP-OMS

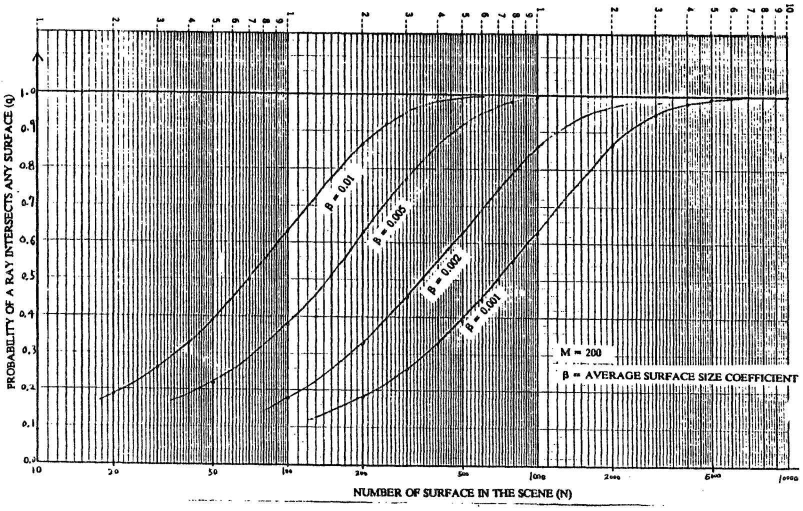


Figure 6. PROBABILITY OF A RAY INTERSECTING ANY SURFACE vs NUMBER OF SURFACE IN THE SCENE

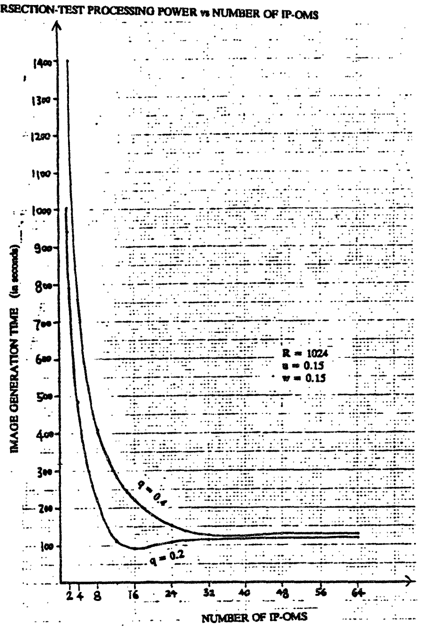


Figure 10. IMAGE GENERATION TIME vs NUMBER OF IP-OMS

A PRACTICAL STATIC DATA FLOW COMPUTER BASED ON ASSOCIATIVE METHODS

Thomas L. Sterling
Ellery Y. Chan

Government Systems Sector
Harris Corporation
P.O. Box 37
Melbourne, Florida 32902

Abstract

The static data flow model of computation offers high performance and scalability by exploiting fine-grained parallelism and flow control constrained only by data precedence. Unfortunately, the token driven mechanism upon which most proposed static data flow architectures are based is inefficient for communication and synchronization, being profligate in its use of memory bandwidth and micro-operations. Associative templates have been proposed as a temporally efficient alternative to tokens. This approach applies associative processing methods to data flow communication and synchronization. This paper presents a practical associative template based architecture that provides effective, fine-grained static data flow computation.

1. Introduction

Efficient techniques for managing fine-grained parallelism are prerequisite to very high performance computation. In the static data flow model [1, 2] of computation, program execution is typically coordinated by tokens [3], directed packets providing communication and synchronization among operation templates. Token-based data flow architectures [4, 5] have memory bandwidth requirements (the number of memory accesses per operation) and serial temporal overhead (the number of primitive micro-operations that must be performed sequentially per instruction execution) greatly in excess of equivalent characteristics for conventional uniprocessors performing the same tasks. The present dearth of data flow machines in the high performance computing arena is due, in part, to the intrinsic inefficiency of the token driven approach. Viable static data flow awaits an alternate execution mechanism.

The *associative template* [6] mechanism evolved from the need to significantly reduce overhead and make better use of communication bandwidth between integrated circuits. It employs associative processing methods to perform program flow control in a data flow context. The associative template mechanism fully satisfies the semantic requirements of the static data flow model, and has previously been described [7] in the context of a single-node system.

Associative diffusion, a related associative processing technique that extends the associative template approach to multi-node systems, is a new method for supporting communication between adjacent nodes in a mesh interconnected topology. For this specific class of system structures, associative diffusion provides nearest neighbor communication without tokens, and without incurring an additional time penalty. It does so by overlapping the domains of associativity across adjacent node boundaries. There is an overhead cost in time for communication between nonadjacent nodes, so this method is best suited for those algorithms that can be statically mapped onto the node array to require mostly nearest neighbor[†] transactions.

Together, the associative template and associative diffusion concepts establish an alternate approach to static data flow architecture. By eliminating tokens, static data flow architectures of much greater efficiency can be realized. However, the unsophisticated implementation of these associative methods can result in specifications that exceed the capability of today's technology in terms of packaging, power dissipation, and electrical characteristics.

This paper presents a new static data flow architecture based on associative templates and associative diffusion. The Associative Template Dataflow (ATD) computer architecture separates the synchronization control and data communication functions of the associative template and associative diffusion mechanisms, providing the low-overhead communication between neighbors at little additional cost, and resulting in a system organization whose components can be realized with current technology.

For certain classes of storage allocation and program structure the architecture permits maximum throughput of critical components. This is achieved by decoupling the synchronization of modules into an ensemble of asynchronously interacting client and server components, maximizing throughput utilization across the interfaces to critical (expensive, performance constraining) system elements, and minimizing the number of such transactions that must be performed per operation execution (template firing).

In the following sections, the associative template and associative diffusion concepts are reviewed in their generalized form, and a simplistic single node architecture is described to demonstrate that the semantic criteria of the static data flow computing model are met. The ATD architecture embodies associative templates and associative diffusion in a practical implementation. The architecture is functionally decomposed, the modules are described in detail, and some of the variations and trade-offs are considered. The concluding discussion focuses on unresolved problems of hardware implementation and execution of real world applications.

2. Background

2.1 The Static Data Flow Model

The static data flow model of computation is a set of semantic policies that must be supported by the underlying execution medium. Foremost among these are:

1) A data flow operator will execute when its operands have been computed by its argument source operators

[†] In the class of interconnection schemes discussed here, nearest neighbor transactions are always of distance one.

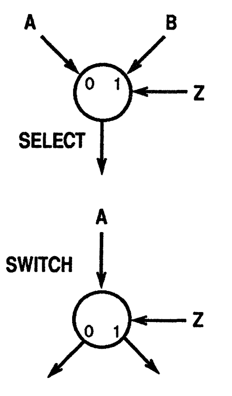


Figure 1. The Select and Switch operators. The Select operator passes either its A or B operand based on the value of the third operand, Z. This template can fire prior to the arrival of the operand not selected. The Switch operator passes its A operand to one of two groups of recipients, based on the value of Z.

(data driven synchronization).

2) Only a single instantiation of each operator at a time is permitted. An operator will execute only when the operators that use its result values (the recipients) have completed their most recent execution and are prepared to accept new operand values.

Each operator in a data flow program is represented by a template, a small data structure that explicitly or implicitly specifies the operation to be performed, the source templates that supply the argument values, the recipient templates that use the result values as operands, and the transient internal state of the operator as it progresses through its execution cycle. Conditional flow control is supported by special templates. The select template chooses as its result value one of two argument values depending on the Boolean value of its third argument. The switch template makes its primary input datum available to one of two sets of prescribed recipient templates depending on the value of its other Boolean operand (see Figure 1).

In most proposed static data flow architectures the assumed implementation mechanism is the token, a small message packet for synchronization and communication among templates. In such architectures each operator execution involves a number of token handling micro-operations and a number of memory accesses to the template store, resulting in temporal overhead and memory bandwidth requirements almost an order of magnitude greater than that of executing the same operation on a conventional RISC microprocessor [8].

2.2 Associative Processing for Flow Control

The conventional application of associative processing [9] has been the searching of sets of data to detect records that contain a field matching a specified key value. Associative mechanisms have been applied to database operations for sorting and searching [10], to cache memories [11, 12] for fast instruction fetching and data access, and to translation lookaside buffers [13] for rapid virtual memory mapping. The associative template mechanism applies associative processing [9 - 13] to program execution flow control. Instead of managing program data as in the case of the previously cited applications associative templates use associative methods to directly modify the control state of an executing data flow computer.

Associative techniques are used to manage data flow control because the control state of the system is distributed among the templates: the generation of a new result value can affect elements of the control state in different parts of the system. In a token driven system, each portion of the control state change invoked by an executing template is performed as a distinct token handling operation. By employing broadcast communication techniques, the knowledge of a template firing event can be distributed to all necessary parts of the control state at the same time. Templates recognize relevant broadcast events and adjust their own part of the program control state. This includes both reporting availability of new operands to recipient templates (templates for which result values are destined), and updating the source templates (templates from which argument values are derived) with the acknowledge status of their recipients. All of these actions can be done simultaneously, but require memories configured with internal logic for the purpose.

2.3 Associative Diffusion for Communication

The initial associative template architecture was developed for a system consisting of a single processing element, or node. The architecture exploits parallelism to sustain peak performance of a pipelined functional arithmetic unit. The concept of associative diffusion was devised to extend associative mechanisms to the important but still restrictive case of communication between adjacent nodes of a multi-node associative template system. Unfortunately, the cost in transistors and package pin count of a direct application of the associative diffusion concept to the implementation of a static data flow computer incorporating associative templates would be prohibitive.

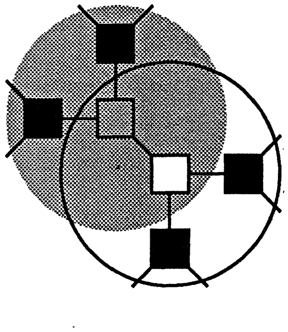


Figure 2. Domains of associativity. The gray node monitors the template activity of the two black nodes and the white node in the gray circle. The white node references values produced by nodes in the outline circle. The domains for the black nodes are not shown here.

Associative diffusion extends the domain of associativity of a node across node boundaries to encompass the operation space of its nearest neighbors. Every template in a node monitors the operation of the other templates contained within its local node and in its adjacent nodes (see Figure 2), and watches the result values produced by templates of its own and its neighboring nodes. In this way, a template can reference result values of other nodes and determine when they become available. Templates can reference arguments across node boundaries and respond to requests for result values from other nodes. A template monitoring argument references from executing templates of its neighboring nodes as well as those of its own node can ascertain the acknowledge status (for data flow synchronization) of recipient templates in its local node and in the neighboring nodes.

Associative diffusion supports nearest neighbor communication that is as fast as communication between templates in the same node. Since that efficiency does not extend beyond adjacent nodes, it best suits applications whose locality properties can be easily mapped onto mesh-like system organizations. Although this is somewhat restrictive, many important classes of scientific computation exhibit such behavior. Even for applications where some longer distance transactions are required, multiple hop transfers can be supported within the scope of this mechanism. Doing so, however, will impact latency time and degrade system performance to a degree proportional to the average communication distance.

2.4 Advantages and Disadvantages

The associative template mechanism circumvents much of the burdensome overhead in time and space imposed by the token mechanism. Some of the specific advantages include reduced memory bandwidth, higher throughput, free acknowledgments, smaller memory, and queue elimination.

With these gains comes the inconvenience of requiring custom integrated circuits for all of the principal elements. The template storage module, which for a token driven system is conventional RAM, must be implemented as a very smart memory. The templates require at least six comparators each, along with other custom logic, even in a single node system.

This paper presents an architecture that, while employing the associative template and associative diffusion concepts to eliminate the need for token mechanisms, assumes a structure that is within the capability of current engineering practices. A direct application of the previously described mechanisms to a system architecture presents serious difficulties in implementation. The number of transistors and pins required for system node-to-node interfacing would appear to be prohibitive. Additional problems of bus loading and power consumption strain the system's feasibility. The machine described below captures the strengths of the associative diffusion mechanism without the expected prohibitive costs. The result is a practical architecture for tokenless static data flow computation.

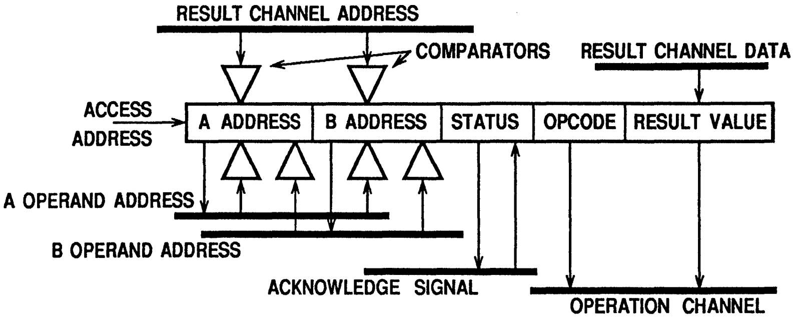


Figure 3. An associative template. The triangles are comparators that monitor the various busses for addresses that match the template's operand address fields. A result address match will set the A or B operand arrival flag. An operand bus address match will cause the template to generate an acknowledge status signal that will be sampled by the operand's source template. When the template fires, it places its opcode onto the operation channel and its operand addresses onto the operand address busses. Later, it receives its new value on the result channel. When it is a source, it places its result value onto the operation channel.

3. A Single Node Architecture

A simple, single-node associative template data flow machine has one high throughput, pipelined *functional computation unit* (FCU), functional in the sense that no internal state is kept between operations. The FCU accepts operation packets containing an opcode, the necessary operand values, and a result address, and produces packets containing the value resulting from the computation performed and the result address. The FCU is driven by the *graph coordinator*, which stores and manages execution of the static data flow program. The graph coordinator embodies the associative template techniques. It fires a new template every cycle, delivering an operation packet to the FCU for processing. At the same time, it assimilates a result from the FCU every cycle and updates the control state of all recipient templates.

The graph coordinator is a collection of associative templates and the logic that controls their execution (dispatching logic). A template for a dyadic operator has five entries: a status field, two operand source addresses, an opcode, and a result value buffer. It receives results computed by the FCU on the *result channel*, and passes more work to the FCU on the *operation channel* (see Figures 3 and 4). When a template fires, its address and the contents of its opcode field are written to the operation channel, and its operand address fields cause the source templates to write their result value fields to the operation channel as shown in Figure 4.

A section of logic having access to every template's status flags, called the *dispatching logic*, determines when a template can fire. A template that is eligible to fire, called *pending*, has its A and B flags set, indicating that its operands have been computed and are available for access; and has its acknowledge flag set, indicating that its current result value is no longer needed by its recipients. The dispatching logic will select one of the pending templates for execution.

The A or B flag of a template is set when the result channel address matches the template's A or B address field, respectively, which occurs when the FCU produces a needed operand value and returns it to the graph coordinator. The acknowledge flag, which performs the static data flow acknowledge synchronization function, stores the negated current value of the wired-OR acknowledgment signal whenever the template's result value is accessed by a recipient. The acknowledgment signal is asserted by other recipients of the template's result value that have not yet fired, signifying that the result value is still needed for future computation. These recipients are identified by matching the contents of either of their source address fields with the contents of either of the operation channel's operand address busses.

The graph coordinator accepts a result value and produces an operation packet during each cycle of its interface. During the cycle, all templates that are recipients of the computed result update their control state simultaneously to reflect its availability. At the same time that the operation packet is being assembled, the acknowledge status for both referenced operand source templates is derived. This wealth of very low level parallelism in the operation of the associative template mechanisms is responsible for its high throughput and interface bandwidth efficiency.

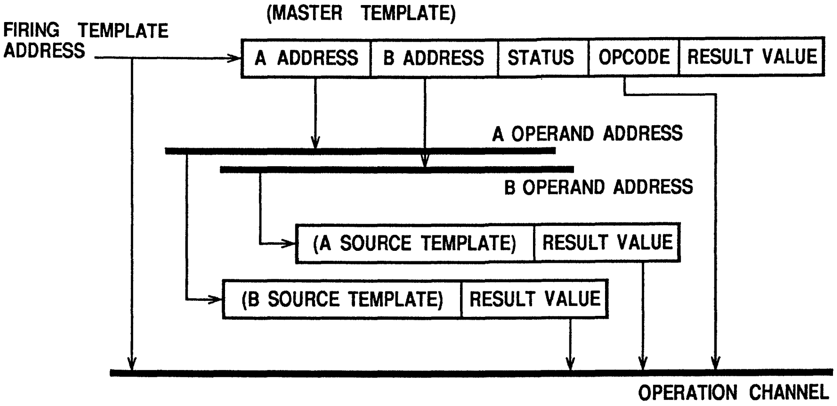


Figure 4. A template firing. When a template's operands have become available and its current result value is no longer needed, it can fire when selected by the dispatching logic. The firing sequence causes the opcode and the master template's address to be placed directly onto the operation channel. The A and B operand addresses cause the source templates to also place their values onto the operation channel. The completed operation packet is submitted to the FCU.

4. The ATD Architecture

The ATD architecture is a practical static data flow architecture embodying the associative template and associative diffusion concepts. This section describes the ATD architecture in detail, building on the background material and presentation of the simple single node system.

4.1 System Level Architecture

The ATD architecture extends the single node architecture to support multiple, interconnected nodes while retaining the efficiencies derived from its associative properties. Very tight coupling of the graph coordinator and the FCU is essential for high performance. The associative domain is augmented to include the nearest neighbors without an overwhelming increase in cost or complexity.

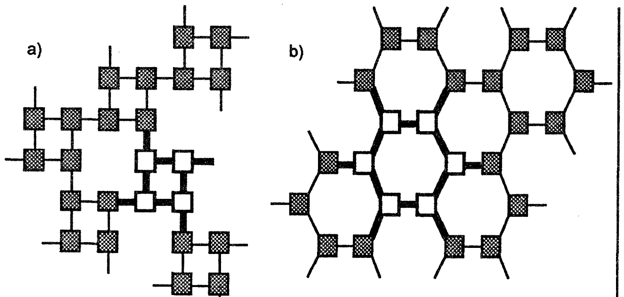


Figure 5. Mesh topologies. Nodes of degree 3 can be used to create mesh structures. In a) degree 3 nodes are combined to form a degree 4 mesh building block. In b) the nodes form a degree 6 building block.

Some simplifying assumptions are imposed on this architecture to reduce complexity of its nodes while assuring system scalability. The system level architecture is chosen to be a large mesh [14] with efficient adjacent node communication. Longer distance transactions may experience proportionally longer latencies and performance degradation. Exploiting locality to eliminate or severely limit long distance transactions opens the way to vast systems comprising millions of nodes without a reduction in the average useful throughput per node, assuming sufficient problem size and node reliability. Many problems of interest, such as systolic algorithms [15], finite element methods [16, 17], and signal processing applications [18] have communication patterns that can be statically mapped onto such a structure.

The mesh system organization can be realized with nodes of only degree 3 (nodes with three external interface ports). The relatively small number of ports for each node is important in constraining the module's complexity while achieving the essence of the associative diffusion mechanism. Two examples of mesh topologies that can be supported with degree three nodes are shown in Figure 5. Cube-connected cycles [19] can also be implemented with degree three nodes.

The system level architecture employs a non-global address space. This is acceptable under the assumption that almost all references are either intra-nodal or to adjacent nodes. For those cases where multiple hop communication is necessary, forwarding templates can be employed at the proportional cost of template storage space and node throughput.

4.2 Node Architecture

The architecture of the ATD node comprises a small number of specially devised elements that implement the associative template and associative diffusion mechanisms. These elements operate in a manner similar to that of the simplified architecture described earlier, but also support communication with three nearest neighbors. The elements are 1) the inter-node interface, 2) the data store, 3) the functional computation unit, 4) the operation packet builder, 5) the graph coordinator, and 6) the opcode store. The structure of a node is shown in Figure 6.

Every node has a functional computation unit that is only responsible for processing operation packets generated within the node. A dominant specification of the architecture is to sustain peak throughput of this unit. Its design is matched to the performance of other units that determine the rate of operation packet generation.

The fields comprising a data flow template are distributed among three of the node elements. The parts of the template that specify the flow graph topology and program control state, namely the operand addresses and the status flags, are stored in the graph coordinator. The operation code for each template is located in the node's opcode store. The result values produced by the templates are held in the node's multiport data memory, the data store.

The opcode store is a simple small memory with the number of entries equal to the number of templates managed by the node and wide enough to hold the number of bits necessary to distinguish among the set of operations performed by the functional computation unit.

The data store stores the results computed by the FCU and provides the result values to the operation packet builder of the local and adjacent nodes upon request. It contains two two-port memories, each with separate read and write ports. The duplicate memories increase the throughput of the data store. The result values of the functional computation unit are stored in both memories simultaneously via their write ports. Each data memory services four sources of read requests, one from its own node and one from each of the connected neighboring nodes. These requests come from the graph coordinators of each of the nodes. The data values are returned to the operation packet builder of the node originating the access.

The operation packet builder constructs operation packets for delivery to the FCU. It acquires the source template address directly from the graph coordinator. The opcode of the operation to be performed is provided by the opcode store. The argument values come from the data memories of either the host node or the neighboring node holding the result value.

The graph coordinator stores the data flow graph topology, maintains the program control state, and determines the order in which the

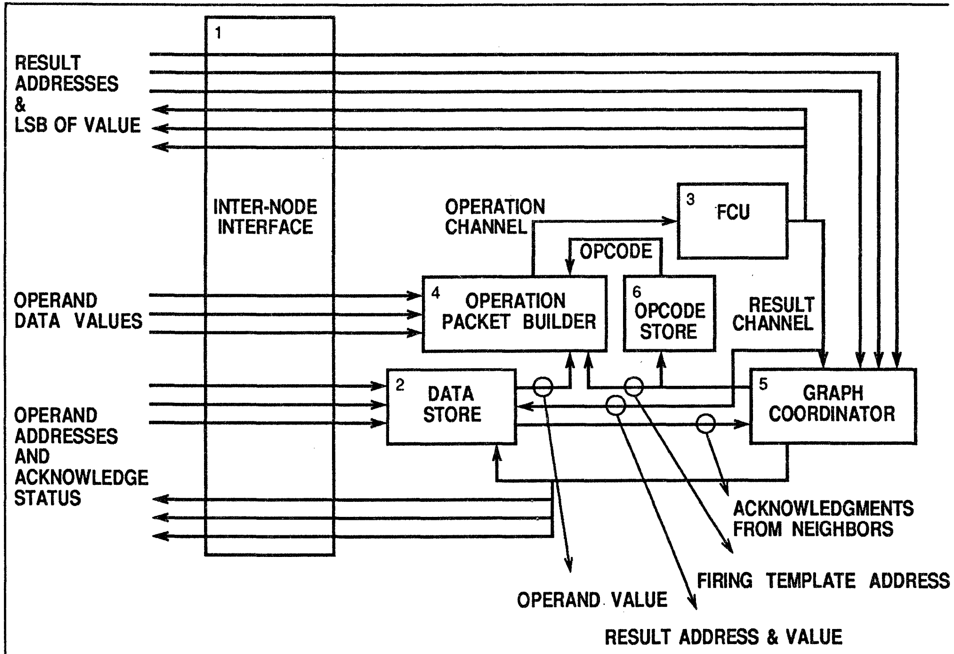


Figure 6. A node of the ATD architecture. Additional communication channels exchange template addresses and data values with neighboring nodes.

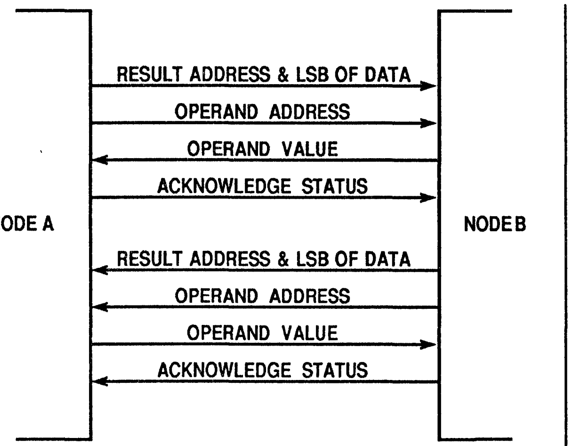


Figure 7. A node and its inter-node connection to one neighbor. Each neighbor connects in the same manner, with the full complement of signals being exchanged.

program templates fire. It receives addresses of fired templates from the functional computation units of its own and its three neighboring nodes. It also gets acknowledge synchronization signals from the graph coordinators of all four nodes. The means by which these two classes of event information are used to update the program control state approximates the domain of associative diffusion for the host node and its three adjacent neighbors. The graph coordinator generates a new three-tuple each cycle specifying the address of the executing template and the addresses of the source templates for its arguments. For each of the two argument template addresses, the graph coordinator provides a single bit acknowledge signal indicating whether or not it contains other templates that still require that operand to be available. A key architectural objective is to maximize graph coordinator utilization.

4.3 The Inter-Node Interface

Adjacent nodes are connected by means of a symmetrical interface as shown in Figure 7. There are four groups of interface signals between two neighbor nodes, A and B. The first group supports data driven synchronization. The result address of operations performed by the functional computation unit of one node are sent to the result bus of the other node's graph coordinator, indicating the availability of the result values for the identified templates.

The next two groups support data access service requests from the nodes' data memories. The service requests originate with the node's graph coordinator. The data values are returned to the requesting node's operation packet builder.

The last group of interface signals provides acknowledge synchronization information between adjacent nodes. Again, two complete paths are provided in each direction. The acknowledge information consists of the acknowledge condition state and the template receiving the acknowledge signal. This information comes from the operation channel of the sending node's graph coordinator and is destined for one of two of the acknowledge ports of the receiving node's graph coordinator. This information is used to set the state of the acknowledge flags for the selected template.

4.4 The Data Store

The *data store* (see Figure 8) resolves operand addresses into operand values. It accepts addresses from each domain member (node), and can return values to any member. The particular method used to accomplish the diffusion of result values across the domain is embodied in the choice of implementation of the data store. One configuration is described here.

The configuration shown in Figure 8 has two banks. Results of computation are stored in the data memory, duplicated in each bank. This structure, allowing two operands to be resolved simultaneously, increases the availability of the operand data.

An address register latches the incoming operand address until it can be resolved by the data memory. Arbitration logic selects one of the waiting addresses, which is then used to retrieve the operand value from the data memory. The selected address is also supplied, accompanied by the acknowledge status to the graph coordinator.

4.5 The Operation Packet Builder

Because the delay between the initiation of the execution sequence of a template (when the graph coordinator selects an eligible template from those that are pending) and the arrival of its operand values can vary, and since there are multiple (in this case, four) sources of operand values, greater throughput can be achieved by allowing the template firing logic to start building operation packets for several templates at once. The short latency incurred by the operation packet builder does not degrade system performance as long as there are enough templates continuously ready to keep the FCU pipeline full. This creates the need for a small buffer area where the opcode and result address are temporarily stored while the operand addresses are being resolved. When the operand data values are obtained, the com-

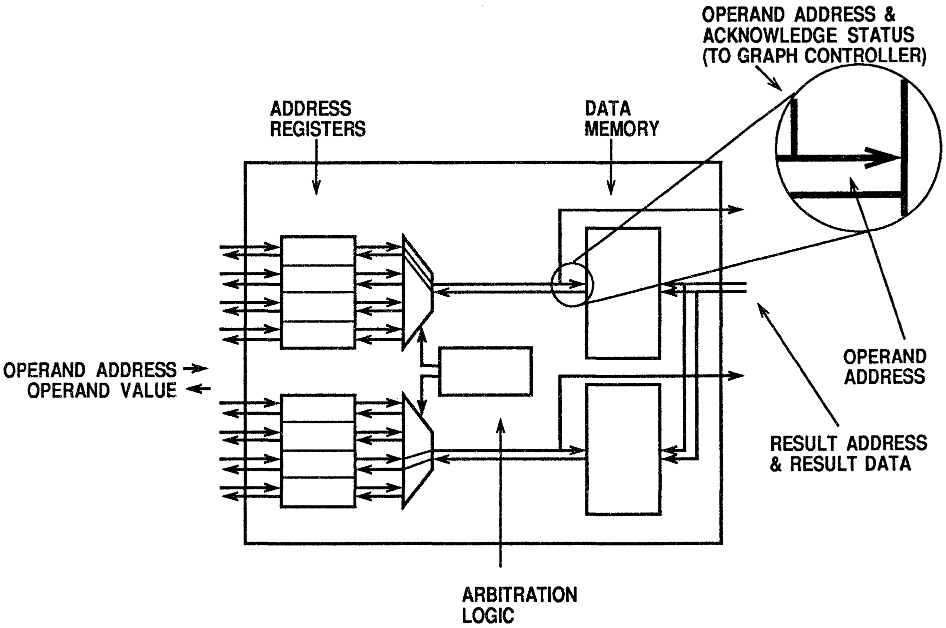


Figure 8. The data store. The data store has two data memories containing duplicate sets of result values, and two access ports, one to each memory, for each node in the domain. As operands are fetched, the accompanying acknowledge status is forwarded to the graph coordinator. New result values are stored as they arrive from the FCU.

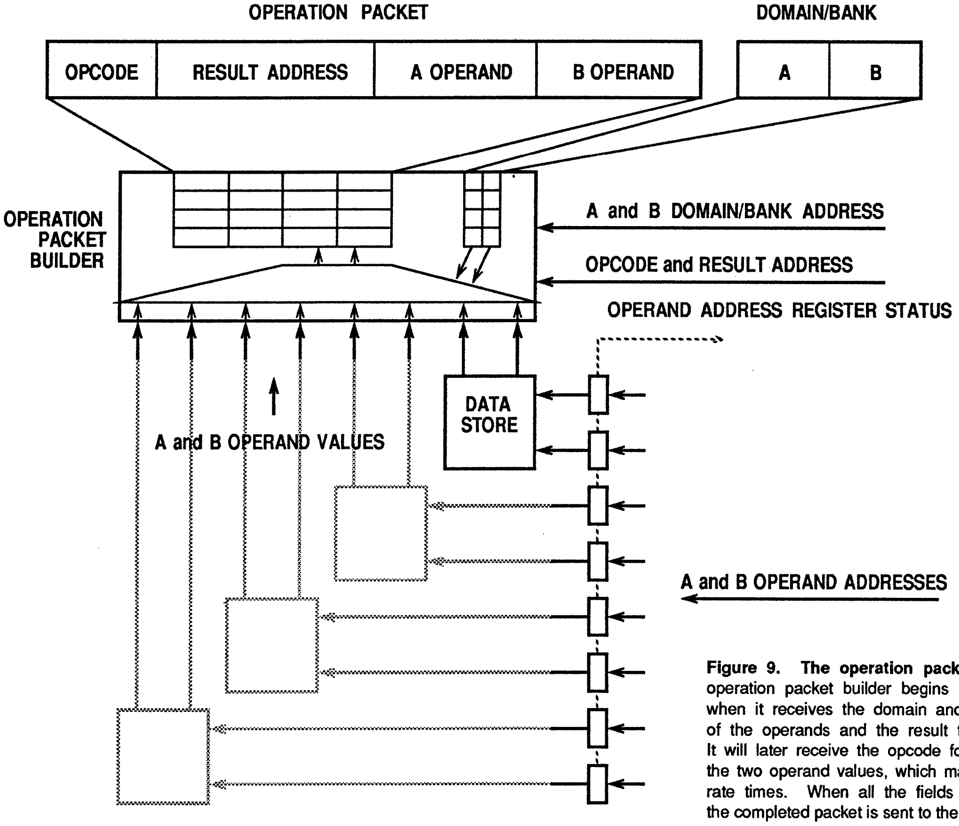


Figure 9. The operation packet builder. The operation packet builder begins building a packet when it receives the domain and bank addresses of the operands and the result template address. It will later receive the opcode for the packet and the two operand values, which may arrive at separate times. When all the fields have been filled, the completed packet is sent to the FCU.

pleted packet is submitted to the FCU. The *operation packet builder* (see Figure 9) accomplishes those tasks.

The operation packet builder accepts an opcode, a result address, and, for each operand, a domain/bank address (three bits specifying the neighbor and which of the two ports). These it places in one of its buffers. The domain/bank address is used to route the arriving operand value to the proper operation packet buffer field. Once an operand value has arrived and has been stored, the operand address register used for the access is free again to be used by another firing template. The status of the operand address registers is used by the graph coordinator, which chooses a pending template that can use currently available registers.

4.6 The Functional Computation Unit

Operations on the operand data values, which include arithmetic and Boolean manipulations, are performed by the *functional computation unit* (*FCU*). The FCU accepts a stream of *operation packets* from the operation packet builder, each containing an opcode, two operand values, and a result address (see Figure 10). Since the FCU is purely functional, the result of any operation will be the same independent of the ordering of the arrival of operation packets. The actual internal FCU architecture is not discussed here, but can be assumed to employ pipelining to increase its throughput, and one or more VLSI floating point or special purpose functional units.

4.7 The Graph Coordinator

The ATD graph coordinator extends the simple single node architecture presented in Section 2 to provide associative diffusion for associative template operation between adjacent nodes. It provides data driven synchronization from neighbor nodes, acknowledgment reporting by neighbors, and template firing based on available resources. These extensions are presented in this section. The new graph coordinator architecture is designed to yield chip, pin, and device counts considered practical by standards of contemporary technology.

4.7.1 Result Value Handling

As previously indicated, result values are no longer stored in the graph coordinator. Instead, a separate dedicated multiport data store is provided. This transfer of functionality reduces the requirements imposed on the graph coordinator, drastically lowering its interface pin count, permitting more uniform chip layout, and concentrating available on-chip devices on the task of flow control. While the result data of the FCUs are not applied to the graph coordinator, the result addresses that identify the source templates of the operations are still supplied. This is necessary for the unit to synchronize on completed operations and update its control state. Furthermore, for the distributed architecture, (as opposed to the single node system) result addresses from adjacent nodes as well as those from the host node's FCU must be monitored associatively.

In the single node system, each template argument source address field employs a field-wide comparator to monitor the result address bus. Associative diffusion requires that the result addresses of all four nodes in an associative domain be equally accessible, and thus four

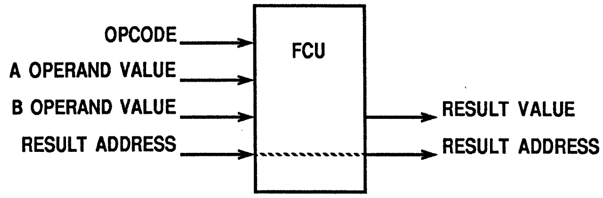


Figure 10. The functional computation unit. The functional computation unit receives a stream of operation packets and produces a stream of result values accompanied by their destination addresses.

result address busses, one from each node's functional computation unit, are used with the ATD graph coordinator. Address widths are expected to be in the range of twelve bits, which would require forty-eight input pins. A few additional pins are needed for conditional templates and timing.

The direct realization of associative diffusion would imply that the single source address field comparator should be expanded to four comparators, one to monitor each of the four result address busses. Fortunately, the computing model is constrained, so this is not necessary. Because of the static allocation of data flow templates, the argument template referenced by a source address field can come from only one of the four domain nodes. Therefore each source address field need monitor only one of the four result address busses. The bus to be monitored is specified by the two most significant bits of the template address. Instead of adding three more comparators, the graph coordinator is augmented with a 4 to 1 multiplexer with input selection controlled by the two most significant address bits (the *domain* bits), as shown in Figure 11. This approach is far less expensive than the four comparator approach in terms of both transistor count and power consumption.

4.7.2 Synchronization by Acknowledgment

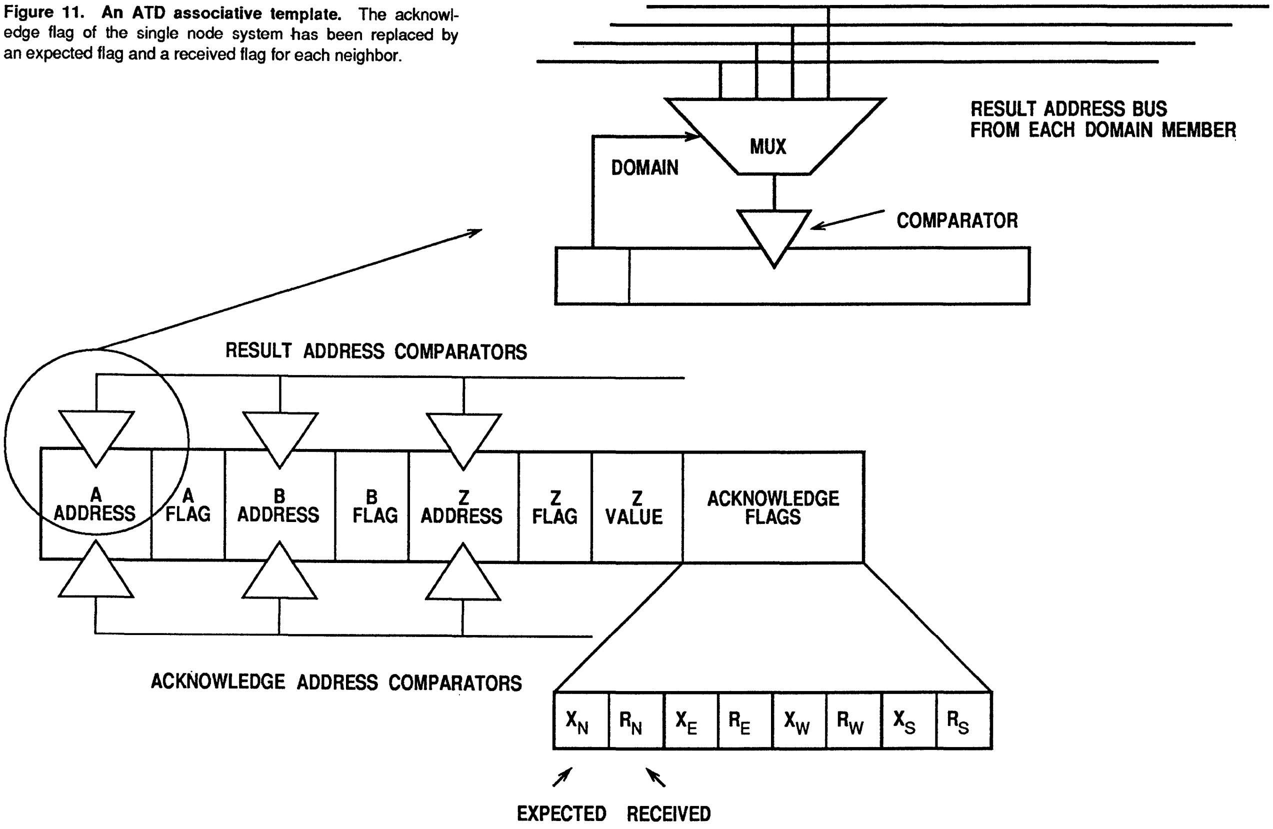
The single node associative template method of synchronizing with recipient (children) templates, referred to here as acknowledge synchronization, can be thought of as consisting of two parts: generating the acknowledge condition state, and recording the current state in the appropriate template's acknowledge flag. When all templates were in the same node, both parts could be performed simultaneously. In the ATD machine, recipient templates may be located in any one of a template's domain nodes. The direct method of implementing associative diffusion would be to tie all nodes of a domain into one large node. To maintain parallelism, there would have to be four times as many operation channel address busses and as many additional comparators per source address field. The costs of such a structure are clearly prohibitive.

The ATD architecture approximates this structure by separately handling the acknowledge synchronization of a node for each of the nodes in its domain. When a node creates an acknowledge condition signal, it needs only to go to one of its four domain nodes, that being the same node to which the argument data request is directed. Instead of broadcasting the condition signal across the entire domain, it is sent only to the node in which the argument template resides. All templates in one node that use the results of a template in another node of their domain participate associatively as in the single node architecture to determine whether all of them are done with that operand. An acknowledge condition signal received from another node indicates whether the entire node is finished with the operand. Thus the first part, that of creating an acknowledge condition signal for a template, is done on a per node basis.

The second part, that of recording the acknowledge condition state, is facilitated by replacing the original acknowledge flag with four flags and four mask bits (see Figure 12). Each flag reflects the acknowledge condition state of one of the four nodes in the domain for the template's result value. If a particular neighboring node contains no resident templates that use the result of a host's template, then the mask bit corresponding to that neighbor node is set. A template's acknowledge status is satisfied when either the flag or its mask bit is set for all of the domain nodes.

To set the flags, the graph coordinator is augmented with two acknowledge select busses. Each of these busses can independently choose a template and load the state of one of the acknowledge flags. One of these busses is associated with each of the node's two data store memories, and the current acknowledge status is stored in the appropriate flag at the same time that the data value for that template

Figure 11. An ATD associative template. The acknowledge flag of the single node system has been replaced by an expected flag and a received flag for each neighbor.



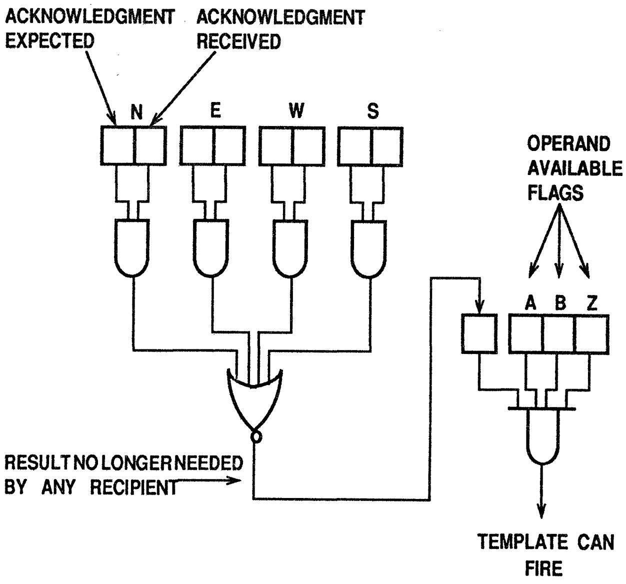


Figure 12. Firing logic. The dispatching logic uses the acknowledge flags and the operand available flags to determine whether a template is eligible for firing.

is being fetched from the data memory. The additional acknowledge busses increase the pin count by twenty-eight. Since these are common select busses, no additional comparators are required to extend the utility of the graph coordinator from single node operation to associative diffusion emulation.

4.7.3 Dispatching

The *dispatching logic* in the single node architecture served the simple task of choosing almost arbitrarily among the pending templates ready for execution. The ATD architecture relies heavily on the dispatching logic for a second critical function, that of resource management. The potential for bottlenecks exists in the ATD architecture because of the possibility of contention for access to the data memories shared among the four nodes of a domain. It is the job of the dispatching logic to prevent such contention from degrading the throughput of the graph coordinator and indirectly the throughput of the FCU.

A node has two interface ports to the data memory of each of its domain's nodes. Each port can only support one data access request from a memory at a time. The dispatching logic receives empty/full signals from all of the ports. The set of full ports restricts the categories of templates (based on the source nodes of the arguments) from which the next one to fire may be chosen. The dispatching logic continues to select templates as long as ports are available to carry out the argument access requests, and as long as templates that make use of available data store ports are pending.

4.8 Template Execution

To summarize the operation of the ATD architecture the execution cycle of an associative template is examined. Assume this template gets its A operand from a local source template and its B operand from a source template in a neighboring node (Figure 13). Also assume that its result values are used by two local recipient templates and one recipient template in a neighboring node.

The cycle begins with the template ready to fire. The dispatching logic of the graph coordinator selects the pending template for execution when a port to the local data memory and a port to the node containing its B operand are available, then sends the address of the firing template to the opcode store and to the operation packet builder. The firing template asserts the contents of its two source argument address fields on the A and B operand address busses within the graph coordinator. The A operand address is applied to the access port of one of the two data memories in the local node. The B operand address is applied to the access port of one of the two data memories in the neighbor node containing the B source template.

An acknowledge condition state signal accompanies each of the two operand access service requests. Other templates in the local node monitor the operation channel and, if either of their argument source template fields match either of the source template addresses on the operation channel, they assert an active signal on the appropriate wired-OR acknowledge signal line indicating that they still require the operand to be available. Otherwise, the templates output an inactive signal on the acknowledge lines. These signals tell the source templates whether or not there are other templates in that node for which the result value must remain available. The firing template's parameters are distributed to the designated data stores, opcode store, and packet builder, and the acknowledge status for each of the arguments is produced. The argument and acknowledge flags of the firing template are then reset.

The operation packet builder chooses a free operation packet buffer and records from which data memory output ports the two argument values are to come. It immediately stores the firing template's identifying address, which is also applied to the opcode store. The operation's opcode is provided during the second cycle and is loaded into the operation packet buffer.

The data store for the neighbor node supplying the B operand arbitrates in a round-robin fashion among the four access ports (from the adjacent nodes) it services. When it comes to the port for the firing template, the data memory reads the contents of its addressed value and returns it to the dedicated output buffer of the local node. This output buffer directly feeds the local operation packet builder. At the same time, the acknowledge condition state accompanying the argument value request is passed to the graph coordinator's acknowledge port along with the source template address. The address selects the source template, and the acknowledge condition state is loaded into the acknowledge flag associated with the local node.

The operation packet builder continuously acquires the contents of data memory output buffers to which values have been written and stores them in the appropriate fields of the designated operation packet buffers. When all components of the operation packet have been assembled in the buffer, the buffer's ready flag is set. Shortly thereafter, the functional computation unit detects the ready condition of the operation packet in the buffer and assimilates its contents.

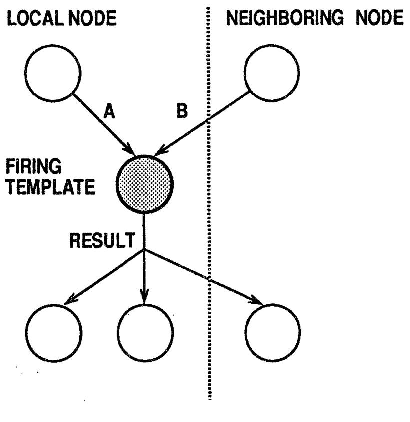


Figure 13. Template Execution. The firing template interacts with other templates in its own node and in a neighboring node.

The functional computation unit processes the operation packet. After some number of cycles, due to the latency of the unit, (which is unspecified by the architecture,) the result value of the operation and the address of the template responsible for its creation are produced. The result value is immediately stored in both halves of the node's data memory via their respective write ports. The constraints of the data flow model guarantee that no location of the data memory will be both written and read at the same time, so conflict cannot occur. The result address is distributed to the graph coordinator result busses of each of the nodes within the domain.

The template source address fields monitor the result busses of the nodes from which their operand values are derived. The A source address field monitors the local result bus with its comparators connected to the bus through its multiplexor set by its address' two most significant bits. The B source address field similarly monitors the result bus of the neighboring node containing the template referenced by the field. When these source templates fire (A locally and B in the neighboring node) the template determines the availability of their result values by detecting a match between the fields' contents and those of the respective result busses. Upon this occasion, the appropriate argument flags, A or B, are set.

As the template's recipient templates fire, they access the node's data memory for the template's result value and return acknowledge condition state signals to the acknowledge ports of the graph coordinator. When the recipient template in the neighboring node fires, there are no other templates using the local template's result value as operands, so the acknowledge condition state returned to the local graph coordinator causes the corresponding acknowledge condition flag to be set. When the first of the two local recipients fires, the acknowledge flag will remain clear because the second local recipient still requires the template's result value to be available. Upon firing of this second local recipient, however, the local acknowledge flag is set because no other templates in the local node require the result value to perform their own operations.

Both the A and B flags are set indicating that both operands are available. The local acknowledge flag and that associated with one of the neighbor nodes (the one containing the recipient template) are set while the masks of the other two acknowledge flags are set because those adjacent nodes do not contain any recipients of the template. Under these conditions, the dispatch logic determines that the template is again ready to fire, thus completing the execution cycle.

5. Conclusions

A new architecture for static data flow computation has been presented that employs associative mechanisms for program flow control and communication in lieu of more conventional token driven techniques. Tokens impose too much overhead for effective fine-grained parallel processing and are wasteful of memory bandwidth. It has been shown that by using associative techniques, associative templates support the semantics of static data flow more efficiently than do tokens. That concept alone, however, has been inadequate to formulate a complete distributed static data flow architecture. It has not provided the means by which interaction among multiple nodes is conducted. A second concept, that of associative diffusion, had also been put forward to fill this gap. It proposed that the domains of associativity of nearest neighbor processing elements, or nodes, be overlapped so that the activities of one node could be directly monitored by its immediate neighbors. This provides the vehicle for extending the associative template mechanisms across node boundaries. While feasible methods of implementing a single node system with associative templates exist, the direct method of extending that architecture with associative diffusion requires a prohibitive amount of logic. The new architecture presented in this paper provides the first practical means without tokens of realizing an associative template static data flow computer with an approximation of associative diffusion for synchronization and communication.

The ATD architecture exhibits substantial promise for high performance parallel computing in general, and static data flow computation in particular. But a number of questions still remain to be investigated before it can be proved worthy of implementation. One is the logic intensity of the control unit. The architecture requires an interface to this element that is entirely realizable. However, the circuitry required per template is substantial. Preliminary designs have established that approximately a thousand transistors are required per template in the control unit. While the layout structure is particularly orderly promising good utilization of chip real estate and simple design, this is still a lot of logic. Current technology can thus produce control units capable of containing about 256 such templates. The ATD architecture supports connecting these units in groups of four to permit a thousand templates per node. Before judgement can be made regarding its acceptability, this cost must be weighed against alternate methods of applying that scale of logic to parallel computing.

A second challenge that must be satisfactorily met is the means by which programs are distributed among the nodes in the assumed mesh system level structure. While a number of classes of problems are known to be easily mapped onto such a structure to maximize nearest node communication, the degree to which intra-node program parallelism is needed to fill the latency cycles of the memory access paths must be studied and automated allocation techniques must be developed.

Finally, a range of data memory/packet builder structures present themselves. How performance varies for these different structures with respect to real world application programs has yet to be understood and needs to be explored. At this point, the success of the ATD architecture is that it demonstrates a complete and viable alternate approach to the token mechanism for static data flow architecture and opens a new area of performance/cost trade-offs in the design of data flow computers. It is hoped that this work, as preliminary as it is, will inspire other researchers in this field to reexamine the data flow computing model in light of these new structures and to consider the potential of their advancement.

Acknowledgments

The authors wish to recognize the valuable contributions to the ATD architecture made by D. S. Wills of the MIT Artificial Intelligence Laboratory. They also wish to convey their appreciation to Professor R. H. Halstead of the MIT Laboratory for Computer Science for his critical evaluation of the associative template concept. Lastly, many thanks to Dr. D. B. Bradley, D. A. Thomae, A. J. Musciano, and G. H. Thaker of Harris Government Systems Sector for their in-depth review of this paper.

References

[1] Karp, R.M., and Miller, R.E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal of Applied Mathematics*, Vol. 14, No. 6, Nov. 1966, pp. 1390–1411.

[2] Dennis, J.B., "Programming Generality, Parallelism and Computer Architecture," *Information Processing 68*, North-Holland, Amsterdam, 1969, pp. 484–492.

[3] Dennis, J.B. and Misunas, D.P., "A Preliminary Architecture for a Basic Data-Flow Processor," *Proceedings of the Second*

Annual Symposium on Computer Architecture, Dec. 1974, pp. 126–132.

[4] Gurd, J.R., Kirkham, C.C., Watson, I., "The Manchester Prototype Dataflow Computer," *Communications of the ACM*, Vol. 28, No. 1, Jan. 1985, pp. 34–52.

[5] Hiraki, K., Shimada, T., Nishida, K., "A Hardware Design of the Sigma-1, a Data Flow Computer for Scientific Computations," *Proceedings of the 1984 International Conference on Parallel Processing*, Aug. 1984, pp. 524–531.

[6] Sterling, T.L., "Intuitive Templates: A Static Data-Flow Architecture without Tokens," *White Paper*, Harris Government Systems Sector, May 25, 1987.

[7] Sterling, T.L., Wills, D.S., Chan, E.Y., "Tokenless Static Data Flow using Associative Templates," submitted to Supercomputing '88, 1988.

[8] Patterson, D., "Reduced Instruction Set Computers," *Communications of the ACM*, Vol. 28, No. 1, Jan. 1985, pp. 8–21.

[9] Foster, C.C., *Content Addressable Parallel Processors*, Van Nostrand Reinhold Co., New York, 1976.

[10] Yau, S.S., Fung, H.S., "Associative Processor Architecture – A Survey," *ACM Computing Surveys*, Vol. 9, No. 1, Mar. 1977, pp. 3–28

[11] Smith, A.J., "Cache Memories," *ACM Computing Surveys*, Vol. 12, No. 3, Sept. 1982, pp. 473–530.

[12] Archibald, J., Baer, J.-L., "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, Vol. 4, No. 4, Nov. 1986, pp. 273–298.

[13] Moussouris, J., *et al*, "A CMOS RISC Processor with Integrated System Functions," *Proceedings 1986 COMPCON*, IEEE, Mar. 1986, pp. 126–131.

[14] Hwang, K., Briggs, F., *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

[15] Kung, H.T., "Why Systolic Architectures?" *Computer*, Vol. 15, No. 1, Jan. 1982, pp. 37–46.

[16] Huebner, K.H., *Finite Element Method for Engineers*, John Wiley & Sons, New York, 1975.

[17] Strang, G., Fix, G., *An Analysis of the Finite Element Method*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[18] Oppenheim, A.V., *Applications of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, N.J., 1978.

[19] Preparata, F.P., Vuillemin, J., "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Communications of the ACM*, Vol. 24, No. 5, May 1981, pp. 300–309.

MAPPING THE DATA FLOW MODEL OF COMPUTATION INTO AN ENHANCED VON NEUMANN PROCESSOR*

Peter M. Maurer

Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

Abstract -- The SAM architecture is an enhanced von Neumann processor that contains inexpensive features for supporting data flow style of parallelism. The architecture gets is name from the basic instructions for supporting parallelism, Split and Merge. It is shown that these instructions can be used to implement the parallel structure of an arbitrary acyclic data flow graph. Features for supporting dynamic parallelism and multiple run-time environments are presented. Implementation issues for supporting instruction execution and the handling of faults and interrupts ar also discussed.

1. Introduction.

One of the main focuses of current research in computer architecture is the design of hardware organizations that support the parallel execution of instructions (see [1] for several examples.). Data flow parallel architectures continue to receive a great deal of attention [3] [4]. In a data flow architecture an instruction may execute as soon as its operands become available, permiting a degree of parallelism bounded only by the flow of data between instructions. In spite of their intuitive appeal, data flow machines have been slow to reach the marketplace, and it appears that much work must be done to make data flow machines competitive with other parallel architectures[5].

In spite of the objections raised by [5], data flow is appealing, and it is reasonable to ask whether it can be adapted to a more conventional architecture. The approach taken in this paper is to start with a von Neumann processor, and by adding features, enable it to execute programs in the highly parallel manner characteristic of data flow machines. The objective is to develop inexpensive parallel architectures that can exploit parallelism without sacrificing compatiblity with existing software. Compatiblity with existing software is important because it represents an enormous investment for the computer user, and it is necessary to preserve this investment. The architecture presented in this paper is called the SAM architecture for reasons that will be explained in section 2. The features of this architecture are similar to those found in multithreading machines [2][6][7], but are somewhat simpler. In spite of this, the features presented here can be used to program some of the more complicated features found in other machines.

Section 2 describes the architecture and the primitive features for supporting parallelism. Section 3 shows how the architecture supports arbitrarily complex static parallelism. Section 4 introduces features that support dynamic parallelism and multiple run-time environments. Section 5 discusses implementation issues, and section 6 draws conclusions.

2. The Basic Architectural Features.

The SAM architecture supports arithmetic and logical instructions as well as conditional and unconditional jumps. Initially it is assumed that conditional jumps perform both a comparison and a conditional jump, and that all instructions are memory to memory. It will be possible to relax. these restrictions later. Two addressing modes are supported, the full-address mode which provides direct addressing using full-width addresses, and the short-address mode which requires fewer address bits than the full-address mode and is used to access the low-address portion of memory. Short addresses may be either direct or indirect. The portion of memory addressable in the short-address mode is called the short-address space, and will be described more fully in section 4. In some implementations, portions of the short-address space may be mapped to registers or a high-speed cache. There are no programmer-addressable registers.

The SAM architecture is a MIMD machine that allows the degree of parallelism to vary with time. Two types of parallelism are supported, static parallelism where the degree of parallelism is determined at compile time, and dynamic parallelism where the degree of parallelism depends in part on the data being processed. There is no upper limit on the degree of parallelism. The features for supporting parallelism are motivated by the differences between the execution histories of sequential machines and those of data flow machines. On a sequential machine each instruction has exactly one predecessor and exactly on successor, while on a data-flow machine, each instruction has several predecessors and successors. In order to support parallelism whose degree varies with time, it is necessary to have instructions that have more than one predecessor and successor. In the SAM architecture the "split" instruction has one predecessor and two successors, while the "merge" instruction has two predecessors and one successor. These instructions form the core around which the rest of the architecture is designed, hence the name "SAM" for "Split And Merge." The split instruction has the format of an unconditional jump, one successor is the branch target, while the other is the following instruction. The split instruction creates two independent instruction streams. The merge instruction has one operand that is normally initialized to zero. When its operand is zero, the merge instruction sets it to 1 and terminates the execution of the current instruction stream. Otherwise the it sets its operand to zero and continues execution with the next instruction. The merge instruction operates atomically on its operand. Figure 1 shows how the combination of split and merge can be used to evaluate the statement "e=(a+b)+(c+d)." with the sub-expressions evaluated in parallel.

```
1         split    L1
2         add      a,b,t1
3         merge    x
4         jump     L2
5    L1:  add      c,d,t2
6         merge    x
7    L2:  add      t1,t2,3
```

Figure 1. Parallel Evaluation of e=(a+b+(c+d).

Most of the instructions Figure 1 are self explanatory. The labels "a," "b," "c," "d," and "e" are the variables named in the expression, while the labels "t1" and "t2" are temporary variables. The label "x" is a temporary variable that is

[*] This research was supported by the university of South Florida Center for Microelectronics Design and Test.

initialized to zero. The split instruction on line 1 causes the add instructions on lines 2 and 5 to be executed in parallel. The first two operands of these instructions are added and the result is placed in the third operand. In this example, a separate merge instruction is placed at the end of each instruction stream. An equivalent way to program this example would be to omit the merge instruction on line 3 and move the label "L2" from line 7 to line 6.

Figure 1 shows that the split instruction adds three or four instructions of overhead to each stream (the two merge instructions cannot execute in parallel). If the end of both streams is moved to line 6, the overhead can be reduced to three instructions per stream. Assuming that all instructions execute in one time unit, each stream must be at least four instructions long for there to be any benefit from the parallelism. introduced by a split. However, if the split and merge instructions execute quickly as compared to the other instructions, the length of the stream could be reduced without negating the beneficial effects of parallelizing the code.

At this point it is assumed that all instruction streams execute in the same environment, which restricts the way code can be parallelized. Methods for removing these restrictions will be discussed in section 4.

3. Translating Data Flow Code to Split/Merge Streams.

The translations presented in this section are based on the intermediate form of data flow code presented in [8]. A program is represented as a combinatorial expression of the form (C op0 ... opn), where C is an Abdali combinator[9], and op0 through opn are the operands of the combinator. An operand may be a constant or another expression. The combinator may be of the form B^n_m, I^n_m, or K_m. If it is of the form B^n_m then op0 will be the name of an instruction and op1 through opn will supply the operands of the instruction. If the combinator is of the form I^n_m, op0 through opn will not be present, and if it is of the form K_m, op0 will be a constant and op1 through opn will not be present. For any expression, all subscripts will have the same value . A subscript of m signifies that m inputs are need to evaluate the expression. Combinators of the form B^n_m are used to evaluate n-input functions, those of the form K_m are used to introduce constants into an expression, while those of the form I^n_m are used to select the nth input from a list of m inputs. For example, the expression x+y+1 can be translated into the expression $(B^2_2 + (B^2_2 + (I^1_2) (I^2_2) (K_2\ 1))$. As was pointed out in [8] these expressions are simply linearized forms of data flow graphs.

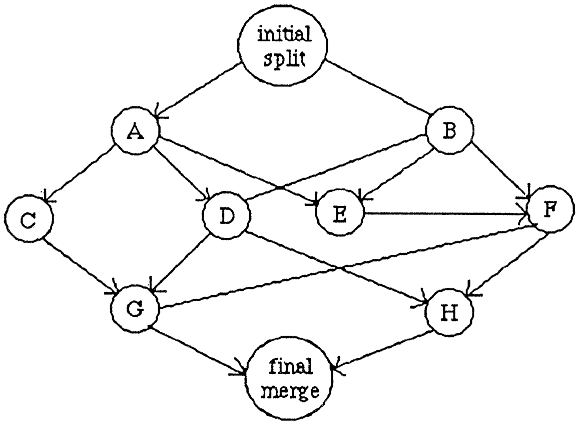
Because the language presented in [8] does not contain conditionals loops or assignments, no provision was made for handling them. In addition because the language is applicative, no provision was made for handling sets of independent statements that communicate by side effects. To make the results of this section as general as possible, it is necessary to introduce the functions "if," "while," "assign," and "set" to handle conditionals, loops, assignments, and sets of independent statements. In addition, the new combinator Q^n_m is introduced to distinguish between sets of statements that are independent, and those that have data dependencies. The combinator Q^n_m is mathematically equivalent to B^n_m, but when code is generated for the expression (B^n_m x0 x1 ... xn) each of the expressions will be evaluated in parallel. When code is generated for the expression (Q^n_m x0 x1 ... xn), the expressions x1 through xn will be evaluated serially. The Q^m_n combinator can also be used near the "leaves" of an expression if it is necessary to place a lower bound on the length of independently executed instruction streams. To illustrate the use of the Q^n_m combinator, consider the usual form of the combinatorial expression for e=(a+b)+(c+d) which is $(B^2_5 \mathrm{assign}(I^1_5)(B^2_5+(B^2_5+(I^2_5)(I^3_5))(B^2_5+(I^4_5)(I^5_5))))$. When code is generated for this expression, the code-generation algorithm will create two independent instruction streams to evaluate the sub-expressions (a+b) and (c+d) in parallel. The two streams will be merged to complete the final addition. The following slight modification in the combinatorial expression will cause the three additions to be executed serially, $(B^2_5 \mathrm{assign}(I^1_5)(Q^2_5+(B^2_5+(I^2_5)(I^3_5))(B^2_5+(I^4_5)(I^5_5))))$.

Code can be generated for combinator expressions in a straightforward manner. A separate instruction stream is created to evaluate each operand of a
B-type combinator, while the operands of a Q-type combinator are evaluated serially. The operands of "if" and "while" functions are always executed serially, although parallelism within the evaluation of the operands is not precluded. When code is generated for the body of a loop, it begins and ends as a single instruction stream, which prevents the iterations of a loop from getting out of sync.

If expressions of the form (Q^n_m set x1 ... xn) are translated using the most straightforward algorithm, parallelism will be lost. For example, consider the following two-statement sequence.

a=b+c
e=(d+f)*a

The subexpressions "b+c" and "d+f" can be executed in parallel, but if the statements are serialized due to the data dependency, this parallelism will be lost. A more sophisticated method of translating these functions is needed. The procedure is easier to visualize if it is assumed that the set of statements has been described as a data flow graph. Each node in the graph represents one statement in the set. (Complex expressions have been broken into separate statements.) All arcs that do not begin and end on a node are omitted, along with all duplicate arcs. The result is a directed acyclic graph with one or more source nodes and one or more sink nodes. Assume that there are j source nodes and k sink nodes. Since the source nodes use only those data items that are assumed to be present before the execution of the set begins, they can all be executed in parallel. The code for the set begins with a j-label msplit instruction, which is the single predecessor of each source node. Similarly, the code for the set ends with a k-label mmerge instruction, which is the single successor of every sink node. Msplit and mmerge are standardized sequences of instructions that create and merge an arbitrary number of streams. An n-label msplit acts as an n-way branch, while an n-label mmerge acts as an n-label branch-target. Their construction is straightforward. The code for a node with m predecessors and n successors begins with an m-label mmerge instruction and ends with an n-label msplit instruction. In each case the labels on the msplit instruction match the labels on the mmerge instruction of the successor nodes. The nodes of the data flow graph can represent arbitrarily complex expressions and are not restricted to individual instructions. A node may have a high degree of internal parallelism as long as it has a single entry and a single exit. This method of translating set functions allows arbitrary acyclic data flow graphs to be implemented using the split and merge instructions. An example of this procedure is illustrated in Figure 2.



```
msplit     LA1,LB1
mmerge     LA1
-----A-------
msplit     LC1,LD1,LE1
mmerge     LB1
-----B-------
msplit     LD2,LE2,LF2
mmerge     LC1
-----C-------
msplit     LG1
...
```

Figure 2. Data Flow Parallelism with Split and Merge

4. Creating Multiple Environments.

Although a high degree of parallelism can be realized with the split instruction in a single environment, multiple environments are needed to support dynamic parallelism and shared subroutines. One method of supporting multiple environments would be to have several data and address registers that are replicated for each instruction stream. Such a mechanism is used in some multiprocessors, but since not all independent instruction streams require separate environments, it is desirable to separate the function of creating an instruction stream from the function of creating a new environment. Recall that the SAM architecture provides a short-addressing mode that is used to access the short-address space. Associated with each instruction stream is a register called the prefix register that contains the location of the short-address space. The prefix register is assumed to contain the high-order address bits of the short-address space, with the low order bits being supplied by the instruction. The instructions "readp" and "writep" are provided to read and write the prefix register. Instruction streams that require separate environments may use these instructions to create a new short-address spaces. The current value of the prefix register is replicated on a split which causes the two independent streams to execute in the same environment. Since the two streams will generally begin execution at two different points in memory, distinct environments can be created for each stream. The merge instruction does not affect the contents of the prefix register. Although the prefix register can be used to solve the problem of dynamic parallelism and the problem of calling the same subroutine in two different instruction streams, the stack-based addressing scheme for passing arguments and saving return addresses cannot be used in a multi-threading environment without elaborate support mechanisms or rigid controls on how it is used. In a multi-thread environment it is not possible to predict when memory for one set of arguments will be deallocated with respect to the memory for another sets. In the SAM architecture, allocation of space is made the responsibility of either the support software or the compiler, because the most efficient method for doing so depends on the problem being solved,. Efficient implementation of the basic features of the architecture should allow many different allocation schemes to be programmed efficiently.

To illustrate how the prefix register can be used to achieve dynamic parallelism, consider the code illustrated in Figures 3 and 4. It is assumed that each instruction stream requires an environment of size 2^x, and that 2^i instruction streams are to be created. In addition to a number of temporary variables, each environment contains its starting address, its size, the number of the current instruction stream, the total number of instruction streams, and a pointer to the parent environment. There is also a word initialized to zero, which will be used as a merge target. Figure 3 illustrates the serial creation of instruction streams, while Figure 4 illustrates the logarithmic creation of instruction streams. The logarithmic initiation of instruction streams operates by creating a single environment of size 2^{x+i} and repeatedly splitting it in half until environments of the proper size have been created. In the process the proper number of instruction streams will be initiated.

```
allocate $2^{x+i}$ bytes;
for j=1 to $2^i$
    init env j and make it current;
    split to shared_code;
endfor
for j=1 to $2^i$
    exec merge in env j;
endfor

shared_code:
    ...
for j=current_stream to $2^i$
    execute merge in environment j;
endfor;
```

Figure 3. Serial Stream Initiation.

Creating a separate environment for each stream causes the overhead for each stream to be greatly increased. When multiple environments are being used, it may be more

```
allocate $2^{x+i}$ bytes;
init as one size $2^{x+i}$ env & make current;
while (env_size > $2^x$)
    env_size = env_size / 2;
    init new env in second half of current env &
        make current;
    split to x;
    restore parent env
x:
endwhile;
... shared code ...
while (total_streams > 1)
    if (curr_strm_num is even)
        make env at env_adr+env_size current;
    endif;
    exec merge; restore parent env;
    divide curr_strm_num and total_streams by 2;
endwhile;
restore parent env;
```

Figure 4. Logarithmic Stream Initiation.

convenient to treat the independent streams as individual processes that communicate through a producer/consumer structure as proposed by several others [2]. The simplest way to model the producer/consumer relationship is to follow the producer by the instruction "split x" and precede the consumer by the instruction "x: merge k" where k is a variable that has been initialized to zero. Overruns can be prevented by preceding the consumer by the instruction "y: merge j" where j is a variable that is initialized to one, and following the consumer with the instruction "split y." This scheme will work only if each data item has a single producer and a single consumer. To support multiple producers and consumers, it is necessary to introduce the hardware equivalent of semaphore P and V operations. The P operation is modeled by the "seq" instruction,which has a single operand. If the operand is non-zero when the "seq" instruction will set it to zero and instruction execution continues with the next instruction. If the operand is zero, both it and the program counter for the current stream remain unchanged. The seq instruction operates atomically on its operand,and executes repeatedly until its operand is set to zero. The detrimental effect of the busy wait can be minimized by spinning the seq instruction off into a separate instruction stream. The seq instruction can be used for multiple-producer multiple-consumer problems and other types of synchronization.

5. Implementation Issues

The SAM architecture will be implemented as a shared pipeline similar to that found in the HEP multiprocessor[2]. The number and function of the pipeline stages is not fixed by the architecture, but for definiteness consider the pipeline pictured in Figure 5. This is a typical pipeline augmented with two additional stages to fetch and write stream descriptors. Each stream descriptor contains the current PC for the stream as well as the current prefix register value. The descriptor may contain other items as explained below.

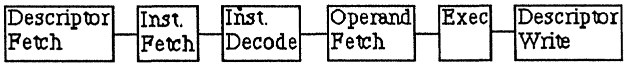


Figure 5. An Augmented Pipeline.

The descriptor fetch stage of the pipeline obtains descriptors from many different sources. In particular, each stage of the pipeline can serve as a source of descriptors, which allows the pipeline to be fully utilized even when only a small number of descriptors exist. When descriptors are fetched from the earlier stages of the pipeline, potential pipeline hazards must be provided for either in the hardware, or by some software scheduling technique.

The descriptor write stage of the pipeline provides for internal buffering for descriptors. When the internal buffer of the descriptor write stage is full and a split instruction creates a new descriptor, several actions can be taken. The descriptor write stage can provide storage management for a circular buffer in some form of backing store, or it can cause a fault to occur when the number of streams reaches a high-water mark. The support software has the choice of suspending the execution of the stream until the number of streams fell below a low-water mark, or of passing the descriptor to a second processor.

To support the simultaneous execution of several processes, each of which may have several instruction streams the SAM architecture provides features for handling interrupts and faults. An interrupt is handled by initiating a new instruction stream in response to an external event. The "return from interrupt" is accomplished by executing a merge instruction with a zero argument. An interrupt vector consists of a pointer to the executable code for handling the interrupt, and a pointer to the short-address space for the interrupt handler.

Since several program faults of the same type may occur simultaneously, it is necessary to have some method of serializing the first portion of the fault handler, which allows the descriptor of the offending instruction stream to be copied into the environment of the fault handler without destroying data that is still needed to process a previous fault of the same type. This problem is solved by adding a global register for masking faults, and a status register to the descriptor of each stream. When a fault occurs and the corresponding fault-type is masked, the descriptor of the offending stream will have a "suspended" bit set in its status register. A descriptor with the suspended bit set propagates through the pipeline without change. When the descriptor reaches the stage where the fault originally occurred, the stage will schedule the fault-handler. if the fault type is now unmasked. A more expensive solution would be to allow the descriptor write stage of the pipeline to queue descriptors waiting for the fault to become unmasked.

At times the support software may need to terminate a process in response to a program fault or other event. Because each process can have many instruction streams active simultaneously, some mechanism is needed to identify and terminate all instruction streams belonging to the terminated process. To solve this problem a process-id, which can be used to identify and terminate instruction streams, is added to the descriptor of each stream. The process-id is copied into the new descriptor when a split instruction is executed but it may be changed by the support software. One way to accomplish this is to combine the assignment of process-ids with memory management. For example, the process-id could be a pointer to the segment or page table for the process. The memory management hardware could be used to force the termination of instruction streams. Another method is to pass the process-id of a failed process to the descriptor write stage of the pipeline and allow this stage to purge all descriptors with matching process-ids.

The addition of status bits to the stream descriptor permits the implementation of privileged instructions, allows more conventional compare and conditional jump instructions to be used, and allows for local masking of faults in the instruction streams. There are a number of implementation issues that remain to be solved, but these should be readily addressed as work on the SAM architecture progresses.

6. Conclusion.

The SAM architecture is the first step in developing an inexpensive method for supporting data flow style parallelism in a von Neumann architecture. The features presented here are intended to be inexpensive to implement, and easy to use by a compiler. Although the split and merge instructions are simple, it has been shown that they can be used to implement arbitrarily complex static parallelism in a single environment. Using the prefix register to create multiple environments, it is possible to implement arbitrarily complex dynamic parallelism at a cost somewhat higher than that for static parallelism. Implementation issues have been discussed that allow for multiple processes as well as interrupt and fault handling. It is hoped that the SAM architecture will soon lead to the development of one or more small inexpensive multiprocessors.

REFERENCES

1. R. H. Kuhn, D. A. Padua (eds.) "Tutorial on Parallel Processing," IEEE Computer Society Press, Silver Spring Md, 1981.

2. B. J. Smith "Architecture and Applications of the HEP Multiprocessor Computer System," Real Time Signal Processing IV, Proceedings of SPIE, 1981, pp. 241-248.

3. Arvind, D. E. Culler, "Dataflow Architectures," Annual Reviews in Computer Science, 1986,.Vol 1, Annual Reviews Inc., 1986, pp. 225-253.

4. J. B. Dennis, "Data Flow Supercomputers," Computer Vol. 13, No. 11, Nov. 1980, pp. 48-56.

5. D. D. Gajski, D. A. Padua, D. J. Kuck, R. H. Kuhn, "A Second Opinion on Data-Flow Machines and Languages," Computer, Vol. 15, No. 2, Feb. 1982, pp. 58-69.

6. L. M. Pedersen, "Design for MISP: A Multiple Instruction Stream Shared Pipeline Processor," Technical Report CSG-37, Coordinated Science Laboratory, Computer Systems Group, University of Illinois at Urbana-Champaign, 1984.

7. P. C. Trealeven, R. P. Hopkins, P. W. Rautenbach, "Combining Data Flow and Control Flow Computing," The Computer Journal, Vol. 25, No. 2, 1982, pp. 207-217.

8. P. M. Maurer, A, E. Oldehoeft, "The Use of Combinators in Translating a Purely Functional Language into Low-Level Data-Flow Graphs," Computer Languages, Vol. 8, No. 1, 1983, pp. 27-45.

9. S. K. Abdali, "An Abstraction Algorithm for Combinatory Logic," The Journal of Symbolic Logic, Vol. 41, 1976, pp. 222-224.

Dynamic Structured Data Flow:
Preserving the Advantages of Sequential Processing in a Data Driven Environment

Israel Gottlieb

Bar Ilan University
Tel Aviv, Israel

Abstract. An architectural model is presented which enjoys the automatic sequencing of parallel operations characteristic of dataflow. However, the processors employed incorporate program counters and execute dependent sequences of actors in a sequential, fetch/execute, Von Neumann fashion. The synthesis of these -- ordinarily opposing -- approaches, is achieved without sacrificing the fine grained parallelism of classical dataflow. As a theoretical model, the machine is shown to achieve uninterrupted sequential execution of all critical paths in an arbitrary algorithm, subject to a single class of system overhead: *context initiation*.

Introduction

Dataflow computing systems have generally been motivated by the need to get away from an underlying machine model which is inherently sequential, to one which naturally supports parallelism. Proceeding from this fundamental perspective, proponents of these systems argue that we require a machine which relates independently to each elementary computing activity, scheduling them for execution subject only to dependence constraints inherent in the algorithm. The notion of independently scheduling each activity however, does not properly follow from the first premise. A cursory look at almost any computation graph shows that there are paths of computation activity of significant length; i.e. groups of activities that are inherently sequential. By saying that these sequentiality constraints are imposed by the algorithm we do not change the fact that they would be much more efficiently executed by a sequential processor than by one-at-a-time scheduling of each instruction. These and other efficiency problems of classical dataflow have been documented by Kuck and others (see for example [5]).

Indeed no processor can be more than a sequential machine; even the execution units in dataflow systems can only do one thing at a time. The goal of parallel architectures is to have many such units working together. This goal should not obscure the fact that an individual processor can perform significantly better by being optimized for sequential processing. Two decades of engineering experience with uniprocessor CPU's has taught us how to incorporate instruction caching, lookahead, pipelining etc.; forcing an execution unit to work on elementary tasks that are independently scheduled is retrogressive. Further, a typical dataflow machine consists of a long pipeline of independent units operating asychronously to each other. Apart from the length of the pipe itself, the full handshake protocols required at each unit to unit interface exact a significant price in performance -- over comparable synchronous systems.

In [3] a basic deficiency of the Von Neumann architecture was eloquently expressed: A single active processor is controlling a passive memory state. Elaborating, we may say that since the contents of memory represent the algorithm, the Von Neumann machine has the algorithm as the passive agent, with an additional artificial level of control being imposed upon it by the single CPU. We require the opposite arrangement: the algorithm should dictate control of the system resources, with processor power being merely one such resource. However, as noted above, the algorithm itself dictates sequential execution along paths in the computation graph (perhaps many such paths in parallel), as often or more often than not.

For algorithms which have static structure, i.e. no branches or loop variables that are determined dynamically, we can partition the computation graph into a set of sequential paths that communicate with each other (see e.g. [8]). Paths are statically allocated to different processors. If an operand being communicated from path *A* to a node *b* on path *B*, does not arrive in time for consumption by the processor executing *B*, the latter is suspended via a form of exception handling. The path is restarted when the operand arrives. The partition chosen assumes a worst case time for each elementary operation and tries to minimize the number of such "operand late" exceptions. A method for finding an optimal partition is given in [7]. The Hughes Data Flow Machine [4] and other projects have incorporated similar techniques in their designs.

This approach is workable for static programs. In dynamic code however, it is clearly not possible to allocate paths beforehand; we have no idea -- before the program actually runs -- how the partition should look. In this paper we propose a parallel architecture in which any processor may be conscripted to execute *dynamically determined segments* of sequential code. Thus the unit which we schedule for execution is an *execution path* rather than a single elementary activity as in classical dataflow. On the other hand we wish to realize Backus' directive for an architecture in which the algorithm is the active agent allocating passive resources. Accordingly, execution paths will be dynamically scheduled in accordance with the arrival of operands; no central control is imposed by a CPU. In particular our approach does not dictate the level of granularity used; parallelism at the lowest level may be exploited. We have dubbed this abstract machine *Dynamic Structured Data Flow* (DSDF).

In what follows we shall assume that a name is associated with each token generated by a computation, after the manner of e.g. the U interpreter [1]. Tokens are matched in a matching store to form executable operand pairs. This approach elegantly solves the problem of different contexts and also provides a convenient conceptual separation of instructions and data. Other reasons for this choice are elaborated in the sequel.

An additional assumption which we shall make in the DSDF architecture is that all processors have local access to all program code. The issue of implementation will be treated in a later section of this paper.

The Execution Discipline

In what follows, we assume an algorithm is represented by a directed graph G, in which arcs represent the flow of data and nodes correspond to activities. A *weight* may be associated with each node corresponding to its execution time.

Definition 1: A *sequential instruction path*(SP) is a subset of the activities in G which are linearly ordered with respect to dependence and where, for activities *a* and *b* in the SP, if $a < b$ then there is no *c* not in the subset such that $a < c < b$. Equivalently, an SP is a path in G.

Definition 2: An *execution process*(EP) will denote an instance of execution of an SP. An EP is created by asssociating it with a particular instruction; not with a general SP. Since a single instruction is also an SP, the resultant process created is in fact an EP consistent with this definition. However, an EP will, in general, progress with its activity in such a way as to execute an arbitrary SP. All activity of an EP is a coherent unit: execution is not data driven but rather proceeds according to sequence. If an operand required for the execution of some node along the SP has not arrived, the result is an *EP disabled exception* . EP's progress according to the *EP execution rule* given below.

Definition 3: A node which represents a unary instruction, or for which one operand has been made available, is called *EP enabled.*

EP execution rule: If a node *v*, with out-degree > 0, has been executed by an EP α then:

1) if the out-degree of v =1, and its successor v' is EP enabled, α proceeds with the execution of v'.
2) if the out-degree of v is > 1, α proceeds with an arbitrary successor v' which is EP enabled.

In all cases, α COMmunicates its result to any successors with which it does not continue. An EP terminates at a node *v* when either a) *v* has out-degree 0, or b) none of its successors are EP enabled.

If an operand is COM'd to a node which is EP enabled, a *context initiation* (CI) results. The CI corresponds to the creation of a new EP.

The net effect is that an EP is created to execute an SP of unknown extent. Moreover, even specifying an EP by initial node and length does not uniquely determine the SP which will be executed; more than one path may be possible.

To fix the above ideas more concretely let us consider the creation, execution and termination of a particular EP α, in a DSDF machine. All operands will be deposited in a central associative memory, with their tags as a search key. We associate with each EP a *Home*(H) datum; this is the datum kept by the processor in its internal register for the duration of an EP execution. It may be thought of as the context for the EP. A processor executing a binary operator node *w* uses its H datum as one operand and the tag of its H datum to address the token memory to retrieve the second operand. However, the value of the token is also sent in the memory access. If the match fails, the access is interpreted as a write; the value and its tag are installed in the memory, to await the arrival of a match. If the match succeeds, the access is interpreted as a read; the operand is returned to the processor. Thus the COM operation in the execution rule is no different in practice from an ordinary access to token memory. We associate the notion of communication with a token when the arc being traversed is not on the execution path of any EP. This is the only case of inter-EP communication in the DSDF machine. Returning to the specific example, some EP β created α when it executed a node *v* which had more than one successor that was EP enabled. That is, an operand was COM'd by β from *v* to say v' , resulting in a CI. In terms of our architecture, the instruction *v* specified two successors, i.e. two context tags. The processor executing *v* successfully retrieved the second operand from the matching store for one of its successors and continued execution with that path. For the other successor v', it sent the tag and value to the store with an indication that it is busy. The matching store controller detects the condition "match-with-processor-busy" and initiates a new EP α. The result of executing v' becomes the H datum for α; the H datum undergoes transformation with each node executed by α and provides the context with which matching operands are retrieved for each subsequent instruction. α terminates when all attempts to retrieve a matching operand fail, or when an instruction specifies no successor.

Decision and loop constructs are handled by the usual branch type instructions rather than the elaborate switches of dataflow. This is possible because SPs are sequential sequences of instructions meant to be fetched/executed by a processor which incorporates the usual program counter. The arrangement for an If-Then-Else construct incorporating a single SP is shown in figure 1(a). *BC* denotes a Branch on Condition.

If more than one SP is involved in an If-Then-Else, we can provide a *compound SP* as a program structuring aid. This is a group of static paths that communicate with one another, and which have a distinguished set of inputs and outputs. The If-Then-Else is then constructed from 3 compound SPs, as shown in figure 2. SP1 and SP2 are the alternate code sequences to be chosen; they must match in their numbers of inputs and outputs. The third SP is a *control token generator* ; it may take some subset of the inputs to SP1/SP2 and outputs the single boolean token which controls the IF construct.

The BC instruction is shown here accepting the boolean control token as a second operand. This operand is in effect communicated to it by an SP within the control token generator. An important issue is how to efficiently implement the distributor. We shall return to this problem in a later section.

Figure 1(b) illustrates looping in the DSDF machine. The usual tag operators for a dynamic machine are employed, allowing many instances of the Loop to execute concurrently. The loop body may be extended to a compound SP in a manner analogous to the IF construct; tag operators are added for each sequential path as required.

We now proceed to characterize the performance advantage accruing to the DSDF execution discipline.

Definition 4: Let v' , a successor of *v*, be a node which experiences a context initiation(CI). At the time of the CI, v' must have been EP enabled. If v' is a binary operation, let the node which provided its first operand be other than *v*. If the EP executing *v* continued with some other successor, say v'', the CI experienced by v' is called *inherent* .

Since *v* continued with some other successor v'', it follows by the execution rule that v'' could not have been executed by an EP arriving via another predecessor, say *w* , as follows: v'' has two predecessors, *v* and *w*, and hence is a binary operation. For the EP executing *w* to have continued with v'', it would have had to find it EP enabled by v, in which case the EP executing v could not have continued it. Hence the EP executing v was the only one that could have continued with either of the nodes v' or v''. Stated differently,

Lemma 1: An inherent CI can be avoided only at the cost of another CI.

Hence the name -- inherent. Note that the question of whether a CI is inherent or not is dynamically determined, as illustrated in figure 3. If b arrives at c before d, there will be no inherent CI, while if d arrives first, the EP executing b will be the only one able to continue with either a or c, hence an inherent CI will occur at one of them.

For a computation graph G, let $G_U(E)$ be the graph derived from G by "unravelling" all loops, specifying all branch decisions and ascertaining the actual execution time of every node, in accordance with some particular execution instance E of G. $G_U(E)$ is a weighted, directed, acyclic graph. If E executed on a DSDF machine, $G_U(E)$ will have been partitioned into a set of SPs which communicate with one another. In particular, the EPs generated by the DSDF execution constitute a path cover for $G_U(E)$ in which arcs in the cover correspond to transformations in place on the Home datum of a processor, and other arcs correspond to communication between EPs. We would expect that an EP may be held up by late arrival of operands which must be communicated by other EP's. Such an EP will have to be suspended and in effect, broken in two EP's -- each of which executes without interruption. Some optimal path cover exists for $G_U(E)$ which allows maximal uninterrupted sequential processing. That is, if a processor is allocated to each of the paths in the cover, the number of uninterrupted sequences executed will be minimal, or equivalently, the average length of uninterrupted sequences will be maximal. This is similar to what is done in [7] for static programs; in our case however, $G_U(E)$ does not exist until execution completes and hence the optimal cover is undefined a priori.

Lemma 2: Let $G_U(E)$ represent an instance of execution of G. If E was executed by a DSDF machine, then the following hold:

1) all nodes in $G_U(E)$ will have executed,
2) all EPs generated in the course of E progress without interruption, and
3) if an EP terminates at a node v then either the out-degree of v is 0 or any CIs experienced by successors of v must be inherent.

Proof: A node v' fails to be executed by the same EP which executed its predecessor v in one of 2 cases:

a) v is not EP enabled. This implies some other predecessor w, which must supply a second operand to v'. Hence when the EP executing w checks v', it will find it EP enabled. Execution will continue with v', or with some other successor v'', which is also EP enabled. For the latter, case (b) below, applies.

b) The EP executing v continued with some other successor. The rule specifies that a CI occurs, a new EP is created and hence v' is executed.

The DSDF machine begins by initiating an EP for each node in G with in-degree 0. by induction and the application of (a) and (b) above, we have that all nodes in $G_U(E)$ are executed.

Assertion (2) follows directly from the execution rule: there are no provisions for suspension/resumption; an EP can only progress or terminate.

Further, a CI at a node v' occurs only if the node is EP enabled and subsequently has an operand COM'd to it from some predecessor v. By the execution rule, there must have been some other successor v'' of v, also EP enabled, to which the EP executing v continued. The CI occuring at v' is therefore inherent, whence all CI's which occur in the course of E must be inherent. ▯

Our basic result follows directly from the Lemma. We say that a path p in $G_U(E)$ is a *critical path*, if $\mathbf{w}(p) \geq \mathbf{w}(q)$ for all paths q, where $\mathbf{w}(p)$ denotes the sum of the weights of the nodes on p. As before, weights correspond to execution times.

Theorem 1: All critical paths in $G_U(E)$ are executed in an uninterrupted sequential manner except for the system overhead associated with CIs. Further, only inherent CIs are experienced by a critical path.

Proof: Let p be a critical path. If p is executed by a single EP, then by Lemma 2(2) it cannot be interrupted. Let v be the node at which the first EP executing p terminated. If the successor v' of v on p is not EP enabled, there must be some other predecessor w of v' on an path q, which supplies the other operand to v', and which has not yet done so. Then p cannot be a critical path to v' because $\mathbf{w}(q) > \mathbf{w}(p)$ up to the node v' and hence p cannot be a critical path in $G_U(E)$.

Thus if the EP executing p -- a critical path -- terminated at v and sent an operand to v', the latter must have been EP enabled. By the execution rule a CI immediately results and an EP is created which continues with v'. By Lemma 2(3) this CI must be inherent. ▯

It is important to note that we have used the notion of a critical path in a somewhat loose fashion. In particular, it is critical paths in $G_U(E)$ that we are treating in Theorem 1. $G_U(E)$ represents execution on a DSDF machine, and CI times are treated as part of the weight value of nodes which experienced them. In terms of the execution times of elementary operations alone, it may be that $\mathbf{w}(q) > \mathbf{w}(p)$ holds for some instance of execution of G, while if CI times are added to the weights, the inequality is reversed. However, only inherent CIs will occur regardless of the amount of time they add to a path's execution. Further, it is clear that as the time overhead of a CI approaches zero, the critical paths of Theorem 1 become determined only by the inherent costs of the operations of the algorithm.

In sum, the EPs generated by a DSDF computation never wait -- if they are on a critical path. Further Theorem 1 demonstrates that the theoretical limit of parallel speedup can be achieved for any algorithm to the extent that we can reduce the CI time.

Since processors in DSDF are active and access the token memory in much the same way as a Von Neumann CPU, many of the pipeline stages of a typical dataflow machine, e.g. packet formation, packet arbitration etc., are eliminated. Tighter coupling and synchronous communcation protocols between processor and memory should be possible.

Conclusion

Theorem 1 highlights the central role played by CIs in system performance. If we bring specialized architectural resources to bear on this particular function, we should expect significantly increased performance. In a related paper [6], a new process spawn technique is described, which is designed to minimize system overhead in the DSDF environment. The method provides a standard interface from algorithm to processors which is used in a uniform fashion to support all basic code constructs which generate multiple processes. The 'distributor' node discussed earlier is easily treated as a special case.

A basic difficulty to be overcome is that of contention by multiple processors for the matching store. The usual solution to memory contention -- distributed access via interleaving of modules -- is not directly applicable to an associative store because a central controller must match the tag against all locations. In the Irvine machine model for example [1,2], each processor node has its own matching unit. Code is statically distributed to the different processors and no runtime variation on this partitioning is permitted. The same type of static criteria are used to decide which matching store in the network or ring is to receive result tokens generated by the program. This approach however, is unlikely to permit realization of the kind of performace potential described in this paper for DSDF: uninterrupted sequential execution of critical paths. The assumption that code and data are allocated statically necessarily implies that processor resources are not, in general, available for any executable activity. Only a matching store that is accessible by all processors would allow this generality of resource distribution. This issue, among others is being explored by a research group at Bar Ilan University which is developing an architecture and programming system based on DSDF principles.

References

[1] Arvind and Gostelow, K.P., "The U-Interpreter," *IEEE Computer*, Vol. 15, No. 2, February 1982.

[2] Arvind, Kathail, V., and Pignali, K., "A Dataflow Architecture with Tagged Tokens," Rep. LCS/TM-174, MIT, September 1980.

[3] J. Backus, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM,* Vol. 21, No. 8, August 1978, pp. 613-641.

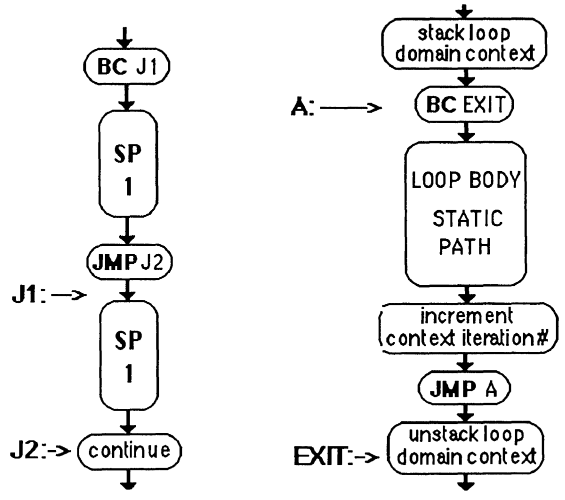
[4] M.L. Campbell, "Static Allocation for a Data Flow Multiprocessor," *Int'l Conference on Computer Architecture,* 1985

[5] D.D. Gajski, D.A.Panda, D.J.Kuck and R.H.Kuhn, "A Second Opinion on Dataflow Machines and Languages," *Computer,* Feb. 82, pp. 58-70.

[6] Gottlieb, I., "*Efficient Process Spawning in Functional Multiprocessor Environments,*" Technical Report TR-CS102PA, Dept. of Computer Science, Bar Ilan University, January 1988.

[7] Gottlieb, I., "The Partitioning of QSDF Computation Graphs," to appear in *Distributed Computing,* 1988

[8] Gottlieb, I., "SDF-The Structured Dataflow Model of Computing and its Architecture," *First Int'l Conference on Supercomputing,* Dec. 1985



(a) Figure 1 (b)

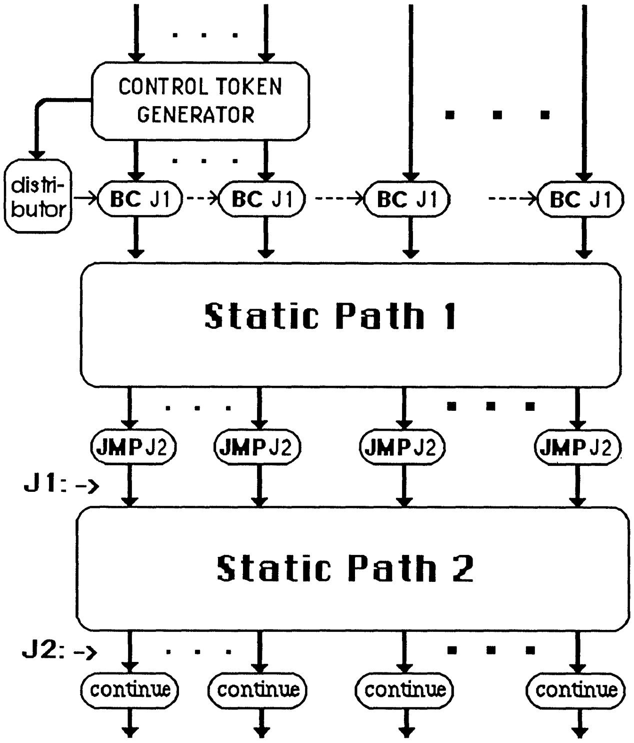


Figure2

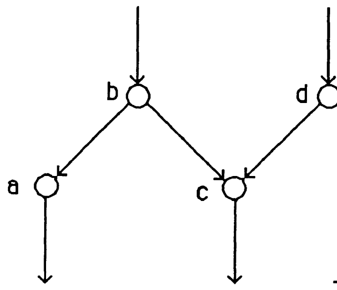


Figure 3

ITERATIVE ALGORITHMS IN A DATA-DRIVEN ENVIRONMENT*

Paraskevas Evripidou and Jean-Luc Gaudiot
Computer Research Institute
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, California
(213) 743-0249

Abstract— Data-flow principles of execution are an elegant way to synchronize many parallel processes in a large scale multiprocessor system. However, the execution by runtime detection of data dependencies also introduces many inefficiencies. In this paper, we apply the data-flow principles to a numerically intensive application: the Jacobi method for solving linear systems. We introduce a modification to the algorithm which allows a full exploitation of the parallelism inherent in the method by "vectorizing" a portion of the calculation and allowing some amount of "look-ahead" in the termination criterion. Resource allocation issues are then considered and we demonstrate by a combination of analytical and simulation methods a priority mechanism which allows both an increase in performance as well as better resource utilization.

1 Introduction

The computing needs of the near future are far beyond the power of any supercomputer available today. Physical constraints are placing an upper bound on the speed of single processors. Current technology is rapidly approaching this limit. A natural solution consists of having many processors collaborating to solve large problems. However the basic principles of von Neumann architectures preclude their extension to parallel execution environments [1]. Data-flow principles of execution on the other hand, offer easy programmability and tolerance to high memory latencies which are inevitable in large scale multiprocessors[2]. Iterative algorithms are very powerful tools for solving linear systems, and are particularly efficient in the solution of large sparse systems. These sparse systems are frequently encountered in the solution of Partial Differential Equations.

The data-flow model of execution [3] represents programs as graphs. The nodes (actors) of a data-flow graph are the instructions of the program. Tokens flow along the arcs carrying data from the producer actors to consumer actors. The static model of execution has been described by Dennis [4]. This model of execution allows only one instantiation of each actor at any given time. In the "dynamic" model (Arvind *et al.*) [2], multiple instantiations of the same actor are allowed. This is done by associating a different color (tagging) with the tokens which belong to different instances of the same actor. The rules for tagging the tokens are referred to as the "U-Interpreter".

Dynamic data-flow provides an efficient way of exploring the parallelism present in an algorithm. It has been shown [2] that the U-Interpreter principles are particularly efficient in conjunction with the FORALL type of constructs. This is the same type of construct which the von Neumann model of execution targets for optimization through vectorization. However, iterative algorithms have been traditionally implemented by using REPEAT-UNTIL and WHILE constructs. The U-interpreter cannot unravel these loops. These REPEAT-UNTIL loops can become more efficient for parallel execution if a FORALL (For $i = 1, n$) is inserted into their body. This allows the U-interpreter to simultaneously unravel n iterations instead of merely one. This paper examines the behavior of iterative algorithms in a dynamic data-driven environment and the enhancement in performance provided by the REPEAT-UNTIL transformation. We analyze our proposed scheme by a deterministic simulator of a dynamic data-flow architecture.

The goal of this paper is thus to study the performance of a dynamic data-flow architecture applied to a numerically intensive problem. A modification of the scheme is then introduced and a new execution priority mechanism is analyzed. In Section 2, we review essentials of the Jacobi method for solving linear systems. We also briefly discuss some data-flow principles relevant to the implementation of iterative algorithms. In Section 3, our transformation technique is introduced and simulation results are provided. The priority mechanism is presented and analyzed in Section 4, while concluding remarks are made in Section 5.

2 Iterative Algorithms in a Data-Driven Environment

Iterative techniques are very frequently used for the solution of systems of equations. An iterative technique to solve an $n \times n$ linear system $\mathbf{Ax}=\mathbf{b}$ starts with an initial approximation $\mathbf{x}^{(0)}$ to the solution $\mathbf{x}$, and generates a sequence of vectors $\{\mathbf{x}^{(k)}\}_{k=0}^{\infty}$ which can be shown to converge to $\mathbf{x}$. The Jacobi method for solving linear systems is shown by equation 1.

$$x_i^{(k)} = \frac{\sum_{j=1,j\neq i}^{n}(-a_{ij}x_j^{(k-1)}) + b_i}{a_{ii}} \qquad \text{for i = 1,2,\ldots,n} \quad (1)$$

*This material is based upon work supported in part by the U.S. Department of Energy under Grant No. DE-FG03-87ER25043

If **A** is strictly diagonally dominant, then for any choice of $\mathbf{x}^{(0)}$, the Jacobi method gives a sequence $\{\mathbf{x}^{(k)}\}_{k=0}^{\infty}$ that converges to the solution of **Ax=b**.

Graph construction

The simulation model follows in principle the U-interpreter model of execution. The architectural model is a 64 Processor hypercube, based on the MIT Tagged Token Dataflow Architecture [2]. Each operation (match, fetch etc.) takes one time unit. Also one time unit delay per communication hop is assumed.

Loop indices generation and the treatment of conditionals is the dominant part of data-flow graphs for iterative algorithms. The lack of global state and the single assignment principle make the data-flow graphs (programs) fundamentally different from conventional programs. The handling of loop indices receives different treatment in a data-flow environment. Consider the following nested loops:

For i in 1,k cross j in 1,l

In a von Neumann environment, the variable i will be updated k times while j will be updated $k \times l$ times. However in a data-flow environment there is no notion of variable, therefore the value i has to be created $k \times l$ times. The same holds for j. In other words, all the indices of the outer loops have to be created as many times as the index of the innermost loop. Another notable characteristic of the data-flow environment is the treatment of conditionals. Each input value of the true block of a conditional has to be gated through a true gate (T). In a similar fashion, each input value to the false block has to be gated through a false gate (F). This means that the tokens carrying the conditional must reach all the gates involved and all gates will fire. Loop indices and conditionals introduce a lot of "communication/synchronization" overhead in data-flow graphs of iterative algorithms. The transformation technique presented in the next section targets this overhead for a more efficient execution.

3 Transformation Techniques

In a data-flow environment all the parallelism present in an algorithm is inherently preserved, nevertheless, some changes in the implementation of algorithms can help take full advantage of the potential of the data-flow principles. In the remainder of this Section, our transformation technique for improving the performance of iterative algorithms in a dynamic data-flow machine is described.

3.1 Transformation algorithm

Iterative algorithms have traditionally been implemented in a step at a time approach. This was very natural at the pre-computer era since scientists and mathematicians would typically manually undertake the procedure. The same approach is very natural in conventional von Neumann architectures, because the (single) human brain is replaced by a single powerful processor. In von Neumann architectures, iterations are handled by REPEAT-UNTIL and WHILE constructs. The stopping criterion is naturally checked at each iteration. These REPEAT-UNTIL constructs severely limit the performance of parallel processors because they cannot be vectorized and/or multitasked. In addition, the performance of parallel processors is restricted by the fact that the stopping criterion is calculated at each iteration which usually involves a lot of synchronization overhead. This synchronization overhead is sequential in nature which has been shown (Amdahl's law) to have a very negative effect on the maximum achievable speedup.

For the great majority of iterative algorithms, we can theoretically estimate the order $O(n)$ of the number of iterations needed. When a computation is expected to take 100 iterations, for example, it does not serve any purpose to test the stopping criterion during the early iterations.

3.1.1 Basic Principles

Reduction of the overhead is possible by inserting a FORALL loop ([For $i = 1, n$] loop) inside a REPEAT-UNTIL construct. This allows us to check the stopping criterion every n iterations. In addition to the reduction of overhead due to the decrease in the number of evaluations of the stopping criterion, we can execute some parts of the various iterations in parallel.

The basic form of the modified Jacobi implementation is shown in Figure 1a. Figure 1b shows the traditional implementation of the Jacobi algorithm.

```
n  = expected_number_of_iterations(...)
  REPEAT
       For i=1,n  do  Jacobi(...)
        check_stopping_criterion(...)
        n = evaluate_n(...)
  UNTIL norm_of_error < tol
```

Figure 1a. The modified Jacobi Implementation.

```
REPEAT
    Jacobi(...)
      check_stopping_criterion(...)
UNTIL norm_of_error < tol
```

Figure 1b. Traditional Jacobi Implementation.

The function `expected_number_of_iterations()` is used to give an initial estimate of the number of iterations needed. The decision will be based on the nature of the problem and the convergence rate of the algorithm. The function `evaluate_n()` estimates the number of iterations needed to achieve the required accuracy.

Unravelling the FORALL loops yields considerable potential for parallel execution. However, it should be noted that almost 70% of a typical iterative program, coded using the U-Interpreter principles, is synchronization overhead related to the interpreter itself.

The "overhead/synchronization" actors are the target of our proposed scheme. In an iterative algorithm, the current iteration depends on all or part of the previous iteration. This means that in a data-flow environment, detection of data dependencies remains at the level of instructions, thereby allowing maximum pipelining among the iterations. For example, as soon as $x_1^{(k)}$ has been calculated, the next iteration $k+1$ can be initiated without awaiting the whole production of the vector $x^{(k)}$.

3.1.2 Estimating the number of iterations

The implementation of the `evaluate_n()` function is application dependent. For our experiments we used a function based on the observed convergence rate of the algorithm. Testing the stopping criterion consists of first calculating the distance $d_n = ||\mathbf{x}^{(n-1)} - \mathbf{x}^{(n)}||$ between the n^{th} approximation and the previous approximation. If this distance d_n is less than the desired value tol, the execution terminates. Otherwise, it proceeds to the next iteration.

The reduction coefficient R_{cf}

$$R_{cf} = \frac{d_{n-1}}{d_n} = \frac{||\mathbf{x}^{(n-1)} - \mathbf{x}^{(n-2)}||}{||\mathbf{x}^{(n)} - \mathbf{x}^{(n-1)}||} \tag{2}$$

indicates how many times the distance d_n at iteration n has been reduced w.r.t. the distance at iteration $n-1$.

Sometimes, a converging iterative process oscillates at the first few steps. To compensate for this phenomenon, the reduction coefficient R_{cf} can be estimated by averaging the effect of t iterations:

$$R_{cf} = \left(\frac{d_{n-t}}{d_n}\right)^{1/t} \tag{3}$$

Assuming that the reduction coefficient is uniform through the iterative process, we should expect that.

$$\frac{d_n}{(R_{cf})^k} = tol \Rightarrow k = \frac{\log(\frac{d_n}{tol})}{\log(R_{cf})} \tag{4}$$

and finally

$$\text{new } n = \lceil k \rceil \tag{5}$$

Equations 2 to 5 form the basic structure of the function `evaluate_n(...)`. Analytical proof for this function is beyond the scope of this paper.

3.2 Simulation Results

Both the Jacobi algorithm and our modified Jacobi implementation were evaluated for various problem sizes and machine configurations. Simulations were performed for problem sizes 3×3 to 32×32. The results, for the 8×8 and 32×32 systems, in terms of simulation time and speedup, are shown in Table I. The numbers shown under the "Repeat" column correspond to the traditional implementation of the Jacobi method and the ones under the "Forall" heading correspond to the modified Jacobi. Actually in order to access the effect of the Forall insertion only, we further modified our modified Jacobi Implementation. The inserted FOR loop (FORALL) is performed only once. The total number of iterations was known beforehand and "forced" into the program graph. The stopping criterion in the traditional implementation of the algorithm was chosen in such a way that exactly 10 iterations were necessary. This was done for all problem sizes. In other words, all the results shown in Table I correspond to an execution of 10 iterations. Also, the stopping criterion (check if $||\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}|| < tol$) is inside the For loop unlike the modified Jacobi implementation of Figure 1a. Therefore both implementations (Repeat and Forall) are identical, the only difference is that the FORALL construct will initiate all the iterations from the beginning. This indeed isolates and exacerbates the effect of the FORALL insertion.

#PEs	Repeat		Forall	
	Time	Speedup	Time	Speedup
	8x8			
1	40948	-	36569	-
2	22029	1.86	21322	1.72
4	13470	3.04	12111	3.02
8	10572	3.87	7564	4.83
16	9088	4.51	5941	6.15
32	8513	4.81	4738	7.71
64	8420	4.86	4164	8.78
	32x32			
1	562269	-	499319	-
2	289632	1.94	253749	1.97
4	157435	3.57	131747	3.78
8	91273	6.16	70379	7.09
16	57240	9.82	38997	12.80
32	42382	13.27	23356	21.38
64	34935	16.09	16975	29.41

Table I: Simuation results for the traditional implementation of Jacobi (Repeat) and the modified implementation (Forall).

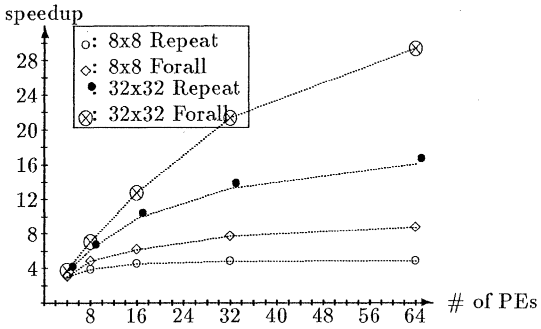


Figure 2: Speedup vs # of Pes for the 8×8 and 32×32 problem for both Jacobi and Modified Jacobi Implemenations.

The speedup vs. # PE's curve, for the 8×8 and 32×32 problems is shown in Figure 2. This plot shows clearly that the implementation with the FORALL outperforms the traditional implementation throughout the whole space.

However, although the FORALL implementation outperforms the traditional implementation over the whole space of experiments and it even projects higher margins of improvement, it is still not as efficient as expected. This modified scheme targets the "overhead/synchronization" actors introduced by the U-interpreter. Since this constitutes about 70% of the total graph, the improvement should have been higher. Among all possible factors, we examine the effect of the fact that the critical path is getting no special treatment. The critical path in this context is the data dependencies among successive iterations, i.e., computation actors.

4 Mechanism for Priority Handling

By completely unraveling more than one iteration at a time, more parallelism is exploited because we have the overhead/synchronization actors of all n iterations initiated from the beginning. However the actors belonging to the actual computation (for the rest of this paper they will be referred to as computation actors while the rest will be referred to as synchronization actors), get no special treatment. Therefore, at any given time t they have to compete with the synchronization actors for machine resources. The probability of a computation actor belonging to iteration i at time t to be allocated a specific resource r is given by:

$$P(i,r,t) = \frac{C_{i,r}(t)}{\sum_{j=1}^{n}(S_{j,r}(t) + C_{j,r}(t))} \qquad (6)$$

where $S_{j,r}(t)$ is the number of synchronization actors of iteration j at time t competing for resource r, and $C_{j,r}(t)$ is the number of the computation actors, also belonging to iteration j at time t competing for resource r and finally n is the number of active iterations. In other words the numerator of the r.h.s. of equation 6 is the number of computation actors waiting for resource r, and the denominator is the total number of actors waiting for resource r. Therefore the expected wait $E_r(i,t)$ time for any computation actor belonging to iteration i to get hold of resource r at any time t is

$$E_r(i,t) = (P(i,r,t))^{-1} = \frac{\sum_{j=1}^{n}(S_{j,r}(t) + C_{j,r}(t))}{C_{i,r}(t).} \qquad (7)$$

Calculating the expected duration of each iteration analytically is not a trivial matter; as demonstrated by equation 6 and 7, it is very complex to estimate how long it will take to gain access to a single resource. However it is clear that if there are many more synchronization actors than computation actors (per iteration) the expected wait time for a resource will be high.

4.1 The priority algorithm

The modified Jacobi implementation is motivated by the fact that otherwise idle processors can be kept busy by dealing with future iteration actors. However as suggested in the previous section the actors of the future iterations are actually competing with the actors of the current iteration. It has been shown in [6] that this indeed extends dramatically the lifetime of the early iterations. Therefore some sort of priority hierarchy is needed to ensure that the early iterations are not delayed. A good candidate to be used as a priority field is the tag associated with each token. The *u.c.s.i.* tag can be mapped by a one-to-one functional $f : tag \rightarrow N$. This means that by sorting the tags of the tokens in the firing queue (or any other queue), we can guarantee that the ordering imposed by the programmer is observed.

If no such strict priority is required then, the preference to tokens expected to be needed first can be enabled by using the iteration part i of the tag *u.c.s.i.* of the outermost level. Whether a loop is incrementing (For i=i_0,n where n> i_0) or decrementing (For i=n,i_n where n> i_n) the iteration part i of the tag is always incrementing. Therefore, if tokens with lower iteration values have priority over other tokens with higher iteration values, the competition for resources remains among actors belonging to the same iteration. In short, the priority mechanism is:

> *Tokens with lower tag iteration identifier i at the outermost level of their tag have priority over other tokens.*

This policy works well with various kinds of loop constructs. However, more complex analyses and policies may be required for more complex graphs.

4.2 Simulation Results

Having successfully tested the influence of both the inserted FORALL and the priority mechanism [6], we proceed by testing the entire modified Jacobi implementation with the priority mechanism enforced. The priority was enforced at the outer FORALL loop (the inserted loop). The same problem sizes, as in the previous Section, were investigated. Simulations results for the 8×8 and 32×32 systems are shown in Tables II and III. The results under column "FOR+PRI" correspond to our modified algorithm implementation with the priority policy enforced. The manner in which the speedup is calculated for this set of results differs from the definition of the speedup used for the previous set of results. Rather than comparing the performance of the algorithm using multiple PEs with that of the same algorithm using a single PE, we compare the performance of the algorithm using multiple PEs with the performance of the standard unmodified algorithm using a single PE. This was done because the object under evaluation was not the architectural model but the modified Jacobi implementation.

Figure 3 shows the plots for 8×8 and 32×32 for both the traditional implementation of Jacobi and the modified implementation with the priority mechanism enforced. The modified implementation (FOR+PRI) outperforms the traditional implementation throughout the whole space. The results for the rest of the problem sizes were similar to the ones presented. The speedup enhancement of the modified over the traditional implementation is in the range of 3.

Overall, the simulation results show that the proposed modification to the algorithm and the priority mechanism

Pes	REPEAT		FORALL		FOR+PRI	
	Time	S	Time	S	Time	S
1	302118	-	-	-	-	-
2	161364	1.82	157013	1.92	151517	1.99
4	97573	3.10	88703	3.41	78406	3.85
8	75778	3.99	54180	5.58	42231	7.15
16	64775	4.66	39684	7.61	27483	10.99
32	60979	4.95	32087	9.41	21571	14.01
64	59213	5.10	27461	11.00	19890	15.19

Table II: Simulation time and speedup (S) for the 8x8 matrix and n=40,20,10,5

# Pes	REPEAT		FOR+PRI	
	Time	Speedup	Time	Speedup
1	1281060	-	-	-
2	657115	1.94	650813	1.96
4	352485	3.63	332168	3.86
8	200807	6.38	173836	7.36
16	123343	10.39	93598	13.69
32	89070	14.38	58292	21.98
64	73164	17.50	39217	32.66

Table III: Simulation time and speedup for the for the 32 × 32 matrix and n=15,5,3

provide very good enhancement to the performance of the iterative algorithms. For "real life" applications with problem sizes in the range of $10^3 - 10^4$ the enhancement in speedup should be much higher. The interested reader can refer to [6] for a more complete and detailed presentation and analysis of the results.

5 Conclusions

In this paper, we have identified an important source of inefficient operations in data-driven machines. We have presented a program graph optimization scheme which can be applied to iteratively-based algorithms in dynamic data-flow environments. In our scheme, we rewrite the conventional WHILE (also referred to as REPEAT) operator and replace it by a WHILE/FORALL construct. This allows a more efficient "block" execution which bypasses much of the overhead traditionally associated with data-flow computations by enabling a certain amount of "look-ahead" on the termination criterion. We have verified our scheme by a combination of analytical and deterministic simulation means. We have shown that the speedup of the modified implementation is considerably enhanced over the traditional implementation. This is brought about by the larger amount of instructions which can be executed concurrently in an asynchronous fashion.

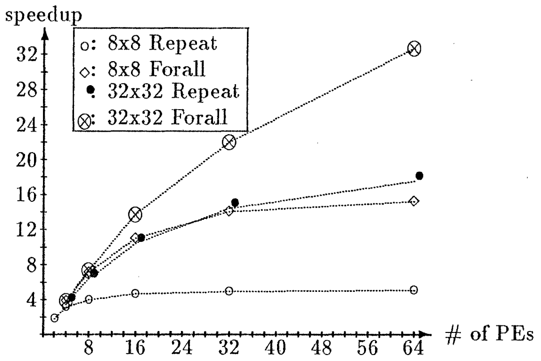


Figure 3: Speedup vs # of PEs for the 8×8 and 32×32 problem for both Jacobi and Modified Jacobi Implementations with the priority Mechanism enforced.

However, it has been discovered that this very "anarchy" in the scheduling of operations would underutilize the resources by favoring "low-yield" operations (*i.e.,* those overhead instructions which spawn few other actors in the "computation" part of the program while conversely enabling more "overhead/synchronization" actors: a case of bureaucratic folly!). In order to make sure that the computation work would be performed immediately, possibly at the expense of overhead work in higher iterations, we have hence developed a hierarchy mechanism which tends to give higher priority to the execution of instructions in lower iterations. Both analytical and simulation results show that the priority mechanism reduces the lifetime of the individual iterations of the modified algorithm. This yields considerably better resource utilization and faster execution. Higher performance is achieved as a direct combined effect of the modified algorithm and the priority mechanism.

References

[1] J. Backus, "Can programming be liberated from The von-Neumann Style? A functional style and its algebra of programs," *Communication of the ACM 21(8)*, (Aug., 1978), pp. 613-641.

[2] Arvind, R. S. Nikhil, "Executing a program on the MIT Tagged-Token Dataflow Architecture",*Parallel Architectures and Languages Europe,* volume II, (June, 1987), Springer-Verlag, pp. 1-26.

[3] IEEE Computer magazine, *Special issue on data-flow systems* (Feb., 1982.)

[4] J. B. Dennis, "First version of a data flow procedure language," In *Proceedings of the Colloque sur la Programmation,* (Apr., 1974), Springer-Verlag, pp. 362-376.

[5] Arvind, and R. E. Thomas, *I-Structures: An efficient data type for functional languages* Rep. LCS/TM-178, Lab. for Computer Science, MIT, (June, 1980).

[6] P. Evripidou, and J-L Gaudiot. *Numerical Algorithms in a Data-Driven Environment,* Technical Report CRI 88-19, Computer Research Institute, USC Department of Electrical Engineering-systems (Apr., 1988).

Graceful Degradation Schemes for Static/Dynamic Wavefront Arrays*

S. N. Jean C. W. Chang

University of Southern California
Signal and Image Processing Institute
Department of Electrical Engineering
University Park MC-0272
Los Angeles, CA 90089, U.S.A.

S. Y. Kung

Princeton University
Department of Electrical Engineering
Princeton, NJ 08544, U.S.A.

Abstract

In this paper, we propose two *graceful degradation* schemes for two-dimensional wavefront arrays. The first scheme is the *static dataflow scheme* which can be applied to both run-time and compile-time applications. The second scheme is the *dynamic dataflow scheme* which is mainly used for run-time applications. For the static scheme, topics discussed include *program complexity* and *load balancing*. For the dynamic scheme, the focus is on the *routing control*. Without broadcasting, a *distributed* routing algorithm which is *self-adaptive* to different faulty patterns is developed. An upper bound analysis of the system survival probability for both schemes is presented. Results of Monte-Carlo simulations for both schemes are compared and the tradeoff between *fault-tolerant capability* and *hardware complexity* is explored.

1 Introduction

VLSI/WSI processor arrays have regular and modular structures that match the computational requirements of most signal and image processing algorithms. Their parallel/pipelined processing characteristics will satisfy the very high computational throughputs in real-time applications. However, it is almost impossible to guarantee that an array with a large number of processing elements (PEs) will have all the PEs running correctly in a mission time. Therefore, fault-tolerant techniques must be incorporated into these systems. A desired objective of a fault-tolerant design is to maximize reliability while minimizing the corresponding hardware and time overhead.

Fabrication defects and operational faults on wafers are inevitable in today's IC technology. The motivation for incorporating fault tolerance in VLSI/WSI processor arrays is two-fold: **yield enhancement** in *fabrication time* and **reliability improvement** in *run-time*. As to yield enhancement, the fabrication defect problem can be solved by using static restructuring techniques [9] to connect the good components.

As to reliability improvement, operational faults have a much lower probability of occurrence as compared to production defects. Reconfiguration and graceful degradation have been used to deal with operational faults. A host-driven fault-stealing reconfiguration method is proposed by Sami and Stefanelli [11] to replace faulty PEs with good spare PEs. On the other hand, a distributed reconfiguration algorithm is proposed in [5]. In these methods, spare PEs are used and thus the size of the physical array is greater than the size of the logical array. When the logical array size is equal to the physical array size, graceful degradation techniques, which use *time redundancy* instead of *space redundancy*, should be applied. The row/column elimination method by Fortes and Raghavendra [2] is a typical example. Their design uses switches to bypass the whole row (or column) of a faulty PE and thus reduce the size of the logical array. Since the size of the logical array is reduced, the algorithm needs to be recompiled and the host computer is involved. A lot of time may be consumed in the recompilation and the propagation of data/control signals between the host computer and the array. This paper propose graceful degradation methods which require no recompilation of algorithms.

In VLSI array processing, it is critical to avoid large clock skews in synchronizing systolic computing network. A simple solution is to take advantage of the dataflow computing principle such as in wavefront array processing [6]. Conceptually, a wavefront array equals a *systolic array* plus *static dataflow computing*. Thus the requirement for correct *timing* in the systolic array is now replaced by a requirement for correct *sequencing* in the wavefront array. *Graceful degradation* schemes are proposed in this paper for two-dimensional wavefront arrays.

The paper is organized as follows: In Section 2, the array topology and fault assumptions are described. In Section 3, an upper bound analysis of the system survival probability is presented. Distributed adaptive routing algorithms with static and dynamic dataflow are described in Sections 4 and 5, respectively. Finally, we summarize some comparisons in Section 6.

2 Two Dimensional Grid Network Model

*This research was supported in part by the National Science Foundation under Grant MIP-87-14689, and by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization, administered through the Office of Naval Research under Contract No. N00014-85-K-0469 and N00014-85-K-0599.

Topology Most signal and image processing algorithms are dominated by filtering, transfer techniques, and some key linear algebraic methods. These algorithms, possessing common properties such as regularity, recursiveness and locality, can be efficiently computed in processor arrays with *mesh-type* interconnections. For example, Figure 1(a) shows some arrays with mesh-type interconnections. Different algorithms may require different kinds of interconnections. An array which exactly match the requirement of an algorithm is called a *logical array.* Apparently, the array that people build, the *physical array*, can not be the same as all the different logical arrays. In this paper, a **torus** topology (or called k-ary 2-cube interconnection network [1]), as shown in Figure 1(b), is provided for the *physical array.* Note that in a torus array, the failure of boundary PEs can be treated the same as that of internal PEs. From the implementation point of view, the wraparound interconnections of a torus are feasible on PC boards and wafers. Since adjacent PEs on the array are interconnected directly, no external switches [4] are required among PEs. Thus some VLSI area and hardware design work can be saved. Furthermore the model requires no global wires like that used in the host-driven global reconfiguration technique [10] or that used in the row/column elimination method [2]. In our model, bidirectional information flows are allowed in the logical array. Compared to unidirectional information flow as in [11], this model will have broader applications.

In the proposed grid network,

1. Each PE is self-tested (including computation and routing parts) and its test results are transmitted to adjacent PEs. Note that these *fault signaling wires* are not shown in Figure 1(b).

2. Since a faulty PE may contaminate data which pass through it, a PE will disconnect the communication links between itself and a neighbor faulty PE. This will force data flow only through good PEs and links.

3. When the system is failed by one part of the array, a system failure signal should be generated so that the host and the other part of the array can be notified. This system failure signal is very critical and a bad method would result in a very unreliable system. A *double checking* procedure should be used to make sure the signal is not from a faulty PE. The adoption of global links is not a good way to solve this problem from this point of view. Thus the system failure signal is also propagated locally with O(N) time penalty for an array with size $N \times N$. PEs which receive a system failure signal is responsible for executing the double checking procedure.

Fault Assumptions In this paper, we make the following fault assumptions:

1. The *self-testing* part is fault free. Some kinds of hardware redundancies, e.g., TMR, may be incorporated to assure this assumption.

2. The *communication links* are fault free. Some *error-correcting-code* techniques may be incorporated so that this assumption can be made.

3. *Fault signaling wires* to neighbor PEs are fault free. Because the information transmitted over each wire is only one bit, a robust design can be developed without causing too much hardware overheads.

4. *System failure signaling* is fault free.

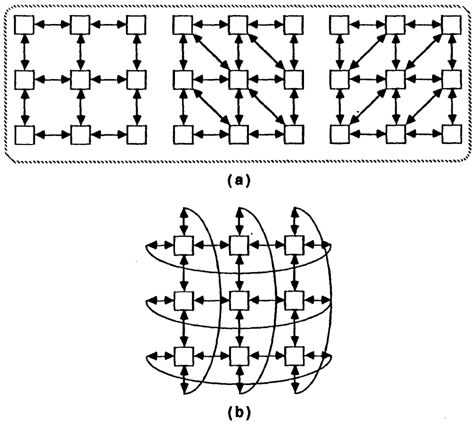


Figure 1: Two dimensional processor array grid network model. (a) Logical arrays, (b) Physical array – Torus.

Algorithm Matching Many algorithms, when executed on a processor array, require more communication links than what a torus array can support. For example, a southeastern link may be required for some algorithms. The problem of mapping a logical array to a physical array is called a *matching problem.* In order to implement such algorithms on a torus array, data which can not be transmitted over direct links should route through several links to reach their destinations.

1. If *synchronous* design is to be adopted, then the communication links have to be *time-shared* in order to accommodate all the data links required by algorithms. A time-shared scheme for the communication links was proposed in [7]. Their idea is to use buffers of different size for different data so that the data usages of the link can be time-multiplexed. Being able to determine the buffer size, they are able to use synchronous arrays to handle those algorithms.

2. If asynchronous design is to be adopted, then either *static* or *dynamic* dataflow schemes can be used.

    - A data token may route through several communication links to reach its destination. But the sequence of transmitting multiple data tokens within each PE is predetermined and can thus be incorporated into the program within each PE. This is a *static dataflow* approach.

- All data tokens can be tagged with information about their destination PEs. In this case, data tokens reach their destination via some routing algorithms. The routing path is not predetermined. This is a *dynamic dataflow* approach.

In this paper, only asynchronous designs are considered.

Two Steps in Graceful Degradation For a graceful gradable dataflow array, two steps are involved.

1. Once a PE is faulty, its task should be reassigned to a good PE. This is called *job reassignment.*
2. Once job reassignment is fixed, data *routing* is needed to implement the logical array on the faulty physical array. Two routing schemes are discussed in this paper, namely, static and dynamic dataflow approaches.

3 Job Reassignment and Upper Bound Analysis

Job Reassignment Job reassignment strategies directly affect the control complexity of PEs and the system fault-tolerance capability. In general, the more flexible the job reassignment is, the more complicated the PE hardware will be. However, the increase of reassignment flexibility also increases the fault-tolerance capability. To simplify the discussion, the job handled by each PE is assumed to be non-breakable. Since the slowest PE in an array determines the array throughput, jobs of different faulty PEs should be reassigned to different good PEs. Here we focus on the design with fixed reassignment, i.e., the job of faulty PE is always assigned to its left neighbor PE. Thus a good PE possesses at most two PE jobs.

Definition: *The system survival probability,* $\mathbf{P}_S$, is the probability that the system (array) works with tolerable degradation in performance given that the system works initially.

If r is the PE reliability, i.e., the probability that a PE is good (given that the PE is good initially), then for an array with size $N \times M$,

$$\mathbf{P}_S = \sum_{i=0}^{N\times M} r^{(N\times M)-i}(1-r)^i D_i \qquad (1)$$

where D_i is the number of successful faulty patterns with i faulty PEs.

Upper Bound Analysis of $\mathbf{P}_S$ In our scheme, the task of a faulty PE is always reassigned to its left neighbor PE. Thus no two adjacent PEs in the same row will be allowed to be faulty at the same time. Here we want to calculate the probability that a system fails due to the existence of two adjacent PEs in the same row. First, a *linear* array is considered. The result is then extended to a *ring* array. Finally, results for two-dimensional torus arrays are obtained.

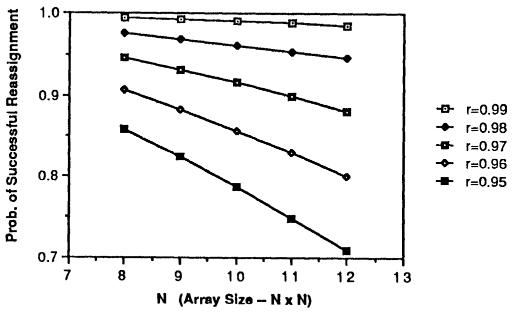


Figure 2: Upper bound for system survival probability vs. array size.

1. Assume there are M PEs in a *linear array* and q of them are faulty; denote the number of successful reassignments as $\mathbf{f}(M,\ q)$. Then $\mathbf{f}(M,\ q)$ can be computed from the following recurrence equations.

$$\begin{cases} \mathbf{f}(M,q) &= \mathbf{f}(M-1,q)+\mathbf{f}(M-2,q-1) \\ & \quad M \geq 2,\ \lfloor \frac{M}{2} \rfloor \geq q \geq 1, \\ \mathbf{f}(M,0) &= 1 \quad M \geq 1, \\ \mathbf{f}(M,1) &= M \quad M \geq 2, \\ \mathbf{f}(M,q) &= 0 \quad \text{otherwise.} \end{cases}$$

2. If the array structure is a *ring*, the number of successful reassignments, $\mathbf{g}(M,\ q)$, can be computed by noting the following relation between $\mathbf{f}(\cdot,\cdot)$ and $\mathbf{g}(\cdot,\cdot)$.

$$\mathbf{g}(M,q) = \mathbf{f}(M-1,q) + \mathbf{f}(M-3,q)$$

3. Assume that the array size is N rows and M columns in our **torus** model. If there are q faulty PEs and denote the number of successful reassignments as $\mathbf{h}(N,\ M,\ q)$, then

$$\mathbf{h}(N,M,q) = \sum_{k=0}^{q} \mathbf{h}(N-1,M,q-k)\mathbf{g}(M,k) \quad N \geq 2, \qquad (2)$$

with $\mathbf{h}(1,M,q)=\mathbf{g}(M,q)$.

Once we obtain $\mathbf{h}(N,\ M,\ q)$, we may compute $\mathbf{P}_S$,

$$\mathbf{P}_S = \sum_{q=0}^{N\times M} \mathbf{h}(N,M,q) r^{(N\times M)-q}(1-r)^q \qquad (3)$$

Let's assume the 2-D array is square (*i.e., N=M*) and compute $\mathbf{P}_S$. The result is shown in Figure 2.

The accurate performance of the model depends on achievement of both job reassignments and token routings. In the above analysis, we consider successful job reassignment cases without any routing consideration. Hence the results in Figure 2 are the *upper bound* performance.

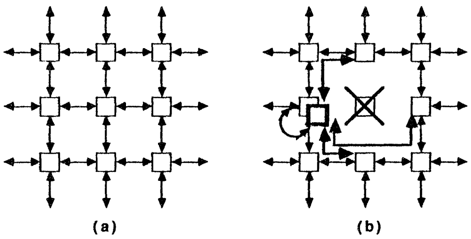


Figure 3: The communication patterns: (a) before the fault occurrence; (b) after the fault occurrence and task reassignment.

4 Static Dataflow Scheme

For a wavefront array, the dataflow sequences within each PE are predetermined and can be handled by the program within each PE. Here we propose a static dataflow graceful degradation scheme.

Definition: An *N(a,b) neighbor region* of a PE is a $a \times b$ region with the PE as the center of the region. Here a and b are odd integers.

First let's consider the case with single faulty PE. The tasks originally handled by a faulty PE can be shared by PEs on its $N(3,3)$ neighbor region. The load sharing scheme can be predetermined and thus can be handled by programs. Note that PEs located outside the $N(3,3)$ neighbor region need not change their dataflow sequences.

The more complicated faulty patterns we try to handle, the more complicated the program would be. To simplify the programming, we propose to handle only cases where **no two $N(3,3)$ neighbor regions of faulty PEs overlap**.

Program Complexity A comparison of communication patterns as shown in Figure 3 illustrates the complexity of the program to be written into each PE. Once a fault occurs, PEs on the $N(3,3)$ neighbor region should be notified. Thus eight error signaling wires should be used for each PE. Furthermore, when a fault occurs, the routing functions of PEs in the $N(3,3)$ neighbor region are different. In Figure 3(b), there are seven PEs with different faulty routing functions and one PE (the one on the right-upper corner) with the normal routing function. To adapt to different faulty patterns, each PE must be able to execute eight different routing functions, i.e., the normal function and seven emergency functions. These eight functions are predetermined and can be implemented either by hardware, some kind of router, or by software programming.[1]

Note that arrays with different number of faulty PEs possess almost the same throughput since their slowest PEs have the same number of PE jobs. If a PE job can be shared by several PEs, then *load balancing* should be taken into account. Since load balancing can be predetermined and handled by programs, the degradation of array throughput can be minimized without increasing the hardware complexity.

Monte-Carlo Simulation In this scheme, $\mathbf{P}_S$ is the probability that, in the array, no two $N(3,3)$ neighbor regions of faulty PEs overlap. Thus a faulty pattern is successful if and only if no two $N(3,3)$ neighbor regions of faulty PEs overlap. A necessary and sufficient condition for a faulty pattern to be successful is all the faulty PEs are located outside the $N(5,5)$ neighbor regions of other faulty PEs.

To estimate $\mathbf{P}_s$, a Monte-Carlo simulation was performed for different array size (N) and different PE reliability (r). In the simulation, to estimate D_i (cf., Eq. 1), 100,000 random faulty patterns, each with i faulty PEs, are used and the number of successful faulty patterns are counted. In this way, we estimate D_i and then use Eq. 1 to compute $\mathbf{P}_s$. The results for different PE reliability, r, are shown in Figure 4. Note that for a system without fault-tolerance capability, the system reliability is $r^{N \times N}$ and is also shown in Figure 4.

Compile-Time Environment As we can see, $\mathbf{P}_S$ for this scheme is not very attractive. The problem is the requirement of non-overlapping $N(3,3)$ neighbor regions to reduce the complexity of the program residing in each PE. If higher $\mathbf{P}_S$ is required for run-time environment, the dynamic dataflow scheme, as explained later, may be used. However, if the compile-time fault tolerance is to be considered, then $N(3,3)$ neighbor regions need not be non-overlapping and $\mathbf{P}_S$ can be improved drastically. This is explained below.

Compile-time fault-tolerance is for arrays which are designed as "general purpose" machines with array compilers to compile *array programs* into *PE programs*. Users write programs in the array level and produce array programs. To execute an array program, the array is first examined to see if there are any faulty PEs. Then, adaptive to different faulty patterns, the array compiler produces PE programs. At last, PE programs are loaded into the array and executed.

In this case, the error detection time can be longer (compared to run-time environments), more complete fault location operations can be enforced, and communication link errors may be detected (i.e., the fault assumptions can be reduced). The PE programs loaded into PEs may be different. But each PE needs only one PE program disregard the faulty pattern. In this case, the faulty pat-

[1] Some checkpoints or resynchronization are required to restart PEs on the same neighbor region.

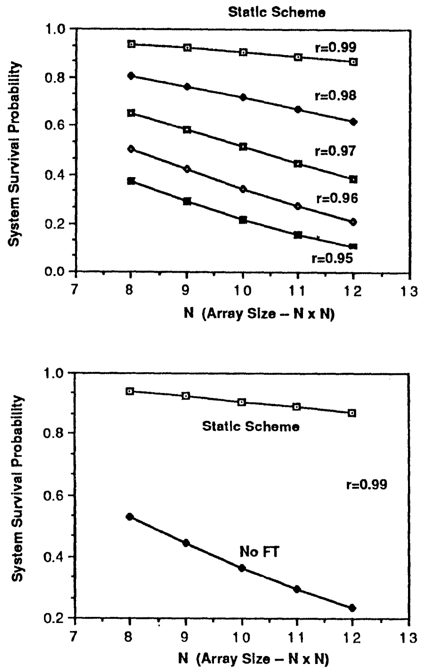


Figure 4: System survival probability with/without static graceful degradation.

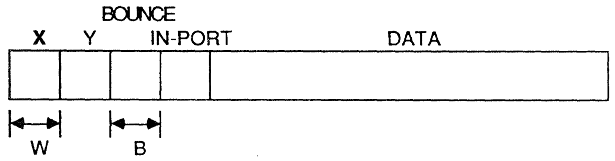


Figure 5: Protocol of data token.

tern can be more complicated without greatly increasing the complexity of each PE's program. Thus the system can handle some faulty patterns with overlapping $N(3,3)$ neighbor regions of faulty PEs and drastically increase $\mathbf{P}_S$.

5 Dynamic Dataflow Scheme

An array with dynamic dataflow is an array whose data are tagged with some header. By extending the header, graceful degradation may be achieved.

A data token definition (see Figure 5) for fault-tolerance can be stated as follows:

1. X/Y field: used to denote the relative position of the current PE (which the token resides) to the destination PE **before reassignment.**[2] If the current PE and the destination PE locations are (i, j) and (i_D, j_D) respectively, then x/y, the value of the X/Y field, is defined as $x = i - i_D$ and $y = j - j_D$.

2. IN-PORT field: used to denote the original input port of the destination PE. In a torus array, two bits are required to distinguish the four input ports (see Figure 1(a)). If the links in the array are unidirectional, one bit is sufficient to distinguish the two input ports.

3. BOUNCE field: used to indicate the number of times the token is kept away from the destination PE (because of the existence of faulty PEs).

Routing Algorithm The basic idea is to locally update the X/Y field so that the destination PE is gradually approached. A 0/0 value in the X/Y field means the destination PE is reached. Then the IN-PORT field can be used to distinguish the source PE. If the destination PE is faulty, then the data token will reach the reassigned PE with its X/Y field as $-1/0$. Note that the X/Y field can be used for the reassigned PE to distinguish whether the token is for itself or for its faulty right neighbor PE.

When a faulty PE is encountered during a routing, a data token might be kept away from its destination PE. The BOUNCE field is used to constrain the number of "bounces", i.e., the action that forces a data token to leave its destination PE. For example, the number of bounces cannot be more than 3 if 2 bits are used for the BOUNCE field. When the number of bounces for a data token is over 3, the system is declared "failed". Apparently, the larger the number of bits for the BOUNCE field, the higher the number of faulty PEs the system can tolerate and thus the higher $\mathbf{P}_S$. The purpose of the BOUNCE field is to avoid infinite loops which will be explained later.

In the routing algorithm, no *backtracking* is allowed. That means once leaving a PE, a data token is not allowed to return to the PE immediately. This is to avoid some useless routing steps.

Loop Free Requirement Because of the lack of global information, most distributed routing algorithms need to solve the problem of *infinite looping* [8,3]. That is, tokens may be trapped in some loops forever. In a fault-tolerant array, data token may be trapped in infinite loops for some peculiar faulty patterns.

To solve the infinite loop problem, a straightforward way is to use an AGE field in the data token to indicate the number of links the data token has traveled through. Apparently, if an upper bound is set on the AGE field of a token, it is impossible to have an infinite loop. The problem with this scheme is too many bits would be used for the AGE field. By using the BOUNCE field, the proposed algorithm can avoid infinite loops with less number of bits.

[2] Since all fault signaling wires are for local communication, the source PE may not know the failure of the destination PE. Thus the location of the destination PE before reassignment must be used.

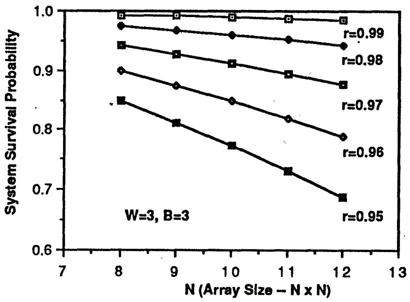


Figure 6: System survival probability for dynamic dataflow scheme with **W**=3 and **B**=3.

Monte-Carlo Simulation Let **W** be the number of bits in the X/Y field. and **B** be the number of bits in the Bounce field. To take routing into account, a Monte-Carlo simulation was performed where, for each specific case (with specific array size and specific number of faulty PEs), 100,000 random faulty patterns were used. In the simulation, the unidirectional mesh array communication is assumed to be the logical array. The simulation results for **W**=3 and **B**=3 are summarized in Figure 6.

Tradeoff Since **W** constrains the range which tokens can flow and **B** sets an upper bound on the number of faulty PEs which can be tolerated, increasing **W** and **B** will improve the routing capability and $\mathbf{P}_S$. However, the hardware cost is increased accordingly. Monte-Carlo simulations were made to illustrate the system survival probabilities and the results for four cases (**W**=2, **B**=2; **W**=2, **B**=3; **W**=3, **B**=2; and **W**=3, **B**=3) are summarized in Figure 7.

Note that $\mathbf{P}_S$ for the case **W**=3 and **B**=3 are very close to the upper bounds. It means that very few routings fail and thus no more bits should be used for **W** and **B**.

Communication Overhead To illustrate the communication overhead of the routing scheme, we define the *maximum routing distance* for a successful faulty pattern as the length of the longest routing path. The *average maximum routing distance* of an array is the average of the maximum routing distances over all the successful faulty patterns. This parameter indicates the involved communication overhead and can be expressed as

$$\sum_{i=0}^{N\times M} \begin{pmatrix} N \times M \\ i \end{pmatrix} L_i r^{(N\times M)-i}(1-r)^i$$

where L_i is the average maximum routing distances for successful faulty patterns with i faulty PEs and can be estimated by simulations. Table 1 shows the simulation results for the average maximum routing distance.

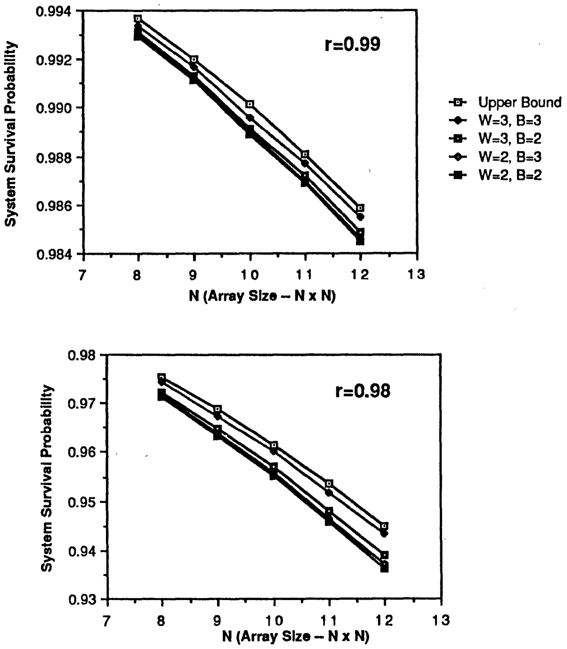


Figure 7: System survival probabilities for cases with different **W** and **B**.

r	Array Size	W=2 B=2	W=2 B=3	W=3 B=2	W=3 B=3
0.99	8x8	2.45	2.45	2.45	2.45
	9x9	2.73	2.74	2.74	2.74
	10x10	2.97	2.97	2.97	2.97
	11x11	3.21	3.21	3.21	3.21
	12x12	3.42	3.42	3.42	3.42
0.98	8x8	3.38	3.38	3.38	3.39
	9x9	3.66	3.67	3.67	3.68
	10x10	3.91	3.91	3.91	3.93
	11x11	4.10	4.10	4.11	4.13
	12x12	4.26	4.26	4.26	4.28
0.97	8x8	4.00	4.01	4.01	4.05
	9x9	4.27	4.27	4.28	4.32
	10x10	4.49	4.50	4.51	4.55
	11x11	4.65	4.66	4.66	4.72
	12x12	4.79	4.80	4.81	4.88
0.96	8x8	4.46	4.47	4.48	4.55
	9x9	4.71	4.72	4.73	4.80
	10x10	4.92	4.93	4.95	5.04
	11x11	5.11	5.13	5.14	5.26
	12x12	5.30	5.32	5.33	5.47
0.95	8x8	4.81	4.82	4.85	4.98
	9x9	5.10	5.13	5.14	5.29
	10x10	5.33	5.35	5.37	5.53
	11x11	5.54	5.57	5.58	5.79
	12x12	5.75	5.78	5.79	6.04

Table 1: The average maximum routing distances.

6 Conclusion

Two *graceful degradation* schemes are proposed in this paper for wavefront arrays in run-time applications. A graceful degradation scheme without tagging data is proposed for the static dataflow array. For the dynamic dataflow array, a scheme which extends the header of data token is stated. An upper bound analysis of the system survival probability for both schemes is presented. Simulations were made to estimate the system survival probabilities of both schemes. It is found that the dynamic dataflow scheme exhibits higher system survival probabilities (see Figure 8) which are very close to the upper bounds. This is at the expense of higher hardware complexity. In the compile-time environment, the system survival probability can be improved for the static dataflow scheme by relaxing the constraints that no two $N(3,3)$ neighbor regions of faulty PEs overlap. It is noted that although the array compiler will be somewhat more involved, the hardware complexity remains the same.

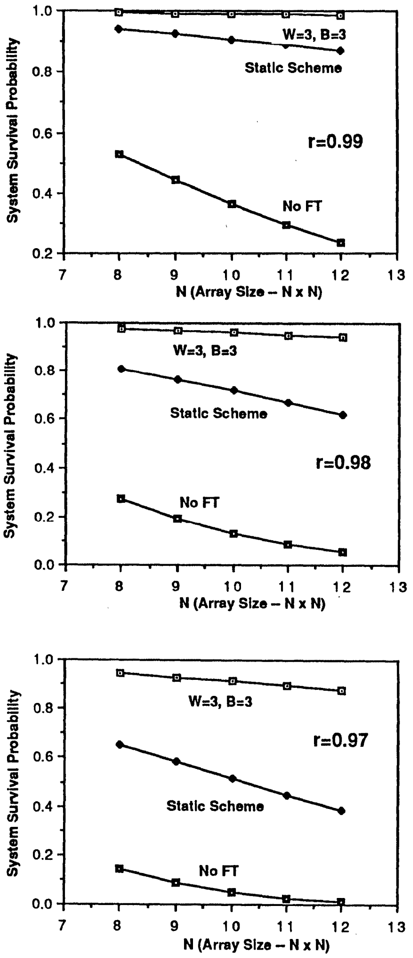


Figure 8: Comparison of static scheme and dynamic scheme in terms of the system survival probability.

References

[1] W. J. Dally. Wire-efficient VLSI multiprocessor communication networks. In *Advanced Research in VLSI*, pp. 391–415, 1987.

[2] J. A. B. Fortes and C. S. Raghavendra. Gracefully degradable processor arrays. *IEEE Transactions on Computers*, pp. 1033–1044, November 1985.

[3] J. M. Jaffe and F. H. Moss. A responsive distributed routing algorithm for computer networks. *IEEE Transactions on Communications*, pp. 1758–1762, July 1982.

[4] I. Koren and D. K. Pradhan. Yield and performance enhancement through redundancy in VLSI and WSI multiprocessor systems. *Proc. IEEE*, pp. 699–711, May 1986.

[5] S. Y. Kung, C. W. Chang, and C. W. Jen. Real-time reconfiguration for fault-tolerant VLSI array processors. *Proc. Real-Time Systems Symposium*, pp. 46–54, December 1986.

[6] S. Y. Kung, S. C. Lo, S. N. Jean, and J. N. Hwang. Wavefront array processors: from concept to implementation. *IEEE Computer Magazine*, pp. 18–33, July 1987.

[7] S. Y. Kung, S. N. Jean, and S. C. Lo. Matching algorithm to array processors. In *ACM-IEEE Proc. of FJCC*, October 1987, pp. 357–365.

[8] P. M. Merlin and A. Segall. A fairsafe distributed routing protocol. *IEEE Transactions on Communications*, pp. 1280–1287, September 1979.

[9] W. R. Moore. A review of fault-tolerant techniques for the enhancement of integrated circuit yield. *Proc. IEEE*, pp. 684–698, May 1986.

[10] M. Sami and R. Stefanelli. Fault-tolerance and functional reconfiguration in VLSI arrays. In *Proc. ISCAS 1986*, pp. 643–648, 1986.

[11] M. Sami and R. Stefanelli. Reconfigurable architectures for VLSI processing arrays. *Proc. IEEE*, pp. 712–722, May 1986.

Data-Driven Multiprocessor Implementation of the Rete Match Algorithm*

Jean-Luc Gaudiot, Sukhan Lee, and Andrew Sohn
Computer Research Institute
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-0781

Abstract

Much effort has been expended on developing special architectures dedicated to the efficient execution of production systems. While data-flow principles of execution offer the promise of high programmability for numerical computations, we demonstrate here that the data driven principles can also be applied to symbolic computations. In par´icular, we consider a mapping of the Rete match algorithm on the MIT Tagged Token Data-flow Architecture. The results of a deterministic simulation of this multiprocessor architecture demonstrate that artificial intelligence production systems can be efficiently mapped on data-driven architectures.

1. Introduction

In rule-based production systems, it is often the case that the rules and the facts needed to represent a particular production system in a certain problem domain would be very large. It is thus known that simply applying software techniques to the matching process would yield untolerable delays. Indeed, as [Fo82] has pointed out, the time taken to match patterns over a set of rules can reach 90% of the total computation time spent in expert systems. The need for faster execution of production systems has spurred research in both the software and hardware domains. The conventional control flow model of execution is limited by the "von Neuman bottleneck" [Ba78]. Architectures based on this model cannot easily deliver large amounts of parallelism [AI83]. The data driven model of execution has therefore been proposed as a solution to these problems. These principles have been surveyed by [Ga87]. The purpose of this paper is to demonstrate the applicability of data-flow principles of execution and of architecture design to the solution of artificial intelligence (AI) oriented problems. For this purpose, a subset of production systems problems, the Rete match algorithm has been chosen.

Section 2 briefly introduces production systems and the Rete algorithm. Section 3 discusses mapping the Rete algorithm to data-flow architectures. There, we identify the problems associated with the Rete algorithm in a multiprocessor environment and give solutions to these problems through the allocation and distribution policies we have developed. In section 4, simulations are carried out and performance observations obtained for a data-driven environment are compared to those of a conventional control-flow approach. Concluding remarks as well as future research topics are discussed in section 5.

2. Production systems and the Rete algorithm

A production system is a program composed entirely of conditional statements called *productions* or *rules*. The *left hand side* (LHS) is the condition part of a production rule, while the *right hand side* (RHS) is the action part. The collection of all the production rules in a production system forms a rule base, called a *production memory*. The productions in the production memory operate on a *working memory* which is a set of assertions called *Working Memory Elements* (WMEs). Both patterns and WMEs have a list of elements, called *Attribute Value Pairs* (AVPs). The value of an attribute can be either fixed (in lowercase) or variable (in uppercase).

Production Memory

Rule 1	Rule 2
[(a Z) (b Y)]	[(p 1) (q 2) (r X)]
[(c X) (d Y)]	[(c X) (d W)]
[(p 1) (q 2) (r X)]	[(l 5) (m 6) (n W) (o Z)]
$\Longrightarrow$	$\Longrightarrow$
[Modify (c Y) (d X)]	[Remove 1st pattern]

Working Memory

1: [(p 1) (q 2) (r *)]
2: [(p 1) (q +) (r =)]
3: [(a +) (b 6)]
4: [(l 5) (m 6) (n 6) (o 2)]
5: [(l 5) (m 7) (n 6) (o @)]
6: [(c *) (d 6)]

A typical execution cycle of production systems is composed basically of three steps: *matching, conflict resolution*, followed by *rule firing*: In the matching cycle, the LHSs of all the production rules are matched against the current WMEs to determine the set of satisfied productions. The conflict resolution cycle selects one production,

*This material is based upon work supported in part by the National Science Foundation under Grant No. CCR-8603772 and by the USC Faculty Research and Innovation Fund.

if the set of satisfied productions is non-empty. The actions specified in the RHS of the selected productions are performed in the rule firing cycle. In this paper, we limit ourselves to the *matching step only* since it takes most of the computation time in the evaluation of production systems.

The Rete match algorithm [Fo82] is one of the best known approaches used in the matching of objects in production systems. It constructs a condition dependency network, saves in memory the information concerning the changes in the working memory between production cycles, and then utilizes them at a later time. This is based on the observation, called *temporal redundancy* [BF85], that there is little change in the working memory between production cycles. The Rete algorithm further reduces the matching time by sharing identical tests among productions. It stems from the fact that the productions have many similar or identical parts, called *structural similarity.* A condition-dependency network for the two rules listed above has been constructed in Figure 1. The network consists of several types of nodes: root node, one-input nodes, two-input nodes, negated two-input nodes and terminal nodes (see [Fo82] for details).

3. Data-flow implementation of the Rete algorithm

In this section, we identify the necessary mapping schemes to suit the Rete match algorithm and the data-flow multiprocessor. Bottlenecks in the Rete algorithm are identified and possible solutions are suggested.

3.1. Suitability of the data-driven execution model

Executing the Rete algorithm on a data-flow multiprocessor has many advantages over execution on a conventional control-flow computer: First, the execution principles of the Rete algorithm are driven by incoming data tokens, *i.e.*, execution may proceed whenever data are available. In any situation, multiple firings of actor in data-flow and comparison tests in the Rete algorithm are possible. Second, both are based on the single assignment principle, *i.e.*, no data modifications except arrays. Third, both a data-flow machine and the Rete algorithm need dependency graphs. Fourth, the requirement for the memorization capability in two-input nodes of the Rete algorithm assumes a good structure handling technique and this can be effected by using the I-Structure Controller in Arvind's Dynamic Data-flow machine. Finally, the dynamic data-flow architecture allows an easy manipulation of the counters (see [Fo82]) since the counter for negated-pattern processing can be treated the same as other tags in the dynamic architectures.

3.2. The Rete algorithm in a multiprocessor environment

Mapping production systems onto multiprocessor systems has been done in several ways in the recent literature. *Direct mapping* employed by [SM84, St87] for DADO uses "full distribution," which allocates one rule to an available PE to achieve the production-level parallelism. In [Gu84] a relevancy between the rules and the WMEs is identified and used to directly allocate rules to PEs. It has been suggested by [Bi85] that the semantic network can directly be viewed as a data-flow graph. Each node in the semantic network corresponds to an active element capable of accepting, processing, and emitting value tokens traveling asynchronously along the arcs. The other approach suggested by [TM85] may be considered an *indirect mapping.* In this approach, all productions are analyzed and grouped according to the dependency existing between productions to enable parallel firing of rules.

The mapping scheme adopted for our simulation, however, is somewhat different from the forementioned approaches. The motivation for the choice of an alternative method is in two facts: First, the architecture we have adopted is based on data-flow principles of execution. Since the parallel model employed in this paper exploits parallelism at the production level, condition level, and further attribute-value pair level, the mapping scheme must be efficient to utilize all the possible forms of parallelism inherent to both data-flow principles and the Rete algorithm.

Second, the Rete algorithm presents two bottlenecks which substantially degrade the performance of the production system in our parallel machine: Since the root node distributes tokens one at a time to all PEs, tokens will pile up on the input arcs as shown in Figure 2. This is due to the fact that rules cannot be copied to all PEs. The second inefficiency can also be seen on Figure 2. Assume that m tokens are received and matched on the left input arc of the two-input node. Further assume that a token is received and matched on the other input of the two-input node. The arrival of this last token will trigger the invocation of m comparisons with the values received and stored in the *left memory* LM_3 of the two-input node. On the average, there will be $O(m)$ such tests. Should the situation have been reversed and n tokens be in the *right memory* RM_6, a token on the left side would provoke $O(n)$ comparisons. The internal workings of this two-input node are therefore purely sequential. In order to avoid the wasted time in searching through the entire memory, an effective allocation of two-input nodes and one-input nodes should be devised.

3.3. Allocation of productions

The allocation policy we are going to use does not follow the structural similarity discussed in section 2. Those condition patterns that are shared by different productions are copied and allocated to different PEs. It is based on the fact that by copying shared patterns and allocating to different PEs the overhead in inter-processor communication can be substantially reduced. However, this policy will consume a lot of processor space and be costly as the number of productions that share patterns or part of patterns increases.

Suppose that n PEs are available. They are logically partitioned into $\sqrt{n}$ groups, where each group has $\sqrt{n}$ PEs. Those condition patterns that have i AVPs in each pattern are allocated to PEs in group i. Each two-input node is split into two memories; left- and right memories. Memories are allocated to PEs, where the corresponding one-input nodes are allocated. Those memories that have no corresponding one-input nodes are are allocated to PEs in Group 0. Allocating a memory to a PE will ensure an even distribution of processing load across the processor space. At the same time, we can realize parallel matching in condition level. Terminal nodes are not explicitly allocated to PEs for our simulation.

Based on the above allocation policy, the network is allocated to PEs, shown in Figure 3. PEs are partitioned into 5 different groups, where Group 1 is not used in our example since no condition pattern has only one AVP. Consider the first pattern of Rule 2, [(p 1) (q 2) (r X)], for example. The sequence of nodes in the pattern and the left memory for that pattern are labeled 9 through 12 in Figure 1 (9 through 11 are one-input nodes). Since the pattern has 3 AVPs, it is classified into Group 3 and allocated to PE_1 of Group 3, designated $PE_{3,1}$. The second pattern of the Rule 2 has 2 AVPs and right memory, labeled 13 through 15. It is classified into Group 2 and allocated to PE_2 of Group 2, designated $PE_{2,2}$. In general, the number of PEs needed to allocate productions is proportional to the number of inter-element feature tests in the productions.

3.4. Dynamic WMEs distribution

In order to overcome the bottleneck at the root node we propose one scheme which simultaneously distributes many different tokens to many PEs at a time if many WMEs are available at the same time for distribution. WMEs that have i AVPs never match patterns that have j AVPs such that $i < j$. The network shown in Figure 1 is, therefore, modified to a network with multiple root nodes, as depicted in Figure 4. Whenever the new WMEs that are generated due to the rule firings become ready for distribution, PEs distribute WMEs based on the group numbers attached to the WMEs.

Assume that the two rules are compiled and allocated to the PEs according to the allocation policy described in section 3.3. Suppose further that a set of WMEs shown in section 2 is available and is about to be distributed into the network. If the Rete algorithm distributes one WME at a time to the network through the root node in Figure 1, it would take 6 time units to distribute them. Furthermore, a number of comparison tests which are performed at the very first one-input nodes (1, 4, 9, 13, and 16) will reach 36 (= 6 PEs × 6 WMEs). This is depicted in Figure 5(a), where one WME at a time is sequentially distributed to *all* PEs. For example, when WME_1 is distributed, all 6 PEs to which patterns are allocated make a comparison test simultaneously. Only two PEs, $PE_{3,0}$ and $PE_{3,1}$, will succeed in matching. This forces the machine to operate in Single-Instruction-stream-Multiple-Data-stream (SIMD) execution mode although it has a Multiple-Instruction-stream-Multiple-Data-stream (MIMD) processing capability.

Applying our distribution policy, the 6 WMEs are partitioned into 3 groups and the group numbers are assigned to WMEs. WMEs 3 and 6 get group #2 while 1 and 2 get #3 and 4 and 5 get #4. The total number of comparison tests performed at the very first one-input nodes in three sequences reduces to 12 (= 2×2 + 3×2 + 1×2), as shown in Figure 6(b). There are three bins in Figure 6(b), where each bin corresponds to a group. In each group, WMEs are sequentially distributed to PEs belonging to the corresponding group in the PE space. However, between groups WMEs are simultaneously distributed. The speed-up for the distribution policy would then be 36/12=3 for the given set of WMEs. The number of groups in WMEs determines the speed-up. In the worst case, only one WME can be distributed to all PEs at a time as shown in Figure 6(a). Note that in the original Rete algorithm, a sequential distribution, analogous to our worst case, would be implemented. Instead, our improvement provides the extra parallelism although this scheme depends heavily on the fact that WMEs will be evenly classified to all groups.

4. Simulation and performance evaluation

4.1. Simulation

In this simulation, various WMEs and rules are used. First, one-input nodes and array operations are tested by 1 PE. Simulation results show that a sequence of one-input nodes takes about 15 simulation time units. Each additional matching takes 13 time units. Second, three conditions of the Rule 2 are tested separately one at a time with various WMEs. Simulation results indicate that WME 1 matches against WME 6 of RM_{15} in 76 time units. Third, two patterns are executed in parallel by two PEs and take about 200-500 time units to match WMEs depending upon the number of WMEs that have reached either LM_{12} or RM_{15}.

4.2. Performance evaluation

The following assumptions are made in the simulation: No tokens wait for their partner for more than 1 time unit. The routing time for a token to reach any PE is set to 1 time unit. Each PE can execute 10 comparison tests at a time. On the average, there are 3 patterns per rule. One simulation time unit is set to 1μsec. With the simulation results and assumptions listed above we identify the following results:

1. T_o, the time units for a PE to process one-input nodes and variable bindings with 1 WME, < 20.

2. T_{t1}, the time units for a PE to process a two-input

node with one WME, < 100,

3. T_{t2}, the time units for 2 PEs to process a two-input node with various WMEs, < 125.
4. T_n, the time units for 2 PEs to process a negated two-input node, < 300.
5. T_r, the time units to instantiate a rule that has 1 regular and 1 negated two-input nodes, $= T_t + T_n \approx 400$.

Suppose that a certain production system has rules with average number of two inter-element features (1 two-input node and 1 negated two-input node) per rule and that there is only one WME matched through the one-input nodes and stored in each memory. The data-flow model would instantiate a rule in 400 simulation time units, which is equivalent to 0.4 msec. If there are more than 1 WME matched through one-input nodes and stored in each memory, T_r, the time taken to instantiate a rule will be proportional to the number of WMEs stored in each memory, as verified by our simulation results. When there are on the average n WMEs in each memory, $T_r \approx 400n = 0.4n$ msec in the absence of conflict resolution.

When the conflict resolution step (10% of total computation time [Fo82]) is taken into account, $T_r = 0.4(1 + 10/90)n \approx 0.5n$ msec, where n is an average number of WMEs stored in a memory. This T_r in turn gives $1000/0.5n = 2000/n$ rule firings/second. Compared to the analysis of the implementation of OPS5 onto DADO [Gu84], the choice of a data-flow multiprocessor gives a $2000/100n = 20/n$ fold in speed-up since DADO is estimated to be able to fire below 100 rules/second.

5. Conclusion

In this paper, we have explored the potential of data-flow multiprocessor systems for the efficient implementation of symbolic computations. Among the various data-flow architectures proposed, The MIT Tagged Token Data-flow Machine has been chosen for our simulation model. As a benchmark of symbolic computations, the Rete match algorithm has been chosen.

Inefficiencies in the implementation of the Rete algorithm on parallel machines have been identified and possible solutions to the problems have been worked out in our data-flow environment. Simultaneous distribution of many WMEs to many PEs has proven effective in delivering the parallelism inherent to the Rete algorithm and allowed by a given configuration of our data-flow architecture. Allocating conditions to different PEs, we have completely distributed $O(n)$ iterations throughout the system.

The Rete algorithm has been successfully implemented into a data-flow processing environment. The results we obtained reveal that symbolic computations on a data-flow multiprocessor computer can indeed be processed efficiently. Comparison with conventional computers has shown that a high speed-up could be obtained from this approach. However, some problems in applying data-flow principles of execution remain unsolved. One of the problems is the programmability in high-level language. Also, a complete implementation of the conflict resolution step will be next undertaken. In conclusion, it appears that the data-flow principles of execution are not limited to numerical processing but will also find applications in some AI problems.

References

[AI83] Arvind, Iannucci, R.A., "Two fundamental issues in multiprocessing: the data-flow solutions," MIT Laboratory for Computer Science, MIT/LCS/TM-241, September 1983.

[Ba78] Backus, J., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM 21*, 8 (Aug. 1978), pp.613-641.

[BF85] Brownston, L., Farrell, R., Kant, E., Martin, N., "Programming Expert Systems in OPS5," Addison-Wesley Publishing Company, 1985.

[Bi85] Bic, L., "Processing of Semantic Nets on Data-flow Architecture," in *Artificial Intelligence 27*, 1985, pp.219-227.

[Fo82] Forgy, C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," in *Artificial Intelligence 19*, September 1982, pp.17-37.

[Ga87] Gaudiot, J.L., "Data-driven multicomputers in digital signal processing applications," in *Proceedings of the IEEE*, September 1987.

[Gu84] Gupta, A., "Implementing OPS5 Production Systems on DADO," in *Proc. IEEE International Conference on Parallel Processing*, August 1984, pp.83-91.

[St87] Stolfo, S.J., "Initial Performance of the DADO2 Prototype," in *IEEE Computer*, January 1987, pp.75-83.

[SM84] Stolfo, S.J., Miranker, D.P., "DADO: A Parallel Processor for Expert Systems," in *Proc. IEEE International Conference on Parallel Processing*, August 1984, pp.74-82.

[TM85] Tenorio, M.F.M., Moldovan, D.I., "Mapping Production Systems into Multiprocessors," in *Proc. IEEE International Conference on Parallel Processing*, August 1985, pp.56-62.

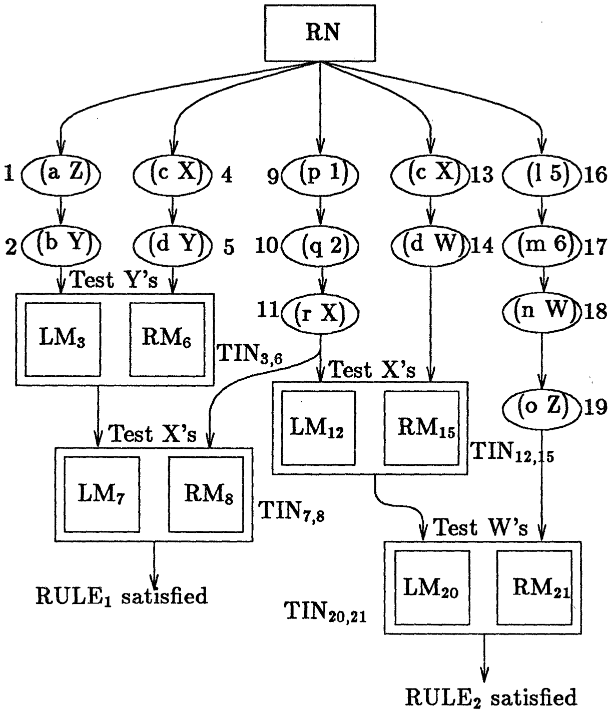


Figure 1: The condition-dependency network.

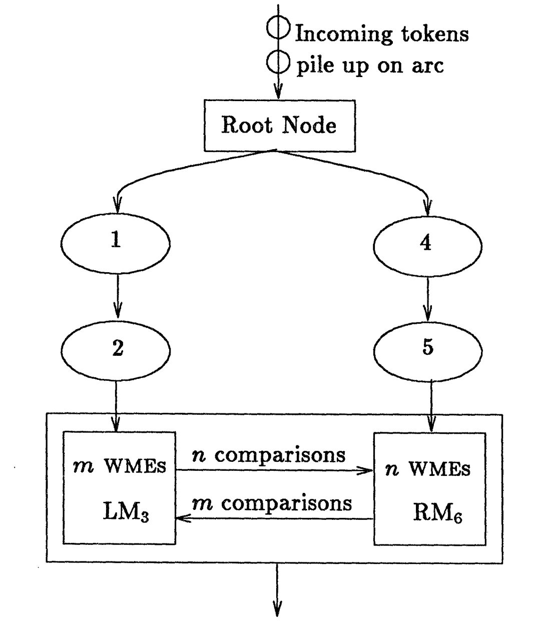


Figure 2: Two bottlenecks in the implementation.

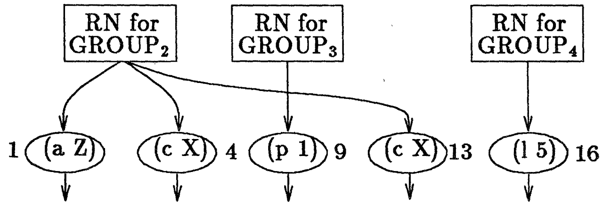
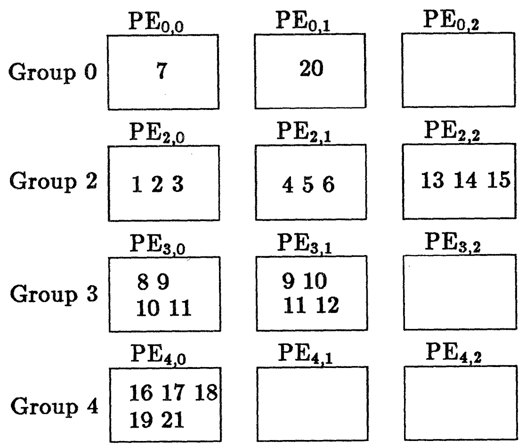


Figure 4: A modified condition-dependency network.



	$PE_{0,0}$	$PE_{0,1}$	$PE_{0,2}$
Group 0	7	20	
	$PE_{2,0}$	$PE_{2,1}$	$PE_{2,2}$
Group 2	1 2 3	4 5 6	13 14 15
	$PE_{3,0}$	$PE_{3,1}$	$PE_{3,2}$
Group 3	8 9 10 11	9 10 11 12	
	$PE_{4,0}$	$PE_{4,1}$	$PE_{4,2}$
Group 4	16 17 18 19 21		

Figure 3: A simple redundant allocation policy.

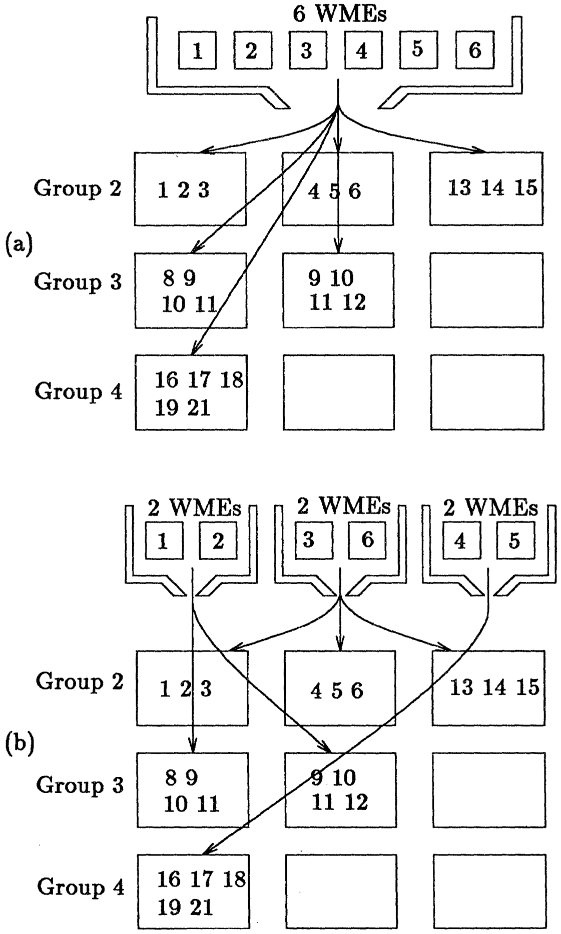


Figure 5: (a)Sequential- (b)Parallel distribution of WMEs.

ON MEASURING THE PERFORMANCE OF A MASSIVELY PARALLEL PROCESSOR*

Anthony P. Reeves
Department of Computer Science
University of Illinois at Urbana–Champaign
Urbana, Illinois 61801

Maria Gutierrez
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Abstract

An important performance characteristic of a parallel processor is its ability to implement data permutations; this is especially true for massively parallel processors which have restricted interconnection networks. Efficient programming of a massively parallel processor requires a non–conventional programming language. The overhead incurred when using a high–level programming language is also an important performance issue. The performance of a number of fundamental algorithms which have been implemented on NASA's Massively Parallel Processor is presented and the data permutation capability of the MPP is examined. These algorithms include: data permutations, the FFT, convolution, and arbitrary data mappings. The MPP is programmed in the high level language Parallel Pascal and the impact of using a simple implementation of this language is estimated.

1. Introduction

In order to analyze the performance of a massively parallel processor system it is necessary to characterize its data permutation ability. In this paper a characterization scheme is proposed which involves measuring the ability to perform a set of regular permutations. These permutations occur in many scientific problems and knowledge of their performance may also be useful in guiding a programmer to develop efficient programs. The performance of these permutations on NASA's Massively Parallel Processor (MPP) is presented and the overhead due to their implementation in a high level language is also given.

Massively Parallel systems are suitable for the large class of scientific applications which involve regular operations on large data arrays. The main interest is in evaluating the systems performance for these well matched applications. Many of these applications involve matrix operations such as the Fast Fourier Transform (FFT) and matrix convolution; the performance of both of these operations on the MPP is considered in detail.

In the remainder of this section the MPP is described and the performance of its primitive operations is presented. In section two the performance of the MPP for a number of important data permutations is considered in detail. The performance of the MPP for matrix convolution is considered in section three and the FFT is considered in section four. The convolution operation can be analyzed by means of the primitives presented in section one while analysis of the FFT requires knowledge of the data permutations presented in section two. Finally, in section five, a heuristic data mapping algorithm is considered which is data dependent and requires information to be acquired form the processor array in order to determine the instruction sequence.

1.1. The Massively Parallel Processor

The Massively Parallel Processor consists of 16384 bit–serial Processing Elements (PE's) connected in 128 x 128 mesh [1]. That is, each PE is connected to its 4 adjacent neighbors in a planar matrix. The two dimensional grid is one of the simplest interconnection topologies to implement, since the PE's themselves are set out in a planar grid fashion and all interconnections are between adjacent components.

The PE's are bit–serial, i.e. the data paths are all one bit wide. This organization offers the maximum flexibility, at the expense of the highest degree of parallelism, with the minimum number of control lines. The minimal architecture of the MPP is of particular interest to study, since any architecture modifications to improve performance would result in a more complex PE or a more dense interconnection strategy.

The MPP Processing Element

The MPP processing element is shown in Figure 1. All data paths are one bit wide and there are 8 PE's on a single CMOS chip with the local memory on external memory chips. Except for the shift register, the design is essentially a minimal architecture of this type. The single bit full adder is used for arithmetic operations and the Boolean processor, which implements all 16 possible two input logical functions, is used for all other operations. The NN select unit is the interface to the interprocessor network and is used to select a value from one of the four adjacent PE's in the mesh.

The S register is used for I/O. A bitplane is slid into the S registers independent of the PE processing operation and it is then loaded into the local memory by cycle stealing one cycle. The G register is used in masked operations; when masking is enabled only PE's in which the G register is set perform any operations. Not shown in Figure 1. is an OR bus output from the PE. All these outputs are connected (ORed) together so that the control unit can determine if any bits are set in a bitplane in a single instruction. On the MPP the local memory has 1024 words (bits) and is implemented with bipolar chips which have a 35 ns access time. The clock cycle time is 100 ns which is sufficient for a memory access and an ALU operation.

The main novel feature of the MPP PE architecture is the reconfigurable shift register. It speeds up integer multiplication by a factor of two and also has an important effect on floating–point performance.

*This work was funded in part by NASA Grant 5–403

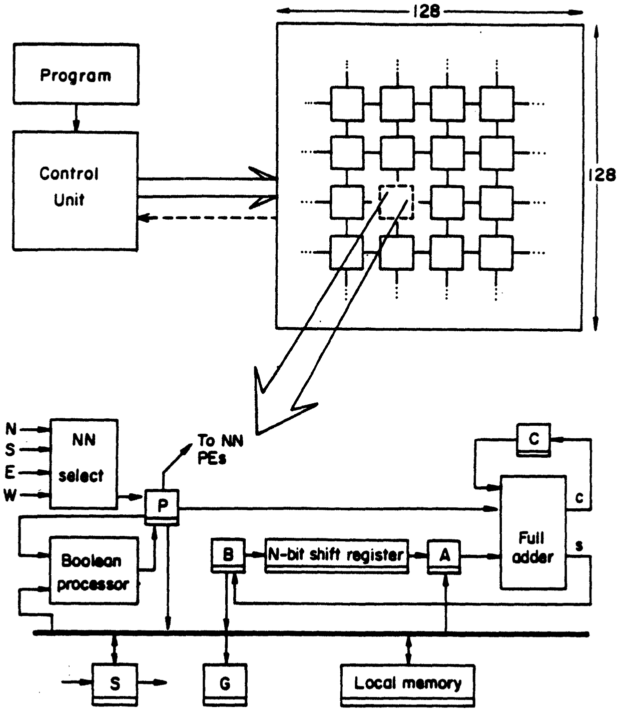


Figure 1. The MPP Processing Element

Array Edge Connections

The interprocessor connections at the edge of the processor array may either be connected to zero or to the opposite edge of the array. With the latter option rotation permutations can be directly implemented. A third option is to connect the opposite horizontal edges displaced by one bit position. With this option the array is connected in a spiral by the horizontal connections and can be treated like a one-dimensional vector of 16384 elements.

The MPP Control Unit

A number of processors are used to control the MPP processor array; their organization is shown in Figure 2. The concept is to always provide the array with data and instructions on every clock cycle. The host computer is a VAX 11/780; this is the most convenient level for the user to interact since it provides a conventional environment with direct connection to terminals and other standard peripherals. The user usually controls the MPP by developing a complete subroutine which is down loaded from the VAX to the main control unit (MCU) where it is executed. The MCU is a high speed 16-bit minicomputer which has direct access to the microprogrammed processor array control unit (PCU). It communicates to the PCU by means of macro instructions of the form "add array A to array B". The PCU contains runtime microcode to implement such operations without missing any clock cycles. A first in-first out (FIFO) buffer is used to connect the MCU to the PCU so that the next macro operation generation in the MCU can be overlapped with the execution in the PCU. A separate I/O control unit (IOCU) is used to control input and output operations to the processor array. It controls the swapping of bitplanes between the processor array and the staging memory independent of the array processing activity. Processing is only halted for one cycle in order to load or store a bitplane.

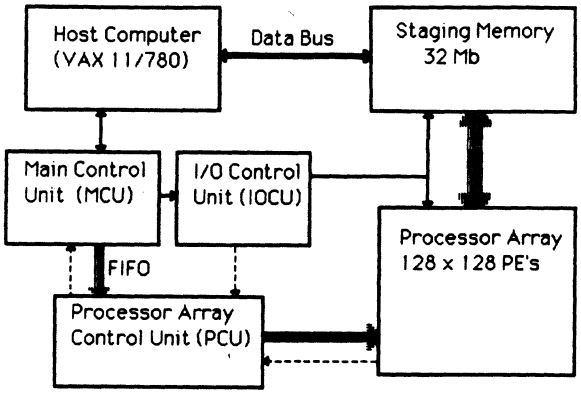

Host Computer (VAX 11/780)
Data Bus
Staging Memory 32 Mb
Main Control Unit (MCU)
I/O Control Unit (IOCU)
Processor Array 128 x 128 PE's
FIFO
Processor Array Control Unit (PCU)

Figure 2. The System Organization of the MPP

The staging memory is a large data store which is used as a data interface between peripheral devices and the processor array; it provides two main functions. First, it performs efficient data format conversion between the data element stream which is most commonly used for storing array data to the bitplane format used by the MPP. Second, it provides space to store large data structures which are too large for the processor array local memory.

1.2. Parallel Pascal

Parallel Pascal is an extended version of the Pascal programming language which is designed for the convenient and efficient programming of parallel computers. It is the first high level programming language to be implemented on the MPP. Parallel Pascal was designed with the MPP as the initial target architecture; however, it is also suitable for a large range of other parallel processors. A more detailed discussion of the language design is given in [2].

In Parallel Pascal all conventional expressions are extended to array data types. There are three fundamental classes of operations on array data which are are primitives on processor arrays but which are not available in conventional programming languages, these are: data reduction, data permutation and data broadcast. These operations have been included as primitives in Parallel Pascal. Mechanisms for the selection of subarrays and for selective operations on a subset of elements are also important language features.

The Parallel Pascal compiler generates a parallel P-code [3]. A code generator has been developed for NASA which generates MCU assembly language for procedures which are to directly use the MPP. Runtime support is provided in both the MCU and the PCU. No code for the PCU is directly generated by the compiler. Also, the code generator does not have a conventional optimization stage.

1.3. Performance Measurements

The execution times of several primitive operations implemented in Parallel Pascal were measured using program loops and timing routines. For example, to measure the time for a parallel array multiplication the following program segments were timed:

```
var a,b: Parallel array [1..128,1..128] of real;

Program segment #1:
For i := 1 to 1000 do
begin
a := b
end;

Program segment #2:
For i := 1 to 1000 do
begin
a := a * b
end;
```

The time required to execute the first program segment was 6.60 msec, and the time to execute the second program segment was 87.7 msec. The difference between these time measurements is due to the 1000 array multiply operations. Therefore, the time required for a parallel multiplication is given by:

$$t_{rm} = \frac{t_{ps2} - t_{psr1}}{1000}$$

where the t_{rm} is the time for a real multiplication, and t_{ps1} and t_{ps2} are the measured times for program segment #1 and program segment #2 respectively.

1.4. MPP Primitive Operations

The cost of the basic primitive parallel operations of the MPP, when programmed in Parallel Pascal, were measured using the procedure outlined above. When calculating the execution times of operations on 8-bit integers or Boolean data types, the time to execute program segment #1 is equal to 6.48 msec. The measured operation costs are presented in Table 1. Optimal times for these operations were estimated for the processor array by itself; these are also presented in Table 1. Optimal floating point arithmetic times were obtained from [1], and the remaining optimal times were derived by counting the clock cycles for optimal microcode instruction sequences applied to the PE array hardware. The difference between the measured and optimal times is due to three main factors: (a) MCU overhead, (b) overhead introduced by the Parallel Pascal compiler, and (c) overhead in the PCU microcode.

From Table 1. we can see that Boolean operations are the least efficiently implemented. The MCU adds an overhead of 3 or more μsec. per operation, which in the case of Boolean operations, dominates the execution times and causes an order of magnitude in loss of performance. On average, the Boolean measured times are about 20 times slower than the corresponding optimal times. For floating point operations, the 3 or more μsec. overhead is negligible since the execution times of the operations are on the order of tens or hundreds of μsecs. The floating-point add operation required about twice the stated optimal time. We do not know the reason for this; one possibility is that the run time implementation is significantly different from that used by Batcher in [1]. It is important to note that the implementation of floating-point operations is programmable since the PE's are bit serial. For example, Batcher used the IBM format while the current MPP run-time system implements the VAX floating-point format. Finally, it was noted that shift operations are least efficient when either the number of bits to be shifted is small or the shift distance is small.

Table 1. Optimal and Measured execution times of some typical operations.

Operation	Optimal time	Measured time
assignment r	6.4	6.6
assignment i8	1.6	1.8
assignment b	0.2	1.6
add r	38.1	75.8
mult r	75.8	81.1
mult r × s	43.9	87.7
sin r	–	334.
add i8	2.5	4.0
mult i8	8.8	9.0
mult i8 × s	7.0	7.5
div i8	17.6	24.1
mod i8	17.6	24.0
trunc	38.1	145.
round	38.1	145.
and	0.3	13.7
or	0.3	13.8
not	0.3	3.4
odd(i8)	0.5	2.9
any	0.5	2.4
min(r)	32.0	71.5
max(r)	32.0	71.2
min(i8)	8.0	33.1
max(i8)	8.0	33.1
compare i8	2.5	3.9
where b	0.1	3.1
shift(r,0,1)	3.2	9.9
shift(r,0,64)	205.	212.
shift(r,64,64)	410.	505.
shift(i8,0,1)	0.8	4.3
shift(i8,0,64)	51.2	62.9
shift(i8,64,64)	102.	126.
shift(b,0,1)	0.1	4.3
shift(b,0,64)	6.4	7.6
shift(b,64,64)	12.8	14.5
procedure call	–	150.(*)

time in μsec.
s = scalar, r = array of real, i8 = array of 8-bit integer, b = array of Boolean.
(*) The measured time for a procedure call varies according to the variables passed on the call.

1.5. Optimal Performance Estimation

In the following, when an algorithm is presented, the corresponding measured and estimated execution times will be given. The measured times were obtained using the timing functions of the MPP as previously discussed for primitive operations. The estimated optimal times were calculated by tracing the Parallel Pascal code of the algorithm and adding the optimal execution times, given in Table 1, of the encountered array instructions. Any scalar operations, executed by the MCU, were assumed to be concurrent with the execution of the array operations, and thus, were not taken into account. As an example, the following is a program segment of the *shuffle* permutation, for which the calculation of the estimated execution time is shown.

```
x := 1; y := 0                                          -1-
tmx1 := mx;                                             -2-
tmx2 := mx;                                             -3-

num := 2;                                               -4-
while num < tn do                                       -5-
 begin
   tmx1 := shift(tmx1, -x, -y); (* shift down *)        -6-
   where id = num do                                    -7-
      mx := tmx1;                                       -8-
   num := num + 2;                                      -9-
 end;
```

where x and y are integers, and *num* and *tn* are 8-bit integers ($tn = 128$); *id* is a parallel array of 8-bit integer; *mx, tmx1,* and *tmx2* are parallel arrays of either 32-bit reals, 8-bit integers, or Boolean. Let the number of bits in the data elements beN_b .

The optimal execution time is equal to the sum of:

- 2 assign array of $N_b = 2 \times (0.2 \times N_b)$ (-2- and -3-)

the following are executed $\frac{tn}{2} - 1$ (i.e. 63) times:

- shift array of N_b by $1 = 0.3 \times N_b$ (-6-)
- where b + compare i8 = 0.1 + 2.5 (-7-)
- assign array of $N_b = 0.2 \times N_b$ (-8-)

Note that an assignment operation takes two cycles per bit, and a shift operation takes per bit two cycles (for load and store) plus the shift distance. The total optimal time is equal to $31.9 \times N_b + 163.8$ μsec. For example, for arrays of 8-bit integer where $N_b = 8$, the optimal time is 419.1 μsec.

1.6. The Transfer Ratio

A comparative measure, the *transfer ratio* [4], is used to express the cost of an algorithm. The *transfer ratio* is defined as the ratio of the time for the data transfer over the time for an elemental operation. The time for an elemental operation is defined as the average between the time of a multiplication and the time of an addition on the processor array. Based on the optimal execution times given in Table 1., the time for an elemental operation for 32-bit floating point elements is 57 μsec, and for 8-bit integer elements it is 5.65 μsec. For Boolean elements, the time for an elemental operation is taken to be two clock cycles i.e., 0.2 μsec [4].

2. Data Permutations

The performance of the MPP for a number of classical regular data permutations is considered here; however, the method presented is not limited to these permutations. A permutation function is performed on an ordered set of N elements, and it is defined by a one-to-one function $\pi(x)$ [5]. Both x and $\pi(x)$ are integers between 0 and N-1; x and $\pi(x)$ represent the addresses of the elements before and after the permutation, respectively.

2.1. The Shift Permutation

The near-neighbor interconnection network of the MPP can only directly implement the *shift* permutation. This permutation is possible because the PE array has, in addition to the near neighbor connections, toroidal end-around edge connections. Any other permutation can only be achieved through the *shift* permutation.

In Parallel Pascal, the *shift* permutation is specified with the built in function 'rotate'. Its arguments consist of the array to be shifted, and the amount and direction of the shift. For the MPP, the cost of a *shift* permutation (t_s) depends on the amount of the shift (d) and on the number of bits of the array elements (N_b). The optimal cost for the shift permutation is given by

$$t_s = (2 + d) * 0.1 * N_b \ \ \mu sec$$

2.2. The Test Permutations

The permutations which have been implemented for the MPP are *exchange* (ϵ), *shuffle* (σ), *butterfly* (β), and *bit reversal* (ρ). Where applicable, the respective *Sub* and *Super* permutations were also implemented. Conceptually, a *Sub* permutation involves a partitioning of the vector into groups of adjacent elements such that each group is mapped into itself only. A *Super* permutation involves partitioning the vector into groups of adjacent elements such that each group moves the same distance in the permutation. In general, the implementation of a *sub* or *super* permutation involves less work on the MPP than for the regular form of the permutation.

A permutation will be defined by considering the binary representation of **x**:

$$x = (b_n, b_{n-1}, \ldots, b_1)$$

$$x = b_n\, 2^{n-1} + b_{n-1}\, 2^{n-2} + \ldots + b_1$$

The above expressions represent the binary address of an element in $N = 2^n$, and a permutation is defined by the permutation on the bits of this address [5].

The Exchange Permutation

$$\epsilon_{(k)}(x) = (b_n, \ldots, \bar{b}_k, \ldots, b_1) \quad where \quad 1 \le k \le n$$

The *exchange* permutation consists of complementing bit k of the input address. Thus, this permutation consists of exchanging every pair of elements, where two elements form a pair if their addresses are the same except for the kth bit.

In the *exchange* permutation all the elements move an equal distance, except that half of the elements move in one direction and the other half move in the opposite direction. This procedure is accomplished in two steps:

1 – Calculate the amount of the shift which is equal to 2^{k-1}

2 – Perform the exchange of pairs: where $b_k = 1$ then shift up (or left), and where $b_k = 0$ then shift down (or right).

The Shuffle Permutation

$$\sigma(x) = (b_{n-1}, b_{n-2}, \ldots, b_1, b_n)$$

The *shuffle* permutation consists of a circular left shift of the bits of the input address. The resulting permutation consists of splitting in half the set of N elements, and then interleaving them like in a perfect card shuffle.

The algorithm for the shuffle permutation is as follows:

1 – Map the upper half of the input matrix by shifting the elements down a total of $\frac{N}{2} - 1$ steps. For each shift down, one element will be located in the correct position, and therefore stored in the result array.

2 – Map the lower half of the input matrix. The same procedure of step **1** is followed except that the elements are shifted up.

The *Sub–Shuffle* permutation is specified by:

$$\sigma_{(k)}(x) = (b_n, \ldots, b_{k+1}, b_{k-1}, \ldots, b_1, b_k)$$

In the *sub-shuffle* permutation, the set of elements is divided into 2^{n-k} groups, each one of size 2^k, and a perfect shuffle is performed on each of the subgroups. The *sub-shuffle* algorithm is the same as for the *shuffle*, except that there are 2^{n-k} halves, and total amount of the shift is $2^{k-1} - 1$.

The *Super–Shuffle* permutation is specified by:

$$\sigma^{(k)}(x) = (b_{n-1}, \ldots, b_{n-k+1}, b_n, b_{n-k}, \ldots, b_1)$$

The *super-shuffle* permutation, performs a perfect shuffle on the whole set, except that now an 'element' consists of a group of 2^{n-k} elements. The *super-shuffle* algorithm is similar to the *shuffle* algorithm, except that instead of shifting by one, a shift by 2^{n-k} is performed at each step.

The Butterfly Permutation

$$\beta(x) = (b_1, b_{n-1}, \ldots, b_2, b_n)$$

The *butterfly* permutation consists of exchanging the most significant bit (MSB) and the least significant bit (LSB). Three cases arise from this exchange: First, if the bits are equal (i.e. both are equal to 1 or to 0), the permuted addresses are unchanged, and therefore the corresponding elements remain in their initial positions. Second, if MSB = 1 and LSB =0, the corresponding elements have to move up $2^{n-1} - 1$ locations away. Third, if the MSB = 0 and LSB =1, the corresponding elements have to move down $2^{n-1} - 1$ locations away.

Similarly to the *butterfly*, the *sub-butterfly* permutation consists of exchanging bit **k** (MSB) and bit **1** (LSB); and the *super-butterfly* permutation consists of exchanging bit **n** (MSB) and bit **n–k+1** (LSB); The binary representations of these two permutations are:

$$\beta_{(k)}(x) = (b_n, \ldots, b_{k+1}, b_{k-1}, b_1, \ldots, b_k)$$

$$\beta^{(k)}(x) = (b_{n-k+1}, \ldots, b_{n-k+2}, b_n, b_{n-k}, \ldots, b_1)$$

The butterfly algorithm consists of two steps:

1 – Generate two Boolean masks: 'shiftup' indicates the positions where the MSB of the address is equal to 0 and the LSB is equal to 1, and 'shiftdown' indicates the positions where the MSB is equal to 1 and the LSB is equal to 0. Nothing needs to be done where MSB = LSB.

2 – Perform the shifts: where 'shiftup' is true, the elements are obtained with a shift up by a distance of $2^{n-1} - 1$; where 'shiftdown' is true, the elements are obtained with a shift down by a distance of $2^{n-1} - 1$.

The same algorithm applies for the *sub* and *super butterfly* permutations, except that in the *sub-butterfly* a $2^{k-1} - 1$ shift is performed, and in the *super-butterfly* a $(2^{k-1} - 1)(2^{n-k})$ shift is performed.

The Bit Reversal Permutation

$$\rho(x) = (b_1, b_2, \ldots, b_{n-1}, b_n)$$

The *bit reversal* permutation consists of reversing the order of the bits of the input address. Similarly, the *sub-bit reversal* at bit k reverses the k least significant bits: bit **k** to bit **1**, and the *super-bit reversal* reverses the k most significant bits: bit **n** to bit **n–k+1**. The binary representations of these two permutations are:

$$\rho_{(k)}(x) = (b_n, \ldots, b_{k+1}, b_1, b_2, \ldots, b_k)$$

$$\rho^{(k)}(x) = (b_{n-k+1}, b_{n-k+2}, \ldots, b_n, b_{n-k}, \ldots, b_1)$$

The *bit reversal* permutation can be achieved with a series of bit exchanges between the pairs of corresponding MSB and LSB, i.e. between b_n and b_1, b_{n-1} and b_2, . . . , b_{n-m} and b_{m+1} where $m < \left\lfloor \frac{n}{2} \right\rfloor$. Therefore, a bit reversal can be realized by a series of *butterflies* where the distance of each shift is calculated from the bits position numbers. Exchanging MSB bit **j** with LSB bit **i**, requires a shift of length $2^{j-1} - 2^{i-1}$. The algorithm consists of $\left\lfloor \frac{n}{2} \right\rfloor$ iterations, where each iteration consists of:

1 – Determine which pair of bits is to be exchanged, and calculate the amount of the shift.

2 – Similarly to the *butterfly*, create the shifts masks, and then perform the shifts.

The same algorithm applies for the *sub* and *super bit reversal* permutations, except that the bit exchanges are performed only to the **k** least significant bits or the **k** most significant bits, respectively. In both cases, the number of iterations is equal to $\left\lfloor \frac{k}{2} \right\rfloor$.

2.3. Performance Evaluation

Each permutation was coded in Parallel Pascal and was run on the MPP using three types of data: 32-bit floating point, 8-bit integers, and Boolean. Program details and the timing results are given in [6]. These programs accept a parameter *k* and generate the masks for the permutations each time that they are called. Some of the overhead incurred by the initialization code could be avoided.

The results from the optimal time estimations are used first to characterize the performance of the MPP processor array hardware for data permutations, then these are combined with the measured results to characterize the efficiency of the Parallel Pascal compiler and the MPP control units.

Due to the orthogonality of the test permutations, the performance of the MPP can be characterized by considering only one dimension of the processor array. On the MPP each permutation is performed concurrently on each row of the processor array; i.e., 128 sets of 128 elements (the performance for permuting the columns is identical). Since all the permutations considered are orthogonal with respect to the two dimensions of the MPP mesh connections, these results may be simply extended to the case of permuting a 16384 element vector (or 128 x 128 matrix); the transfer ratio cost will be doubled and the compiler efficiency will remain the same.

For each permutation, expressions for the optimal execution times (to_π) for a one dimensional permutation on the MPP have been derived from the optimal execution times of the primitive operations given in Table 1. These execution times are optimal in the sense that they represent the cost of a direct translation of the highl level language program without any overhead from the program control unit; i.e. the processor array is doing useful work on every clock cycle. In some cases a faster realization could be achieved by careful programming at the microcode level. In Table 2. expressions are given for *sub*-permutation costs $to_{\pi,\,k}$ and *super*-permutation costs $to_{\pi,\,\tilde{k}}$, where k is an integer in the range of 1 to 7 which is related to the group size. In the time expressions,N_b represents the number of bits in the data elements: N_b is equal to 32 for floating point elements, to 8 for integer elements, and to 1 for Boolean elements. The time unit used in all expressions is μseconds.

Table 2. Optimal Permutation Costs for one dimension of the MPP.

π	Cost
$to_{\epsilon,\,k}$	$52 + N_b \; * \; (0.6 + 0.1 \; * \; 2^k)$
$to_{\sigma,\,k}$	$17.6 + 0.4 \; * \; N_b + (0.5 \; * \; N_b + 2.6) \; (2^k - 2)$
$to_{\sigma,\,\tilde{k}}$	$17.6 + 0.4 \; * \; N_b + [(12.8 \; * \; 2^{-k} + 0.4)N_b + 2.6](2^k - 2)$
$to_{\beta,\,k}$	$0.4 \; * \; N_b + 24.2 + 0.1 \; * \; N_b \; 2^k$
$to_{\beta,\,\tilde{k}}$	$13.4 \; * \; N_b + 24.2 - 25.6 \; * \; N_b \; * \; 2^{-k}$
$to_{\rho,\,k}$	$(113 + 0.6 \; * \; N_b) \left\lfloor \frac{k}{2} \right\rfloor + 0.2(2^k - 2^{\left\lceil \frac{k}{2} \right\rceil} - 2^{\left\lfloor \frac{k}{2} \right\rfloor} + 1) \; N_b$
$to_{\rho,\,\tilde{k}}$	$(113 + 0.6 \; * \; N_b) \left\lfloor \frac{k}{2} \right\rfloor + 25.6(1 - 2^{-\left\lceil \frac{k}{2} \right\rceil} - 2^{-\left\lfloor \frac{k}{2} \right\rfloor} + 2^{-k})N_b$

The permutation *transfer ratios* for the different data types are given in Figures 3–5. The horizontal axis spans the range of *sub* and *super* permutations with the usual permutation at k = 7 since, when N = 128, $\pi(x) = \pi_{(7)}(x) = \pi^{(7)}(x)$. For the exchange permutation, the kth *sub* and *super* permutations are considered to be identical. Note that the transfer ratio is plotted on a logarithmic scale and that the actual time in μsecs is also provided at the right side of each graph.

The transfer ratios are in the range 1–20 for floating point data, 6–200 for integer data and 100–2000 for Boolean data. Recall that the transfer ratio is the number of arithmetic operations of that data type which can be performed in the time of one permutation. Ideally, if the transfer ratios can be made less than 1 then permutations will not impact the performance of the system. In many cases the range for floating point numbers will not be a problem since a number of operations are typically performed between permutations in most algorithms. However, there is a significant difference in performance between the shuffle and bit reverse permutations compared to the others and these permutations should be avoided if possible on the MPP.

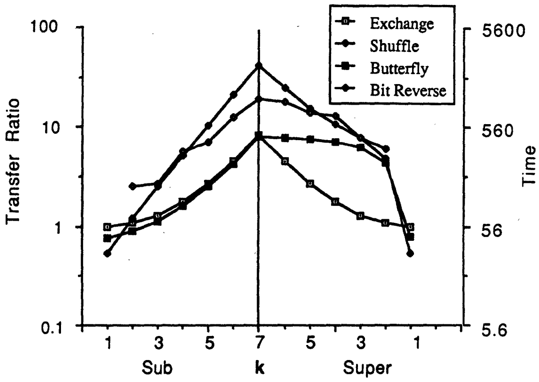


Figure 3. Transfer Ratios for Floating Point data Permutations

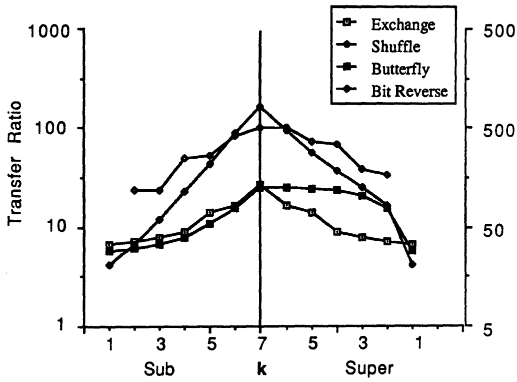


Figure 4. Transfer Ratios for 8-bit Integer Permutations

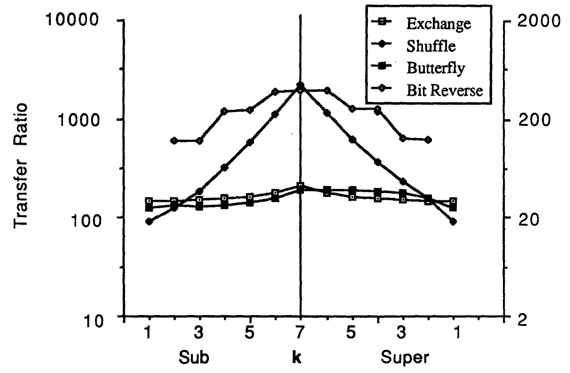


Figure 5. Transfer Ratios for Boolean data Permutations

For the integer and Boolean data these permutations may incur a significant overhead unless they occur very infrequently in the algorithm. For small data sizes the additional operations necessary to implement the permutation dominate the permutation cost especially for the shuffle and bit reverse permutations. Since the performance of the permutations differs by more than an order of magnitude; the careful selection of the most appropriate permutation for a task is even more important than for the floating point case.

The efficiency of the programming language and control units for a permutation is expressed by the ratio of the optimal time to the measured time. The percentage efficiency of the Parallel Pascal implemented permutations is shown for the different data types in Figures 6–8. The efficiency for the permutations ranges from 30–75 percent for floating point data, 20–50 percent for integer data and 10–28 percent for Boolean data.

The thee main potential causes for inefficiency are the lack of code optimization, MCU overhead, and the loss of useful cycles in the PCU microcode. The first inefficiency causes the processor array to do extra useless work such as copying arrays to temporary buffers, the second occurs when the MCU has too many operations to perform between MPP macro operations such that the FIFO buffer empties and the PCU waits idle for the next operation, the last case may occur when the microcede architecture is unable to perform all necessary operations in a single 100 ns cycle such that the array must miss a useful processing cycle.

It is difficult to ascertain the exact cause of inefficiency from Figures 6–8.; however, it is possible to determine the benefit which could be achieved with an optimally coded control system. The implementation of floating point data permutations is already reasonably efficient in most cases while the implementation of the Boolean data types is not very efficient. The results suggest that the main loss of efficiency is caused by the MCU overhead. The code generated by the Parallel Pascal Compiler is of a similar efficiency for each data type; furthermore, it is reasonable to assume that the PCU unit can efficiently manage single bitplane data. On the other hand, Boolean data places the largest load on the MCU since it must generate macro instructions for the PCU at a much higher rate than for other data types.

3. Convolution

Convolution is an important operation in many signal processing and image processing applications. A two–dimensional convolution involves *convolving* a large data matrix D with a given small matrix W, the *convolution kernel*, as specified by

$$R[i,j] = \sum_{x=-m}^{m} \sum_{y=-m}^{m} W[x,y] * D[i+x, j+y]$$

where W is of size $(2m + 1) \times (2m + 1)$.

Conceptually, the convolution result for a given element $R[i,j]$ is obtained by superposing the W kernel onto the matrix (with the center of W at position i,j), and multiplying each kernel element with the corresponding matrix element. The convolution result for i,j is then equal to the summation of these products.

On the MPP, D is distributed on the processor array and a convolution operation is implemented by a series of shift–multiply–add operations (one for each element of the kernel). The performance, on the MPP, of a two–dimensional convolution operation involving a 5X5 kernel was examined. The measured execution time for matrices with 8–bit integer elements was 987 µsec, and for matrices with 32–bit floating point elements was 5.28 msec. This is a processing rate of 830 MOPS for 8–bit integer data and 155 MFLOPS for 32–bit floating point data. The time required for just the arithmetic computations is 325 µsec (33% of the total time) for 8–bit integer data and 3.92 msec (74% of the total time) for 32–bit floating point data.

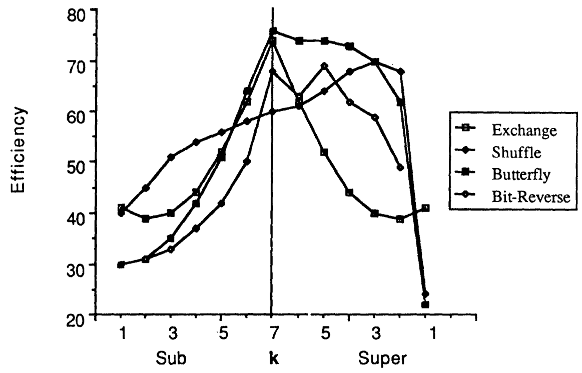


Figure 6. Efficiency of Floating Point data Permutations

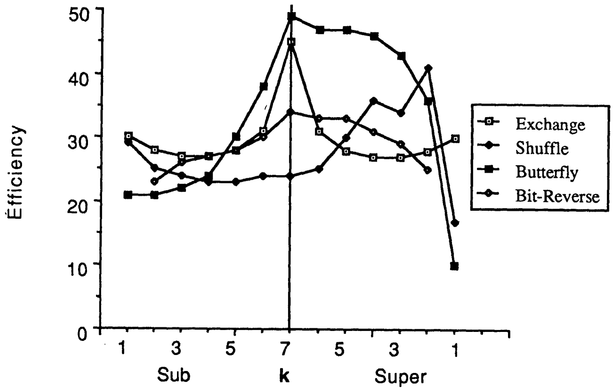


Figure 7. Efficiency of 8-bit Integer Permutations

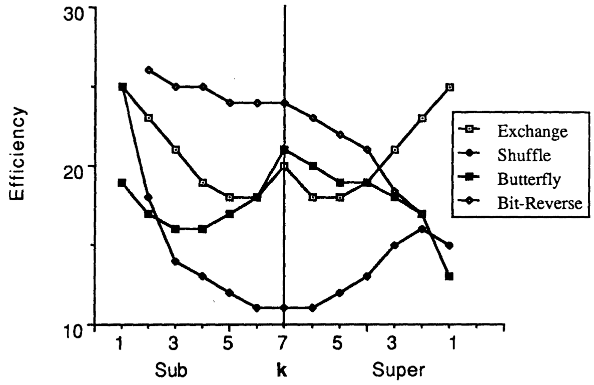


Figure 8. Efficiency of Boolean data Permutations

4. The Fast Fourier Transform

A Fast Fourier Transform (FFT) program, that involves the β, σ, and ρ permutations, was developed. The Fourier Transform is the frequency domain representation of a function and it is frequently used in several different scientific applications. The FFT is a fast method to compute the Discrete Fourier Transform (DFT), since it reduces the calculation of the FT from $O(n^2)$ for the DFT to $O(n \log_2 n)$. An N-point DFT is specified by:

$$X(n) = \sum_{k=0}^{N-1} x_o(k) * w^{nk}, \quad n = 1, 2, \ldots, N-1$$

where $w = e^{-j\frac{2\pi}{N}}$ and N is a power of 2 [7].

4.1. The FFT Algorithm

The input to the FFT program is an N×N array, and the FFT is performed concurrently on either the rows or the columns of the array, depending on the coordinate chosen. Consequently, a two dimensional FFT is achieved by performing an FFT on the columns and then on the rows, or vice versa.

In general, the algorithm used to perform a FFT to a given row of length N (or column) is defined as follows:
For a bit reversed result:

$$FFT_n = \beta_{(n)}\, W\, \beta_{(n-1)}\, W \ldots \beta_{(2)}\, W\, \sigma\, W \quad where\ N = 2^n$$

For a naturally-ordered result:

$$FFT_n = \beta_{(n)}\, W\, \beta_{(n-1)}\, W \ldots \beta_{(2)}\, W\, \sigma\, W\, \rho \quad where\ N = 2^n$$

β, σ, and ρ refer to the permutations presented in the previous section; **W** represents the following operation:

$$x_l = x_{l-1} + w^p * y_{l-1}$$

$$y_l = x_{l-1} - w^p * y_{l-1}$$

where $w = e^{-j\frac{2\pi}{N}}$, p represents the power of w, l represents the iteration number, and x and y are dual nodes ([7] p. 154).

This is only one formulation of the FFT algorithm; for example, an alternative well known FFT formulation is obtained if every β is replaced by a σ. The best permutation sequence to use depends upon the relative speeds of the regular permutations discussed in section 2. For the applications programmer, the performance of the permutations on the total system is required. The transfer ratios obtained from the time measurements, normalized by the attainable floating point arithmetic speed of 80 μs are shown in Figure 9. From this graph it can be seen that the butterfly permutation is an order of magnitude faster than the shuffle permutation and sub butterflys are faster still. Therefore, it will always be better to use butterfly permutations on the MPP rather than shuffles whenever possible. Furthermore, a reasonable computational performance may be anticipated since all butterfly permutations have a transfer ratio of less than 10 which is the order of the number of operations involved with each node pair calculation.

As mentioned previously, the FFT algorithm has complexity $O(n \log_2 n)$, but on the MPP, it takes $\log_2 n$ steps because there are N PE's for N elements, as opposed to one PE for N elements (N = 128). In the MPP, a step or iteration l consists of the following:

1 – Perform the $\beta_{(n-l+1)}$ permutation except when $l = n$, in this case a σ permutation is performed. At this point all the dual nodes pairs are located in adjacent PE's.

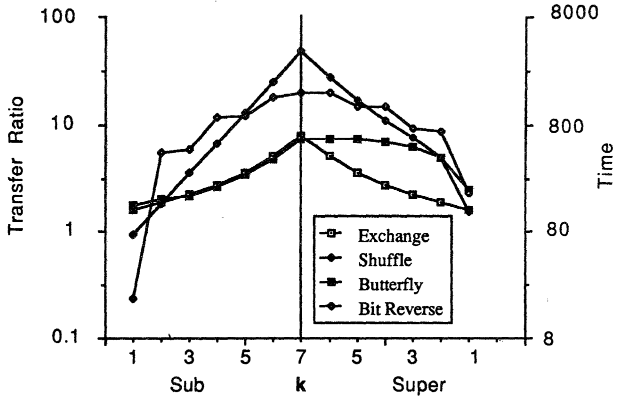


Figure 9. Measured Transfer Ratios for Floating Point data Permutations

Table 3. FFT operation Times

Operation	optimal time	measured time
$\beta_{(7)}$	851	993
$\beta_{(6)}$	451	617
$\beta_{(5)}$	252	445
$\beta_{(4)}$	154	379
$\beta_{(3)}$	107	771
$\beta_{(2)}$	85	362
σ	4390	7290
W	437	765
$pgen\ l = 2$	41.9	294
$pgen\ l = 3$	64.6	393
$pgen\ l = 4$	87.3	493
$pgen\ l = 5$	110.0	593
$pgen\ l = 6$	133.0	692
$pgen\ l = 7$	4.2	23
Total FFT	14820	22420

2 – Calculate the weights, w^p's. First the values of the p's are generated, these are dependent on l and on the corresponding array position. Then the real part of w is equal to $cos(\frac{2\pi}{N} p)$ and the imaginary part is equal to $-sin(\frac{2\pi}{N} p)$.

3 – Calculate x_l's and y_l's, i.e. perform the W operation presented above. Initially, the x_{l-1}'s are in the even numbered PE's and a copy is send to the corresponding dual odd numbered PE; the opposite is done to the y_{l-1}'s. Then the complex multiplication and addition are performed.

4.2. Performance Evaluation

An FFT_7 has been investigated on the MPP which is defined by:

$$FFT_7 = \beta_{(7)}\, W\, \beta_{(6)}\, W\, \beta_{(5)}\, W\, \beta_{(4)}\, W\, \beta_{(3)}\, W\, \beta_{(2)}\, W\, \sigma\, W$$

The times for the various operations of the FFT algorithm are given in Table 3. The execution times of these permutation functions are more than the execution times of the same permutations presented in section 2, because, in the FFT, the data elements used are complex numbers i.e. two 32-bit floating point elements. The execution time of the *W* operation is independent of *l*, the iteration number. The *pgen* function generates the *p* values and its execution time depends on the value of *l*.

The measured time for the FFT is greater than the estimated optimal time by a factor of 1.5. The execution times of the FFT's are dominated by arithmetic operations (*W* calculation) and by shifting operations (permutations); no reduction functions are used. The MCU overhead is likely to be minimum since most of the operations are array operations of 32-bit floating point elements; therefore, the differences between estimated and measured times are probably mainly due to the lack of code optimization and the inefficient implementation of Parallel Pascal primitive operations.

As a performance measure of the FFT, the number of MFLOPS was calculated. An operation is defined as either a floating point multiplication or an addition. It was determined that the *sine* and *cosine* operations, used in the calculation of the weight factors, have a measured execution time equal to 334 μsec, and thus, a sine or cosine operation on the MPP has a cost equivalent to four floating point multiplication operations.

In an FFT, two main calculations are performed at each iteration: first, the weights factors are generated, and second, the *W* operation is performed. In general, at each iteration 9 floating point operations per processor are necessary for the weights calculation and 8 are necessary for the *W* operation. There are seven iterations on 128^2 processors, and the FFT takes 0.0224 seconds. Therefore, the FFT calculation achieves an approximate rate of 87 MFLOPS.

In addition to the MFLOPS calculation, the percentage of time spent on shift operations and the percentage of time spend on arithmetic calculations were found. The shift operations consist of the *butterflies* and the *shuffle* permutations, and the arithmetic operations consists of the weight factors calculation (*pgen* and the sine and cosine operations), and of the *W* operation. The FFT program spends approximately 46% of the execution time on shift operations, and approximately 54% on arithmetic operations.

5. Arbitrary Data Mapping

A parallel data mapping for a two-dimensional matrix may be specified by two coordinate-index matrices. These are called the **r** matrix (for row index) and the **c** matrix (for column index). A mapping from an input matrix M to a result matrix R is specified by

$$R[i,j] = M\Big[r[i,j],\ c[i,j]\Big]$$

Two algorithms have been explored to perform this mapping function: a simple direct algorithm and a heuristic algorithm.

5.1. The Simple Algorithm

The simple algorithm requires every element of the input matrix to be passed by every position of the output matrix. When an element is located at the appropriate output position, that is, when it has moved the correct distance according to the **r** and **c** matrices, its value is then stored at this position.

```
For i := 1 to n do
  begin
    For j := 1 to n do
     begin
       where r = i and c = j do
         R := M;
       M := rotate(M, 0, 1);
     end;
   M := rotate(M, 1, 0);
  end;
```

The cost complexity of this algorithm is $O(n^2)$ where n is equal to the number of rows (or columns) in the matrix. This algorithm always requires n^2 iterations and a total of n^2+n data rotations.

5.2. The Heuristic Algorithm

A heuristic algorithm [8] has been developed for the MPP which takes advantage of uniformity or locality existing in the movement of the data elements. Its performance depends upon the available locality in the specific mapping operation. The general concept is that the matrix M is only shifted to positions where data elements are to be mapped to R. In order to know how far M can be moved in one step it is necessary to find the shortest distance to the next needed displacement. This is achieved by *min* reduction functions applied to relative versions of the c and r matrices. Therefore, the number of iterations is reduced but each iteration is more complex since a calculation involving two or more reductions is needed to determine the displacement to the next iteration.

The worst case cost of the heuristic algorithm is O(n^2), as this algorithm may conceivably require up to n^2 iterations; however, this has never been observed in practice.

5.3. Performance Evaluation

The performance results obtained from applying the heuristic algorithm to a set of six different matrix rotation mappings are shown in Table 4. The performance of the simple algorithm (which is the same for all data mappings) is shown in the last row of Table 4. The first two columns give the number of iterations and bit-shift operations required for each mapping. The next column gives the number of reductions required by the algorithm The last four columns give the estimated and measured timing results for 8 and 32-bit data; all times in this table are in milliseconds.

The results show that, even with the more complex iterations, the heuristic algorithm is significantly faster than the simple algorithm for most of the given data mappings when the optimal times are considered. However, in practice, the simple algorithm is usually faster. The measured results for the simple algorithm were about 4 times longer than the optimal results for 8-bit data and 2 times longer for 32-bit data which is comparable to previous results. For the heuristic algorithm the corresponding figures are about 12 for 8-bit data 8 for 32-bit data.

Three main factors which contribute to the unusual behavior of the heuristic algorithm are as follows. First, the implementation of the primitive reduction functions is not very efficient (see Table 1.). The implemented functions are more

Table 4. Cost of the Heuristic Algorithm for Matrix Rotation

map θ	iterations	rotations	reductions	to (8–bit)	tm (8–bit)	to (32–bit)	tm (32–bit)
map15	910	1451	2666	50.47	577.4	75.19	579.0
map30	2619	3359	7759	146.2	1684.0	216.2	1687.0
map45	4069	5838	12098	228.4	2628.0	339.1	2634.0
map60	3882	7975	11553	220.1	2511.0	331.5	2523.0
map75	2232	9029	6659	130.4	1451.0	205.3	1468.0
map90	128	8129	382	13.62	87.99	36.19	109.1
simple	16384	16512	0	153	600	350	685

than 4 times slower than the optimal case and 44% of the estimated execution time of the heuristic algorithm is accounted for by these reductions. The simple algorithm does not involve any reductions. Second, after a reduction function is executed the result is used in a conditional branch statement in the MCU. This causes the FIFO buffer to empty and the MCU must do additional work before the next macro instruction can be generated. Third, 20% of the optimal estimated time is accounted for by Boolean array operations; Boolean operations are the least efficiently implemented when controlled from the MCU. On average Boolean operations are about 20 times slower than the optimal case. The simple algorithm does not involve any Boolean operations.

Conclusion

The capability of the MPP to perform a set of regular permutations has been studied in detail. The results indicate that the optimal implementation times for floating point data transfers may be reasonable for many applications but the permutation of small length data may not be very efficient. The performance of the total system for permutations is also quite good for floating point data but significant savings might be made if the shorter data type permutations were reprogrammed in PCU microcode. The analysis techniques presented could be applied to other highly parallel architectures.

Several characteristic algorithms have been considered: convolution which is implemented with a few primitive operations, the FFT which involves significant data permutations, and the heuristic algorithm which has a data dependent behavior. In general, the Parallel Pascal Compiler performed well for large data types and deterministic algorithms (which provided the lightest load for the MCU). It did not perform as well for complex algorithms involving Boolean data or reduction functions; however, in this case it was still quite adequate for algorithm prototyping. It is not clear that an optimizing compiler would be very much faster, for the difficult algorithms, unless it generated PCU microcode for critical sections.

In terms of processing speed, using Batchers figures [1] the peak performance of the MPP is 288 MFLOPS; from our primitive operation measurements the fastest rate we could expect to attain is 210 MFLOPS (due to the slower add time). For the convolution algorithm a rate of 155 MFLOPS was attained and for a 128 x 128 FFT the rate was 87 MFLOPS. These algorithms were conveniently programmed in Parallel Pascal. Furthermore, 128 x 128 is the worst case size for the FFT implementation on the MPP; for either larger or smaller matrix sizes the comparative overhead due to interprocessor communication would be less.

References

1. K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers* **C–29**(9) pp. 836–840 (September 1981).
2. A. P. Reeves, "Parallel Pascal: An Extended Pascal for Parallel Computers," *Journal of Parallel and Distributed Computing* **1** pp. 64–80 (1984).
3. J. D. Bruner and A. P. Reeves, "A Parallel P–Code for Parallel Pascal and Other High Level languages," *1983 International Conference on Parallel Processing,* pp. 240–243 (August 1983).
4. A. P. Reeves, "The Massively Parallel Processor: A Highly Parallel Scientific Computer," pp. 239–252 in *Data Analysis in Astronomy II,* ed. V. Di Gesu, Plenum Press (1986).
5. R. W. Hockney and C. R. Jesshope, *Parallel Computers,* Adam Hilger Ltd, Bristol (1981).
6. M. Gutierrez, *Algorithms and Performance Analysis for the Massively Parallel Processor,* MS Thesis, Cornell University, January 1988.
7. E. O. Brigham, *The Fast Fourier Transform,* Prentice–Hall, Englewood Cliffs, N.J. (1974).
8. A. P. Reeves and C. H. Moura, "Data Mapping and Rotation Functions for the Massively Parallel Processor," *Proceedings of Computer Architecture for Pattern Analysis and Image Database Management,* pp. 412–419 (November 1985).

Parallel OPS5 on the Encore Multimax

Anoop Gupta
Department of Computer Science, Stanford University, Stanford, CA 94305

Charles L. Forgy, Dirk Kalp, Allen Newell, and Milind Tambe
Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213

Abstract

Until now, most results reported for parallelism in production systems (rule-based systems) have been simulation results -- very few real parallel implementations exist. In this paper, we present results from our parallel implementation of OPS5 on the Encore multiprocessor. The implementation exploits very fine-grained parallelism to achieve significant speed-ups. For one of the applications, we achieve 12.4 fold speed-up using 13 processes. Our implementation is also distinct from other parallel implementations in that we parallelize a highly optimized C-based implementation of OPS5. Running on a uniprocessor, our C-based implementation is 10-20 times faster than the standard lisp implementation distributed by Carnegie Mellon University. In addition to presenting the performance numbers, the paper discusses the amount of contention observed for shared data structures, and the techniques used to reduce such contention.

1. Introduction

As the technology of production systems (rule-based systems) is maturing, larger and more complex expert systems are being built both in industry and in universities. Often these large and complex systems are very slow in their execution, and this limits their utility. Researchers have been exploring many alternative ways for speeding up the execution of production systems. Some efforts have been focussing on high-performance uniprocessor implementations [2, 10], while others have been focussing on high-performance parallel implementations [3, 6, 11, 9, 12, 13, 14]. This paper focusses on parallel implementations.

Until now, most results reported for parallelism in production systems have been simulation results. In fact, very few real parallel implementations exist. In this paper, we present results from our parallel implementation of OPS5 on an Encore Multimax shared-memory multiprocessor with sixteen CPUs. The implementation, called PSM-E (Production System Machine project's Encore implementation), exploits very fine-grained parallelism to achieve up to 12.4 fold speed-up for match using 13 processes. Our implementation is distinct from other parallel implementations in that we parallelize a highly optimized C-based implementation of OPS5. This is in contrast to other efforts where slow lisp-based implementations are being parallelized. Running on a uniprocessor, our C-based implementation is 10-20 times faster than the lisp implementation of OPS5 distributed by Carnegie Mellon University. A consequence of parallelizing a highly-optimized implementation is that one must be very careful about overheads, else the overheads may nullify the speed-up. One need not be as careful when parallelizing an unoptimized implementation. In this paper, we first discuss the design of an optimized implementation of OPS5, and then discuss the additions that were made for the parallel implementation. For the parallel implementation, we discuss the synchronization mechanisms that were used, the contention observed for various shared data structures, and the techniques used to reduce such contention.[1]

The paper is organized as follows. Section 2 presents some background information about the OPS5 language, the Rete match algorithm, and the Encore Multimax multiprocessor. Section 3 gives an overview of the parallel interpreter and then goes into the implementation details describing how the rules are compiled and how various synchronization and scheduling issues are handled. Section 4 presents the results of the implementation on the Encore multiprocessor. Finally, in Section 5 we summarize the results and conclude.

2. Background

This section is divided into three parts. The first subsection describes the basics of the OPS5 production-system language -- the language which we have implemented in parallel. The second subsection describes the Rete algorithm -- the algorithm that forms the basis for our parallel implementation. The third subsection describes the Encore Multimax computer system -- the multiprocessor on which we have done the parallel implementation.

2.1. OPS5

An OPS5 [1] production system is composed of a set of *if-then* rules called *productions* that make up the *production memory*, and a database of temporary assertions called the *working memory*. The assertions in the working memory are called *working memory elements* (wmes). Each production consists of a conjunction of *condition elements* corresponding to the *if* part of the rule (also called the *left-hand side* of the production), and a set of *actions* corresponding to the *then* part of the rule (also called the *right-hand side* of the production). The actions associated with a production can add, remove or modify working memory elements, or perform input-output. Figure 2-1 shows a production named *find-colored-block* with two condition elements in its left-hand side and one action in its right-hand side.

```
(p find-colored-block
   (goal   ^type find-block   ^color   <c>)
   (block ^id <i>   ^color <c>   ^selected no)
 -->
   (modify 2    ^selected   yes))
```

Figure 2-1: A sample production.

The production system *interpreter* is the underlying mechanism that determines the set of satisfied productions and controls the

[1] This paper presents only a summary of the results. A more in depth analysis is presented in [5].

execution of the production system program. The interpreter executes a production system program by performing the following *recognize-act* cycle:

- **Match**: In this first phase, the left-hand sides of all productions are matched against the contents of working memory. As a result a *conflict set* is obtained, which consists of *instantiations* of all satisfied productions. An instantiation of a production is an ordered list of working memory elements that satisfies the left-hand side of the production.
- **Conflict-Resolution**: In this second phase, one of the production instantiations in the conflict set is chosen for execution. If no productions are satisfied, the interpreter halts.
- **Act**: In this third phase, the actions of the production selected in the conflict-resolution phase are executed. These actions may change the contents of working memory. At the end of this phase, the first phase is executed again.

A working memory element is a parenthesized list consisting of a constant symbol called the *class* of the element and zero or more *attribute-value* pairs. The attributes are symbols that are preceded by the operator ^. The values are symbolic or numeric constants. For example, the following working memory element has class **C1**, the value **12** for attribute **attr1** and the value **15** for attribute **attr2**.

```
(C1     ^attr1  12     ^attr2  15)
```

The condition elements in the left-hand side of a production are parenthesized lists similar to the working memory elements. They may optionally be preceded by the symbol –. Such condition elements are called *negated* condition elements. Condition elements are interpreted as partial descriptions of working memory elements. When a condition element describes a working memory element, the working memory element is said to *match* the condition element. A production is said to be *satisfied* when: (1) For every non-negated condition element in the left-hand side of the production, there exists a working memory element that matches it; (2) For every negated condition element in the left-hand side of the production, there does not exist a working memory element that matches it.

Like a working memory element, a condition element contains a class name and a sequence of attribute-value pairs. However, the condition element is less restricted than the working memory element; while the working memory element can contain only constant symbols and numbers, the condition element can contain variables, predicate symbols, and a variety of other operators as well as constants. Variables are identifiers that begin with the character "<" and end with ">" -- for example, <i> and <c> are variables. A working memory element matches a condition element if they belong to the same class and if the value of every attribute in the condition element matches the value of the corresponding attribute in the working memory element. The rules for determining whether a working memory element value matches a condition element value are: (1) If the condition element value is a constant, it matches only an identical constant. (2) If the condition element value is a variable, it will match any value. However, if a variable occurs more than once in a left-hand side, all occurrences of the variable must match identical values. (3) If the condition element value is preceded by a predicate symbol, the working memory element value must be related to the condition element value in the indicated way.

The right-hand side of a production consists of an unconditional sequence of actions which can cause input-output, and which are responsible for changes to the working memory. Three kinds of actions are provided to effect working memory changes. *Make* creates a new working memory element and adds it to working memory. *Modify* changes one or more values of an existing working memory element. *Remove* deletes an element from the working memory.

2.2. The Rete Match Algorithm

In this subsection, we describe the Rete algorithm used for performing the match-phase in the execution of production systems. The match-phase is critical because it takes 90% of the execution time and as a result it needs to be speeded up most. Rete is a highly efficient algorithm for match that is also suitable for parallel implementations. A discussion of alternative match algorithms can be found in [3]. The Rete algorithm gains its efficiency from two optimizations. First, it exploits the fact that only a small fraction of working memory changes each cycle by storing results of match from previous cycles and using them in subsequent cycles. Second, it exploits the similarity between condition elements of productions (both within the same production and between different productions) to reduce the number of tests that it has to perform to do match. It does so by performing common tests only once.

The Rete algorithm uses a special kind of a data-flow network compiled from the left-hand sides of productions to perform match. The network is generated at compile time, before the production system is actually run. Figure 2-2 shows such a network for productions p1 and p2, which appear in the top part of the figure. In this figure, lines have been drawn between nodes to indicate the paths along which information flows. Information flows from the top-node down along these paths. The nodes with a single predecessor (near the top of the figure) are the ones that are concerned with individual condition elements. The nodes with two predecessors are the ones that check for consistency of variable bindings between condition elements. The terminal nodes are at the bottom of the figure. Note that when two left-hand sides require identical nodes, the algorithm shares part of the network rather than building duplicate nodes.

To avoid performing the same tests repeatedly, the Rete algorithm stores the result of the match with working memory as state within the nodes. This way, only changes made to the working memory by the most recent production firing have to be processed every cycle. Thus, the input to the Rete network consists of the changes to the working memory. These changes filter through the network updating the state stored within the network. The output of the network consists of a specification of changes to the conflict set.

The objects that are passed between nodes are called *tokens*, which consist of a *tag* and an *ordered list of working-memory elements*. The tag can be either a +, indicating that something has been added to the working memory, or a –, indicating that something has been removed from it. No special tag for working-

```
(p p1 (C1 ^attr1 <x> ^attr2 12)          (p p2 (C2 ^attr1 15 ^attr2 <y>)
      (C2 ^attr1 15  ^attr2 <x>)               (C4 ^attr1 <y>)
    - (C3 ^attr1 <x>)                        -->
  -->                                          (modify 1 ^attr1 12))
      (remove 2))
```

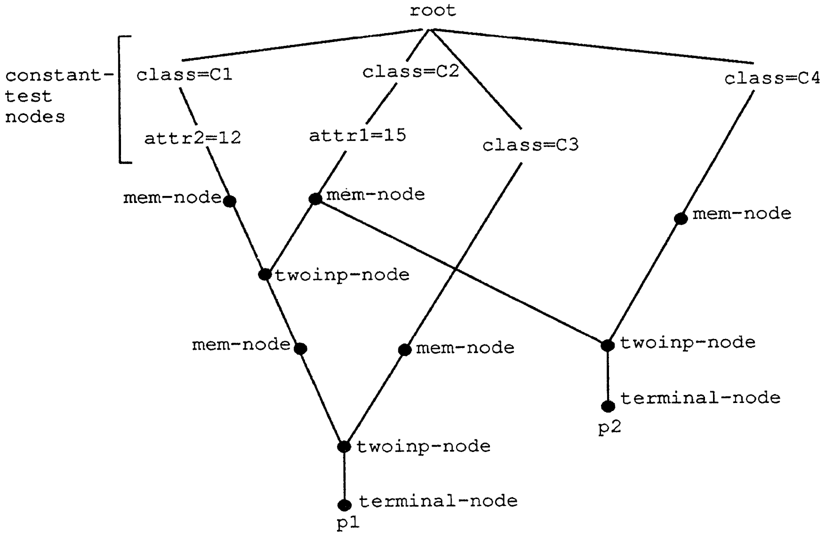


Figure 2-2: The Rete network.

memory element modification is needed because a modify is treated as a delete followed by an add. The list of working-memory elements associated with a token corresponds to a sequence of those elements that the system is trying to match or has already matched against a subsequence of condition elements in the left-hand side.

The data-flow network produced by the Rete algorithm consists of four different types of nodes. These are:

1. **Constant-test nodes:** These nodes are used to test if the attributes in the condition element which have a constant value are satisfied. These nodes always appear in the top part of the network. They have only one input, and as a result, they are sometimes called *one-input* nodes.

2. **Memory nodes:** These nodes store the results of the match phase from previous cycles as state within them. The state stored in a memory node consists of a list of the tokens that match a part of the left-hand side of the associated production. For example, the right-most memory node in Figure 2-2 stores all tokens matching the second condition-element of production p2.

3. **Two-input nodes:** These nodes test for joint satisfaction of condition elements in the left-hand side of a production. Both inputs of a two-input node come from memory nodes. When a token arrives on the left input of a two-input node, it is compared to each token stored in the memory node connected to the right input. All token pairs that have consistent variable bindings are sent to the successors of the two-input node. Similar action is taken when a token arrives on the right input of a two-input node.

4. **Terminal nodes:** There is one such node associated with each production in the program, as can be seen at bottom of Figure 2-2. Whenever a token flows into a terminal node, the corresponding production is either inserted into or deleted from the conflict set.

The most commonly used interpreter for OPS5 is the Rete-based Franz Lisp interpreter. In this interpreter a significant loss in the speed is due to the interpretation overhead of nodes. In the OPS5 implementation we present in this paper, the interpretation overhead has been eliminated by compiling the network directly into machine code. While it is possible to escape to the interpreter for complex operations during match or for setting up the initial conditions for the match, the majority of the match is done without an intervening interpretation level. This has led to a speed-up of 10-20 fold over the Franz Lisp interpreter (see Table 4-4). In addition to this speed-up, our parallel implementation gets further speed-up by evaluating different node activations in the Rete network in parallel.

2.3. Encore Multimax

In this subsection, we describe the Encore Multimax shared-memory multiprocessor -- the computer system on which parallel OPS5 runs. The Multimax consists of 2-20 CPUs, each of which is connected to the shared-memory through a high performance bus. The shared-memory is equally accessible to all of the processors, in that each processor sees the same latency for memory accesses.

The processors used in our Encore Multimax are National Semiconductor NS32032 chips along with NS32081 floating point coprocessors, each processor capable of approximately 0.75 million instructions per second. There are two processors packaged per board and they share 32 Kbytes of cache memory. The processor boards use a combination of write-through strategy and bus-watching logic to keep the caches on different processor boards consistent. The bus used on the Encore Multimax is called the *Nanobus*. It is a synchronous bus and it can transfer 8 bytes of new information every 80 nanoseconds, thus providing a data transfer bandwidth of 100 Mbytes/second.

The version of Encore Multimax available to us at CMU has 16 processors, 32 Mbytes of memory, and runs the MACH operating system developed at Carnegie Mellon University. The operating system provides a UNIX-like interface to the user, although the internals are different and several extensions have been made to support the underlying parallel hardware. It provides facilities to automatically distribute processes amongst the available processors and it provides facilities for multiple processes to share memory for communication and synchronization purposes. The results reported in this paper correspond to this configuration of the Encore Multimax.

3. Organization and Details of the Parallel Implementation

When studying parallelism in production systems (or in any other application for that matter), it is important to compute the speed-ups with respect to the performance of the most efficient uniprocessor implementations. It is indeed quite easy to obtain large speed-ups with respect to inefficient implementations of the application, but such results have little practical utility. In the case of OPS5, the most efficient uniprocessor implementations are currently based on the Rete algorithm and they compile the Rete network directly into machine code and use global register allocation. Such compilation into machine code gives approximately 10-20 fold speed-up over Rete-based lisp implementations of OPS5 (see Table 4-4). For this reason, our parallel implementation of OPS5 on the Encore is also Rete-based and compiles the Rete network directly into (NS32032) machine code.[2] Another effect of parallelizing a highly efficient implementation versus an inefficient one is that the number of instructions executed in each parallel subtask (for the same task decomposition) is smaller in the highly efficient implementation. This is equivalent to exploiting parallelism at a finer granularity, and as a result, the issues of synchronization and scheduling are more critical.

3.1. High-Level Structure of the Parallel Implementation

The parallel OPS5 implementation on the Encore (PSM-E) consists of one *control process* and one or more *match processes*. The number of match processes is a user specified parameter, but it is fixed for the duration of any particular run. The system is generally used in a mode where the computer contains at least as many free processors as there are processes in the matcher; this permits each process to be assigned to a distinct processor for the duration of the run (provided the operating system is reasonably clever about assigning processes to processors).

The control process is responsible for performing conflict resolution, evaluating the right-hand side of rules, handling input/output, and all the other functions of the interpreter except for performing match. It is also responsible for starting up the match processes at the beginning of the run and killing them at the end of the run. The match processes do nothing except perform the match. The match processes pipeline their operation with the control process. Thus when RHS evaluation begins, the match processes are idle. However, as soon as the first working memory change is computed, information about that change is passed to the match processes and they start to work. The control process continues evaluating the RHS, and as more changes are computed, the information is passed immediately to the match processes for them to handle as soon as they are able. When the control process finishes evaluating the RHS, it becomes idle and waits for the match processes to finish. When the last match process finishes, the control process performs conflict resolution and then begins evaluating the next RHS, thus starting the cycle over again.[3]

To perform match, the match processes use the Rete algorithm described in Section 2.2. The match processes exploit the dataflow-like nature of the Rete algorithm to achieve speed-up from parallelism. In particular, a single copy of the Rete network is held in shared memory. The match processes cooperate to pass tokens through the network and update the state stored in the memory nodes as indicated by the tokens. The match is broken into fairly small units of work called *tasks*, where a task is an independently schedulable unit of work that may be executed in parallel with other tasks. In our parallel implementation:

- Small groups of constant-test node activations constitute a task. Multiple constant-test nodes are processed as a group, because individual constant-test node activations take only 3 machine instructions to execute, and that is too fine a granularity.
- The memory nodes in the Rete network are coalesced with the two-input nodes that are below them. Each activation of these coalesced two-input nodes constitutes a single task. The reasons for this coalescing are discussed in [4]. As an example, the task corresponding to the left activation of a two-input node involves: (i) the addition/deletion of the incoming token to the left memory node; (ii) comparison of this token with all tokens in the opposite memory node checking for consistent variable bindings; and (iii) scheduling of matching token pairs for execution as new tasks. Note that multiple activations of the same two-input node constitute different tasks and these can be processed in parallel.
- Each individual terminal node activation constitutes a task.

In our current implementation, each task is represented by a data object called a *token*. The token in the parallel implementation is essentially the same as that used in the sequential Rete matcher (as described in Section 2.2), except that it has two extra items of information: the address of the node to which the token is to be sent, and if that node is a two-input node, an indication of whether to send it to the left or right input. The list of tokens that are awaiting processing is held in a central data structure called a *task queue*. The individual match processes perform match by executing the following loop.

[2] Note that the argument in the beginning of this paragraph does not say that one has to use the same algorithm (as the most efficient uniprocessor one) for the parallel implementation. It just turns out in our case, that the efficient uniprocessor algorithm is also very good for parallel implementation.

[3] For simplicity, we are ignoring two kinds of optimizations that are possible. First, it is possible to overlap conflict-resolution with match. Second, if *speculative* parallelism is used (we are willing to be wrong in our prediction sometimes and know how to recover from the error), it is possible to make a guess about the production that will fire next and to evaluate its right-hand side before conflict-resolution is completely finished. We choose to ignore these two optimizations for the present, because conflict-resolution and RHS evaluation are not the bottlenecks in our current implementation.

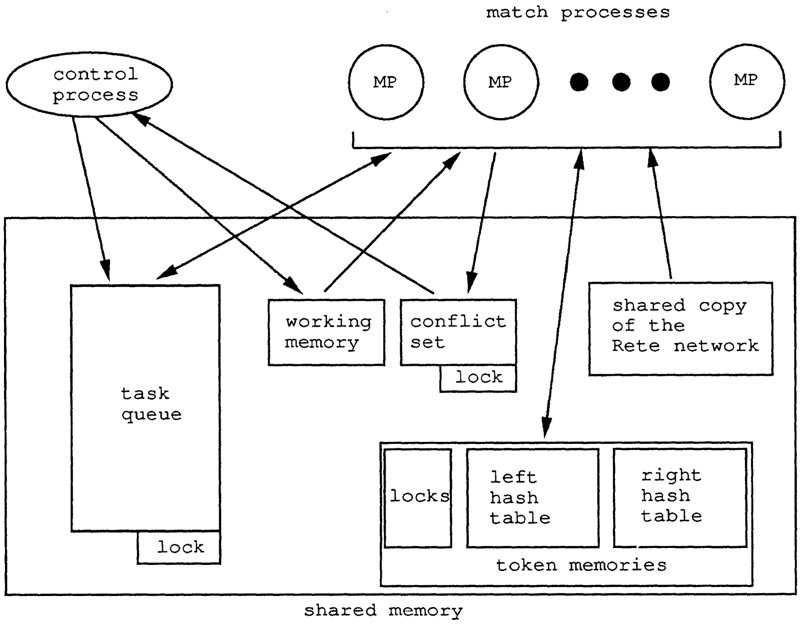


Figure 3-1: Use of shared-memory by various processes.

1. Remove a token from the task queue. If the queue is empty, wait until something is added.
2. Process the token. If new tokens are to be sent out, push them onto the task queue.
3. Go to step 1.

3.2. Implementation Details

All communication between processes (both the match processes and the control process) takes place via shared memory. The virtual address spaces are set up so that the objects in shared memory have the same virtual address in every process. Hence processes can simply pass pointers around in essentially the same way routines within a single process can. For example, the tokens are created in shared memory, and the address of a given token is the same in every virtual address space in the system. Thus when a process places a token onto the central task queue, all it really has to do is to put the address of the token into the task queue. Figure 3-1 shows how the shared-memory is used to communicate between the various processes.

Synchronization within the program is handled explicitly by executing interlocked test-and-set instructions. The synchronization primitives provided by the operating system (for example, semaphores, barriers, signals, etc) are not used because of the large overhead associated with them. When a process finds that it is locked out of a critical region it spins on the lock, waiting for a chance to enter the region. In order to minimize the amount of bus traffic generated by the spinning processes, a "test and test-and-set" synchronization mechanism is used. In this scheme, a process uses ordinary memory-read instructions to test the status of a lock until it finds that it is free; then the process uses a test-and-set interlocked instruction to re-read the lock and set it (if it is still free). Note that while the lock is busy, the process spins out of its cache and does not use the bus. This is more efficient than using only the "test-and-set" interlocked instruction for the lock. In this case, the process generates bus traffic to perform the writes while it is busy waiting.

The control process communicates with the match processes primarily through the shared task queue. Whenever the evaluation of an RHS results in a change to working memory, a token is created and marked as being destined for the root node of the network. The control process pushes these tokens onto the task queue in exactly the same way as the match processes push the tokens they create. The tokens are picked up and processed by waiting match processes. When the evaluation of an RHS begins, the match processes are idle. The first token created by the control process causes the match processes to start up. After the first token, the control process proceeds in parallel with the match processes.

Depending on the granularity of tasks (number of instructions executed per task) that are scheduled using the task queue and depending on the number of processors that are trying to access the task queue in parallel, it is quite possible that a single task queue would become a bottleneck. For this reason, Gupta [4] proposed a hardware task scheduler for scheduling the fine-grained tasks. So far we have not implemented the hardware scheduler, and in this paper we present results only for the case when one or more software task queues are used.

After the control process finishes evaluating the RHS, it must wait for the match processes to finish before it can perform the next conflict resolution operation. A global counter, TaskCount, is used to determine when all the match processes have finished. This counter contains the sum of:

- the number of tokens that are currently on the task queue, and
- the number of tokens that are being processed by the match processes.

This count is maintained quite simply. Every time a token is put onto the task queue, the counter is incremented. Every time a match process finishes working with a token, the counter is decremented. The match phase is finished when the counter goes to zero.

Shifting our focus back to the evaluation of individual two-input node activations, we note that instead of having separate memories for each two-input node, the matcher has two large hash tables which hold all the tokens for the entire network. One hash table holds tokens for left memories of two-input nodes, and the other for right memories of two-input nodes. An alternative scheme is to have separate hash tables for each two input node, but such a scheme was considered to be wasteful of space. The hash function that is applied to the tokens takes into account:

- the values in the token which will have equality tests applied at the two-input node, and
- the unique identifier of the two-input node which stored the tokens.

This permits the two-input nodes to locate any tokens that are likely to pass the equal-variable tests quickly. It also permits multiple activations of the same two-input node to be processed in parallel.

The processing performed by the individual node activations in the parallel implementation is similar to the processing done in the sequential matcher with two exceptions:

- Code has been added to the two-input nodes to handle conjugate token pairs.
- Sections of code that access shared resources are protected by spin locks to insure that only one process at a time can be accessing each resource.

A *conjugate pair* is a pair of tokens with opposite signs (an add token request and a delete token request), but which refer to the same working memory element or list of working memory elements. Conjugate pairs arise in the match operation for a variety of reasons, which are too complex to go into here (see [4]). They occur in both sequential and parallel implementations of Rete, but they present much greater problems in a parallel system. The reason for this is that in a parallel system it is not possible to insure that the tokens will be processed in the order in which they are generated, and consequently in some cases a token with a – (delete) flag will arrive at a two-input node before the corresponding token with the + (add) flag. The parallel matcher code handles this by saving the – tokens that arrive early on an *extra-deletes-list* without otherwise processing the token. When the corresponding + token arrives both tokens are discarded.

Many resources in a parallel system have to be protected with mutual-exclusion locks -- the task queues, the count of the number of active tokens, the conflict set, etc. Most of these are relatively straight-forward to protect and a simple variation of standard spin locks is used. The exception is the locks used to control access to the token hash tables. There are several different operations that are performed on the token hash tables, for example, searching for matching tokens, adding and removing tokens, adding and removing conjugate tokens, and we would like many of these operations to proceed in parallel without having any undesirable effects. Because of the importance of the hash tables to the performance of the system, several locking schemes were implemented and tried. Two of these schemes are described here.

The first scheme, the simple one, is easy to describe and it provides a departure point for describing the second more complex one. We define a "line" as a pair of corresponding buckets (buckets with the same hash index) from the left and right hash tables along with their associated extra-deletes lists. In this scheme, each line in the hash table has a flag controlling its use.[4] The flag takes on two values: *Free* and *Taken*. When a process has to work with the hash table, it examines the flag for the line it needs. If the flag is *Free*, it sets the flag to *Taken* and proceeds to perform the necessary operations; when it finishes, it sets the flag back to *Free*. If a process finds the flag set to *Taken*, it waits until the flag is set to *Free*. Of course, the act of testing and setting the flag must be an atomic operation. This synchronization scheme works, but it is a potential bottleneck when several tokens arrive at a node about the same time, and if all of them require access to the same hash table line.

The second scheme is a complex variant of the *multiple-reader-single-writer* locking scheme. It permits several tokens to be processed in the same line at the same time, though even here, some serialization of the processing is necessary when destructive modifications to the lists of tokens are performed. This scheme requires two locks, a flag, and a counter for each line in the hash table. The flag takes on three values: *Unused*, *Left*, and *Right*, to indicate respectively that the line is not currently being processed, that it is being used to process tokens arriving from the left, or that it is being used to process tokens arriving from the right. The counter indicates how many processes are using that line in the hash table; it is needed only so that the last process to finish using the line can set the flag back to *Unused*. The first lock insures that only one process at a time can access the flag and the counter. When a process first tries to use a line in the hash table, it gets this lock, and checks the flag. If the flag indicates that tokens from the other side are being processed, the process releases the lock and put the token back onto the task queue. If the flag allows the process to continue, it sets the flag if necessary, increments the counter, and releases the lock. For the remaining time that the process uses this line in the hash table, it leaves the flag and the counter untouched; finally, when the process finishes using the line it decrements the counter and if appropriate sets the flag to *Unused* (again, all within a section of code protected by this lock). All this is to insure that tokens from two different sides are not processed at the same time. The second lock is used to insure that only one process at a time can be modifying the token lists. Recall that the first task in processing a two-input node is to update the list of tokens stored in the memory node. To do this, the process gets the modification lock, searches the conjugate or regular token list, and it either adds the token to or deletes it from one of these lists. When it has finished, it releases the modification lock and proceeds with searching the tokens in the opposite hash-table bucket to find those that satisfy the variable binding tests.

More complex locking schemes can be devised and, in fact, were implemented and tested. One other scheme that was tried permitted more than one process to be searching the token lists to find tokens to delete; in this scheme the only serialization of the tasks occurred when the actual destructive modification of the token list was performed. As in all implementations, the main

[4] Note that any given operation on the token hash tables requires access to only a single line of the hash tables. In other words processing a single node activation never requires access to multiple hash table lines.

Table 4-1: Uniprocessor versions on Microvax-II.

PROGRAM	VS1 List-based memories (sec)	VS2 Hash-based memories (sec)	Total number of WM-changes processed	Total number of node activations
Weaver	101.5	85.8	1528	371173
Rubik	235.2	96.9	8350	554051
Tourney	323.7	93.5	987	72040

tradeoff to keep in mind is that in an attempt to speed-up the rare cases, one should not slow-down the normal case.

3.3. RHS Evaluation and Conflict Resolution

In our system, the rules' RHSs are compiled into a form of threaded code which is interpreted at run time [8]. Interpreting the threaded code is slower than executing the compiled code, but since RHS evaluation is not a bottleneck to the performance, threaded code, which is simpler to compile was considered fast enough. Conflict resolution in the system is handled by code written in the C language. This code is executed by the control process.

4. Results

We present results for the parallel execution of three production-system programs in this paper. These are:

- Weaver [7], a VLSI routing program by Rostom Joobbani with 637 rules.
- Rubik, a program that solves the Rubik's cube by James Allen with 70 rules.
- Tourney, a program that assigns match schedules for a tournament by Bill Barabash from DEC with 17 rules.

We have chosen Weaver because it represents a fairly large program and it demonstrates that our parallel OPS5 can handle real systems. Rubik is a smaller program that demonstrates some of the strengths of our parallel implementation and the Tourney program demonstrates some of the weaknesses of our parallel implementation.

4.1. Results ior the Uniprocessor Implementations of OPS5

Before we did a parallel implementation on the Encore, we initially did several uniprocessor C-based implementations of OPS5. In this subsection, we present results for two of these uniprocessor implementations, vs1 and vs2, for the Microvax-II workstation.[5] The performance results for vs1 and vs2 implementations are shown in Table 4-1. The base version is vs1, and it is characterized by the use of linear lists to store tokens in node memories, just as uniprocessor lisp implementations do.[6]

The second version, vs2, uses a global hash table to store all memory-node tokens, as discussed in the previous section. If there are equality tests at the two-input node, the hash-table based scheme (i) reduces the number of tokens that have to be examined in the opposite memory to locate those that have consistent variable bindings, and (ii) for deletes, it reduces the number of tokens that have to be examined in the same memory to locate the token to be deleted. The statistics for the reduction in tokens examined in the opposite memory for the three programs are given in Table 4-2. Note the statistics are computed only for those node activations where the opposite memory is not empty. The statistics for the reduction in tokens examined in the same memory for delete requests are given in Table 4-3. As can be seen from the two tables, the savings are substantial, especially for the Tourney program. The time-saving effect of hash-based memories can be seen from numbers in Table 4-1.

The second last column in Table 4-1 gives the total number of wme-changes processed during the run for which data are presented, and the last column gives the total number of node activations processed during the run (this is also equal to the number of tasks that are pushed/popped from the task queue in the parallel version). Dividing the time in column vs2 by the number of tasks, we get the average duration for which a task executes. This has implications for the amount of synchronization and scheduling overhead that may be tolerated in the parallel implementation. Doing this division we get that the average duration of a task for Weaver is 230 microseconds (or approximately 115 machine instructions, as the VAX executes about 500,000 instructions per second), for Rubik is 175 microseconds, and for Tourney is 1300 microseconds.

Finally, Table 4-4 gives the speed-up that our uniprocessor C-based implementation achieves over the widely available Franzlisp-based OPS5 implementation when running on the Microvax-II. As the table shows, we get a speed-up of about 10-20 fold over the Franzlisp based implementation. The problem in the past has been that due to lack of availability of better uniprocessor performance numbers, researchers have ended up comparing the performance of their highly optimized parallel OPS5 implementations with the slow Franzlisp-based implementation. We think that such apples to oranges comparison can be misleading, and we hope that in the future the performance of parallel implementations would be compared to the performance of this optimized uniprocessor implementation.

[5] The results are presented for Microvax-II and not for Encore, because the uniprocessor implementations were done on the Microvax and only one of these was later taken over to the Encore.

[6] Note that memory nodes are not shared in either vs1 or vs2 versions of OPS5, unlike in the Franzlisp version of OPS5. This optimization was not used in vs1 or vs2 because it is not possible to share memory nodes in the parallel implementations of OPS5 (see [4]), and we did not want to spend the effort just for the uniprocessor implementations.

Table 4-2: Number of tokens examined in opposite memory.

PROGRAM	Tokens in opp mem for left actvns		Tokens in opp mem for right actvns	
	lin mem	hash mem	lin mem	hash mem
Weaver	10.1	7.7	5.2	1.0
Rubik	31.0	3.8	1.6	1.8
Tourney	47.6	5.9	270.1	23.3

Table 4-3: Number of tokens examined in same memory for deletes.

PROGRAM	Tokens in same mem for left actvns		Tokens in same mem for right actvns	
	lin mem	hash mem	lin mem	hash mem
Weaver	6.2	3.6	7.0	5.1
Rubik	23.5	2.6	8.1	3.7
Tourney	254.4	40.1	3.8	2.9

Table 4-4: Speed-up of C-based over Franzlisp-based implementation

PROGRAM	VS-lisp Lisp-based implemen. (sec)	VS2 Hash-based memories (sec)	Speed-up VS-lisp/VS2
Weaver	1104.0	85.8	12.9
Rubik	1175.0	96.9	12.1
Tourney	2302.0	93.5	24.6

4.2. Results for the Multiprocessor Implementation of OPS5

While the uniprocessor C-based implementations of OPS5 were done on the Microvax-II, the parallel version was done on the Encore Multimax multiprocessor. In this section, we present speed-up numbers for our implementation on the Encore and the results of our experiments as we varied (i) the number of task queues that were used and (ii) the locking structures used for token hash-table buckets.

Table 4-5 shows results for the case when a single task queue is used and when *simple* locks (described in Section 3.2) are used with the token hash-table buckets. The first column simply gives the name of the programs. The second column gives the time taken to do the match when only one process is used (time for conflict-resolution and RHS evaluation is not included). The timing numbers in the second column correspond to version vs2 discussed earlier. The numbers here are larger than the corresponding numbers in Table 4-1 because the NS32032 processor used in Encore is slower than the Microvax-II processor and because of the presence of extra synchronization and scheduling code in the parallel implementation. The numbers given in the remaining columns are speed-up figures with respect to the time given in the second column. The number of processes used in the parallel match are given in the second row from the top in the table. These numbers are expressed as "1+k", where the "1" indicates the control process and the "k" indicates the number of match processes. The third row from the top indicates the number of task queues used, which is one for all entries in this table.

In Table 4-5, the speed-up for the case when number of processes is "1+1" is in two cases greater than one. This is because the set of node activations is different when the RHS evaluation and match are proceeding in parallel (even though match is being done by only one process), as compared to the case when match does not start until RHS evaluation is completely finished. The speed-up with multiple match processes is also quite disappointing for all three programs and especially for Tourney. Possible reasons are: (i) contention for access to the single task queue, (ii) contention for access to the hash-table buckets, and (iii) low intrinsic parallelism in the programs. We now explore the effects of removing the first two bottlenecks by using multiple task queues and by using more complex hash-table locking schemes.

Table 4-6 presents results for the case when multiple task queues are used, while retaining simple hash-table locks. The speed-up increases significantly for both Weaver and Rubik, indicating that the contention for pushing and popping task queues must have been a bottleneck. The speed-up for Weaver for 1+13 processes goes up from 3.9-fold to 8.2-fold and that for Rubik goes up from 6.3-fold to 11.4-fold. The speed-up for Tourney remains about the same at 2.4-fold. To get more insight into these results, we instrumented the task queue to get data on contention. The results are shown in Table 4-7. The table shows the contention among the processes for the centralized task queue as the number of match processes is increased. We see from the table that as the number of processes is increased, there is indeed significant contention for the single task queue in case of Weaver and Rubik. For Tourney, there does not seem to be significant contention for the task queue, and that is why the speed-up does not increase when multiple task queues are used. The contention numbers drop from 24.62, 26.89, and 8.93 for single task queue to 4.85, 6.12, and 4.75 for eight task queues for Weaver, Rubik, and Tourney respectively when 13 match processes are used.

Examining the speed-up for Rubik in Table 4-6, it is interesting to note that we get 3.9-fold speed-up for Rubik using only 3 match processes. In this case, the speed-up is larger than the number of match processes because when the Rete network is evaluated in parallel, it is quite possible that the total number of node activations evaluated and their complexity is less than that in the sequential implementation. Of course, the final result of the match is still the same as the sequential implementation.

In Table 4-8 we present results for the case when multiple task queues are used and when complex multiple-reader-single-writer locks (described in Section 3.2) are used for controlling entry to the token hash tables. We expected the complex locks to benefit those programs that (i) generate cross-products, that is, there are multiple activations of the same two-input node from the same side that need concurrent processing, and (ii) have long lists of tokens in hash-table buckets, where the complex locks help by allowing multiple processes to read the opposite memory at the same time.

Table 4-5: Speed-up for single task queue and simple hash-table locks.

PROGRAM	Uniproc Execution Time (sec)	Speed-ups with multiple processes					
		1+1	1+3	1+5	1+7	1+11	1+13
		1 Que	1 Que	1 Que	1 Que	1 Que	1 Que
Weaver	119.9	1.02	2.55	3.65	3.97	3.91	3.90
Rubik	257.9	1.00	2.80	4.47	5.48	6.18	6.30
Tourney	98.0	1.10	1.90	2.70	2.59	2.43	2.41

Table 4-6: Speed-up for multiple task queues and simple hash-table locks.

PROGRAM	Uniproc Execution Time (sec)	Speed-up with multiple processes					
		1+1	1+3	1+5	1+7	1+11	1+13
		1 Que	2 Que	4 Que	8 Que	8 Que	8 Que
Weaver	118.2	1.02	2.88	4.51	5.80	7.56	8.15
Rubik	253.6	1.07	3.93	6.41	8.49	10.66	11.42
Tourney	97.7	1.12	2.02	2.17	2.33	2.47	2.30

Table 4-7: Contention for the centralized task queue. Measured by the number of times a process spins on the lock before it gets access to the task queue.

PROGRAM	contention for the central task queue					
	1+1	1+3	1+5	1+7	1+11	1+13
	1 Que	1 Que	1 Que	1 Que	1 Que	1 Que
Weaver	1.03	2.68	6.31	11.58	20.05	24.62
Rubik	1.01	2.63	5.92	10.58	22.66	26.89
Tourney	1.00	1.57	2.53	3.94	7.22	8.93

Table 4-8: Speed-up for multiple task queues and multiple-reader-single-writer hash-table locks.

PROGRAM	Uniproc Execution Time (sec)	Speed-ups with multiple processes					
		1+1	1+3	1+5	1+7	1+11	1+13
		1 Que	2 Que	4 Que	8 Que	8 Que	8 Que
Weaver	134.9	1.02	3.02	4.63	6.14	8.18	9.02
Rubik	289.4	1.04	3.98	6.40	9.01	11.33	12.35
Tourney	100.8	1.07	2.06	2.58	2.40	2.57	2.67

However, programs for which the above two conditions are not true may slow down when complex locks are used, because of the extra overhead that they incur due to complex locks. Table 4-9 presents some results about contention when simple locks are used versus contention when complex locks are used. We see that the contention for the hash-table buckets decreases for all three programs when complex locks are used, although the increase in speed-up is not much. However, Table 4-9 does give an indication as to why we are getting very poor speed-up for the Tourney program. The poor speed-up for the Tourney program is due to the large contention for the hash-table locks resulting from multiple node activations trying to access the same hash-table bucket. This, in turn, is the result of a few culprit productions in Tourney that have condition elements with no common variables. By modifying two such productions using domain specific knowledge, we could increase the speed-up achieved using 1+13 processes from 2.7-fold to 5.1-fold.

Table 4-9: Contention for token hash-table locks. Measured by the number of times a process spins on a lock before it gets access to the hash-table bucket.

PROGRAM	contention with simple locks				contention with mrsw locks			
	6 processes		12 processes		6 processes		12 processes	
	left	right	left	right	left	right	left	right
Weaver	20.4	1.0	51.2	1.4	4.7	2.0	15.7	2.1
Rubik	11.0	1.1	23.0	1.5	3.7	2.0	12.9	2.1
Tourney	137.1	4.9	377.7	15.7	49.9	2.9	134.9	33.3

5. Conclusions

In this paper we have presented the details of a parallel implementation of OPS5 running on the Encore Multimax. The first observation is that it is important to speed-up an *optimized* sequential implementation, otherwise most of the benefits are lost. For example, speeding-up the Franzlisp implementation by 10-20 fold from parallelism just brings us to the uniprocessor speed of the C-based implementation. Furthermore, the issues in parallelizing an optimized implementation are different from those in an unoptimized implementation, because only very limited overheads can be tolerated in an optimized implementation.

The second observation we make is that it is possible to obtain significant speed-ups for OPS5 using fine-grained parallelism on a shared-memory multiprocessor. We get up to 12.4 fold speed-up for Rubik using 13 match processes. However, this does not work for all programs. The Tourney program, because of the presence of cross-products [4], resisted all our attempts to obtain higher speed-up. The average length of the individual tasks in our parallel implementation varies between 100-700 machine instructions for the three programs that we studied. In trying to exploit this fine-grained parallelism, we found that the scheduling of tasks on processors was a major bottleneck. We found it essential to use multiple task queues (instead of a single task queue) to obtain reasonable speed-up. For the Rubik program, going from one task queue to multiple task queues increased the speed-up from 6.3-fold to 11.4-fold.

The other variation that we explored to reduce the contention for shared data structures was in the complexity of locks used for hash-based memory nodes. We used both simple spin-locks and complex multiple-reader-single-writer locks. We observed that special note must be taken of *rare-case* versus *normal-case* execution. Trying to handle rare cases efficiently can slow down the normal case, and can result in overall poorer performance. For example, the provision of complex hash-table locks reduced the contention for the hash-table buckets, but it slowed down the overall execution speed of the programs.

6. Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Air Force Avionics Laboratory under Contract N00039-85-C-0134 and by the Encore Computer Corporation. Anoop Gupta is also supported by DARPA contract MDA903-83-C-0335 and an award from the Digital Equipment Corporation.

References

1. Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, 1985.

2. Charles L. Forgy. The OPS83 Report. Tech. Rept. CMU-CS-84-133, Carnegie-Mellon University, Pittsburgh, May, 1984.

3. Anoop Gupta, Charles Forgy, Allen Newell, and Robert Wedig. Parallel Algorithms and Architectures for Production Systems. 13th International Symposium on Computer Architecture, June, 1986.

4. Anoop Gupta. *Parallelism in Production Systems.* Ph.D. Th., Carnegie-Mellon University, March 1986. Also available from Morgan Kaufmann Publishers Inc..

5. Anoop Gupta, Charles Forgy, Dirk Kalp, Allen Newell, and Milind Tambe. Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis. To appear in International Journal of Parallel Programming.

6. Bruce K. Hillyer and David E. Shaw. "Execution of OPS5 Production Systems on a Massively Parallel Machine". *Journal of Parallel and Distributed Computing 3* (1986), 236-268.

7. Rostam Joobbani and Daniel P. Siewiorek. Weaver: A Knowledge-Based Routing Expert. Design Automation Conference, 1985.

8. Peter M. Kogge. "An Architectural Trail to Threaded-Code Systems". *Computer March* (1982).

9. Edward J. Krall and Patrick F. McGehearty. "A Case Study of Parallel Execution of a Rule-Based Expert System". *International Journal of Parallel Programming 15*, 1 (1986), 5-32.

10. Theodore F. Lehr. The Implementation of a Production System Machine. Hawaii International Conference on System Sciences, January, 1986.

11. Daniel P. Miranker. *TREAT: A New and Efficient Algorithm for AI Production Systems.* Ph.D. Th., Columbia University, 1987.

12. Kemal Oflazer. Parallel Execution of Production Systems. International Conference on Parallel Processing, IEEE, August, 1984.

13. Raja Ramnarayan, Gerhard Zimmerman, and Stanley Krolikoski. PESA-1: A Parallel Architecture for OPS5 Production Systems. Hawaii International Conference on System Sciences, January, 1986.

14. M.F.M. Tenorio and D.I. Moldovan. Mapping Production Systems into Multiprocessors. International Conference on Parallel Processing, IEEE, 1985.

A Taxonomy of Synchronous Parallel Machines

Lawrence Snyder
Department of Computer Science
University of Washington
Seattle, Washington 98195[1]

Abstract: A new classificational scheme is presented which is consistent with Flynn's taxonomy but is more expressive. The crucial idea is to recognize that a reference stream is composed of both values and addresses; their treatment exposes critical features of an architecture. This insight, together with the accompanying formal mechanism built on top of it, enables a large variety of recently developed (since Flynn's work) machines to be distinguished, including VLIW, multigauge, systolic arrays, and the Connection Machines. Though the resulting taxonomic structure is illuminating, the most important result of the classification is the discovery that synchronous execution is NOT a defining property of computer architectures, but is a derived property, a consequence of other architectural features. The evidence for this result and the consequences for machine classification are presented.

1 Introduction

In 1966 Flynn [1] introduced his classification of computers. This taxomony proved to be very useful, giving us terminology like SIMD and MIMD that endures to this day. The taxonomy, however, has long been described as too coarse, unable to distinguish between computers that seem to computer architects to be quite different. Though other classifications have been offered [2-5], the fact that Flynn's classification has lasted for so long without being replaced and enhanced is a testament to the difficulty of discovering something better.

In this paper a new taxonomy is presented for synchronous parallel computers. It has no pretentions of being complete nor of capturing all features of synchronous parallel computers. The taxonomy does clarify important distinctions among recently developed parallel computers, such as the VLIW machines, multigauge machines and certain SIMD machines such as the Connection Machines.

The key idea of the taxonomy is to quantify the components of the fetch/execute cycle that process I-streams and D-streams. To make fine distinctions among machines, one must separate these reference streams into their address and value components, because addressing and value processing are crucial features by which machines differ.

Using this kind of analysis, a taxonomy is constructed. Many of the machines that are placed into different classes here would have been classified by Flynn's scheme as SIMD, so this approach permits finer distinctions to be made. Only a small number of classes have been described, and only one or two machines per class have been identified. Thus, there remain substantial opportunities for further research.

Perhaps the most important result derived from the taxonomy concerns the property of "synchroneity". The author and apparently many other researchers have treated synchroneity as a primary classificational property; we have spoken of "the synchronous vs. the asynchronous" machines as if this should be an important way to distinguish between machines. It is not. The criterion used for classifying machines in this taxonomy tells when a machine must have all of its instructions start at the same time, and when it is not necessary. This determination is based on how the machine addresses and processes instructions and data. Machines which must begin all instructions at the same time will automatically be synchronous; for those machines that need not begin their instructions at the same time it is an "engineering decision" whether to make them synchonous or asynchronous. Thus the quality of being synchronous is a derived property: A machine must have it because of other features, or it is a noncritical implementation feature.

2 Preliminaries

A *reference stream*, S, of a computer is a finite set of infinite sequences of pairs,

$$S = \{\ (a_1 < t_1 >)(a_2 < t_2 >)\ .\ .\ .,\\ (b_1 < u_1 >)(b_2 < u_2 >)\ .\ .\ .,\ .\ \ .\ \ .,\\ (c_1 < v_1 >)(c_2 < v_2 >)\ .\ .\ .\}$$

the first component of each pair being a nonnegative integer, called an *address*, and the second component being an n-tuple of nonnegative integers, called *values*, such that n is the same for all tuples of all sequences. An element of a reference stream is called a *reference sequence*. An *I-stream* is a reference stream whose values are interpreted as *instructions*; a *D-stream* is a reference stream whose values are interpreted as *data*.

The interpretation of these definitions is simple. The elements of reference sequences are address, value pairs, the values simply being the contents fetched from (or stored to) memory at the address. A sequence of elements can be thought of as the history of the addresses and values moving between a processor and its memory space. An I-stream is made up of a finite set of these sequences, the number depending on how many instruction sequences the machine can process at one time; and a D-stream is made of a finite set of data sequences, the number depending on how many distinct operations the machine can perform at one time.

Although the I-streams and D-streams have been defined in an intuitive manner, their form is not convenient

[1] This research funded in part by the Office of Naval Research Contract N00014-86-K-0264, National Science Foundation Grant CCR-8416878 and Air Force Office of Scientific Research Contract 88-0023.

for analysis. Accordingly, the following reassociation must be performed. Let

$$S = \{ (a_1 < t_1 >)(a_1 < t_2 >) \ldots, \\ (b_1 < u_1 >)(b_2 < u_2 >) \ldots, \ldots, \\ (c_1 < v_1 >)(c_2 < v_2 >) \ldots \}$$

be a reference stream. Define two sequences: The *address sequence of* S, denoted S_a, is a sequence whose i^{th} element is a tuple formed from the addresses from the i^{th} elements of each sequence of S,

$$S_a = < a_1 b_1 c_1 >, < a_2 b_2 c_2 >, \ldots$$

and the *value sequence of* S, denoted S_v, is the sequence whose i^{th} element is a tuple formed by concatenating the value tuples from the i^{th} elements of each reference sequence of S,

$$S_v = < t_1 u_1 v_1 >, < t_2 u_2 v_2 >, \ldots .$$

Notice that although a reference stream is a set of sequences, address and value sequences are just sequences of tuples.

It is possible to interpret these definitions as grouping the corresponding addresses and corresponding value tuples of a reference stream S into S_a and S_v, respectively.

Let S_x be a sequence of n-tuples; the *width* of the sequence, $w(S_x) = n$.

Proposition 1: Let S be a reference stream with n-tuple values, then

$$w(S_a) = \mid S \mid \text{ and } w(S_v) = n \mid S \mid$$

where | X | denotes the cardinality of the set X.

A *computation* is a pair (I,D), where I is an I-Stream and D is a D-stream. Computers are classified by the computations they execute. A computer *executes* the computation (I,D) provided it presents $w(I_a)$ instruction addresses to memory to be fetched simultaneously, it decodes and interprets $w(I_v)$ instructions simultaneously, it presents $w(D_a)$ operand addresses to memory simultaneously, and it performs $w(D_v)$ operations on distinct data values simultaneously. The computer is described by the notation

$$I_{w(I_a)w(I_v)} \; D_{w(D_a)w(D_v)}.$$

Notice that we speak of *the* computation executed by a computer. This is a definitional simplification, and is sufficient since any desired sequence of instructions or data is a subsequence of the infinite streams of the computation. Observe the relationship between this point and the Enumeration Theorem of recursive function theory.

Let d_1, d_2, d_3 and d_4 be predicates called *class designators*; then a machine is said to be member of the *class* denoted by

$$I_{d_1 d_2} D_{d_3 d_4}$$

if and only if $d_1(w(I_a)), d_2(w(I_v)), d_3(w(D_a))$, and $d_4(w(D_v))$. (Commas may occasionally be inserted between the subscripts for clarity.)

Example 2: By appropriating for our class designators Flynn's "s" and "m" to denote the predicates *"is-one"* and *"is-many"*, it is possible to classify some familiar machines using the mechanisms developed so far.

Let a von Neumann machine, which Flynn classified as SISD, execute the computation (I,D). From his classification we have

$$\mid I \mid = \mid D \mid = 1.$$

By Proposition 1, then, we have

$$w(I_a) = w(D_a) = 1.$$

Moreover, since instructions are decoded serially, $w(I_v) = 1$ and since they are executed serially, $w(D_v) = 1$. Therefore the von Neumann machine is described as

$$I_{1,1} D_{1,1}.$$

It is classified with the present notation as

$$I_{ss} D_{ss}$$

since the predicate "s"is true for all four widths.

Now consider two machines that Flynn's taxonomy lumped in the SIMD category, the MPP and the Illiac IV. (Ignore for the moment the fact that these have bit serial and word parallel PEs, respectively.) The single instruction stream means $\mid I \mid = 1$ for both machines. By the same reasoning just used for the von Neumann machine, the instruction streams for both machines are described as I_{ss}.

For data, consider the MPP first. Recall that the MPP controller broadcasts the same data memory address to all PEs [6], and so the machine has a single D-stream in *our* terminology; $\mid D \mid = 1$. However, a value is fetched from each PE memory, so the values of this stream are 16384-tuples. Thus,

$$w(D_a) = 1 \text{ and } w(D_v) = 16384,$$

which certainly satisfies the "multiple" class designator. So, the MPP is described as

$$I_{1,1} D_{1,16384}$$

and is classified as

$$I_{ss} D_{sm}.$$

The MPP has a "multiple data stream" but the multiplicity applies only to the data values, not to the data value addressing.

For the Illiac IV on the other hand, the controller broadcasts a base address to all PEs, each of which may produce its own address by adding in the contents of a local index register [7]. This means that $\mid D \mid = 64$; there are 64 operand address streams simultaneously produced by the machine and each of them references a single value, i.e. each data address is associated with a 1-tuple. Accordingly,

$$w(D_a) = w(D_v) = 64$$

and the Illiac IV is described as

$$I_{1,1}\ D_{64,64}$$

which places it in the

$$I_{ss}D_{mm}$$

class. It has "multiple data streams" too, but its multiplicities are for addressing and data reference. Clearly, the present taxonomy retains the distinctions achieved by Flynn, but it is also capable of making finer distinctions.

3 Discussion

It is possible to give an intuitive interpretation to much of the foregoing formalism. The key idea is to recognize that the formalism quantifies funtional components of a fetch/execute cycle. Thus, the machine described as

$$I_{av}D_{a'v'}$$

presents instruction addresses to memory for a threads of control (presumably from a PCs but data flow computers qualify as well); it receives v different instructions back from memory at once and interprets them; it presents a' different operand addresses to memory for data values, and it receives v' data values back and operates upon them concurrently. So, when the MPP is described as

$$I_{1,1}D_{1,16384}$$

it is immediately obvious that its PEs all use the same address for accessing their operand values, even though they are capable of independently performing operations on the resulting data.

The interpretation of the classification is intended to carry the implication that if the n-tuple of values $< t_i >$ is received from memory upon presentation of address a_i then the machine is capable of processing all n elements at once. This applies to both instructions and data. So even if a computer makes a memory reference to address a_i and fetches k words, perhaps to cache them, if it only processes one of them, then $n = 1$ in this model.

Finally, notice that our classificational scheme is a completely formal system with a precise meaning. Its utility in classifying computers depends entirely on our interpreting this formalism as meaningful. Though it is possible for two scientists to differ in their interpretation, and thus to differ on a classification, the underlying scheme is unambiguous.

4 Properties of Address and Value Sequences

To simplify discussing computer families, it is convenient to adopt a simple abbreviation. The expression

$$I_{d_1d_2}D_{d_3d_4}$$

will be abbreviated by the string

$$d_1d_2d_3d_4$$

Thus, the von Neumann machine class is abbreviated *ssss*, while the MPP is in *sssm*. String expressions will be used as shorthand to abbreviate several classes.

There are several important properties of this taxonomic system which influence the kind of machine classes definable.

Proposition 3: Any machine $I_{av}D_{a'v'}$ satisfies the inequalities:

$$a \leq v \text{ and } a' \leq v'$$

These inequalities follow from the fact that in a reference stream every address is paired with at least one value, so the width of the address stream is a lower limit to the width of the corresponding value stream. The interpretation of these inequalities seems intuitively correct: The number of addresses presented to memory should never exceed the number of values returned. As a corollary, any nonempty machine class will satisfy these relationships, where the definition of the relation is suitably extended.

Convention 4: Any machine $I_{av}D_{a'v'}$ will satisfy the inequality:

$$v \leq v'$$

Unlike the preceding propositions which are artifacts of the taxonomy's abstraction, this convention is adopted primarily for semantic consistancy. Its interpretation is that the number of instructions being interpreted should not exceed the data available.

Since it is a convention, it is open to debate. On the positive side the convention helps avoid "problem" machines like Flynn's MISD; this machine doesn't make much sense and has often been criticized. Here, the convention is worthwhile, considering that the finer control of this taxonomy permits greater opportunities to create such dubious classes. On the negative side, adopting the convention might prevent accurately describing certain machines, though none has come to the author's attention. Since a taxonomy is descriptive (as opposed to being prescriptive) and given that architects are not likely to have their creativity constrained by this convention, we adopt it.

5 More Machine Classes

The efficacy of a classificational system usually depends to some degree on interpretation. (It always does in biological taxonomy.) Usually there is a large range (sometimes a continuum) of values that a property can assume, and we wish to assign certain segments of this range to different classes. But there may not be any effective way to identify the boundaries of these ranges, and so membership is often a matter of judgement. This characteristic will persist for this taxonomy, but confusion can be minimized by being somewhat more precise about the terminology that we've already used.

Define the class designators as follows:

- s is the predicate "equals 1",
- c is the predicate "from 1 to some (small) constant", and
- m is the predicate "from 1 to an arbitrarily large finite number"

Though the c and m designators have no upper limit in principle, they are intended to convey two different meanings. When the c designation is used the range has a hard upper limit usually due to internal constraints in the architecture and cannot be easily increased by a substantial amount. An example might be the number of instructions that can be packed in the instruction word of a VLIW machine[8]; for any given word size it is fixed, and even though the word size can be increased this is probably not the intended nor the rational way to generalized the given machine. The m designation, however, is used when the quantity can be easily generalized or scaled. An example is when additional PEs can be added as with the MPP.

These distinctions are not always clear, of course, and judgement must be applied. An example is the question of how to classify a machine with processors connected to a bus [9]. In principle, there is no limit to how many processors can be attached to a bus, but with the addition of each processor the congestion increases, and this is an internal constraint reducing the performance. Is this a "c" or "m" case? Arguments can be made on both sides; we leave the question open for the moment.

It is now possible, using the class designators, Proposition 3 and the convention to define a number of machine classes. Notice that there is no attempt to be complete in either defining classes or categorizing machines:

$I_{ss}D_{ss}$ von Neumann machines.

$I_{ss}D_{sc}$ "packed" von Neumann[10]; the machine can fetch several distinct data values from fixed postions from one address and simultaneously apply the same operation to them. Many machines have some instructions of this form, e.g. performing 2 half word adds on the word at a given address; all (ALU) instructions for a machine in this class would have this capability.

$I_{ss}D_{sm}$ SIMD Parallel Machines with no addressability, such as the MPP, the Connection Machine 1 [11] and systolic arrays [12].

$I_{ss}D_{cc}$ SIMD Multigauge machines [13]; these are von Neumann machines which can (optionally) split their datapath to process multiple, independent operand streams at once.

$I_{ss}D_{mm}$ Addressable SIMD Parallel Machines, such as Illiac IV and the CM2 [14].

$I_{sc}D_{cc}$ VLIW Machines [8]; the machine fetches and executes several instructions stored in one instruction address.

$I_{cc}D_{cc}$ MIMD Multigauge machines [13]; these are von Neumann machines which can (optionally) split their fetch/execute cycle to process multiple, independent instructions concurrently.

$I_{mm}D_{mm}$ MIMD Parallel Machines, including machines such as the Ultracomputer[15] and the Cosmic Cube[16].

Clearly, the list is not complete in terms of either the classes listed or the machines recognized as members of any given class. Much work remains.

6 Discussion of the Taxonomy and The Origins of Synchronous Computation

One is struck by at least two aspects of the foregoing classification: A large and diverse set of machines are lumped into the last classification, *mmmm*, and nowhere in the taxonomy has the synchronous requirement been mentioned, except in the paper's title. These two observations are related.

In effect the taxonomy uses as its "criterion for classification" the number of repeated instances of the principle functional activities of the fetch/execute cycle. So, machines are distinguished by how many instructions they can decode at once or how many operations on separate data they can perform simultaneously. But these are not the characteristics we think of as distinguishing the different MIMD parallel computers. Rather, we think of them as being different depending on whether or not they have global shared memory or what their interconnection topology is. These are features unrelated to the fetch/execute cycle. So, lumping MIMD parallel computers in the *mmmm* class says only that by the criterion applied, they are all equivalent.

This is unsurprising and is not evidence of weakness in the classification. Indeed, it might point to why efforts to find criteria suitable for classifying all parallel machines have so far been unsuccessful: Qualities that are important for some computers are for other computers, unimportant, irrelevant or even misleading as a guide to classification. To the extent that the taxonomy provides insight by its classifications, the "criterion" amounts to being a useful way of looking at some computers. (For cases where topology matters for sychronous machines, e.g. between the MPP and the CM1, see the next section.)

Interestingly, the "criterion" apparently mandates the synchronous property. By using "the number of repeated instances of the principle functional activities of the fetch/ execute cycle" as the basis for classification, we are thinking of machines as either a single f/e cycle that has certain components replicated, or multiple copies of a f/e cycle. In the former case (all legal machines of the form *sy*, $y \in \{s,c,m\}^3$) synchronous execution is mandated because there is only one cycle running. With the current structure of the taxonomy this leaves only the *cy* classes and *mmmm*. Though there is no requirement in the model that these be synchronous, the class designators provide some basis for deciding: The c designation carries with it the implication of "other constraints" limiting the extent to which the f/e cycle can be replicated; it is reasonable to presume that these constraints might well mandate synchronous execution. (For example, they do mandate synchronous execution for the MIMD Multigauge members of the *cccc* class.) However the *m* designation carries no such constraints and the possibility of arbitrary scaling would seem to imply a weaker coupling consistant with asynchronous execution.

Thus, except for the mmmm *machines, these computers are synchronous* because *of the structure of their reference streams, and not the other way around.* Synchronous execution is a derived property. *Indeed, the* mmmm *class is the only one where there is a possibility for making an "engineering" choice between synchronous and asynchronous.*

7 Applications of the Taxonomy

So far the foundations of a classificational scheme have been laid, a taxonomy of synchronous machines defined, and a number of machines assigned to classes. This is about all that can be presented here. The taxonomy's structure is more extensive and the analysis is more complete. For example, consider the following:

The classification has thus far clustered machines together in broad groups, but in doing so it has glossed over significant differences among them. For example, the Illiac IV and the CM2 (Connection Machine 2 [14]) are both in the class *ssmm*, yet they have the following differences, among others, some or all of which are probably significant:

	Illiac IV	CM2
Number of PEs	64	65,536
Datapath width (bits)	64	1
Freq'cy PE gen'ates op'nd addrs (instr.)	1	32
Topology	"torus"	trunc. bin. cube

Blurring such distinctions is a necessary role for taxomonies: The higher the classificational level the greater the distinctions being ignored. When detail matters, however, these distinctions must be expressed. The classifictional scheme has been extended to capture such things as the size of the data items, the bandwidth to memory, topology, etc. These added features not only permit a description of the differences between the Illiac IV and the CM2, but they have suggested apparently new architectures [17].

8 Acknowledgment

The results described here have their antecedents in recent work with Chyan Yang, and years earlier, with Dan Reed. It is a pleasure to thank these gentlemen for the many hours of pleasant and incisive technical discussions which have obviously influenced this research.

9 References

[1] M. J. Flynn, Very High Speed Computing Systems, *Proceedings of the IEEE*, 54:1901-1909, 1966

[2] T. Y. Feng, Some Characteristics of Associative/Parallel Processing, *Proceedings 1972 Sagamore Computer Conference*, Syracuse University, pp. 5-16, 1972

[3] W. Handler, The Impact of Classification Schemes on Computer Architecture, *Proceedings International Conference on Parallel Processing*, IEEE, pp. 7-15, 1977

[4] D. Siewiorek, C. G. Bell, and A. Newell, *Principles of Computer Structures*, McGraw-Hill, 1980

[5] Kai Hwang and Faye A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 32-40, 1984

[6] Kenneth E. Batcher, The design of a massively parallel processor, *Transactions on Computers* C-29:836-840, IEEE, 1977

[7] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, A. H. Sameh and D. L. Slotnick, The Illiac IV System, *Proceedings of IEEE* 60(4):369-388, 1972

[8] J.A. Fisher, The VLIW Machine: A Multiprocessor for Compiling Scientific Code, *Computer* 17(7), 1984.

[9] G. Fielland, D. Rogers, 32-bit computer system shares load equally among up to 12 processors, *Electronic Design*, pp. 153-68, 1984.

[10] L. Snyder, An Inquiry into the Benefits of Multigauge Parallel Computation,*Proceedings of the International Conference on Parallel Processing*, IEEE, pp. 488-492, August 1985.

[11] W. Daniel Hillis, *The Connection Machine*, The MIT Press, 1985.

[12] H. T. Kung and C. E. Leiserson, Systolic Arrays, In, Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison Wesley, 1980

[13] Chyan Yang, An Investigation of Multigauge Architectures, *Technical Report* 87-10-05, Department of Computer Science, University of Washington, Ph.D. dissertation, 1987.

[14] Thinking Machines Corporation Connection Machine Model CM-2 Technical Summary, *Technical Report*, HA87-4, 1987.

[15] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, *Transactions on Computers*, C-35(2):175-189, 1983

[16] C. E. Seitz, The Cosmic Cube, *Communications of the ACM* 28(1):22-33.

[17] Lawrence Snyder, A Taxonomy of Computers, In preparation.

PARALLEL EXECUTION SCHEMES IN A PETRI NET

Won Ho Chung[*], *Ha Ryoung Oh*[*], *Hyung Lee-Kwang*[**], *Kyu Ho Park*[*] and *Myunghwan Kim*[*]

[*] Department of Electrical Engineering, KAIST,
P.O. Box 150 Cheongyangni, Seoul, Korea

[**] School of Electronic and Computer Science, KIT,
300-31, Daejeon, Choong-Nam, Korea.

ABSTRACT - In constructing modules for parallel execution, an important thing to be considered is what execution scheme will be taken for parallelism. In this paper, we develop some parallel execution schemes in the Petri net model of a task and present a module construction algorithm for each scheme. The maximum firing rule is used for large amount of parallelism. This is useful in the early stage of software development for a multiprocessor.

1. INTRODUCTION

In a parallel processing system, a task is assumed to consist of modules interconnected with each other and a module consists of actions. Petri nets and related graph models provide an important formalism for modeling and analyzing asynchronous and concurrent activities of a task. These models have been widely used for representing and analyzing those tasks in various applications [1]-[3]. However, little work has been done for constructing modules for parallel execution, which can be run on a multiprocessor. There is an important thing to be solved, *i.e.*, what execution scheme will be taken for the parallelism. An execution scheme was proposed in [4] for this problem. But, that scheme requires a lot of synchronization, so the timing overhead for task control becomes large when it is implemented in a multiprocessor.

In this paper, we develop some Petri net based parallel execution schemes and present an algorithm for constructing modules by using each of the schemes. They are focused on increasing asynchronous activity and thus reducing the control overhead. The maximum firing rule is used for large amount of parallelism. Further, how the execution scheme will be realized as a software, which is effective in the respects of modularity, modifiability, and so on, is considered. It is based on the hierarchical decomposition technique proposed in [5]. Thus, if a task can be represented by a tree-structured form with a certain hierarchy, task assignment in a multiprocessor with arbitrary number of processors becomes no longer the NP problem [6]. We omit the definitions of Petri nets and related details including languages, and follow ones in [7].

2. PRELIMINARIES

In a Petri net, a place p is an input (or an output) place of a transition t if and only if there exists at least an arc (p, t) (or (t, p)). The bag of all input (or output) places of a transition is called the *precondition* (or *postcondition*) and denoted as I(t) (or O(t)). Thus, a marked Petri net N is defined as a 5-tuple N = (P, T, I, O, M_0). We consider the places with non-zero markings for the representation of a state. Thus, the initial marking of the net in Fig. 1, is represented by a state $\{3p_1\}$. We regard the firing of a transition as an action and a module consists of actions.

Now, we give a formal definition of maximal parallelism. It is justified by the maximal-firings of which each element is a maximal set of transitions firable in a state.

Definition-1 (Maximal-firings M_f)
Let N = (P, T, I, O, M_0) be a marked Petri net, S be a state represented by a bag of places and ρ(y) be the power set of a set y. All the enabled transitions in the state S, M_e(S) is the set defined as

$$M_e(\text{S}) = \{t_i \in \text{T} \mid \text{I}(t_i) \subseteq \text{S}\}$$

The maximal-firings in S, M_f(S) is the set of all maximal X's over M_e(S) such that

$$\text{X} \in \rho(M_e(\text{S})) \text{ and } (\textstyle\sum_{t_i \in X} I(t_i)) \subseteq \text{S}.$$

Remarks for justification : Obviously, the M_e(S) is the set of all the transitions enabled in the state S but may not be simultaneously firable, because some transitions may share their input places. Therefore, every set of simultaneously firable transitions in S must reside in the power set ρ(M_e(S)). Since each element X of the M_f(S) is defined as one of the elements in the power set, its domain is M_e(S). Moreover, since an element X must satisfy the condition $(\sum_{t \in X} I(t)) \subseteq$ S, all transitions in X are simultaneously firable in S. Finally, we select only maximal ones among such X's, therefore M_f(S) is the set of all maximals of simultaneously firable transitions. Note that transitions which share input places need not be in conflict. If the marking has sufficient tokens in the shared input places to enable each contending transition individually, then those transitions can fire simultaneously ||

All the transitions in each element of M_f fire simultaneously, thus we call such a firing scheme *maximum firing rule*.

Some concurrency metrics are extracted from M_f(S) and each state of a Petri net in execution is characterized by using them. Note that the M_f(S) has the general form of

$$M_f(\text{S}) = \{X_1, X_2, \cdots, X_N\}, \text{ where } 1 \leq \text{N} \leq |\rho(M_e(\text{S})|,$$

and each $X_k \in M_f$(S) has the form of

$$X_k = \{t_{k0}, t_{k1}, \cdots, t_{kn}\}, \text{ where } 0 \leq \text{n} \leq |\text{T}|.$$

Definition-2 (Concurrency Metrics)
Let M_f(S) = $\{X_k \mid 1 \leq \text{k} \leq \text{N} \text{ integer}\}$.
(a) *degree of concurrency* : C(S) = $\sum_{k=1,N} |X_k|$,
(b) *degree of decision* : D(S) = $|M_f(\text{S})|$,
(c) *concurrency to decision ratio* : CDR(S) = C(S)/D(S).

For example, in the Petri net shown in Fig. 1, suppose S_1 is given by $\{p_1, p_2, p_3, p_4\}$, then $M_f(S_1) = \{\{t_1, t_3\}, \{t_2, t_3\}\}$. In this case, C(S) = 4 and D(S) = 2 so that CDR(S) is 2, which means that the average number of transitions which can fire simultaneously in S_1 is two.

According to C(S) and D(S), each state of a Petri net in execution under the maximum firing rule can be classified into five classes such as *class-0, class-1, class-2, class-3 and class-4* as shown in Table-1 [8]. Every state of a marked Petri net in execution under the maximum firing rule belongs to one of these classes. They show *what* transitions and *how (sequentially or disjunctively or concurrently)* they can fire.

3. PARALLEL EXECUTION SCHEMES

Three parallel execution schemes are developed and a module construction algorithm for each of them is presented. We use the classes of states for describing those construction algorithms, because each state of a Petri net in execution can be identified by its class. First, two partial states called the firing state and idle state are defined and a state S is decomposed to them.

Definition-3
For a given state S, the firing state and the idle state for each $X_k \in M_f$(S) are defined as $S_f^k = (\sum_{t \in X_k} I(t))$ and S_i^k = S − S_f^k respectively, where each state is a bag of places.

As shown in Table-1, since there is no parallelism in those states of the class-1 and class-2, execution schemes for them are sequential or disjunctive. The class-4 is just the combined case with the class-2 and class-3. Thus, class-1, class-2 and class-4 are excluded in our discussion of parallel execution schemes. From now on, we only consider the class-3 and each procedure corresponding to class-1, class-2 and class-4 is separately described in Fig. A1.

```
Procedures for class-1, class-2 and class-4
    Class-1 : (* $M_f(S) = \{\{t\}\}$, deterministic and sequential *)
      begin
          $S_f = I(t)$ ; $S_i = S - S_f$ ; execute the action t ;
          $S' = O(t) + S_i$ ; invoke this algorithm for S' ;
      end
    Class-2 : (* $M_f(S) = \{\{t_1\}, \{t_2\}, \ldots, \{t_N\}\}$, disjunctive and sequential *)
      for k = 1 to N
        begin
          $S_f^k = I(t_k)$ ; $S_i^k = S - S_f^k$ ; execute the action $t_k$ ;
          $S^k = O(t_k) + S_i^k$ ; invoke this algorithm for $S^k$ ;
        endfor
    Class-4 : (* $M_f(S) = \{X_1, X_2, \cdots, X_N\}$, disjunctive and concurrent *)
      for k=1 to N
            follow the procedure Class-2 or Class-3 according to $|X_k|$ ;
            (* The procedure Class-3 will be described in each scheme *)
      endfor
```

Fig. A1. The procedures for class-1, class-2 and class-4.

3.1 Lock-Step Synchronization (LSS) Scheme

The LSS scheme is based on a *lock-step manner with a single-step parallelism*. The scheme for a state S belonging to the class-3 is as follows: 1) find $M_f(S)$, 2) decompose S to the firing state and the idle state, 3) all the actions in $X \in M_f(S)$ are executed simultaneously and 4) a next state is generated by merging the resulting output states ($O(t_k)$) with the idle state. If the next state is belongs to class-3, this procedure is continued with the next state, otherwise follow the procedures in Fig. A1. Therefore, if the class-3 states are continued, the module construction using the LSS scheme is represented as a task σ_{LSS} by using a prefix language.

$$\sigma_{LSS} = (\cdot \;\; m_1 \;\; m_2 \cdots m_n), \tag{3-1}$$

and each m_i is

$$m_i = (|| \;\; \omega_1 \, \omega_2 \cdots \omega_k), \tag{3-2}$$

where the dot (·) and a double-bar (||) are used as control operators representing sequential and concurrent operations respectively. In (3-2), each ω_i is a single action. Thus, the module construction with this scheme is represented by a sequence of parbegin/parend of actions. Suppose that a multiprocessor consists of a single control unit (CU) and related number of processing elements (PE), and they are interconnected by an appropriate communication network. An advantage of this kind of parallelism is that the module construction for a task is easily accomplished in the CU, and thus its allocation to the related number of PE's is simple. However, this scheme has a major defect that the synchronization must take place whenever each module m_i is completed, and then another module m_j is allocated to the PE's by the CU. The number of synchronizations is large. Moreover, in constructing modules for parallel execution with this scheme, the completion of each module m_i will take the maximum time among the parallel actions, *i.e.* $T(m_i) = \max\{T(\omega_i)\} + \alpha$, where α is the time penalty required for a synchronization. The control overhead of parallel execution for a task with this scheme is too high if the class-3 states are continued.

Now, we present two other execution schemes for reducing the number of synchronizations.

3.2 Partial State Branching (PSB) Scheme

This PSB scheme, in contrast with the LSS scheme, exploits asynchronous parallelism in depth to reduce the overhead caused by large number of synchronizations in the LSS scheme. The basic idea is when concurrent actions are executed in parallel, the local output state $O(t_k)$ of each action is independently branched and the M_f is exploited in each of the local output states. The same procedure is repeated until each state is terminated (class-0) or duplicated. An algorithm for constructing modules with this scheme is shown in Fig. A2.

```
<ALGORITHM PSB>
(* HLIST and QLIST are global data structures*)
(* and they are initialized to be empty *)
(* The HLIST is a buffer for storing those states already examined, *)
(* and the QLIST is a fifo queue storing those states to be examined *)
Step 1: if ($S \in$ HLIST) then return ; (* duplicated state is excluded *)
        else save S into HLIST ;  (* for duplication check *)
Step 2: Find the $M_f(S)$ and the CLASS of S ;
Step 3: case CLASS
          Class-0 : break ;  (* terminated state *)
             (* refer to Fig. A1 for class-1, class-2 and class-4 *)
          Class-3 : (* $M_f(S) = \{\{t_1, .. , t_n\}\}$ : deterministic-concurrent *)
           begin
             (* Parallel module, use double bars*)
             $S_f = \sum_{k=1,n} I(t_k)$ ; $S_i = S - S_f$ ;
             for k = 1 to N
               begin
                 execute the action $t_k$ ; $S^k = O(t_k)$ ;
                 invoke this algorithm for $S^k$ ;
               endfor
           endclass
        endcase
        $S' = \sum_{k=1,n} O(t_k) + S_i$ ;
        invoke this algorithm for S'
endPSB.
```

Fig. A2. A parallel module construction algorithm with PSB scheme.

If we also assume that the class-3 states are continued, then the module construction by using the PSB scheme will be described by a task σ_{PSB}

$$\sigma_{PSB} = (\cdot \;\; m_1 \;\; m_2 \cdots m_n \;) \tag{3-3}$$

and each m_i is also expressed by

$$m_i = (|| \;\; \omega_1 \, \omega_2 \cdot\cdot \, \omega_k \;), \tag{3-4}$$

but $\omega_i = a_i$ is a single action or $\omega_i = (\cdot \; a_i^1 \, a_i^2 \cdot\cdot \, a_i^j)$ is a sequential module consisting of actions or $\omega_i = (|| \; a_i^1 \, a_i^2 \cdot\cdot \, a_i^k)$ is a parallel module consisting of actions. The difference between the modules constructed by the LSS and the PSB can be found from (3-2) and (3-4). The ω_i in (3-2) must be a single action while the ω_i of the PSB scheme may be a sequence of actions or a parallel module as described by (3-4). This means that the asynchronousity is higher than that of the LSS scheme, *i.e.*, $|\omega_i|_{LSS} \le |\omega_i|_{PSB}$. However, we can find that subparallelism may occur in the PSB scheme because each ω_i can be a parallel module and thus each a_i can also be a parallel module. For example, the following task described by σ has such a subparallelism.

$\sigma = (\cdot \;\; (|| \;\; (\cdot \;\; a \;\; (|| \;\; (\cdot \;\; c \;\; e) \;\; (\cdot \;\; d \;\; f)) \;\; h) \;\; (\cdot \;\; b \;\; g)) \;\; i)$

When the subparallelism occurs, the number of PE's is a decision factor whether the subparallelism will be serialized or not. If it is less than the maximum edge-cut (3 in this case), some parallelism will be lost. However, if it is at least equal to three, the maximal parallelism can be achieved.

3.3 Extended Partial State Branching (XPSB) Scheme

This is an extension of the PSB for higher asynchronousity than the PSB scheme. In the PSB scheme, all states terminated locally are merged regardless of possibility of execution of another action in each terminated state (See Fig. A2). Such possibility can appear at the time when each locally terminated state is joined with a part of S_i. Hence, in the XPBS scheme, when all states are terminated locally during the PSB scheme, S_i is distributed to those terminated states and further executable actions are explored. If some actions are executable in the state joined with a part of S_i, they are further executed. More precisely, during the PSB algorithm (Fig. A2), a part of the idle state is distributed to those terminated states that can have executable actions when they are joined with a part of S_i. The idle state is changed when this kind of distribution occurs. When each terminated state is not executable any more even if the idle state is partially or totally joined, a next state is generated by merging those terminated states with the changed idle state (if it exists). When the same part of S_i denoted by P(S_i) is distributed to more than one terminated state, the following conflicts can occur.

- the executable actions are the 1) same or 2) different.

The first-come/first-fit strategy can be used for the first case, because the executable actions are the same. However, for the second case, the P(S_i) is not considered for the distribution but the other P(S_i) is considered for the distribution to those states or the other terminated states are considered for the distribution of the P(S_i). If this scheme is used for exploiting parallelism, higher asynchronousity than the PSB scheme can be obtained and thus rare synchronization is achieved, because $|\omega_i|_{PSB} \leq |\omega_i|_{XPSB}$. Since the subparallelism can also occur in this scheme, the same concept as the PSB is applied. Therefore, the module construction by using this scheme has the same descriptions as ones represented in (3-3) and (3-4), but their task sizes are different. The construction algorithm using this scheme is given in Fig. A3.

Let σ_{LSS}, σ_{PSB} and σ_{XPSB} be tasks obtained by using the LSS, PSB and XPSB schemes respectively, and let $N_s(\sigma)$ be the number of synchronizations occurring in the task σ. Then we can find that

$$N_s(\sigma_{LSS}) \geq N_s(\sigma_{PSB}) \geq N_s(\sigma_{XPSB})$$

This is appeared as an overhead due to the synchronization, because this overhead degrades the performance of a task in a parallel processing environment using those parallel execution schemes. Less the total synchronization overhead, better the performance of a system.

3.4 Parallel Behavior Representation

A labeled directed graph called an *AND/OR reachability graph* is used for representing the parallel execution of a task. It is a labeled directed graph, G = (V, E, L), where V is a set of nodes, E $\subseteq V \times V$ is a set of directed edges and L : E → T is an edge labeling function mapping a transition to each edge. Obviously, each node represents a state and each directed edge is labeled by an action. The graph is a reachability graph of a marked Petri net under the maximum firing rule, so that it must be able to describe both of parallel and disjunctive behaviors. Thus, as mentioned before, *double-bar (||)* and *plus (+)* are used for indicating a parallel module and a disjunctive module respectively. The former is assigned to the class-3 nodes and the latter is assigned to the class-2 nodes. Both of them are used for class-4 nodes because they have both operational properties of class-2 and class-3 nodes.

There are four branching rules prescribed for the construction of an AND/OR reachability graph. They are the *sequential branching with a single action (SBS)*, *sequential branching with n multiple actions (SBM)*, *disjunctive branching (DB) of n actions* and *parallel branching (PB) of n actions*. Each of the branching rules is represented graphically as shown in Fig. 2 by using the concurrency and disjunction marks (|| and +). They are described as follows:

```
<ALGORITHM XPSB>
Step 1: if (S ∊ HLIST) then return ; (* duplicated state is excluded *)
        else save S into HLIST ;  (* for duplication check *)
Step 2: Find the $M_f$(S) and the CLASS of S ;
Step 3: case CLASS
        Class-0 : break ;
            (* refer to Fig. A1 for class-1, class-2 and class-4 *)
        Class-3 : (* $M_f$(S) = {{$t_1$, .., $t_n$}} *)
         begin
           $S_f = \sum_{k=1,n} I(t_k)$ ; $S_i$ = S - $S_f$ ;
           for k = 1 to N      (* Start PSB *)
             begin
               execute the action $t_k$ ; $S^k$ = O($t_k$)
               invoke this algorithm for $S^k$ ;
             endfor
         endclass
        endcase
         (* all the states are terminated and thus $S_i$ is distributed *)
        for each $S^k$
           if $S^k$ can be executable when a P($S_i$) is joined
             then  begin
                       $S^k = S^k$ + P($S_i$) ; $S_i = S_i$ - P($S_i$) ;
                       invoke this algorithm for $S^k$ ;
                   end
        endfor
        (* there are no more executable states *)
        S' = $\sum_{k=1,n} O(t_k) + S_i$ ; (* a next state *)
        invoke this algorithm for S'
endXPSB.
```

Fig. A3. A parallel module construction algorithm with XPSB scheme.

(1) **Sequential Branching with a Single action (SBS)**
$(S = S_f + S_i) \rightarrow^{t} (S' = O(t) + S_i)$, where $S_f = I(t)$

(2) **Sequential Branching with n Multiple actions (SBM)**
$(S = S_f + S_i) \rightarrow^{t_1 || .. || t_n} (S' = \sum_{k=1,n} O(t_k) + S_i)$,
where $S_f = \sum_{k=1,n} I(t_k)$ and use the mark ||.

(3) **Disjunctive Branching (DB) of n actions**
for k= 1 to n
$(S = S_f^k + S_i^k) \rightarrow^{t_k} (S^k = O(t_k) + S_i^k)$,
where $S_f^k = I(t_k)$
endfor (use the mark +)

(4) **Parallel Branching (PB) of n actions**
for k = 1 to n
$(S = S_f + S_i) \rightarrow^{t_k} (S^k = O(t_k))$,
where $S_f = \sum_{k=1,n} I(t_k)$
endfor (use the mark ||)

As we see, there are two different cases in the sequential branching such as the SBS and the SBM which are used for class-1 nodes and class-3 nodes respectively. The SBM is the synchronous parbegin/parend represented by a single step parallelism, while the PB is an asynchronous parbegin/parend. An important aspect is that the SBS, SBM and DB generate the global states as next states while the PB generates the partial states as next states. Therefore, it is notified that the PB is necessary to describe the PSB and the XPSB. However, it is not required for the description of the LSS scheme.

The following elimination rule is also used to detect whether a node must be branched or not.

Elimination rule :
If a state is terminated or duplicated, any branching is not carried out.

The termination is detected by checking whether the state belongs to the class-0 or not and the duplication is detected by checking a queue (HLIST) which stores all the states generated before. Therefore, the elimination rule can be implemented by the procedure checking the class of a state and the queue.

The maximal-firings is computed in a current state and characterized by one of the five classes, so that the AND/OR reachability graph is dynamically constructed by the enumeration of classes of states and actions together with several data. Thus, each graph corresponding to each of the parallel execution schemes the LSS, the PSB and the XPSB can be constructed by using those algorithms represented in Fig. A1, A2 and A3. We have used the depth-first method for their constructions.

4. A PROCESS FOR SOFTWARE GENERATION

For the design representation of a software using Petri nets, various control modules have been developed for representing software structures [3], [9]. We classify them into 4 control modules such as a *begin−end*, an *if−then−else* module, a *do−while* module and a *parbegin−parend* module. Each of those control modules can be expressed by a Petri net language. To represent the PN language as a prefix language, four control operators with same precedence are used. They are the serial operator (·), the disjunction operator (+), the concurrency operator (||) and the loop operator (*) and they are used for the four control modules respectively. Note that the *do−while* module is regarded as a loop of several actions or modules with a single loop counter *lc*, and we assume that the iteration is finite. For example, a PN language $(a \cdot (b + (c \,||\, d)) \cdot e \cdot f)^{lc}$ is represented by the prefix notation (* (· a (+ b (|| c d)) e f) *lc*), where *lc* is a loop counter.

A 4-step process is proposed for realizing an architectural software.

(Step-1) Construct a AND/OR reachability graph describing a parallel execution scheme.

(Step-2) Get the Petri net language described by a prefix notation.

(Step-3) Decompose the language and generate a hierarchical relation among decomposed modules.

(Step-4) Generate a software structure with the hierarchical relation.

The steps (1) and (2) can be carried out by using the algorithms represented in Fig. A1, A2 and A3. The step (3) decomposes a Petri net language and generates a hierarchical relation among decomposed modules represented by a tree called a *fork tree*. The decomposition of a language gives us a decomposition of the task. The root node (*module*(0, 0)) of a fork tree is just the given language. In (Step-4) a hierarchically structured program for the task described by a Petri net language is generated by using the four control modules.

5. PERFORMANCE COMPARISON BY AN EXAMPLE

An example illustrating each parallel execution scheme is given for a Petri net model and performance comparison between them is discussed. The Petri net shown in Fig. 3 is the model of a simple CPU which consists of three operation cycles under the maximum firing rule. They are an arithmetic cycle (AC), a store cycle (SC) and a branch cycle (BC). All of them start from the instruction decoding action *a* and are classified by the action. We exclude the branch cycle because there is no parallelism in the cycle. The two task-cycles, which are constructed by applying the LSS, PSB and XPSB schemes to AC and SC, are represented by the Petri net languages with prefix notations as follows:

For the LSS scheme,

AC_{LSS} = (· a b_1 c (|| f l) g_2 i n)
SC_{LSS} = (· a b_2 d (|| g_1 l) j n)

For the PSB scheme,

AC_{PSB} = (· a b_1 c (|| (· f g_2 i) l) n)
SC_{PSB} = (· a b_2 d (|| g_1 l) j n)

For the XPSB scheme,

AC_{XPSB} = (· a b_1 c (|| (· f g_2 i) l) n)
SC_{XPSB} = (· a b_2 d (|| (· g_1 j) l) n)

Consider the arithmetic cycle (AC) for the comparison between the LSS and the PSB scheme. We can find that the task-cycle of the AC_{LSS} consists of a 7-module sequence with one parallelism, where the two actions - *f* and *l* - are executed simultaneously. And the task-cycle of the AC_{PSB} consists of a 5-module sequence, where the module (· f g_2 i) and the action *l* can be simultaneously executed. Therefore the asynchronousity for the PSB is higher than that of the LSS. Furthermore, the execution according to the PSB is equal to or faster than that according to the LSS when they are implemented in a 2-processor system, because the execution of the module (· (|| f l) g_2 i) takes longer time than that of the parallel module (|| (· f g_2 i) l).

Consider the store cycle (SC) for the comparison between the PSB and XPSB schemes. The task-cycle of SC_{PSB} consists of a 6-module sequence, where the two actions g_1 and *l* can be executed in parallel, while the task-cycle of SC_{XPSB} consists of a 5-module sequence, where the module (· g_1 j) and the action *l* can be executed in parallel. The asynchronousity for the XPSB is higher than that of the PSB. In the same sense, the execution according to the XPSB is equal to or faster than that according to the PSB when they are implemented in a 2-processor. Moreover, if the execution of the action *l* is longer than that of the module (· f g_2 i), the PSB scheme is much better than the LSS. Also, if the execution of the action *l* is longer than that of the module (· g_1 j), the XPSB is much better than the PSB.

From this example, we can find that, the PSB scheme is more efficient than the LSS, and the XPSB than the PSB due to higher asynchronousity. With enhancement of asynchronousity in the PSB and the XPSB schemes, the timing overhead for task control in a multiprocessor is reduced from the LSS scheme. Each of three parallel execution schemes describes the parallelism for a special computer architecture. For example, the LSS scheme describes the synchronized parallelism with lock-step manner and thus it may be better that the LSS scheme is applied for SIMD machines such as array processors, while the PSB and the XPSB schemes are for a message-based distributed system or a loosely-coupled multiprocessor system (especially tree-structured machine) due to high asynchronousity among parallelly processable modules, and thus the overhead for the task control is reduced.

6. CONCLUSION

We have presented three parallel execution schemes in a Petri net and discussed differences between them. A module construction algorithm for each of them is given and implemented in a functional language Lisp. Main consideration is to increase the asynchronous activity for reducing the number of synchronizations when a task is executed in a multiprocessor. The extent of asynchronousity differs by an employed scheme, but their parallelism is kept to be maximal, because the execution of a Petri net follows the maximum firing rule characterized by simultaneously executing the maximal-firings in each state. We have shown that the timing overhead for the control of a task in a multiprocessor is reduced by increasing the asynchronousity. The schemes are followed by the hierarchical software construction which represents the dynamic behavior of a task. The overall procedure can be automated by a 4-step process. This is useful in the early stage of software development for a multiprocessor.

8. REFERENCES

[1] M. Diaz, "Modeling and Analysis of Communication and Corperation Protocols Using Petri Nets Based Models," Computer Networks, Vol. 6, North-Holland, 1982, pp.419-441.

[2] C. V. Ramarmoothy and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent System Using Petri Nets," IEEE Trans. Software Eng., Vol. SE-6, pp.440-449, Sept. 1980.

[3] T. Agerwala,"Putting Petri Nets to Work," IEEE Computer, pp. 85-94, Dec. 1979.

[4] H. D. Burkhard, "On the Priorities of Parallelism: Petri Nets under Maximum Firing Strategy", Logics of Programs and Their Applications, Lecture Notes in Computer Science, Berlin, Springer-Verlag, No. 148, pp86-97

[5] H. Lee-Kwang, J. Favrel and G. R. Oh, "Hierarchical Decomposition of Petri Net Languages," IEEE Trans. on SMC, Vol. SMC-17, No. 5, Sept./Oct. 1987, pp.877-878.

[6] H. S. Stone and S. H. Bokhari,"Control of Distributed Processes," IEEE Computer, Vol. 11, pp. 97-106, July 1978

[7] J. L. Peterson, *Petri Net Theory and The Modeling of Systems*, Englewood Cliffs, Prentice-Hall, 1981.

[8] W. H. Chung, H. Lee-Kwang, K. H. Park and M. Kim, "State Characterization By Maximal Set of Concurrently Firable Actions in Petri Net Based Models," IEEE Region 10 Conference, Seoul, Aug. 1987.

[9] T. Murata, "Modeling and Analysis of Concurrent Systems," in Handbook of Software Engineering, edited by C. V. Ramamoorthy and C. R. Vick, Van Nortrand, Rein-Hold, 1984.

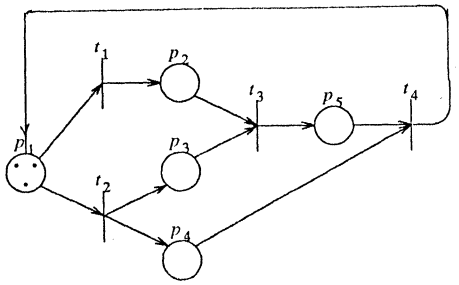
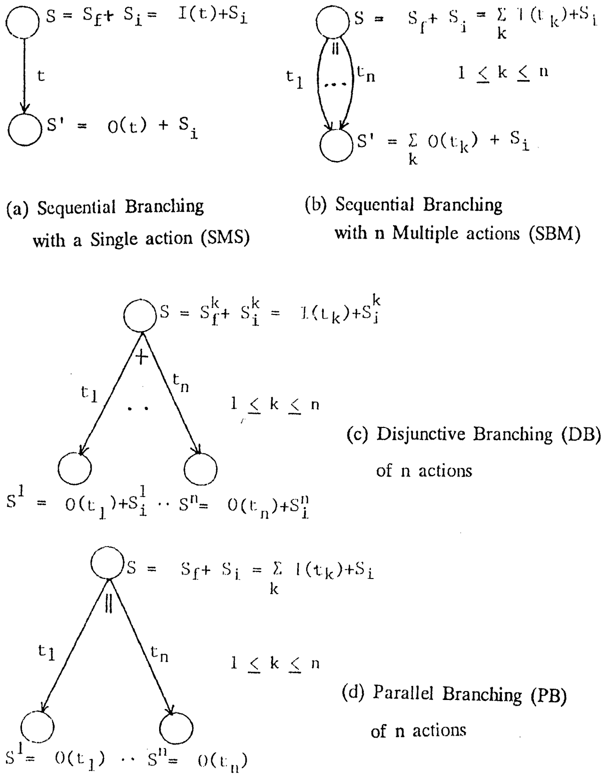


Fig. 1. A marked Petri net with $S_0 = \{3p_1\}$.

Table-1. The state characterization by C(S) and D(S)

Class	Form of M_f(S)	C(S)	D(S)	Remarks
0	{ }	0	0	no action
1	{{t}}	1	1	deterministic-sequential
2	$\{\{t_1\}, \{t_2\}, \ldots\}$	C(S) = D(S) ≥ 2	D(S) = C(S) ≥ 2	disjunctive-sequential
3	$\{\{t_1, t_2, \ldots\}\}$	≥ 2	1	deterministic-concurrent
4	$\{X_1, X_2, .., X_N\}$ where $X_k = \{t_{k0}, \ldots, t_{kn}\}$	C(S) > D(S) ≥ 2	C(S) > D(S) ≥ 2	disjunctive-concurrent



$S = S_f + S_i = I(t) + S_i$
t
$S' = O(t) + S_i$
$S = S_f + S_i = \sum_k I(t_k) + S_i$
$\|$
t_1 ... t_n
$1 \leq k \leq n$
$S' = \sum_k O(t_k) + S_i$
$S = S_f^k + S_i^k = I(t_k) + S_i^k$
$+$
t_1 . . t_n
$1 \leq k \leq n$
$S^1 = O(t_1) + S_i^1 \cdot\cdot S^n = O(t_n) + S_i^n$
$S = S_f + S_i = \sum_k I(t_k) + S_i$
$\|$
t_1
t_n
$1 \leq k \leq n$
$S^1 = O(t_1) \cdot\cdot S^n = O(t_n)$

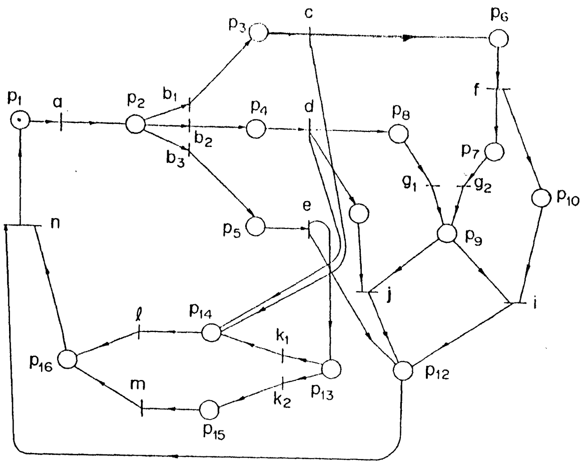
(a) Sequential Branching with a Single action (SMS)

(b) Sequential Branching with n Multiple actions (SBM)

(c) Disjunctive Branching (DB) of n actions

(d) Parallel Branching (PB) of n actions

Fig. 2. Graphical representation of four branching rules.



Transition modules	Preconditions	Postconditions
a	$\{p_1\}$	$\{p_2\}$
b_1	$\{p_2\}$	$\{p_3\}$
b_2	$\{p_2\}$	$\{p_4\}$
b_3	$\{p_2\}$	$\{p_5\}$
c	$\{p_3\}$	$\{p_6, p_{14}\}$
d	$\{p_4\}$	$\{p_8, p_{11}, p_{14}\}$
e	$\{p_5\}$	$\{p_{12}, p_{13}\}$
f	$\{p_6\}$	$\{p_7, p_{10}\}$
g_1	$\{p_8\}$	$\{p_9\}$
g_2	$\{p_7\}$	$\{p_9\}$
i	$\{p_9, p_{10}\}$	$\{p_{12}\}$
j	$\{p_9, p_{11}\}$	$\{p_{12}\}$
n	$\{p_{12}, p_{16}\}$	$\{p_1\}$
k_1	$\{p_{13}\}$	$\{p_{14}\}$
k_2	$\{p_{13}\}$	$\{p_{16}\}$
l	$\{p_{14}\}$	$\{p_{16}\}$
m	$\{p_{15}\}$	$\{p_{16}\}$

$S_0 = \{p_1\}$

Fig. 3. A Petri net model of a simple CPU.

HIGH-SPEED VECTOR INSTRUCTION EXECUTION SCHEMES OF HITACHI SUPERCOMPUTER S-820 SYSTEM

Hideo Wada, Koichi Ishii, Shigeko Yazawa and Shun Kawabe
Computer Development Department
Kanagawa Works, Hitachi, Ltd.
1 Horiyamashita, Hadano, Kanagawa 259-13
Japan

Abstract. The HITACHI supercomputer S-820 has been developed as Hitachi's top end supercomputer. It is also rated as one of the most powerful supercomputers in the world.

To achieve the performance goal, the S-820 employs advanced vector execution control. Of the features of the S-820's vector processing, this paper discusses parallelism between scalar and vector processing, elementwise parallel execution and instruction stacking.

Parallelism between scalar and vector processing greatly speeds up processing of short vectors.

Elementwise parallel execution greatly speeds up calculations which have few terms.

Instruction stacking greatly speeds up processing of short vectors and increases the efficiency of elementwise parallel execution.

1. Introduction

With the progrss of science and technology, demand is increasing for processing large-scale, complex data in many fields, such as structural analysis, molecular science, nuclear fusion, semiconductors and natural resource exploration.

Many supercomputers capable of processing large amounts of data as vectors have been developed to meet such growing demand. Their application is expanding to new fields such as computational experiments and large-scale simulations, which typically require more computation power and larger data storage capacities than other applications.

The HITACHI supercomputer S-820 has been developed to meet these systems requirements based on state-of-the-art LSI technology. This paper introduces high-speed vector instruction execution schemes of the S-820.

These high-speed vector instruction execution schemes of the S-820 have been developed aiming at three goals described as follows.

(1) Performance enhancement of short vector calculations

Before starting the vector calculations, the preprocessing for the vector calculation such as generating the addresses of vectors is executed.

Since the time required for this preprocessing is independent of vector length, the proportion of it to the vector calculation time, which is in proportion to vector length, becomes large in case of short vectors. On the contrary, in case of long vectors, the time required for the preprocessing becomes negligible compared with vector calculation time.

Thus the time required for the preprocessing becomes a great start up time and degrades the performance of short vector calculation.

Therefore performance of not only long vector calculations but also short vector calculations should be enhanced.

(2) Performance enhancement of calculations which consist of few terms

The peak performance of a supercomputer is determined as follows.

$$\text{MFLOPS} = \frac{\text{Number of arithmetic units}}{\text{Pipeline period (micro. sec)}} \qquad (1)$$

To increase vector processing performance, the processor usually must have as many arithmetic units which can operate in parallel as possible.

But in case of calculations which consist of few terms, such as the calculation which consists of only one addition of two vectors, many arithmetic units are idling while the calculation is in execution, since few (i.e. not all) arithmetic units are necessary for the calculation.

As a result, the performance of the processor becomes by far lower than its peak performance.

Therefore the performance of calculations which consist of few terms should be enhanced.

(3) Reduction of loss time caused by instruction switching

When the execution of the instruction in an arithmetic unit ends and the next instruction is given to it, the loss time is caused by instruction switching. This loss time has greater effect on the performance of the processor as vector calculation time becomes shorter.

Therefore this switching time should be reduced.

To achieve the first goal (1), the highly parallelized processing between the scalar processor and the vector processor has been realized.

This scheme is presented in section 6.1.

To achieve the second goal (2), the elementwise parallel processing scheme has been developed.

This scheme is presented in section 6.2.

To achieve the third goal (3), the instruction stacking scheme has been developed.

This scheme is presented in section 6.3.

2. Architecture

The architecture of the S-820 is shown in Fig.1. The S-820 consists of the scalar processor and the vector processor. The architecture of the scalar processor is compatible with Hitachi's general purpose computer M-series systems. Thus the S-820 supports standard data processing environments, such as TSS(Time Sharing System) and RJE (Remote Job Entry).

The architecture of the vector processor includes 90 vector instructions, 32 sets of vector regis-

36

ters, 16 sets of vector mask registers, 32 sets of scalar registers, 48 sets of vector address registers, vector address translation feature and the vector processing timer.

HITACHI M-series Architecture :		Extended Vector Architecture	
M-series instructions	216	Vector instructions	90
General-purpose registers	16	Vector registers	32
Floating-point registers	16	Vector mask registers	16
Control registers	16	Scalar registers	32
Address translation		Vector address registers	48
Storage protection		Vector address translation	
TOD clock		Vector processing timer	
CPU timer			
Clock comparator			
Extended channel system			

Fig.1 S-820 Architecture

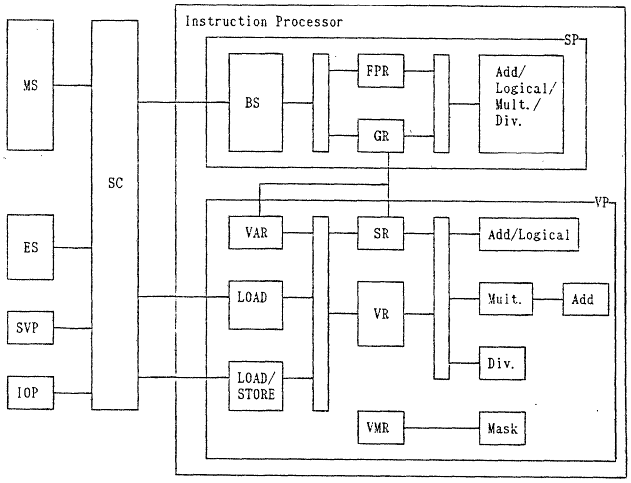
3. Processor Organization

The S-820 is available in two models:
S-820 model 80 (S-820/80)
S-820 model 60 (S-820/60).

The S-820/80 is twice as powerful as the S-820/60 and has a larger maximum storage capacity.

The 820 consists of vector processor, the scalar processor, the main storage, the extended storage, the input/output processor(s) and the service processor.

Fig.2 shows the processor organization. Table 1 shows the specification of the S-820.



MS : Main Storage
ES : Extended Storage
SC : Storage Controller
SVP : Service Processor
IOP : I/O Processor

BS : Buffer Storage
FPR : Floating-Point Register
GR : General Registers
SR : Scalar Register
VAR : Vector Address Register
VR : Vector Register
VMR : Vector Mask Register

Fig.2 Processor Organization

Table 1 Specification of S-820

Model			S-820/60	~~S-820/80~~
Peak Performance			1.5 GFLOPS	3.0 GFLOPS
Proc Unit	No. of Vector Instructions		90	
	Registers	General Registers	16 (32bits)	
		Floating-point Registers	16 (64bits)	
		Vector Registers	32 x 256Words	32 x 512Words
		Vector Mask Registers	16 x 256bits	16 x 512bits
		Scalar Registers	32 (64bits)	
	Data Length	Integer	32 bits	
		Floating-point	32/64 bits	
		Logical	64 bits	
	Vector Processing Timer		Supported	
	Buffer Storage		256 KB	
Main Storage	Capacity (MB)		64, 128, 256	128, 256, 512
	Error Checking		1 Bit Error Correction and 2 Bit Error Detection	
Extended Storage	Capacities (GB)		0.5-6.0	0.5-12.0
	Error Checking		2 Bit Error Correction	
	Max Transfer Rate (GB/s)		1 or 2	
I/O Proc	No. of Channels		16, 32, 48, 64	
	Max Transfer Rate (MB/s)		288	

4. Outline of Vector Processing

Like the predecessor S-810[1], the S-820 features parallel operations between scalar and vector processing, as illustrated below.

For example, a DO loop in a FORTRAN program:

```
      DO 20 I = 1, N
   20 A(I) = B(I) x C(I)
```

is compiled into a chain of vector instructions:

```
VLD    VR0, B
VLD    VR2, C
VEMD   VR4, VR0, VR2
VSTD   VR4, A
```

?

Where,

VLD : Vector Load
VEMD : Vector Elementwise Multiply
VSTD ; Vector Store
VRi : i-th Vector Register

No common decode

A(I), B(I) and C(I) are vectors. The execution of this program is shown in Fig.3. The scalar processor (SP) performs preprocess for this chain of instructions and passes control to the vector processor (VP).

After this preprocessing, the SP issues an EXVP (Execute Vector Processing) instruction. The VP begins vector processing, i.e., fetches the vector instruction chain, executes a VLD instruction to load vector B to vector register 0, a VLD instruction to load vector C to vector register 2, a VEMD instruction to multiply vector B by vector C and to store the result in vector register 4 and a VSTD instruction to store the result held in vector register 4 in vector A.

During the vector processing by the VP, the SP can execute another scalar processing or preprocessing for next vector processing. Parallel processing between the SP and the VP shortens total program execution time.

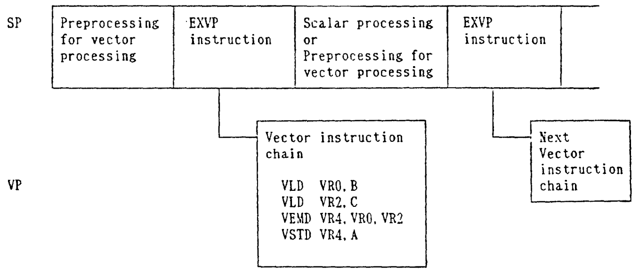

SP
Preprocessing for vector processing
EXVP instruction
Scalar processing or Preprocessing for vector processing
EXVP instruction
VP
Vector instruction chain
VLD VR0, B
VLD VR2, C
VEMD VR4, VR0, VR2
VSTD VR4, A
Next Vector instruction chain

Fig.3 Parallel Operation between Scalar and Vector Processing

5. Outline of Vector Instruction Execution Control

5.1 Structure of Vector Instruction Control Unit

Fig.4 shows the structure of the vector instruction control unit of the S-820.

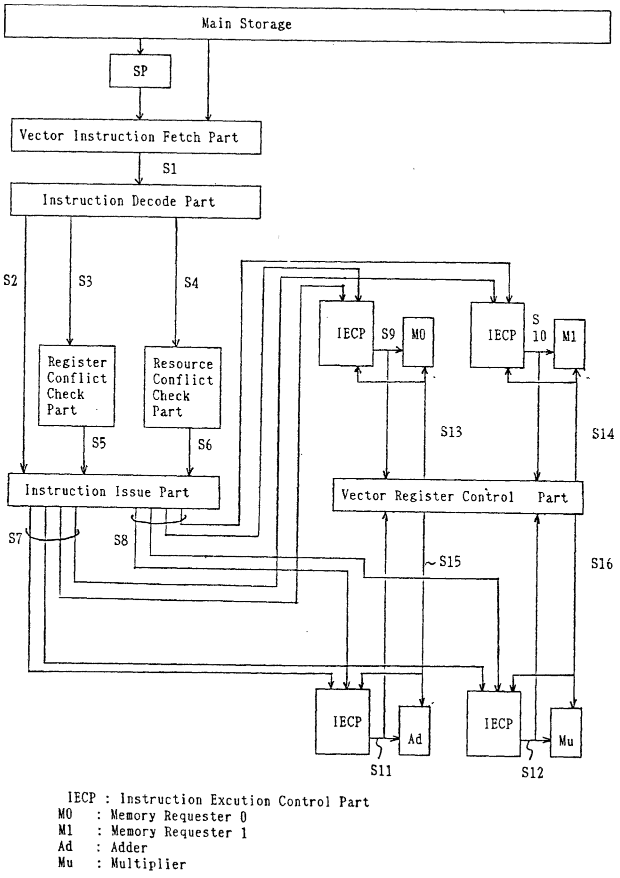


Fig.4 Structure of Vector Instruction Control Unit

The main storage stores instructions and data operands. The vector instruction fetch part fetches a chain of vector instructions from the main storage when the SP issues an EXVP instruction and then it sends vector instructions in a chain to the instruction decode part.

The instruction decode part puts the vector instructions in the queue and decode them.

The register conflict check part checks whether the destination register of the current vector instruction becomes a source for the next vector instruction, and has the information about the status of each vector register.

The instruction issue part sends the decoded vector instruction to the instruction execution control part.

Memory requester 0, memory requester 1, adder and multiplier are called "resources".

These resources are explained briefly as follows. Memory requester sends main storage access requests to the storage controller. Adder executes addition on operands. Multiplier executes multiplication on operands. Each resource processes four vector elements in parallel(See section 6.2)

These four resources must execute different vector instructions in a pipeline manner while maintaining the correct sequence of access (storing and referencing) of vector registers.

The resource conflict check part checks the conflict in use of any of the resources between the current vector instruction and the next vector instruction, and has the information about the status of each resource.

The instruction execution control part, which is attached to each resource, controls the execution of the vector instruction in each resource.

The vector register control part controls data transfer to and from the vector registers.

5.2 Vector Instruction Execution Process

In this section the vector instruction execution process of the S-820 is explained by means of Fig.4.

As mentioned in section 4, vector instructions are started by an EXVP instruction, which is a scalar instruction. The SP senses the condition of the VP and starts the VP when the VP is ready for execution of vector instructions.

The vector instruction fetch part sends the vector instructions to the instruction decode part at a rate of one instruction per machine cycle.

At first the vector instruction which is sent from the vector instruction fetch part is decoded in the instruction decode part, that is, the vector instruction is decoded into op-code, vector length, vector register number, resource number and so forth.

After decoding the vector instruction, the instruction decode part sends the aforementioned information to the instruction issue part, sends the register number of the vector register designated in the vector instruction to the register conflict check part and sends the resource number of the resource used in the vector instruction to the resource conflict check part.

In the register conflict check part, conflict in use between the vector registers used in the decoded vector instruction and the vector register used in the vector instruction in execution is checked.

In the same manner conflict in the use of resources is checked in the resource conflict check part.

When no conflict is detected, the instruction issue part sends the information necessary for the execution of the vector instruction, that is, opcode, vector register number, mask information, chaining information and other control information , which is shown as data path S8 in Fig.4, and instruction start signal, which is shown as S7 in Fig.4, to the instruction execution control part corresponding to the resource which executes the vector instruction.

When any conflict is detected, the instruction issue part waits until the conflicts disappear.

Each instruction execution control part sends the instruction to the corresponding resource by way of one of the data paths S9, S10, S11 and S12 and instructs the resource to execute the instruction.

The resources and the vector register control part executes the instruction sent from the instruction execution control part.

The vector register control part reads the vector data operands from the vector register designated in the instruction and sends them to the resource which needs them.

The vector register control part also writes the vector data which are sent from the resources to the vector register designated in the instruction.

When the vector register control part ends reading/writing, it informs the resource and the instruction control part by the signals S13, S14, S15 and S16 and also informs the register conflict check part.

When a resource ends the operation, it informs the resource conflict check part and updates the resource status in the resource conflict check part.

It should be noted that vector instructions executed by different resources can operate in parallel when no conflict is detected, since each resource is independent of other resources.

6. Vector Instruction Execution Schemes

6.1 Highly Parallelized Processing between the Scalar Processor and the Vector Processor

As described in section 4, parallel processing between the SP and the VP effectively shortens the total program execution time.

In order to further enhance performance the S-820 employs the following two processing schemes:

(1) Vector processing linking
While the VP is executing a vector operation the execution of the next chain of vector instruction can be started.

(2) Vector processing signaling
Even when the VP has not completed all the operations in the chain of the vector instructions, the part of the scalar program which needs the result of a part of vector operations can be started, when it is ready.

Fig.5(a) shows parallel processing between the SP and the VP described in section 4, taking an example of a chain of vector instructions which is divided into two vector operations.

First the SP executes preprocessing for vector operations ①,②.

Next the SP starts the VP by means of EXVP instruction.

While the VP executes the vector operation, the SP can execute scalar processing such as preprocessing of the next vector operations.

As soon as current vector processing has completed, the SP can start the VP for the next vector operation which has been kept waiting.

Fig.5(b) illustrates how vector processing linking improves the performance.

In case that the control information for the vector operation ② is not necessary for the vector operation ①, the SP starts the VP immediately after preparing the control information necessary for the vector operation ①.

Next, the SP sets up the VP with the information necessary for the vector operation ② and signals "linking operation" to the VP.

The VP links the vector operation ② with the vector operation ①, i.e., executes the vector operation ① and the vector operation ② in a synchronized manner.

As compared with Fig.5(a), the degree of parallelism is increased. As a result overall performance is enhanced.

The compiler automatically partitions a chain of vector instructions into two or more smaller chains so that parts of them can be processed in an overlapped manner.

Fig.5(c) illustrates how vector signaling enhances performance, taking an example of a chain of vector instructions which is divided into two vector instructions, where the first vector instruction generates a scalar result which is necessary for following scalar processing.

In Fig.5(c), the vector instruction ① is the only instruction in a vector instruction chain which stores the calculation result in a scalar register. Such vector instruction is given the "signaling flag" by the compiler.

The vector instruction ① stores the result in the scalar register, which can then be accessed by a subsequent scalar instruction ③.

The vector instruction ① in such a case is given the "signaling flag" as mentioned above.

When the vector instruction ① finishes the operation, the VP tells the SP to start subsequent scalar instruction ③.

Since the scalar instruction ③ does not wait until the vector instruction ② which immediately follows the vector instruction ① completes the operation, the SP and the VP operate in parallel after the completion of the operation of the vector instruction ①.

As a result overall performance is enhanced.

The realization of these two schemes, vector processing linking and vector processing signaling , utilizing vector instruction control unit shown in Fig.4 is described as follows.

To realize vector linking, the vector instruction fetch part executes the following control.

When an EXVP instruction without a "linking flag" is issued by the SP, the vector instruction fetch

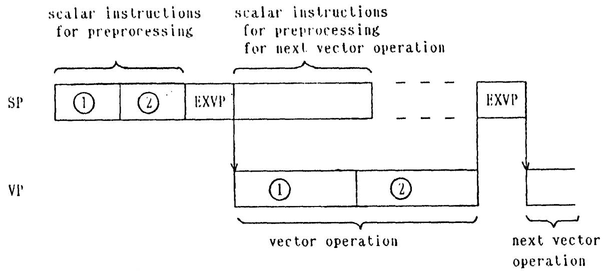

scalar instructions for preprocessing
scalar instructions for preprocessing for next vector operation
SP
①
②
EXVP
EXVP
VP
①
②
vector operation
next vector operation

Fig.5(a) Parallel Processing of the Scalar Processor and the Vector Processor

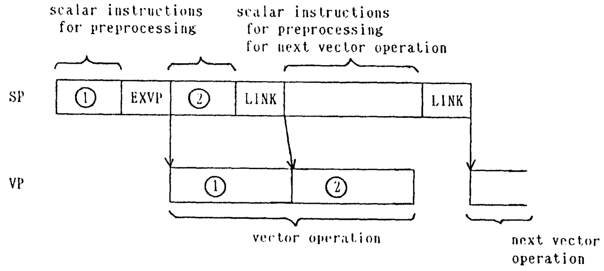

scalar instructions for preprocessing
scalar instructions for preprocessing for next vector operation
SP
①
EXVP
②
LINK
LINK
VP
①
②
vector operation
next vector operation

Fig.5(b) Performance Enhancement by Vector Processing Linking

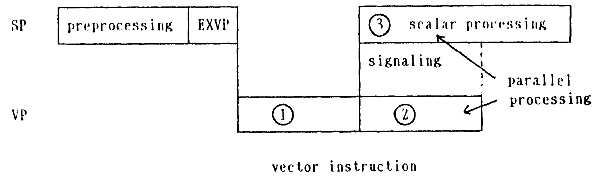


Fig.5(c) Performance Enhancement by Vector Processing Signaling

part senses the instruction decode part and the instruction issue part. In case that there are either instructions being executed or instructions in the instruction decode part, it does not send a chain of vector instructions to the instruction decode part.

When an EXVP instruction with a "linking flag" is issued by the SP, the vector instruction fetch part send a chain of vector instructions to the instruction decode part in case that there is no instruction in instruction decode part.

To realize vector signaling, the vector register control part executes the following control.

When the vector register control part ends writing the result in the scalar register which is specified in the vector instruction with "signaling flag", it sets the specific register called signal register.

The SP senses this signal register and executes subsequent scalar instruction which needs the data held in the scalar register when it finds the signal register to be one.

These "linking flag" and "signaling flag" are specified by the compiler.

6.2 Elementwise Parallel Processing Scheme

The elementwise parallel processing operation is realized as follows.

The arithmetic unit of the S-820 model 80 consists of four fully segmented pipelines. These four pipelines are operated concurrently. Thus memory requester 0, memory requester 1, adder and multiplier in Fig.4 all consist of four fully segmented pipelines. Therefore four elements can be processed in one machine cycle, whereas a single pipeline scheme would allow only one element to be processed in one machine cycle.

Fig.6 shows elementwise parallel processing for addition of two vectors in the vector register NO.0(VR0) and the vector register NO.4(VR4). The result is to be stored in the vector register NO.8 (VR8).

At first, element NO.0, NO.1, NO.2 and NO.3 are processed concurrently as follows. Adder in Fig.4 consists of four fully segmented pipelines, which are called Adder0, Adder1, Adder2 and Adder3.

The vector register control part reads these four elements from VR0 and VR4 and sends them to these four adder units, Adder0, Adder1, Adder2 and Adder 3 concurrently.

Next, the vector register control part sends executing signal to the Adder0, Adder1, Adder2 and Adder3 in parallel.

Finally, the vector register control part writes the four results of additions to VR8 in parallel.

These procedures are repeated for further elements.

Thus four consecutive elements are processed concurrently as one unit.

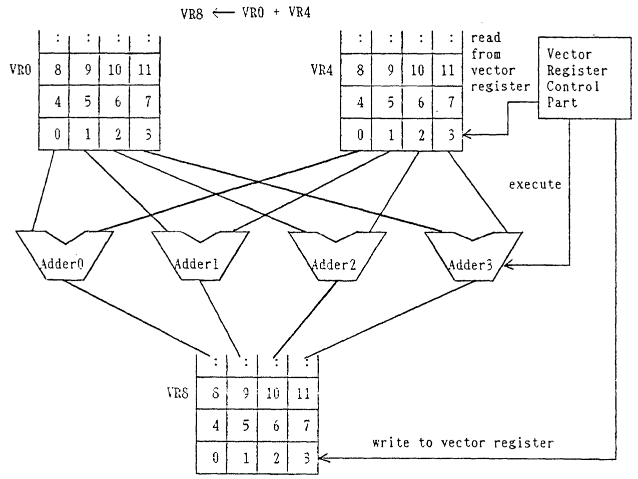


Fig.6 Elementwise Parallel Processing

Fig.7(a) and Fig 7(b) show processing by means of single pipeline scheme and elementwise parallel pipeline (processing) scheme respectively, taking an example of vector calculation

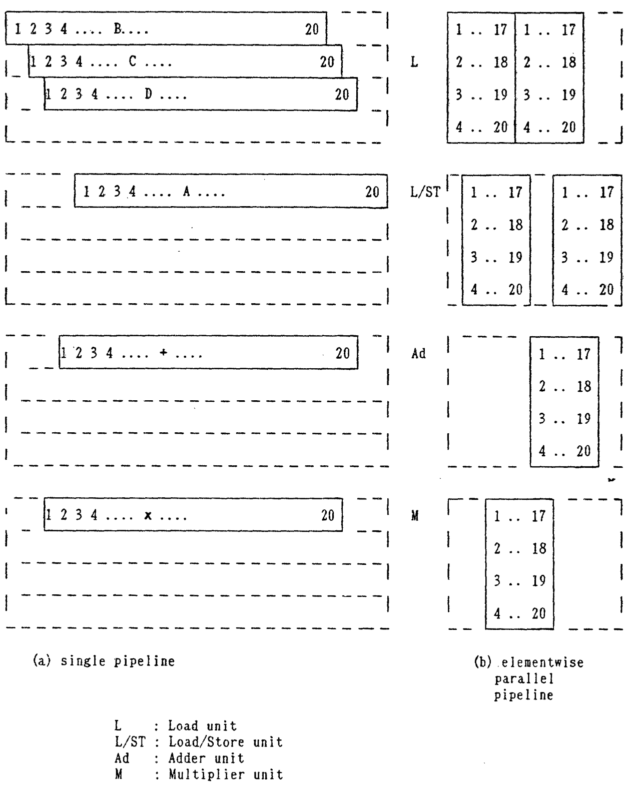
$$Ai = Bi \times Ci + Di. \quad (2)$$

The vector length is assumed to be 20. The number in Fig.7 means the element number.

In Fig.7, load unit, load/store unit, adder unit and multiplier unit correspond to memory requester 0, memory requester 1, adder and multiplier respectively in Fig.4.

Loading of vector B, loading of vector C and loading of vector D are executed by the load unit.

Multiplication of vector B and vector C is exe-



(a) single pipeline

(b) elementwise parallel pipeline

Fig.7 Performance Enhancement by Elementwise Parallel Processing

cuted by the multiplier unit.

Addition of the result of the multiplication, that is B x C, and vector D is executed by the adder unit.

Storing the final result, that is B x C + D, in a vector A is executed by the load/store unit.

Since each unit in elementwise parallel processing consists of four fully segmented pipelines, the time required in each operation such as loading and adding is one fourth of that in single pipeline scheme.

Thus the overall performance is improved.

6.3 Instruction Stacking Scheme

The point of this paper is "Instruction Stacking Scheme" described below.

The aim of "Instruction Stacking Scheme" is to reduce the loss time caused by instruction switching. "Instruction Stacking Scheme" is employed in the instruction execution control part.

Fig.8 shows the structure of the instruction execution control part.

The instruction execution control part can have up to two vector instructions including the instruction in execution, and starts the execution of vector instructions in order of their arrival.

The current instruction register holds the vector instruction being executed. The next instruction register holds the next vector instruction.

These two instruction registers play the role of stacks, that is, two instructions can be stacked for each resource. Therefore these two instruction registers are also called instruction stacks.

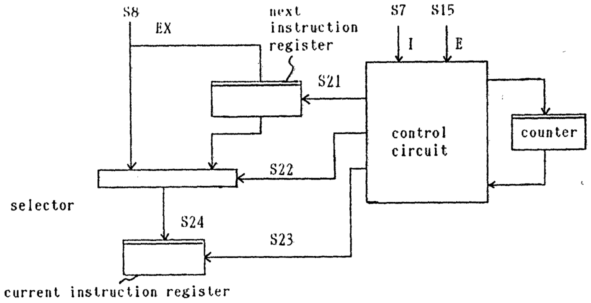


Fig.8 Structure of the Instruction Execution Control Part

As an example, the instruction execution control part for the adder is explained below.

Data path S8 means the information necessary for execution of the instruction, that is, op-code, vector register number, mask information, chaining information and other control information and the signal S7 is the instruction start signal which means that the vector instruction is issued as explained in section 5.2. The signal S15 means that the execution of the instruction by the resource and the vector register control part ended.

In Fig.8 data path S8, signal S7 and signal S15 are called "EX", "I" and "E" respectively, and data path S8 is also called "instruction".

The counter consists of 2 bits and represents the number of instructions included in the instruction execution control part.

The control circuit updates the counter, issues the signal S21 which sets the next instruction register, issues the signal S23 which sets the current instruction register and issues the signal S22 which switches the selector.

Fig.9 is the status transition graph of the counter. Each node represents one value of the counter. The value 00/01/10 means that there are 0/1/2 instructions in the instruction execution control part. The value 11 signifies that a malfunction has occurred and a machine check signal is sent to the SP in this case.

In Fig.9, "I" means that signal "I" is issued and "Ī" means that signal "I" is not issued. This convention applied also to "E" and "Ē".

The status transition of one example case is explained as follows.

Suppose that the signal "I" is not issued and the signal "E" is issued when the contents of the counter are "10".

In this case, since the execution of the instruction in the current instruction register ended and the new instruction is not issued, the contents of the current instruction register is replaced by those of the next instruction register and the next instruction register becomes empty. As a result the value of the counter becomes "01", since there is one instruction in the instruction execution control part.

The control circuit in Fig.8 operates as follows. Table 2 shows the values of signals S21, S22 and

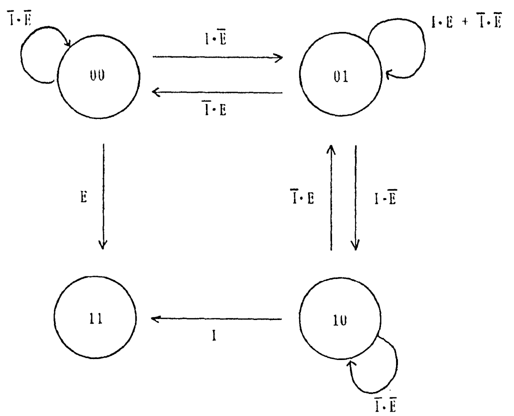


Fig.9 Status Transition Graph of the Counter

Table 2. The Value of Control Signals in the Instruction Execution Control Part

counter	00	01	01	10
I	1	1	1	0
E	0	0	1	1
S23	1	0	1	1
S21	0	1	0	0
S22	0	0	0	1

S23. In Fig.8 signals S21 and S23 set the next instruction register and the current instruction register respectively and signal S22 selects the contents of the next instruction register to data path S24 when its value is 1 and selects data path S8 when its value is 0.

In this example case, the value of the counter, that of signal "I" and that of signal "E" are "10", "0" and "1" respectively. According to Table 2 the value of signal S22 becomes 1 and that of signal S23 becomes 1 so that the contents of the current instruction register may be replaced by the contents of the next instruction register in this case.

The other cases can be explained in the same way.

The resource conflict check part has the status of these two instruction stacks, that is, the current instruction register and the next instruction register so that superfluous instruction start signals (S7) may not be issued.

By controlling the instruction execution as described above, outline of system control becomes as follows.

When there is no instruction in execution, the instruction sent from the instruction issue part is held in the current instruction register.

When there is an instruction in execution, the instruction sent from the instruction issue part is held in the next instruction register and the contents of the current instruction register is replaced by those of the next instruction register upon completion of the instruction.

6.4 Effect of the Instruction Stacking Scheme

By employing the instruction stacking scheme described in 6.3, immediately after the execution of the instruction in each resource ends, the next instruction can be given to the resource.

As a result, the resource can be used more efficiently since vector data to be processed by the current instruction and the next instruction can be given to the resources without a break.

In order to show the effect of the instruction stacking scheme, the timing charts of the case where successive instructions use the same resource are shown in Fig.10.

The vector length of every instruction is assumed to be 3. In Fig.10, the number represents the instruction.

Fig.10(a) shows the case that there is no instruction stack. Fig.10(b) shows the case that there are two instruction stacks, that is, the current instruction register and the next instruction register in Fig.8.

In Fig.10, the instruction start signal corresponds to S7 and the execution ending signal corresponds to S13, S14, S15 or S16 in Fig.4.

In Fig.10(a), the execution on elements has one vacant cycle at the point of instruction switching.

In Fig.10(b), the execution on elements is continuous and resources do not have idle time.

Next, the time T from the instruction start signal for the first instruction to the execution ending signal for the N-th instruction is compared between Fig.10(a) and Fig.10(b) as follows. L means the vector length.

Fig.10(a) : $T = N \times L + N - 1$ (3)

Fig.10(b) : $T = N \times L$ (4)

Therefore, the effect of the instruction stacking scheme to no instruction stack case is

$P = (3) / (4) = 1 + 1/L - 1/NL$ (5)

When number of instructions is large, (5) becomes

$P = 1 + 1/L$ (6)

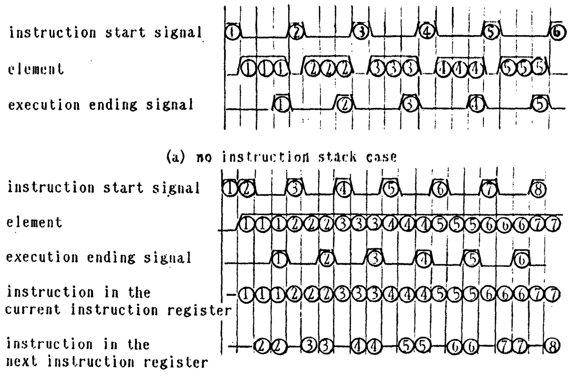
Table 3 shows the effect on the S-820 model 80(4 elements are processed concurrently by elementwise parallel processing).

Table 3 shows that the instruction stacking scheme greatly improves the performance in case of short vectors.

In order to enhance the performance, supercomputers generally employ elementwise parallel pipeline processing scheme, and furthermore the degree of elementwise parallel pipeline becomes larger, that is, the number of elements which can be processed concurrently becomes larger.

As a result the number of elements to be processed per pipeline becomes smaller.

In this situation, this instruction stacking scheme which has a great effect on processing of short vectors becomes important.



(a) no instruction stack case

(b) two instruction stacks case

Fig.10 Timing charts of the Instruction Stacking Scheme

Table 3 Effect of the Instruction Stacking Scheme

vector length	number of elements per pipeline	effect (P-1)
1 - 4	1	100%
5 - 8	2	50%
9 - 12	3	33%
13 - 16	4	25%
:	:	:
37 - 40	10	10%
:	:	:

7. Performance Evaluation

Table 4 shows the algebraic average of the measurement results of Lawrence Livermore Laboratory's 14 Kernels under the following 4 conditions.

① : normal condition
② : without vector processing linking
③ : without vector instruction stacking
④ : without vector processing linking and vector instruction stacking

Table 4 Algebraic Average of the Measured Results of Lawrence Livermore Laboratory's 14 Kernels

condition	①	②	③	④
MFLOPS	419.4	400.3	394.5	380.9

The following considerations are derived from Table 4.

The effect of vector processing linking is

③ / ④ - 1 = 3.6 % (7)

The effect of vector instruction stacking is

② / ④ - 1 = 5.1 % (8)

The effect of combination of vector processing linking and vector instruction stacking is

① / ④ - 1 = 10.1 % (9)

The effect of elementwise parallel processing has not been measured. Therefore it is roughly estimated as follows.

The following two assumptions are made.

First, performance improvement of the S-820 on the S-810 is assumed to be attributed to reduction of machine cycle, elementwise parallel processing, vector processing linking, vector processing signaling and vector instruction stacking.

Second, the effect of vector processing signaling is assumed to be negligible since the cases where vector processing signaling applies are somewhat limited.

Since machine cycle of the S-820 is 57.1% of that of the S-810 and the algeblaic average of 14 Kernels on the S-810 is 137.9 MFLOPS, the effect of elementwise parallel processing is

$$\frac{380.9\ (④\ \text{in Table 4})}{137.9} \times 0.571 - 1 = 58\% \quad (10)$$

8. Conclusions

This paper presented high-speed vector instruction execution schemes of the Hitachi supercomputer S-820.

Parallelism between scalar and vector processing , elementwise parallel processing and instruction stacking have been introduced as these schemes.

In particular instruction stacking greatly improves the performance of processing of short vectors and increases the efficiency of elementwise parallel processing.

By the performance measurement on Lawrence Livermore Laboratory's 14 Kernels, vector processing linking (which enhances parallelism between scalar and vector processing), vector instruction stacking and elementwise parallel processing improve the overall performance by 3.6%, 5.1% and 58% respectively (, where the effect of elementwise parallel processing is estimated value).

References

[1] T.Odaka, S.Nagashima, S.Kawabe, "Hitachi Supercomputer S-810 array processor system." SUPERCOMPUTERS, Class VI Systems, Hardware and Software, S.Fernbach (Editor), Elsevier Science Publishers B.V., pp.113-136, 1986

CHARACTERIZATION OF MEMORY CONFLICT LOADING ON THE CRAY-2

D. A. Calahan

Dept. of Electrical Engineering & Computer Science

University of Michigan

Ann Arbor, MI, 48109

Abstract -- Empirically-derived models are constructed of the shared-memory accessing delays associated with the dynamic-memory CRAY-2.

Introduction

Although memory accessing studies of conflict delays have been carried out for the CRAY X-MP [1][2][3][4], the introduction of the slow massive dynamic memory of the CRAY-2 (abbr. C-2) - producing typical algorithm delays in the range of 20-40% - has emphasized the need for further analysis of this phenomenon.

The origin of this conflict problem is principally not collisions along multiple memory paths as studied in the 1970s [5], but rather from the bank reservation time (T_{br}), i.e., the time for a chip to recover from an access. Although its effect can be lessened by memory reorganization, the qualitative effects of a large T_{br} can be compensated principally by increasing the number of banks, which is limited by space and hardware considerations with fast clock periods.

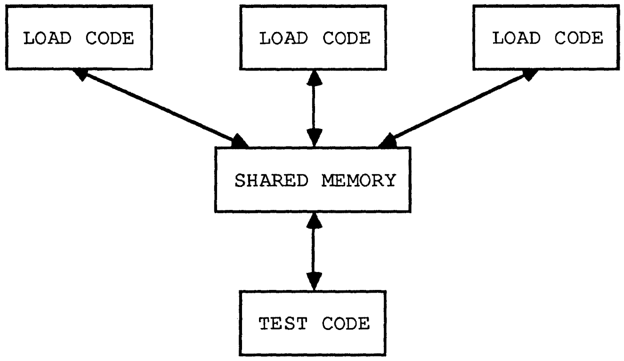
Once a memory technology is proposed, then the principal question is the number of processors a given number of banks can support. Unfortunately, little insight is offered by current literature on the critical load parameters to be accommodated. There is, for example, no descriptor such as "hit-ratio" common in cache memory design.

The C-2 common memory organization[a] is not as amenable to mathematical analysis (or even simulation) as that of the CRAY X-MP, in part because the C-2 contains a variety of buffering and bank-enhancement techniques ("pseudo-banking") to compensate for the small number of banks relative to T_{br}. Consequently, this paper proposes load parameters derived from measurements made on a dedicated C-2 - a "black-box" approach - and develops related mathematical and graphical models.

Experimental Model

The experiments to be reported in this report are based on use of a dedicated C-2. The four processors are conceptually divided into one processor executing a test code instrumented to measure delays and 0-3 active processors executing a load code (Figure 1); load processors are selectively rendered inactive by running a no-access code. The following experiments involve selection of test and load codes which are intended give insight into factors which influence the effects of load codes on a test code.

Figure 1. Experimental model



Model Parameter Selection: The Scalar/Vector Loading Anomaly

It has been noted in measurement of conflict-induced performance degradation that vectorized codes incurred inexplicably large delays relative to scalar codes with a similar total number of memory accesses. Table 1 gives illustrative percent delays incurred in a test code when identical load codes are run on three processors. Both a scalar and vector load code were run against four representative test codes. The table graphically shows that, although a scalar test code is marginally impacted by another scalar load, the same scalar load produces large degradations in a vector test code similar to that produced by a high-access-rate matrix multiply vector load. That is, a low-access scalar code presents a large apparent load to vector codes running in other processors.

Table 1. Algorithm delay (%) of test codes. All codes are Fortran. Run on NAS C-2 on 5/10/86.

	Load Codes	
Test Codes	Scalar (SORT)	Vector (MXM) (Matrix multiply)
Scalar		
GATHER	2.6%	18.3%
SORT	3.2%	26.3%
Vector		
FLUIDS KERN.	32.0%	34.7%
UNROLLED M*V	73.1%	85.6%

Measurement Probes

To develop a measurement standard not dependent on accessing peculiarities of an application code, the accessing delay will be measured directly in two test codes which perform scalar and vector accesses only. These will be termed "probes", because their purpose will be to illuminate the discrepancies noted from Table 1.

These probes will be calibrated by measuring access delays on a dedicated C-2 under a variety of loads with known key parameter values. As a result, an equation for each probe will be developed in terms of these parameters. Measurements with these probes can then be inverted to find the parameters of an unknown load.

[a] See reference [6] for a partial description.

Mathematical Model of Probes

With Table 1 as a guide, the parameters chosen to represent the load are

1. the average startup rate (R_s) of accesses from all loads (startups/cp), and
2. the average rate (R_a) of accesses from all load (accesses/cp).

For a vector access with length VL, $R_a = R_s$*VL. The dependence on R_s represents the influence of accessing irregularity introduced by scalars and short vectors; the effect of memory traffic of any origin is given by dependence on R_a.

The access delay (D) measured in the probe is defined as the extra time in cps either (1) to secure an scalar element in a register for the scalar probe or (2) to clear the memory path after a vector access in the vector probe. It is to be represented in a N-variable Taylor's series truncated after the Mth order, viz, with variables R_a and R_s and M=3

$$D = a_1 + a_2R_a + a_3R_s + a_4R_aR_s + a_5R_a^2 + a_6R_s^2 + a_7R_a^2R_s + a_8R_aR_s^2 + a_9R_a^3 + a_{10}R_s^3 \qquad (1)$$

The delays for the scalar and vector test probes will be indicated D_{ps} and D_{pv}, respectively.

Memory Saturation

The limiting rate at which accesses can be made occurs when all memory banks are continuously reserved due to T_{br}. If the load accessing rate is R_a, the test code accessing rate is R_{tca}, and the number of banks is N_b, then define the memory saturation fraction as

$$F_{sat} = (R_a + R_{tca})T_{br}/N_b \qquad (2)$$

$$\leq 1 \qquad (3)$$

The proximity of F_{sat} to unity will be a measure of the load intensity.

Scalar Probe

Calibration

In the scalar probe, a read is performed whenever the previous scalar read is secured in a register - a high but representative rate for scalar codes. Successive reads are one address apart to avoid self-conflicts. Average delays in five groups of 4000 reads are recorded (20000 total); these group averages are scanned for consistency and a resulting overall average determined.

To calibrate the probe, the load codes are chosen to contain only unit-stride vector accesses, but these are performed at selectable no-load rates. To compensate for the regularity of the scalar test code, the load codes are chosen to have irregular accessing, viz,

(1) vector lengths are chosen uniformly random over 13 selected ranges given by VL_{min} = {1,2,4,8,16,32,64,1,8,16,24,1,4} and VL_{max} = {1,2,4,8,16,32,64,63,56,48,40,15,12}, yielding the average load vector length vl = {1,2,4,8,16,32,64,32,32,32,32,8,8}; the latter six vls test the use of the mean vector length as a load descriptor irrespective of the standard deviation of vector lengths;

(2) the addresses of the first access of each vector are uniformly randomly distributed across 256 banks, to accommodate the branch-doubling effects of pseudo-banking in extending 128 physical banks;

(3) the average no-load startup rate of load vectors are identical for all active load processors during a single experiment, but are varied at six selectable rates between experiments.

The thirteen vls and six startup rates yield 78 experiments. In addition, it was noted that the accessing rates of the load codes are slowed by conflicts; the load codes are consequently themselves instrumented to measure their actual average accessing rates (another 78 experiments). These measured loaded average startup rates will be denoted r_s; the averaged measured values for R_a are denoted r_a and determined from $r_a = r_s$*vl. All experiments are performed automatically by multitasking. Approximately three minutes of elapsed time are required to carry out the 156 experiments on the C-2.

An elementary requirement of the calibration process is that the loads be chosen so that the delays are in the range of those of daytime loads. In these 78 tests, the average measured probe delay was 28cp, whereas a typical daytime delays on the NAS 80-nsec system is 35cp.

Table 2. Results of least squares fitting of probe data.
80-nsec dynamic C-2 memory (NAS)

(a) Scalar probe

		# of terms	Correl. Coefficient	RMS error/ ave. value
Order=1				
	r_s	2	.053	66.5%
	r_a	2	**.9987**	**3.45%**
	r_s,r_a	3	.9987	3.43%
Order=2				
	r_s	3	.831	34.8%
	r_a	3	.9992	2.73%
	r_s,r_a	6	.9992	2.63%

(b) Vector probe

		# of terms	Correl. Coefficient	RMS error/ ave. value
Order=1				
	r_s	2	..799	36.7%
	r_a	2	.564	50.4%
	r_s,r_a	3	.967	15.6%
Order=2				
	r_s	3	..831	34.0%
	r_a	3	.567	50.4%
	r_s,r_a	6	**.992**	**7.60%**
Order=3				
	r_s	4	.833	33.7%
	r_a	4	.567	50.4%
	r_s,r_a	10	.993	7.19%

The measured data was fitted in the least squares sense with the expression of Eq. (1),using only r_s, only r_a, and then both r_s and r_a as variables. The order of fit was also appropriately varied (see discussion of vector probe fitting). The results of curve fitting are shown in Table 2(a), with the linear fit in r_a only chosen (bold type). The resulting linear model has the form

$$D_{ps} = 1.61 + 6.59\, r_a \qquad (4)$$

This expression invites the following observations.

(1) Model parameters. The fact that a good fit - as indicated by the correlation coefficient and the RMS error - is achieved with a function of only r_a in spite of the above variety of loading conditions supports the intuitive notion that access delays in scalar reads are explicit functions only of total memory traffic and not of load vector lengths or the frequency of vector startups. That is, 64 scalar accesses produce the same loading to scalar reads as a single 64-length vector access; otherwise, D_{ps} would be a function of r_s.

(2) Constant term. The constant term of represents the effects of memory refresh on scalar reads.

(3) Linear term coefficient. The coefficient of r_a is a succinct direct measure of the effective access delay caused by loading. Unfortunately, measurements on other C-2 memories [7] show that these coefficients are not obviously relatable to T_{br}, as might be expected.

Use of the Scalar Probe

Since the scalar probe measures only r_a, it becomes a convenient measurement of total memory accessing. Toward this goal, measurements were made of the NAS 80-nsec C-2 during a daytime load by averaging 5 million scalar probe accesses. The following estimate of total memory accessing was obtained, using Eq. (4).

$$D_{ps} = 5.2 \text{ cp} \dashrightarrow r_a = .544 \text{ accesses/cp}$$

Below, r_a will be used below to determine a complete memory load characterization.

Vector Probe

Calibration

The vector probe consists of reading 64-length unit-stride vectors initiated at successive addresses 64 banks apart so as to scan all 256 pseudo-memory banks in four reads. In the vector test code, each vector read is initiated whenever the memory path becomes available, a maximum rate representative of many vector codes for which the single memory path is a data flow bottleneck. The same selection of loads are used as in the scalar probe calibration above (156 experiments total). The average probe delay measured in these calibration runs is 4.7 cp, close to a typical measured daytime value of 5 cp.

Least-squares fitting and modeling

The results of fitting equations to the vector read probe data are shown in Table 2(b). It was decided to choose the second order fit D_{pv} in both r_a and r_s (bold type) because

(1) fits in only r_a or only r_s of any order yielded unacceptable RMS error, and

(2) in choosing between first, second, and third order fits involving both r_a and r_s, the second order fit produced by far the greater improvement from lower orders.

It should be noted that, although higher-order approximations are necessarily better fits for the measured data, they often suffer from higher volatility over the entire range of fitting and offer less insight in extrapolations beyond the range of measurement.

A graphical model

The leasts squares fit corresponding to the choice in Table 2(b) is

$$D_{pv} \;=\; 3.34 - 6.23r_a + 418.r_s + 8.22r_a^2 + 467.r_sr_a - 691.r_s^2 \qquad (6)$$

The constant term (3.34 cp) represents the effects of memory refresh; this agrees with no-load measurement.

Contour representations of D_{pv} are shown in Figure 2. Since r_a=vl*r_s, constant-vl lines can be drawn radially through the origin; they are shown for vl=1, 8, and 64. This presentation invites the following comments.

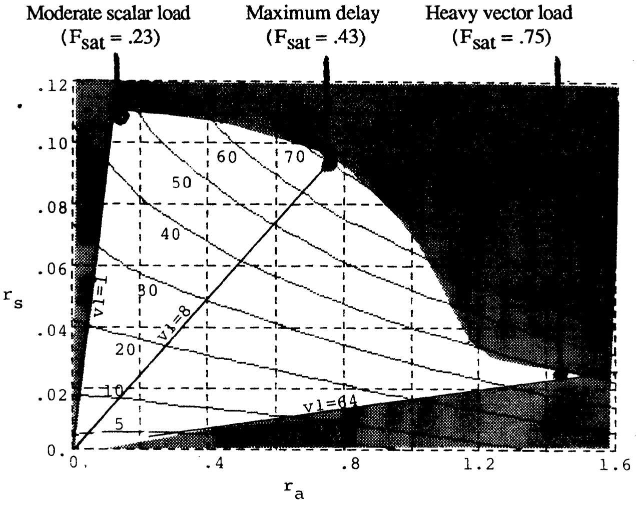
(a) It is clear that the contour curvature increases as vl decreases, correlating with previous observations concerning the degrading effects of scalars.

(b) In the region denoted "heavy vector load", the r_a cannot be increased by increasing no-load accessing rate, indicating a saturated condition. Eq. (2) yields $F_{sat} = .75$, using $N_b = 128$.

(c) With preselected no-load startup rates, a maximum delay of 69cp is produced when vl=8, represented by the radial line shown. This combines the worst conditions of high vector startup rate (r_s) and high total memory activity (r_a), and shows that short vectors rather than scalars can be the most destructive of performance.

It is felt that Figure 2 represents the principle feature of this load characterization, namely, the ability to succinctly depict an accurate nonlinear delay model dependent on only two parameters (r_a and r_s), with a third parameter (vl) easily represented and defining the limits of model validity.

Figure 2. Contour representation of D_{pv}
(no measurements in shaded areas)



Super-scalar performance

Gather/scatter accessing

The increasing nonlinearity of the contours of Figure 2 in the scalar region invites more tests to investigate the limitations of the model in this region. Accordingly, the three load processors executed gather/scatter vector operations with VL=1,2,4,8, and 16, to produce a series of scalar experiments with $.11 \le r_s = r_a \le .32$ and $45 \le D_{pv} \le 110$ cp. (Note that the gather/scatter proceeds at a maximum rate of .25 accesses/cp/processor on the C-2). To evaluate the error of the nonlinear model of Eq. (6), the latter was specialized to the scalar case by setting r_a=r_s, yielding

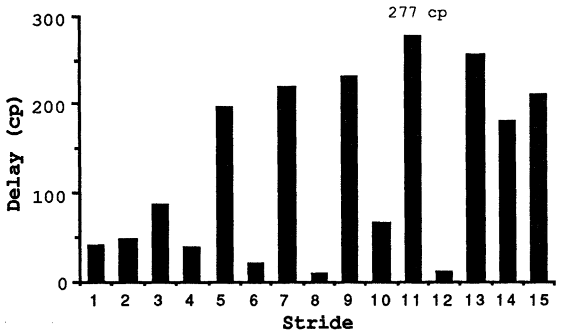
$$D_{ps} \;=\; 3.34 + 412.r_a - 215.r_a^2 \qquad (7)$$

For these 5 points, an error ratio (RMS error/average value) = 6.4% was achieved, in comparison with the ratio of 7.6% of Table 2(b). Thus, the model of Eq. (6) appears quite good extended along the vl=1 line of Figure 2. It must be pointed out that for VL=32 and 64, the group averages read from the probe became erratic, possibly indicating a surging under intense memory loading.

Non-unit-stride accessing

All of the load accessing reported above has involved non-unit-stride vectors. It is felt, however, that typically 10-20% of site accessing is non-unit-stride, principally due to complex arithmetic. To indicate the effects of such accessing, a series of tests were made with 3-processor loads of strides ranging from 1 to 127. Figure 3 shows selective resulting average delays measured by the vector test probe; they are seen to range to 277cp, an astounding delay for a 64-length access. The explanation is likely that successive accesses have the same effect as random scalar accesses, such as occur in the above gather/scatter; however, this vector access proceeds at full rate, unlike gather/scatters on the C-2. The largest average delay observed has been 284 cp for a stride of 75. This indicates the possibility of stride being a dominant load parameter under certain conditions, thus establishing a limit on a model involving only r_a and r_s. As the latter model is refined, it may be possible to add the average load stride as a third parameter. The resulting 3-D contours could offer insight similar to Figure 2 concerning design criticality to a variety of loads.

Figure 3. Delays of vector probe with non-unit-stride loads



Complete Site Memory Load Modeling: The Inverse Problem

In summary of the above models, identification of the memory as 80-nsec dynamic (the C-2 supports two other memory technologies) establishes a set of eight coefficients of $D_{ps}(r_a)$ and $D_{pv}(r_a,r_s)$. Knowledge of the load rate parameters r_a and r_s then permits an estimate of the probe delay, i.e.,

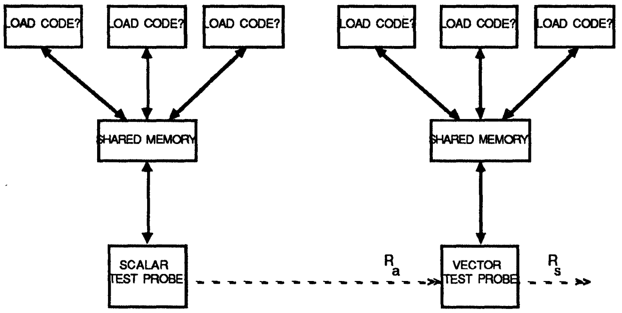
$$\{ r_a, r_s \} ===> \{ D_{ps}, D_{pv} \}$$

It is also clear that the simple functional forms of D_{ps} and D_{pv} permits solution of the inverse problem, i.e.,

$$\{ r_a, r_s \} <=== \{ D_{ps}, D_{pv} \}$$

This raises the prospect of being able to infer both load rate parameters of a site from execution of the two probes. Specifically, Figure 4 shows that r_a is derived from measurement of D_{ps} and use of Eq. (4); r_s is then found from D_{pv}, r_a, and Eq. (6) by solving a quadratic equation.

Figure 4. Modeling from site measurement probes



This process can be viewed as a potential software performance monitoring tool, in lieu of monitoring hardware absent in the C-2. For example, these probes were run with a daytime load on the 80-nsec NAS C-2 memory, obtaining $r_a = .491$ from Eq.(4), $r_s = .083$ from Eq. (6), and vl = $r_a/r_s = 5.56$. The relatively low average load vector length likely occurs because (1) both (scalar) addressing data accesses and floating-point-data accesses are counted in this measure, (2) the above model does not include non-unit-stride accesses, which would have a tendency to appear partly as scalars, and (3) although the average model error has been determined in Table 2, the error in computing the model inverse has not. Thus, this calculation should be regarded as illustrative at the time of this writing.

Conclusions

The above offers a mixture of formal and anecdotal modeling observations, representing the state of information gleaned from an initial series of dedicated tests. As this modeling process continues and all the major load parameters are identified, it may be possible to relate them both to (1) design parameters using simulation, and (2) application code performance The latter, however, will require development of a feedback model allowing the determination of loaded response from an unloaded accessing parameter characterization; this in turn necessitates a method for characterizing application code sensitivity to access delays. In summary, this effort is the first step - parameter identification - in a much longer research study.

Acknowledgements

The programming assistance of Milind Pandit is acknowledged. This research was supported by the Research Institute for Advanced Computer Science and the NAS Projects Office at Ames Research Center, and by the Air Force Office of Scientific Research under Grant AF 84-0096. Dedicated CRAY-2 time was supplied by NAS and the Air Force Weapons Laboratory, Albuquerque.

References

[1] Cheung, T. and Smith, J.E., A Simulation Study of the CRAY X-MP Memory System, Trans. IEEE, vol. C-35, (July 1986) pp. 613-622.

[2] Calahan, D.A., An Analysis and Simulation of the CRAY X-MP Memory System, Proc. First Intl. Conf. on Supercomputing Systems, St. Petersburg, FL, (December 1985) pp. 568-574.

[3] Oed, W., and Lange, O. Modelling, Measurement, and Simulation of Memory Interference in the CRAY X-MP, Parallel Computing, vol 3, no. 4 (October 1986) pp. 343-358.

[4] Calahan, D.A., An Analysis of Vector Startup Access Delays, Trans. IEEE on Computers, in print.

[5] Kuck, D.J., The Structure of Computers and Computations, vol. 1,Wiley (1978).

[6] Bailey, D.H., Vector Computer Memory Bank Contention, Trans. IEEE , Vol. C-36, No. 3 (March 1987) pp. 293-298.

[7] Calahan, D.A., "Measurements on the C-2 Memory System," Technical report in progress.

The Symmetry Multiprocessor System

Tom Lovett
Shreekant Thakkar

Sequent Computer Systems
15450 SW Koll Parkway
Beaverton, Oregon

Abstract

The Symmetry Series [Gif87] is a bus-based, shared-memory multiprocessor system which can contain from two to thirty 32-bit microprocessors with a total performance of around 100 MIPS. Each processor subsystem contains an Intel 80386/80387 microprocessor/floating point unit, optional Weitek 1167 floating point accelerator, and private cache. The system features a 53 Mbyte/sec pipelined system bus, up to 240 Mbytes of main memory, and a diagnostic and console processor. The cache hardware supports two different cache coherence policies: write-through and copyback. Symmetry represents one of the first shared-memory bus-based multiprocessor systems to use both write-through and copyback protocol with split transaction system bus. The performance of the two cache coherence policies has been measured and is compared here for various benchmarks and applications.

1. Introduction

The performance of bus-based shared-memory multiprocessors is limited by the bandwidth supplied by the bus and memory subsystems, and by the demands made on them by each processor subsystem. Typically, such multiprocessor systems use local caches to reduce a processor's demand on the bus. The use of multiple caches on a common bus causes the cache coherence problem. Many different solutions have been proposed to solve this problem [ArB86] with a wide range of cost and performance tradeoffs. The Balance multiprocessor system [TGF88] used write-through caches per processor [MaE84] to reduce the demand on the bus. Our studies [Tha87] showed that many writes generated by each processor in a bus-based shared-memory multiprocessor can be a limitation in performance when increasing the number and speed of the individual processors. Thus this cache policy was a problem in the design of Symmetry whose goal was to have 4-5 times the performance of Balance. Another design requirement of the Symmetry system was that it be an extension of the Balance system and that it remain compatible with the Balance I/O controllers. In addition, the Symmetry boards had to function when installed in Balance systems with Balance memories. This requirement led to the implementation of two different cache policies, one write-through and the other copyback, the choice of which depended on the hardware environment. Copyback caches have been shown to increase performance of a system by reducing the writes to the memory in a uniprocessor environment [AKC86]. Embedded hardware now offered us the ability to measure the performance of the system with the two different caching policies.

First, we review some of the protocols that have been described to solve the cache coherence problem. Next we describe how the Balance architecture was extended for the Symmetry Series. Finally, we describe performance of the two cache coherence schemes using bus utilization as a metric.

2. Multiprocessor Cache Protocols For Bus Based Systems

The use of multiple private caches on a bus causes the cache coherence problem; a write to main memory from a processor or input/output (IO) device must be reflected into the contents of all caches that reside on the bus. Many different cache protocols have been developed to reduce a processor's bus requirements while solving the cache coherence problem.

The simplest approach involves the use of a write-through cache. In a write-through cache each cache block is tagged as either VALID or INVALID. On a read hit the data is returned to the processor from the cache, without any bus traffic required. On a read miss the block containing the requested data is read from main memory and installed into the cache and marked VALID. The data is then passed to the processor. All writes cause the data being written to be passed over the system bus into main memory. If the block is present in the cache the cached copy is also modified. To maintain coherence in a bus-based multiple cache system all caches watch the bus for writes (thus called *snoopy* caches.). When a cache detects a write on the bus to a block marked VALID, the block is simply invalidated. The next access to that block results in a miss and the modified data is retrieved from main memory.

Since all writes are passed directly to main memory, a write-through cache does not reduce write traffic on the shared bus. This can be tolerated in systems where the bandwidth required by an individual processor is a small fraction of the bandwidth available on the bus as observed on Balance systems [Tha87]. As the individual processors get faster, the write-through protocol will consume too much bus bandwidth to support a moderate numbers of processors.

Copyback caches do not send all write traffic through to main memory. The cached copy is written locally, if present, and the modified data is not written back to main memory until the cache block is replaced. Copyback caches have the potential for removing much of the write traffic from the bus in addition to the read traffic, but at some expense in terms of complexity. Copyback caches have been shown to give at least 30% increase in performance over write through caches [AKC86]. This suggests that multiple writes are done to a block before it is replaced. Thus copyback caches represent an attractive choice for bus-based multiprocessor system because they remove these writes from the bus.

In a copyback cache there are at least three states: INVALID, VALID, and MODIFIED, where MODIFIED indicates that the block has been written locally, but main memory has not been updated. When a MODIFIED block is replaced it must be copied-back to main memory. Reads are handled as they are in write-through caches. Most copyback protocols use write allocation, so that write misses cause a block to be allocated in the cache and the data to be written into the newly-installed block. A write-miss is turned into a read-miss on the bus. Write hits cause the data to be written directly into the cache block, leaving the block MODIFIED, and generating no bus traffic.

Several different protocols have been proposed for maintaining coherency in multiple cache copyback systems [ArB86]. Most of these protocols solve the coherence problem by allowing a MODIFIED block to exist in only one cache at a time. Before a write can occur in a cache the cache must have *ownership* of the block it wants to modify. This means that the cache must have the only valid cached copy of that data in the system. A write miss at the cache requires a read on the bus to acquire the data and ownership of the block. Only in the case where the block is already in the cache and known to exist in no other cache is no bus activity required for a write. The data is written back to the memory only when the modified data needs to be replaced to make room for new data. The Berkeley [KEW85] and Illinios [PaP84] protocols are based on the ownership principal. They use write invalidation scheme to invalidate SHARED copies in other caches before a write is allowed to proceed in a cache. Thus these schemes allow multiple readers but only a single writer. Heavy active write sharing can degrade the performance of these systems because the blocks of data must be shuttled back and forth between caches sharing this data. Results from our studies suggest that active write sharing is almost insignificant in the parallel applications we ran on the Balance system [Tha87].

The Dragon [McC85] and Firefly [ThS87] protocols allow MODIFIED blocks to be held in multiple caches, but require all writes to these shared blocks to be broadcast across the bus (ie. uses write broadcast as opposed to write invalidate). A SHARED state is usually added to distinguish between unmodified blocks that are privately held and those that may exist in other caches. This allows writes to non-SHARED blocks to proceed without generating any bus traffic. These schemes work better when there is significant amount of active write sharing in the applications since they prevent shuttling of blocks between caches.

In all of these schemes caches that own MODIFIED blocks must watch the bus for accesses to those blocks and ensure that the response reflects the modifications. There are two basic approaches for this. Either the cache responds to the access itself, or it holds up the access, writes the modified data to memory, and then allows the memory to respond.

Caches must also watch the bus for accesses to blocks marked PRIVATE and SHARED and take actions appropriately. Bus transactions may cause the state of a block to change from PRIVATE to SHARED, or cause the state to change to INVALID.

3. Design Criteria for Symmetry

In 1984 Sequent introduced the Balance Series of multiprocessors, based on a shared-memory architecture and a high speed bus interconnect. The bus contains a 32 bit multiplexed address and data path and uses a split response protocol. The split response protocol releases the bus between a read request and its corresponding response. This allows the bus to be used for other transactions during an otherwise idle period. The bus protocol defines three pipes, a read pipe, a write pipe, and an IO pipe, which allows memory accesses to proceed independent of IO accesses. It also allows reads to proceed independent of write accesses, except that requests to an individual memory subsystem must be serviced by that subsystem in order of receipt. The bus protocol limits the outstanding requests to 3 read requests, 2 write requests and 1 IO request. This is done in a distributed manner, with each requester maintaining a current pipe count. If the count shows that the required pipe is full no new requests will be issued. Responses are always returned in the order of the requests, obviating the need for a requester tag. Requesters make a note of the current pipe count when their request is placed on the bus and count off the responses as they are returned. The bus, memory and processors all run synchronously at 10Mhz. The bus and memory subsystems support a sustained bandwidth of 26.7 Mbytes/sec.

Bus cycles are identified by a 5 bit Cycle Type field which is placed on the bus along with the address or data. The cycle type field identifies the current cycle as an address or data cycle, a read or write operation, and, for address cycles, the size of the transaction. Transactions from one to sixteen bytes are supported.

Bus arbitration is handled by a central arbiter. Processor priority is assigned on a rotating round robin basis with the processor which last used the bus assigned the lowest priority. When a processor receives a bus grant but cannot use the bus because the desired pipe is full, the grant is held until the pipe becomes free. This provides each processor fair access to the bus.

The cache is an 8 Kbyte, 2 way set associative cache [MaE84]. It uses a write-through protocol with bus watching to maintain coherency. The bus watching is implemented with a second set of tags so that bus watching lookups can proceed in parallel with processor cache accesses. Since the processor runs synchronous to the bus, maintaining two sets of tags is simple. Bus watching invalidations occur only on writes. Since write addresses are always followed by at least one write data cycle, the processor tags can be updated during that cycle, stealing the cycle from the processor.

This design was able to support thirty 0.7 MIP processors in many useful applications, with a wide range of benchmarks showing up to 28 effective processors. There were, however, several applications where the write traffic generated by 30 processors was enough to swamp the bus. The size of the cache was also a limitation for certain floating point applications [Tha87]. As we were contemplating implementing the system with the latest generation of microprocessors it was apparent that supporting thirty 3-4 MIP processors would require a different approach.

4. Symmetry Series

In 1986 Sequent began the design of a Symmetry Series multiprocessor system based on the Intel 80386 microprocessor. A major goal was to be able to support as many processors as the Balance Series while maintaining compatibility with the Balance peripheral controllers. Since the new processor had 4-5 times the performance of the Balance processor we needed a 4-5 fold increase in the ratio of available bandwidth to processor demand, without drastically altering the bus subsystem. The bus bandwidth was increased by doubling the width of the datapath to 64 bits. This doubles the sustainable bandwidth to 53.4 Mbytes/sec.

To reduce the demand placed on the bus by the processors we increased the cache size to 64 Kbytes per processor, and implemented a copyback protocol to remove write traffic from the bus. The protocol is similar to the Illinois cache coherency scheme [PaP84]. It has been shown that the performance of the Illinois scheme is as good as the best of the copyback schemes in systems with moderate sharing. Since our studies showed only minimal active write sharing a more complex scheme was not necessary. The Illinois scheme has been shown to have superior performance in handling PRIVATE blocks when compared to other coherence protocols [ArB86]. Our studies suggested that this was a more important consideration.

Two additional cycle type bits were added to the bus to extend the bus protocol to support copyback cache coherency scheme. The first bit is used to identify transactions using the extended 64 bit width of the bus. The second bit allows an address to be tagged with whether or not it should cause an invalidation. This can be used with a read address if a cache needs to insure that it holds the only copy of a block (ie. gain ownership).

5. Symmetry Cache and Bus Protocols

The Symmetry cache and bus protocols are related to each other to support cache coherency in the system. The Symmetry cache protocol [Gif87] makes use of four cache states: INVALID, PRIVATE, SHARED, and MODIFIED.

These states are defined as follows:

INVALID - Block is not currently valid in the cache.

PRIVATE - Block has been read and does not exist in any other cache in the system.

SHARED - Block has been read and may exist in another cache.

MODIFIED - Block has been modified and does not exist in any other cache in the system.

The System Bus in the Balance multiprocessor supported the following cycles to support the write-through protocol:

RA - Read Address cycle
WAi - Write Address with Invalidate cycle
RDF/RDL - Read data first and last cycles
WDF/WDL - Write data first and last cycle.

The System Bus protocol was extended in Symmetry to support the copyback cache coherency scheme by adding the following cycles:

RAi - Read Address with Invalidate
WA - Write Address
IA - Invalidate Address Cycle

In addition, two status lines were added to the bus to support the protocol. The first, SHARED, indicates that a RA cycle on the hit a block that exists in another cache. This lets a requester know whether to install a new block as PRIVATE or SHARED. The second, OWNED, indicates that a RA or RAI cycle on the bus hit a block that is held MODIFIED by another cache. This lets the memory subsystems know that a cache will respond to the request.

The scheme, in general, works as follows:

READ_HIT — No bus activity is required and requested data is supplied to the processor.

READ_MISS — An RA type cycle is issued on the bus. If any cache has a copy of the block of data in PRIVATE or SHARED state it changes its state to SHARED, and asserts the SHARED line on the backplane. If any cache has the data in MODIFIED state it asserts OWNED, responds to the request and changes its local state to INVALID. The state could have been changed to SHARED instead of INVALID but our implementation does not allow this. The memory subsystem observes this transaction, noting the assertion of the OWNED signal, and takes a copy of the data as it is being passed from one cache to the next (called *implied* copyback operation). This allows the cache to relinquish ownership. If no cache signals ownership then the memory responds to the request with its copy of the requested block. The receiving processor sets his tags to PRIVATE, if SHARED was not asserted, or SHARED otherwise.

WRITE_HIT — If the block is in MODIFIED state then this implies that this cache already owns the block and can complete the write. No bus activity is necessary. If the block is in the PRIVATE state, then the cache changes the state to MODIFIED and completes the write. If the block is in the SHARED state then the cache issues an

IA cycle on the bus, causing all other caches to invalidate their copies (ie. write invalidate operation), and changes its state to MODIFIED.

WRITE_MISS An RAi cycle is issued on the bus to obtain the current copy of the block and to signal all other caches to invalidate their copy. If any cache has the copy of the block in MODIFIED state then it responds to the request. Any cache which holds the block in PRIVATE or SHARED state invalidates its copy. If no cache holds the block MODIFIED then memory will respond to the request. The receiving cache installs the block as MODIFIED and completes the write.

I/O devices do not participate in the caching protocol and therefore can issue writes to blocks that caches hold MODIFIED. These WAi cycles are absorbed by the caches which own the block being written.

6. Implementation

The Symmetry system (Figure 1) consists of processor subsystems, memory subsystem, disk controller(s), SCED(s), and Multibus adapter(s). The processor subsystem and memory subsystems are implemented to support the new copyback and bus protocols.

A processor subsystem (Figure 2) consists of an Intel 80386/80387 processor and floating point unit pair, an optional Weitek floating point subsystem, Cache Memory Controllers (CMCs), Bus Interface Controller (BIC) and Bus Data Path (BDP) devices, System Link and Interrupt Controller (SLIC) [BKT87], memory chips for cache address and data fields, some address decoding logic and bus transcievers. Two such systems are implemented per board, and they are identical in all respects except they share a BIC.

The cache coherency and bus protocols form the key part of the Symmetry system and are implemented across three VLSI devices: CMC, BIC and BDP. These devices are all implemented in 1.2 micron CMOS technology; the CMC and BIC are implemented in gate arrays while the BDP in standard cell array.

The CMC has two modes, master and slave, which allow several of them to be cascaded to support set associative cache organization. In current release two CMCs are used to support a 64K byte cache. The CMC is soft configurable to vary block and transfer sizes and support a variety of configurations. The CMC communicates to the BIC when it needs access to the bus. It acts as an initiator of requests when it needs to service cache misses and as a responder to requests for accesses to blocks it holds MODIFIED (*owned* accesses). Such *owned* request addresses are queued inside the BDP. The BDP also queues addresses from bus transactions which cause invalidates. The BDP generates *owned* and *invalidate* requests to the CMC accordingly. The CMC is able to supply data to the 80386 in pipeline mode with zero wait states on cache hits.

The CMC has address, address tag and state tag inputs to allow comparison of address tags and checking of the state for each block of data. Two sets of tags are provided; processor side and bus side to allow concurrent access. The CMC accesses the bus side tags to perform bus-watching, when it needs to interrogate bus side state tags or when it installs a block. The CMC interrogates the bus side tags when it needs to differentiate between the PRIVATE and SHARED states [Gif]. The distinction is not maintained in the processor tags because of the difficulty of atomically changing states across an asynchronous boundary.

The BIC contains two channels which handle both initiator and responder functions. Each processor subsystem on a board uses one of the channels. The memory subsystems also makes use of the BIC but uses only the reponder functions of one channel. The BIC contains logic to arbitrate between the channels, make requests onto the bus, respond to owned and invalidate transactions, load/unload the BDP queues and maintain the state of read, write and IO pipes. The BIC supports the bus watching or *snooping* function to maintain coherency across the system. Following an address cycle on the bus, the BIC informs the CMC of any necessary operations to perform on the bus-side tags. The BIC also receives hit/miss information from the CMC and uses this to load the appropriate BDP queues.

The BDP contains 5 queues, an OWNED REQUEST address queue, an INVALIDATE address queue, a READ RESPONSE data queue, a WRITE DATA queue, and an OUTPUT DATA queue. The OWNED REQUEST queue contains the addresses of cycles which hit modified blocks in this processor's cache. The INVALIDATE queue contains addresses which cause cache blocks to be invalidated. Whenever the OWNED or INVALIDATE queue is loaded an owned or invalidate request is made to the CMC. The READ RESPONSE queue holds the response of read requests generated by this cache's misses. The WRITE DATA queue holds the data associated with write addresses in the OWNED REQUEST queue. The OUTPUT DATA queue holds the data associated with both owned read responses from the cache, and write requests (ie. copybacks) from the cache. Just like the CMC, both the BDP and the BIC are also software configurable to handle different block and transfer sizes.

As an example, a transaction involving a RAI cycle on the bus is handled as follows. The cycle after the address is on the bus, the address is used to search the bus side tags. If a match is detected the state tags for the block are changed to INVALID. In addition, the CMC reports the result of the lookup to the BIC. The BIC uses the result of the lookup to generate load strobes for the two address input queues in the BDP. The BDP latches the address while it is on the bus, and can load it into either queue in the following cycle. If the state is PRIVATE or SHARED the address is loaded into the INVALIDATE queue. If the state is reported to be MODIFIED, the address is loaded into the OWNED REQUEST queue and, during the second cycle after the address, the BIC will assert OWNED on the bus. The memory controller recognizes the OWNED signal and aborts its processing of the request. Once a queue is loaded the corresponding queue empty flag is deasserted to inform the CMC that the queue requires service. These owned or invalidate requests to the CMC normally have higher priority than processor requests. However, if the CMC has made a request to the bus and is waiting for a response of its own it may ignore the request. To avoid deadlock and maintain data integrity the CMC will service the OWNED REQUEST queue if and only if the request in the OWNED queue appeared on the system bus before the request from this CMC. An owned RAI is serviced by transferring the data from the cache into the BDP output queue, and changing the processsor-side state tags to INVALID. The CMC then signals the BIC to respond to the request.

Invalidate address (IA) cycles are similarly handled using the INVALIDATE queue in the BDP. If the operation is just an invalidate no data is transferred. The CMC just changes the processor-side tags to INVALID and pops the queue.

The SLIC on the Symmetry system is used for configuring the system (setting up registers in the VLSI) and for handling interrupts in the system. The gates used by the kernel for mutual exclusion in the Balance system are no longer used because of the faster transparent *parallel* locks [Gif]. Unlike the Balance system both the kernel and users use the same type of locks for mutual exclusion.

The memory subsystem can currently support up to 240 Mbytes of memory on 6 controller/expansion pairs. The controllers support two-way interleaving which allows the subsystem to support the 53 Mbyte/sec bandwidth of the bus. The BIC and BDP are also used on memory controller to support the bus interface and data path functions. The memory controller can respond to both wide and narrow transactions of from 1 to 16 bytes, and are fully compatible with the Balance environment.

The memory controllers perform two special functions that are necessary to support the copyback protocol. The first is the recognition of caches claiming ownership of blocks. When a cache asserts OWNED in response to a request on the bus, the memory must avoid responding to the request. Often the controller is near completion of the request before it recognizes ownership and must abort the request. In cases where the controller was busy when the request arrives it can merely discard the request from the BDP's queue and continue with the following request.

The second function is the *implied copyback* operation. If an RA request for a full cache block is claimed as owned by a cache the memory controller must watch for the response from the cache and grab the data as it is being passed over the bus. The data is written back into memory so that the copy in main memory is up to date and the cache can give up ownership. The BIC monitors the bus for the OWNED signal and maintains a set of flags that the controller logic examines as it pulls addresses out of the BDP. In addition the BIC determines which read response is associated with the owned RA and loads the data into the BDP's queue.

7. Performance Monitoring

The performance of the Balance system was measured using a hardware monitor, DYNAPROBE [Com77], which can measure different events for a given period. The events can be setup using a logic patch board. This proved capable but was severely limited by the number of probe points, which is a real limitation for a multiprocessor system. Thus, on Symmetry a decision was made to incorporate the performance monitoring hooks into hardware which can be accessed by special system software. The hardware includes counters, masks and multiplexing logic. The mask can be set and appropriate events of interest selected before the counters are started. The counters can be stopped and read by system software via SLIC chip.

The types of events that can be measured include all types of accesses to the CMC by the processor, accesses from the bus to CMC (i.e owned and invalidate operations), and state changes. This allows us to detect the accesses to shared blocks, etc. Other events that can be measured include the different types of bus cycles and other aspects of bus protocol. These features give us a unique opportunity to study this architecture and its behavior under different applications.

8. Performance

The performance of a multiprocessor needs to observed in the following domains:

Single Thread Performance
Parallel Program Performance
Multi Stream Performance

8.1. Single Thread Performance

In figure 3 we show the performance of two small integer benchmarks that measure the processor performance across the Balance and different Symmetry configurations. The table shows that the project goal of increasing processor performance to 4-5 times that of a Balance processor has been achieved. Figure 4 shows the floating point performance of the Symmetry processor relative to the Balance processor. Both single precision and double precision performance has increased significantly for linpack and whetstone benchmarks. The **narrow** bus indicates a 32 bit bus while **wide** bus indicates a 64 bit bus.

For small programs such as Dhrystone and Dhampstone benchmarks, in write-through system, the cache size and bus size has affect on performance. Increasing the cache size to two sets (64K bytes) increases the performance of these benchmarks by 5% and increasing the bus width to 64 bits has 3% increase in performance. The copyback system does not show such increases, increasing the cache size and bus width had a small affect for these programs (1%). There is less bus activity for copyback caches since these small program are now entirely running out of the cache.

The 64K byte cache yielded a 99% hit-rate for several integer and floating point benchmarks for both write-through and copyback protocols. This is a significant improvement over the Balance 8Kbyte cache which gave 95% hit-rate for integer benchmarks and 85% for floating-point benchmarks [Tha87].

8.2. Parallel Program Performance

Several parallel benchmarks and applications were run to observe the bus utilization of the Symmetry system with write-through and copyback caches. The benchmark and applications include:

Parallel Linpack Benchmark (LINPACK)
2D Monte Carlo Simulation (SIMC2D)
Butterfly Switch Simulator (SIM)
Ray Tracing (SMOKE)

The bus utilization shows (Figure 5) a significant improvement for the Symmetry system with copyback caches over that with write-through caches. The bus approaches saturation much faster in a system with a write-through cache and appears to reach a limit with fewer than 16 processors. The reason for bus utilization increasing rapidly in a such a system is the number of writes. Active write sharing in parallel applications does not seem to affect the bus utilization adversely since it is insignificant (Figure 6). The IA/RAI cycles indicate sharing on the bus, the RAI cycles reflect the writes to invalid blocks, IA cycles reflect write to shared blocks. The copyback policy removes the most writes from the bus and thus decreases the demand on the bus.

Note that all the results are from a system with a 64K bytes, 2 set cache and wide bus in a 16MHz environment. The benchmarks are not tuned and were run as presented to us. There may be a potential to tune the algorithms and increase their performance.

8.3. Multi Stream Performance

At present, the parallel applications represent explicit attempts at using parallelism in speeding up an application. However, as Sequent and other manufacturers of similar multiprocessors have found, a great advantage can be taken of natural parallelism in a multi-user timesharing environment to deliver superior performance. The processes are scheduled independently across the available processors, thus providing a responsive and available environment. There is little sharing between the processors, and hence little contention exists for resources. This is reflected by both high cache hit-rate and low bus utilization. Since the processes have longer time-slices than on a uniprocessor the large cache provides a large hit-rates (99%) because there are fewer context switches.

In such an environment the performance of the system can be regarded as cumulative of the total number of processors. One way to approximate multi stream performance of a system to consider running *n* copies of a program on a *n* processor system. Figure 7 shows the roll-off in runtime when running 12 copies of a suite of programs on a 12 processor write-through system and 28 copies of the same suite on a 28 processor copyback system. The suite of programs includes several benchmarks (Dhrystones, Sieve, Linpack, Whetstone, Puzzle, Sort), an Nroff application and some scientific applications (e.g, Butterfly, Gauss, Barsim). The roll-off for most of the benchmarks and programs starts occurring early in a write-through system. A gentle roll-off occurs for only one program (Butterfly) in a copyback system. The roll-offs in the write-through systems are really associated with write traffic generated on the bus. The Butterfly roll-off improves with the use of memory interleaving (not shown). Note that this is only an approximation of system performance in such environment.

9. Conclusion

Symmetry represents one of the first bus-based shared-memory multiprocessor to incorporate a copyback cache with a split transaction bus. Embedded hardware incorporates performance monitoring hooks to monitor dynamic behavior of the system. The ability to switch between write-through and copyback protocols has allowed us to observe the behavior of the two protocols for several parallel benchmarks and applications. Results show that a copyback cache has allowed us to incorporate much faster processors in a bus-based shared-memory multiprocessor. Results confirm that there is little active write sharing in parallel applications and justifies our choice for cache coherency protocols.

Acknowledgements

We would like to mention Paul Gifford, Wayne Downer, Bruce Gilbert, Harlan Courtney, and rest of the folks involved in making of Symmetry. They also thank Neal Wyse and Joe DiMartino for their contribution to performance activity; and Eugene Brooks, Jack Dongara and others for the applications and benchmarks.

References

[AKC86]
Alexander, C., Keshlear, W., Cooper, F. and Briggs, F., "Cache Memory Performance in A UNIX environment," *SIGARCH NEWS*, June 1986.

[ArB86] Archibald, J. and Baer, J. L., "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *TOCS*, vol. 4, 4 (November 1986), , ACM.

[BKT87] Beck, B., Kasten, B. and Thakkar, S. S., "VLSI Assist in Building a Multiprocessor ," *Proceedings of APLOS-II* , October 1987.

[Com77] Compten, N., Dynaprobe 7816 User Manual, 1977.

[Gif87] Gifford, P. R., "Symmetry: A Shared Memory Multiprocessor with Copy-back Caches," *Proceedings of ICCD87*, October 1987.

[Gif] Gifford, P. R., "A Write-Back Cache Design for Shared Memory Multiprocessors," *To be published*, .

[KEW85]
Katz, R. H., Eggers, S. J., Wood, D. A., Perkins, C. L. and Sheldon, R. G., "Implementing a Cache Consistency Protocol," *Proceedings of the 12th ISCA*, 1985.

[MaE84] Mayberry, W. and Effland, G., "Cache Boosts Multiprocessor Performance," *Computer Design*, November 1984.

[McC85] McCreight, E. M., "The Dragon Computer System," *Proceedings of the NATO Advanced Science Institute on Microarchitecture of VLSI Computers*, 1985.

[PaP84] Papmarcos, M. and Patel, J., "A low overhead coherence solution for multiprocessors with private cache memories.," *Proceedings of 11th ISCA*, 1984, pp. 348-354.

[ThS87] Thacker, C. P. and Stewart, L. C., "Firefly: a Multiprocessor Workstation," *Proceedings of APLOS II*, 1987.

[Tha87] Thakkar, S. S., "A Performance Analysis Of A Shared Memory Multiprocessor," *Proceedings of ICCD87*, October 1987.

[TGF88] Thakkar, S. S., Gifford, P. R. and Fieland, G. F., "The Balance Multiprocessor System," *IEEE MICRO*, February 1988.

Figure 2: The Processor Subsyster

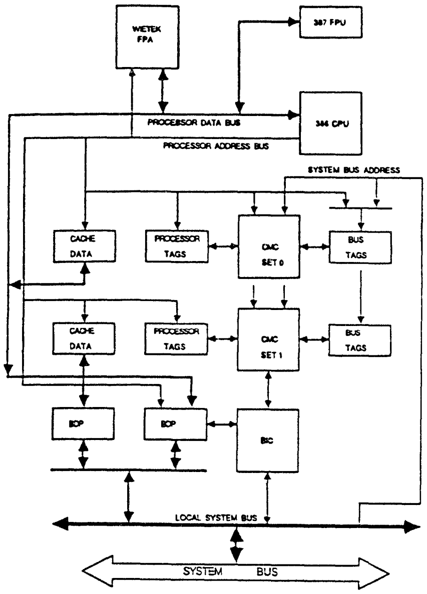


Figure 3: Relative Integer Single Stream Performance

SYSTEM	BALANCE NS32032 10 MHz	SYMMETRY 180386 16 MHz			
CACHE	WRITE-THRU	WRITE-THRU		COPYBACK	
	8K BYTES	32K BYTES	32K BYTES	32K BYTES	64K BYTES
BUS	NARROW	NARROW	WIDE	NARROW	WIDE
DHRYSTONE .1.1	1.0	2.84	3.04	4.75	4.79
DHAMPSTONE	1.0	2.97	3.17	4.74	

Figure 4: Relative Floating Point Single Stream Performance (Narrow Bus)

SYSTEMS	BALANCE NS32081 - 10 MHz	SYMMETRY 180387 - 16MHz		WIETEK - 16MHz	
	WRITE-THRU	WRITE-THRU	COPYBACK	WRITE-THRU	COPYBACK
LINPACK (D	1	2.41	2.58	3.29	3.97
LINPACK (S	1	1.32	1.43	2.35	3.14
WHETD (D)	1	3.64	4.63	4.84	7.41
WHETS (S)	1	3.13	3.82	5.72	8.76

Figure 1: Block Diagram of Symmetry System

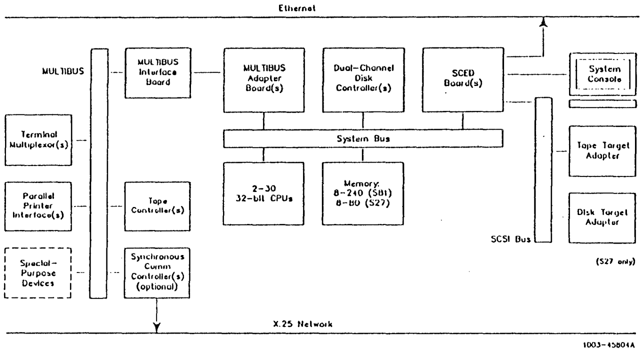


Figure 7: MultiStream Roll-Offs, Write-Thru vs Copyback System

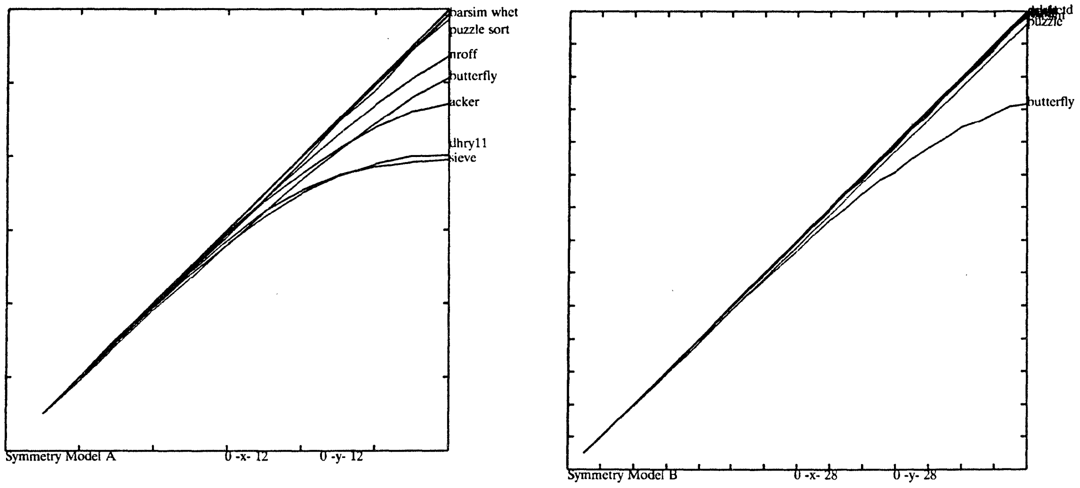


Figure 5: Bus Utilization for Parallel Applications

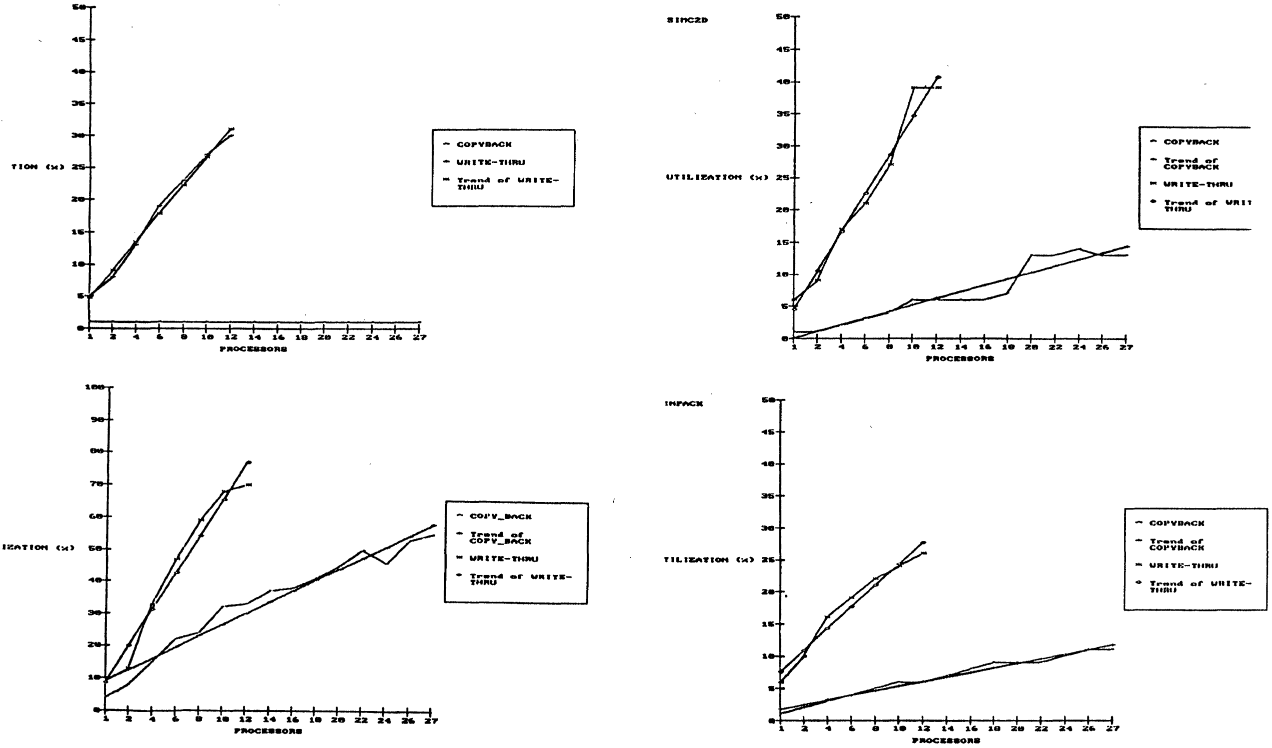
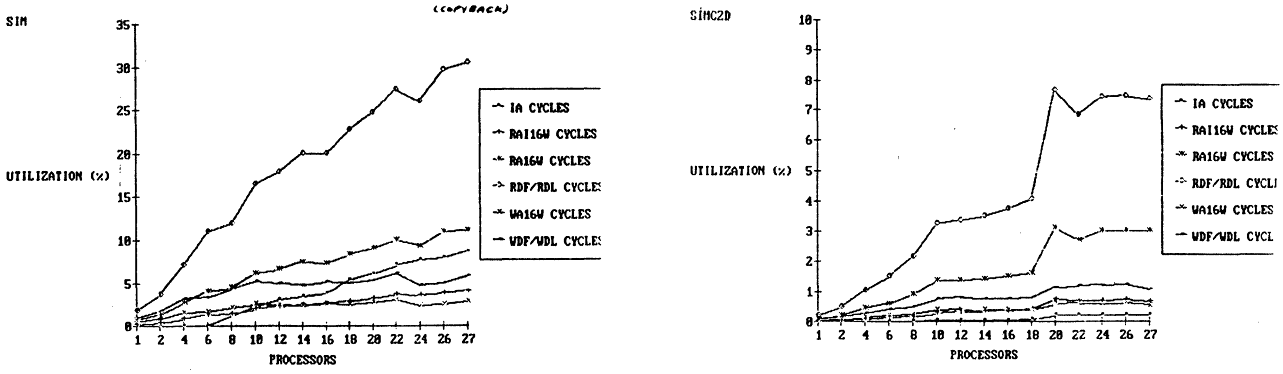


Figure 6: Different Bus Cycles For Sim & Simc2d



Two Parallel Processing Aspects of The CRAY Y-MP Computer System

Steve Reinhardt
Cray Research, Inc.
Mendota Heights, MN 55409

ABSTRACT

The CRAY Y-MP computer system is a parallel processing supercomputer. The architecture of the Y-MP is an evolutionary step from the CRAY X-MP series of computers; in many ways, the Y-MP is a "bigger X-MP". The Y-MP system provides eight CPUs compared to four for the X-MP. On the Y-MP, the CPUs share 32 million words of central memory, compared to a maximum of 16 million words on the X-MP. The clock period of each CPU decreases from 8.5 nanoseconds on recent X-MP models to 6.0 nanoseconds on the Y-MP.

Many of the features of the X-MP which allow it to run common programs fast seem to be features which are particularly hard to scale to systems with more CPUs. In particular, the design and implementation of the shared registers and multiple ports from each CPU to the central memory require care to preserve high performance as the number of CPUs grows. We investigate how the central memory responds to different levels of memory traffic and how the shared register access times affect the size of critical regions in common usage. In short, we look at how well the X-MP architecture scales from four processors to eight.

1. Introduction

This paper looks at the evolution of the CRAY-1 to the CRAY X-MP to the CRAY Y-MP to see how the architectural decisions made during that evolution affect the parallel processing capabilities of the Y-MP. We begin with a short historical perspective and describe the areas of the CRAY Y-MP computer system which are of particular importance to parallel processing. Then we look at the central memory system and how the X-MP and Y-MP respond to frequently occurring memory access patterns. Next we see how the X-MP and Y-MP shared register reference times affect the size (time of execution) of multiprocessing synchronization, which affects the size of critical sections of code which may be profitably processed in parallel. The emphasis is on how the X-MP/Y-MP architecture affects the ability of one program to use the whole machine effectively.

2. The CRAY-1 and CRAY X-MP Computer Systems

The first CRAY-1 computer system was delivered in 1976.[1] The central processing unit (CPU) is of register-to-register type. The clock period is 12.5 nanoseconds. Address registers are 24 bits wide and scalar registers are 64 bits wide. No virtual memory support is provided. Central memory provides one port to the CPU.

The CRAY X-MP represented the first move by Cray Research into parallel processing and began a concentration on parallel processing which continues today. The Cray approach to parallel processing is to make the fastest general-purpose scientific processor possible and then to put together as many of those processors as possible. Architecturally this tends to be an evolutionary, incremental approach.

The basic CPU architecture of the CRAY X-MP is the same as that of the CRAY-1. New features were three ports to central memory and flexible chaining. The first CRAY X-MP computer system (1982) contained two CPUs.[2] When one- and four-CPU models (1984) were introduced, hardware gather/scatter was added. Multi-CPU models provide clusters of shared registers, each of which includes a set of one-bit semaphores and shared address and scalar registers. Early X-MP models had a 9.5 nanosecond clock period; all models produced since 1986 have a 8.5 nanosecond clock period.

3. The CRAY Y-MP Computer System

Each CPU of the CRAY Y-MP computer system is nearly identical to that of the CRAY X-MP. (In fact, X-MP binaries can be run in an X-MP compatibility mode.) To support larger address spaces, address registers on the Y-MP are 32 bits wide. The 8 CPUs run at 6.0 nanoseconds.

The central memory is 32 million words arranged in 256 banks. The banks are interleaved on the low-order bits of the address; thus consecutively addressed words reside in separate banks. Each word is 64 data bits plus 8 check bits for SECDED. The bank cycle time is 5 clock periods (that is, each bank can only be referenced every 5 clock periods). Each CPU provides four ports to memory: two read ports, a write port, and an I/O port. The two read ports and the write port are under direct control of the CPU. The I/O port may be in use independently of the state of the other three ports. Block memory operations can use all three CPU ports simultaneously. Scalar memory operations wait until all block transfers are quiet to ensure correct sequences between block memory operations and scalar operations within a CPU.

A multiprocessing program on the CRAY Y-MP synchronizes its CPUs via a set of shared registers (a *cluster*) which includes 32 semaphore bits, eight 32-bit shared B (SB) registers, and eight 64-bit shared T (ST) registers. The semaphore bits have atomic test-and-set, unconditional set, and clear instructions. The SB and ST registers may be read and written. Access to the SB and ST registers in a parallel program must be controlled with the semaphores.

Physically, the Y-MP is a compact machine. The logic chassis consists of 41 modules: 8 CPU modules, 32 memory modules, and 1 clock module. Each module is 11" x 21.2" x 1.4". Each module lies flat. The clock module is on the bottom, then 16 of the memory modules, the 8 CPU modules, and the other 16 memory modules. The footprint of the mainframe (including power supplies) is 79" long, 32" wide, and the machine is 76" tall.

4. Parallel Processing Scalability

In parallel processing, one major distinguishing point among machines is whether the CPUs share a central memory or whether most of the system memory is local to the CPUs. Shared-memory multiprocessors are generally considered easier to program; private-memory multiprocessors are considered easier to scale.[3] Because the X-MP and Y-MP are shared-memory multiprocessors, we are very interested in scalability of that type of machine. In the CRAY X-MP/Y-MP line, the speed of two particular features is crucial to the parallel processing ability of the machine. The central memory must provide high bandwidth to all processors and minimize the effects of contention. The shared register clusters must synchronize CPUs quickly and allow fast communication with low overhead.

4.1. Main Memory

The CRAY-1 provides one port between the CPU and memory. A program can slow down due to memory conflicts because of I/O references or because the program strides through memory and re-references banks before the bank cycle time has expired. A program which uses only stride-one references will not have bank conflicts. The CRAY X-MP and Y-MP provide three ports between each CPU and memory. Memory banks are grouped into sections (both X-MP and Y-MP) and subsections (Y-MP only). In addition to the possible memory conflicts in a CRAY-1, additional conflicts may arise from two memory operations from the same CPU competing with each other. In a multiple-processor X-MP/Y-MP system, additional conflicts arise from separate CPUs competing with each other.[4] The CRAY Y-MP has no architectural difference from the CRAY X-MP in terms of CPU to memory connections, so the differences in memory contention should be attributable to the different number of processors, number of banks, bank cycle times, number of sections, etc.

3

To illustrate the effects of memory contention, we have chosen one algorithm, matrix multiplication, and coded it in three different ways, each with its own amount of memory contention and bandwidth requirement. Matrix multiplication was chosen because it is important for many applications, well-known, simple to code, and highly parallel. Each algorithm does N-1 floating-point adds and N floating-point multiplies for each element of an NxN matrix. The algorithms run at different speeds even on a single CPU; what we want to emphasize here is the speedup of each algorithm relative to itself. Each algorithm is parallelized by the use of Cray's microtasking software[5] [6].

The first algorithm (VXS) is a FORTRAN code which does 3N memory references for each element of the resulting matrix. Because VXS uses three memory ports per clock period per CPU in its vector portions, it is referred to as a 3-port algorithm.

The second algorithm (SDOT) is a FORTRAN code which does 2N memory references for each element of the resulting matrix; it is a 2-port code. Both VXS and SDOT are Basic Linear Algebra Subroutines (BLAS) Level 1 routines in the LINPACK naming scheme.[7]

The third algorithm (MXV) is a single FORTRAN loop which calls the assembler-coded mxv library routine to multiply a matrix times a vector. The mxv routine does N memory references for each element of the resulting matrix; it is a 1-port code. The mxv routine is a BLAS Level 2 LINPACK routine.

Table 1 has both megaflops per second (MFLOPS) and speedup numbers for the above algorithms for a 1000x1000 matrix on the CRAY X-MP and CRAY Y-MP systems. For this size problem, multiprocessing overheads and granularity of work are not issues, and total performance is related primarily to memory conflicts. MXV requires one port per clock period and scales linearly, since its memory traffic requirement is well within the bandwidth of the machine. SDOT scales moderately well on an X-MP; the higher total memory bandwidth of the Y-MP gives a better 4-CPU speedup on the Y-MP than the X-MP. VXS requires the full memory bandwidth (3 ports) of each CPU; on either machine speed-ups are well below linear. Again, the Y-MP gives a better speed-up for a given number of CPUs. (These runs were made on the prototype hardware which at the time of these tests allowed VXS to run on up to 7 processors. Also, at the time of the tests, the clock period of the prototype was 6.33ns rather than 6.0ns of the production systems.)

1000 x 1000 matrices

1000 x 1000

Algorithm	#CPUs	X-MP Speedup	X-MP MFLOPS	Y-MP Speedup	Y-MP MFLOPS
VXS	1	1.00	164	1.00	215
	2	1.90	312	1.93	417
	3	2.60	427	2.84	612
	4	2.97	488	3.74	805
	5			4.70	1011
	6			5.25	1131
	7			6.31	1358
	8				
SDOT	1	1.00	82	1.00	145
	2	1.93	59	1.97	286
	3	2.82	232	2.90	420
	4	3.69	303	3.75	544
	5			4.66	676
	6			5.34	774
	7			6.32	917
	8			7.15	1038
MXV	1	1.00	220	1.00	296
	2	2.00	440	2.00	592
	3	3.00	659	3.00	886
	4	3.99	878	4.00	1183
	5			5.00	1480
	6			5.99	1772
	7			7.00	2070
	8			8.00	2366

x2

Table 1. Matrix Multiply Speedups

From these results, one can conclude that MXV is the best of these algorithms for matrix multiplication, and that in general the smaller the memory load the better the speed-up. This is not surprising. The speed-ups for different memory loads can also help a programmer predict how well another algorithm (with some known memory load) will scale (subject to degree of parallelism). A machine-independent conclusion is that an important measurement of a parallel machine may be not only how much memory load can it support per CPU, but also how much memory load can it support and still achieve linear speed-ups for highly parallel code.

4.2. Shared Registers

The only architectural difference between X-MP shared registers and Y-MP shared registers is that the Y-MP shared B-registers are 32 bits to match the Y-MP A-registers. X-MP A-registers and shared B-registers are 24 bits. The Y-MP has 9 clusters, the *p*-CPU X-MP has *p+1*.

The time to execute instructions which reference the shared registers has lengthened from the X-MP to the Y-MP. One reason for the slowdown is physical proximity. In the 2-processor CRAY X-MP, each CPU comprises about 140 double modules in half of four columns. The CPUs are situated one on top of the other. For speed, the shared registers are located near the boundary between the CPUs. In the 4-processor CRAY X-MP, each CPU requires the same space as the 2-processor. The CPUs are situated as four corners of a checkerboard, touching at the center of the 8 columns they occupy. The shared registers are located close to the point where all four CPUs meet. In the CRAY Y-MP, each CPU consists of one double module. The CPUs lie flat, one on top of another. The shared registers are distributed across all eight CPU modules. Thus the furthest two CPUs are 7 module slots away from each other. Also, with more CPUs, the shared register arbitration logic has grown. (Shared register access times slowed down from the 2-processor X-MP to the 4-processor X-MP for this reason).

Because of these factors, the access time for the shared registers and semaphores has increased. On the 2-processor X-MP, to test-and-set a cleared semaphore takes one clock period (CP). To read or write a shared register takes 1 CP (assuming no other CPUs are accessing shared registers). On the 4-processor X-MP, to test-and-set a cleared semaphore takes 1 CP. Reading/writing a shared register takes 3 CP for instruction issue (again assuming no conflicts). On the Y-MP, to test-and-set a cleared semaphore takes 3 CP to issue. Reading/writing a shared register takes 3 CP to issue (again assuming no conflicts). However, on the Y-MP, once the instruction issues, the cluster is blocked for another 3-7 CP, depending on the instruction, and a second cluster reference will hold for an extra 3-7 CP. A read from a shared register waits 7 CP after issue before the data is ready to be used. In all X-MP models and the Y-MP, shared register references are slowed down by contention with other CPUs by an amount proportional to the number of CPUs competing for access.

Determining the effect of the slower shared-register accesses on actual programs[8] is difficult because generally they have large enough granularity that time spent in critical regions is small. (A critical region is a portion of code in which only one CPU may be executing at a time.) Instead, we look at how the execution time of the critical regions used in microtasking has increased as the machine type and number of CPUs changes. Predictions of how that effects efficiency for a particular algorithm are algorithm- and granularity-dependent.[9] [10] We use a parallel program which starts a parallel loop and measures how long each CPU takes to enter the parallel region and get its first piece of work. See Figure 1 for the source. (The initial set-up time of getting extra CPUs from the operating system is ignored; the numbers are for the case where all CPUs are waiting at the beginning of the critical region.) Table 2 shows the incremental times for the *i*th processor to get its work after the *i-1*th processor completes. 1

On both the X-MP and the Y-MP, the cost for the initial CPU to set up the loop is larger than the times for subsequent CPUs. The times for subsequent CPUs are not influenced by the number of CPUs competing. This means that adding more CPUs to a program will not slow down critical region executions by prior CPUs.

As the access time for the shared registers grows, the speed advantage of using them instead of memory decreases. The advantage on the CRAY X-MP/2, for instance, is 1 CP to read a shared register versus 14 CP to read from central memory. On the CRAY Y-MP, the difference is 10 CP for the shared register versus 18 CP to memory. Indeed, some work has been done to look at the possibility of using only the semaphores of the shared register sets and not using the SB and ST registers, keeping all data values in memory instead. 2

```
      SUBROUTINE TIMEIT
      IMPLICIT INTEGER ( A - Z )
      PARAMETER ( TRIPLEN = 100 , NUMCPUS = 8 )
      COMMON /CLOCKS/ICPU,ICPU2,OUTCS(NUMCPUS,4)
CMIC$ GUARD
      IF (ICPU.LE.NUMCPUS) THEN    ! determine CPU id
       ICPU=ICPU+1                 ! to store timings later
       MCPU=ICPU
      ENDIF
CMIC$ END GUARD

CMIC$ DO GLOBAL
      DO 100 I=1,TRIPLEN             ! first loop just gets all
       DO 50 JSPR = 1,10             ! CPUs local to subroutine
        X = SQRT(1.0)
 50    CONTINUE
100   CONTINUE

     OUTCS4 = 0
     OUTCS3=IRTC()
CMIC$ DO GLOBAL                       ! use the second loop copy for
      DO 200 I=1,TRIPLEN              ! timing, so all CPUs are local
       IT = IRTC()
       IF(OUTCS4.EQ.0) THEN          ! time for this CPU to enter is
        OUTCS4 = IT                 ! OUTCS4 - MIN(OUTCS(i,3))
       ENDIF                       ! for 1 <= i <= NUMCPUS
       DO 150 JSPR = 1,1000
        X = SQRT(1.0)               ! waste some time
150    CONTINUE
200   CONTINUE

     OUTCS(MCPU,3)=OUTCS3             ! store timings off CPU id
     OUTCS(MCPU,4)=OUTCS4
      RETURN
      END
```

Figure 1. Shared Register Timing Source

However, going to memory incurs a subtle penalty. In the case of a vector code, much of the time of taking the next piece of work (that is, executing the critical region) can be overlapped with vector instructions which have not yet completed. However, if the critical regions kept key variables in memory instead of shared registers, the references to memory for those variables would force a hold issue condition waiting for the vector memory references to complete. Thus the shared registers function in a sense as an independent memory port.

CPUs	X-MP/2	X-MP/4	Y-MP/8
1	78	136	155
2	31	50	67
3		48	67
4		46	67
5			67
6			67
7			67
8			67

Table 2. Critical Region Times (in clock periods)

5. Conclusions

We have looked at two specific aspects of the CRAY Y-MP which are important to using the whole machine for one program, and compared timings for the Y-MP to its X-MP predecessor. In the area of memory contention, the performance of matrix multiplication coded to use only one memory port per clock period per CPU scales linearly with the number of processors. Using three memory ports per clock period per CPU produces less than linear speedups. From this we may conclude that for parallel applications the number of ports from each CPU to central memory may not be as important as the number of ports which can be referenced from the same program on all CPUs and still provide linear speed-ups.

In the area of shared registers we looked at execution times of the standard Cray microtasking critical regions. These timings have slowed down on the Y-MP, but not as much as the timings for the shared register instructions might predict. A positive result of the timings is that extra CPUs do not slow down the speed of the critical regions for prior CPUs. Thus there is no need to worry about getting too many CPUs into a program and hurting overall performance. The slowdown of the critical region highlights the difficulty of providing very fast shared registers as the number of communicating CPUs grows. Since most 4-CPU multitasking codes have granularity sufficient to make synchronization overhead very small, the effect of the longer critical regions on real applications is unclear.

6. Acknowledgements

The Y-MP development groups headed by Les Davis in Chippewa Falls deserve lots of credit for creating a machine on which measurements can be done. Rick Pribnow patiently explained the shared registers. Jeff Nicholson and Mike Booth gave pointers as to how to measure shared register times in a meaningful way. Paul Leskar shared many long hours in the machine room doing software checkout. Chuck Grassl wrote the different matrix multiplication programs used in the memory tests. John Larson and Mark Furtney edited drafts and improved clarity in several places.

7. References

[1] Johnson, P. M., "An Introduction to Vector Processing," Computer Design, February 1978, pp. 89-97.

[2] Chen, S. S., "Large-scale and High-speed Multiprocessor System for Scientific Application - CRAY XMP Series," Proceedings NATO Advanced Research Workshop on High Speed Computation, J. Kowalik, ed., Springer Verlag, Julich, West Germany, June 1983.

[3] Hwang, K., and Briggs, F., "Computer Architecture and Parallel Processing". McGraw-Hill, New York, 1984.

[4] Oed, W., and Lange, O., "Modelling, measurement and simulation of memory interference in the CRAY X-MP", Parallel Computing, Vol. 3 (1986) pp. 343-358.

[5] Booth, M., and Misegades, K., "Microtasking", Cray Channels, Summer 1986, pp. 24-27, Cray Research, Inc., Mendota Heights, MN.

[6] CRAY X-MP Multitasking Programmer's Reference Manual. Publication SR-0222, Cray Research, Inc., 1987.

[7] Dongarra, J.J., Moler, C.B., Bunch, J.R., and Stewart, G.W., Linpack User's Guide. SIAM, Philadelphia, PA, 1979.

[8] Calahan, D.A., "Task granularity studies on a many-processor CRAY X-MP", Parallel Computing, Vol. 2 (1985), pp. 109-118.

[9] Hockney, R.W., "(r(infinity), n(half), s(half)) Measurements on the 2-CPU CRAY X-MP," Parallel Computing, Vol. 2 (1985), pp. 1-14.

[10] Cornelius, H., "Some Timings for Synchronisation on the Multiprocessor System CRAY X-MP", Parallel Computing, Vol. 2(1985) pp. 457-462.

LOOPS AND MULTI-DIMENSIONAL GRIDS ON HYPERCUBES: MAPPING AND RECONFIGURATION ALGORITHMS

Shyh-Kwei Chen, Chung-Ti Liang, and Wei-Tek Tsai

Computer Science Department
University of Minnesota
Minneapolis, Minnesota 55455, U.S.A.

ABSTRACT

This paper investigates reconfiguration algorithms for loops and multi-dimensional grids in a hypercube architecture. The reconfiguration algorithms are invoked when a fault is detected and the original loop or multi-dimensional grid is no longer valid. The reconfiguration algorithms are able to reach an equivalent set of topologies within the same architecture in a distributed manner. We also propose fault-tolerant mapping strategies for embedding a loop or a multi-dimensional grid into a hypercube so that the resulting mapping facilitates the reconfiguration.

1. Introduction

Recently the problem of mapping algorithms to various computer architectures has received much attention in parallel processing [1, 2], VLSI systolic algorithm design [3], distributed processing and fault-tolerant computing [4, 5, 6, 7, 8, 9]. In parallel processing, specific algorithms are mapped into a set of processors connected by a certain inter-connection network. For example, Fast Fourier Transformation and bitonic sorting algorithms can be mapped into perfect-shuffle, butterfly, hypercube, mesh-connected networks and so on [2, 3, 10, 11, 12, 13]. Based on algorithm transformations, any iterative algorithm can be partitioned and mapped into fixed size VLSI systolic arrays [14].

More recently, mapping a basic inter-connection network into a host is also considered by many researchers [15]. This is usually done by exploiting the structures of both the embedded and the host networks. For instance, approaches to map loops, trees, and mess-connected networks into a hypercube have been proposed in [16, 17].

Once we have mapped a basic network into a host, it is desirable that if the host has a fault, it can reconfigure itself so that the basic network can continue operation with minimum interrupts. The reconfiguration process can be either centralized or decentralized. However, the centralized scheme has several drawbacks, e.g., the vulnerability of the global supervisor, the lack of uniformity of each processor, and the tedious information collection, computation, and distribution. Decentralized scheme uses only local information but can achieve the same goal via the cooperation of processors.

Studies have been done on the design of the host for some basic networks so that the reconfiguration can be carried out. In most cases, the host is actually constructed by adding some spare processors and links into the basic network, and the most popular application is to add redundant links and processors to loop and tree networks [18, 19, 20, 21, 22, 23].

However, in many cases, we do not have such luxury to build the underlined host machine, and once the host machine is selected, it is fixed. This is a more realistic assumption, since most machines we use are manufactured by computer companies and, except for minor memory, CPU or I/O upgrade, they can not have drastic changes. Since the host machine is fixed, the problem is then to find intelligent mapping algorithms to map the basic network into the host so that the reconfiguration can be carried out easily. In this paper, we discuss this problem. The host machine of interest is a hypercube and the basic networks are loops and multi-dimensional grids networks.

Hypercube is selected as the host machine because it has many desirable features [16, 24, 25, 26]:

(1) Each node has the same number of direct neighbors n, hence it is possible to overlap n different data transfers from any given node to its n neighbors to fully utilize the high total bandwidth of hypercube;

(2) It has small diameter thus minimizing the communication cost;

(3) A variety of basic graphs can be embedded in a hypercube. For example, trees, loops, multi-dimensional grids, and so on have been mapped into hypercubes; and

(4) Its homogeneity and symmetry properties make each node equivalent in fault detection and recovery, by which we can achieve reconfiguration in a homogeneous and distributed manner;

Some of the properties of hypercubes may be realized more efficiently by other networks individually [8]. For example, DeBruijn graphs have shorter logarithmetic internode distance and richer connectivity, tree networks are most suitable for divide and conquer type algorithms, also grid structures can be embedded with loop and linear arrays. Hypercube, however, has many desirable properties as mentioned above. Most importantly, several versions of hypercube architectures are now commercially available, which makes experimental installment of our algorithms possible. [27]

Loops and multi-dimensional grids networks are chosen since they are widely used for variety of applications [2, 11, 12, 28]. We propose fault-tolerant mapping strategies and corresponding distributed reconfiguration algorithms for the loop networks, and then apply them to the multi-dimensional grids networks.

2. Backgrounds

In [16], topological properties of hypercubes were examined. A hypercube of degree d is an undirected graph with 2^d nodes labeled 0 through $2^d - 1$. There is an edge between a given pair of nodes if and only if the binary representations of their labels differ by exactly one bit. In this paper, we will represent a hypercube of degree d by drawing 2^{d-3} independent 3-cubes. We assume each 3-cube always has the most significant 3 bits (MSBs) labeling as illustrated in Figure 2.1, while the less significant bits appear on top of each 3-cube in a left to right order, i.e., the least significant bit (LSB) is in the rightmost position.

To distinguish the basic network and the host, we use the term **vertex** to represent the processor of basic networks, and **node** that of host. For simplicity, we use M_a as the image of a loop vertex a into the hypercube, i.e., M_a can be viewed as a d-bit binary code. *XOR* is the *exclusive-or* operator and can be performed on at least 2 operands which are binary codes.

It is easy to see that a hypercube of degree d has subgraphs which are loops of length 4, 6, ... , 2^d respectively. If the length of the loop equals 2^d, the nodes of the hypercube are fully utilized. However, if the length is less than 2^d, then $(2^d - n)$ nodes are left. We can use these remaining nodes as spares to adjust the mapping in case of faults so that the new mapping is still a loop of length n.

Another attractive property of hypercubes is that an n-dimensional grid of size $2^{m_1} \times 2^{m_2} \times \cdots \times 2^{m_n}$ can be perfectly mapped in a d-cube, where d is the sum of m_i's [16]. All the 2^d nodes of this d-cube are occupied. However if we use a larger hypercube of degree d+1 as the host graph, then 2^d nodes remain to be spares, which can be used to adjust the mapping in the presence of faults so that the topology of the multi-dimensional grid can still be maintained.

3. Failure Model

The failure model used in this paper follows the model given in [23]. Each processor of the basic network has a unique state. The system is in an operational state if and only if all the distinct states exist. A fault will cause the missing of a state. The system should be able to reconfigure itself distributedly via the local operations of faulty-free processors until the missing state is recovered.

The usages and definitions are as follows.

- A node M_i is in **active** state if it is one of the image nodes of the basic graph and faulty-free. It is in **faulty** state if it is faulty. If none of the above cases applies, then it is in **spare** state. Figure 3.1 illustrates the state transition diagram of each node. We represent the spare state by 0, the faulty state by −1, and each active state by a unique positive integer.
- $S(M_i)$ is the state of node M_i. Let **M** denote a set of nodes, $(M_1, M_2, \ldots, M_i)$. The state of **M** is $S(\mathbf{M}) = (S(M_1), S(M_2), \cdots, S(M_i))$, $i \geq 1$.
- $E(S(M_i))$ represents the fault that state $S(M_i)$ is missing.
- A system state is **valid** if and only if *all* the n active states are present such that the active nodes labeled by the states constitute a loop or multi-dimensional grid. Otherwise it is **invalid**.
- (M_j, M_i) is a **recovery pair** if M_i is responsible for the fault of M_j, and we say M_i is the **parent** and M_j is the **child**. A node M_i can detect the states of all its neighbors. It updates the state information in its local copy, and recovers any faulty child. The recovery actions are assumed faulty-free.

We list other assumptions about the failure model.

- Reliable fault diagnosis mechanisms are assumed available [29, 30, 31]. To ensure the correctness, each node M_i periodically tests itself. If it is faulty, the state of M_i becomes −1.
- A node M_i can detect the states of all its neighbors. It updates the state information in its local copy, and recovers any faulty child. The recovery actions are assumed faulty-free.
- The links are faulty-free.
- Only one error can be present in the system.

4. A Reconfiguration Algorithm for Embedded Loops in Hypercubes

In this section, we describe the system specification, propose a distributed reconfiguration strategy and three initial mapping schemes. More examples and detailed proofs of this paper are in [32].

4.1. State Assignment and Detection

Let L_n be a loop of length n mapped into a d-cube, where n is even and $n \leq 2^d$. Each vertex x of L_n has a unique label representing the state of x. And each node of the d-cube, C_d, has a unique ID represented as a Gray code [2]. Let $C^d[L_n]$ be the resulting mapping. Every node of $C^d[L_n]$ can be viewed as having a unique d-bit **ID code** and up to n+2 possible states : n active states denoted by the distinct labels of L_n, a spare state denoted by 0 and a faulty state denoted by −1. Figure 4.1 illustrates the basic graph L_{10}, and $C^4[L_{10}]$ associating with a labeling.

The state of $C^d[L_n]$ can be expressed as an 2^d-tuple $S(M_1, M_2, \cdots, M_{2^d}) = (S(M_1), S(M_2), \cdots, S(M_{2^d}))$. The system state may become invalid in case of faults. For example, consider the system shown in Figure 4.1(b), (1, 0, 0, 5, 2, 0, 3, 4, 10, 0, 9, 6, 0, 0, 8, 7) is a valid system state, while (1, 0, 0, 5, 2, 0, 3, 4, 10, 0, 9, 6, 0, 0, –1, 7) is invalid since state 8 is missing.

The node in state i responds to the missing of the state i+1, i.e., ($E(i$+1)), where the addition is modulo-n. M_i would check the state of node M_j periodically, save and update the work environment of node M_j. If M_j is faulty, M_i will recognize this fault during its next detection, and provide the saved work environment of M_j to reconfigure the system to a valid state.

The amount of overhead associated with the saving and updating of the work environment depends on the frequency of state updates. This involves the problem of optimal placement of checkpoints, which takes as factors the state vector of each node, the reliability of each node, the degree of urgency of fault recovery etc.

4.2. Strategy and Algorithm

In this section we propose a **distributed, homogeneous** strategy with which each node can recover the fault of state missing of its child.

Lemma 4.1 Suppose vertices i, j, and k are three consecutive vertices of a loop, and (M_j, M_i), (M_k, M_j) are recovery pairs, then there exists a node M_δ, so that M_i, M_j, M_k and M_δ constitute the four corner of a 2-D plane of the hypercube [16].

According to Lemma 4.1, we can get $XOR(M_i, M_j, M_k) = M_\delta$. That means if each active node M_i has the backup work environment of its child, say M_j, the code of M_j, and the code of M_j's child, say M_k, then it is possible to reconfigure the system under any possible single fault $E(S(M_j))$. When the state of M_j has lost, node M_i can detect this fact and compute $XOR(M_i, M_j, M_k)$ to get the code of M_δ which is adjacent to it. It will then reset the node M_δ by sending the missing state, $S(M_j)$, backup work environment and code M_k. If node M_δ is a spare node, the system has reached a consistent valid state so that it can resume operation. However, if $S(M_\delta) > 0$, then a new fault, $E(S(M_\delta))$, occurs which in turn will be detected and recovered by M_δ's parent. It seems that the fault will be propagated along the loop and finally be absorbed by a spare node. It is easy to see that this strategy will result in a homogeneous and distributed algorithm.

However, the strategy does not work for all initial mappings. Consider a counter example shown in Figure 4.2. A loop of length 12 is mapped into a hypercube of degree 4. Suppose $E(2)$ occurs. New faults are generated in sequence of $E(4)$, $E(6)$, $E(8)$, $E(10)$, $E(12)$, to recover the previous fault. The system then gets stuck when it is trying to recover state 12 to a faulty node. Hence the correctness of the proposed strategy really depends on initial mappings.

We exclude the case that the length of the loop equals 4, since no feasible spare nodes are available and any kinds of initial mappings will only lead to a face of the hypercube. Nevertheless, we still can solve this problem by moving two adjacent nodes, one faulty node and one active node, at the same time. In other words, more than local information is required.

4.3. Initial Mappings

To design the initial mappings, two alternative strategies, I and II, may be taken for consideration. Strategy I is a general scheme which can map any loop of even length l into a d-cube, where $4 \leq l \leq 2^d - 2$. It ensures the correctness of reconfiguration process under any single fault, though the overhead may be large. On the other hand, Strategy II is to minimize the number of steps, in terms of state changes, needed to reconfigure a faulty system back to a valid state. It is normally employed when our major concern is to reduce the communication and state switching overhead.

We propose the general scheme in Section 4.3.1 and give two examples for schemes of strategy II in Section 4.3.2.

4.3.1. Mapping I

Definition The parity of a node M_a whose code equals $x_1 x_2 \cdots x_d$ is **even** if $XOR(x_1, x_2, \cdots, x_d) = 0$; is **odd** if $XOR(x_1, x_2, \cdots, x_d) = 1$, where $x_i = 0$ or 1, for $1 \leq i \leq d$.

Definition Suppose a loop of length n has been mapped into a hypercube of degree d. Let $M_0, M_1, \ldots, M_{n-1}$ be the corresponding image nodes. A sequence $b_1, b_2, \ldots, b_n$ is a **transition sequence** if M_{i-1} differs from M_i by the b_ith MSB, for $1 \leq b_i \leq d$, $1 \leq i \leq n-1$, and M_{n-1} differs from M_0 by the b_nth bit. We say b_i is the **transition** from node M_{i-1} to M_i.

Example Consider the initial configuration shown in Figure 4.1(b), let $M_0 = 1000$, i.e., the code of the node whose state is 2. Then $M_1 = 1100$ (in state 3), $M_2 = 1110$ (in state 4), ... , $M_9 = 0000$ (in state 1), and the transition sequence from b_1 to b_{10} equals 2, 3, 1, 4, 1, 3, 1, 2, 4, 1.

It is easily seen that if $b_1, b_2, \ldots, b_n$ is a transition sequence for some mapping, $n > 2$, then $b_i \neq b_{i+1}$. And if $\alpha = b_i$, for $1 \leq \alpha \leq d$ and $1 \leq i \leq n$, then α must appear even number of times.

Definition Let the **transition graph**, as shown in Figure 4.3, denote that (M_j, M_i) is a recovery pair, and M_i differs from M_j by the kth MSB.

Definition Let $A(i)$ denote the maximun length of a loop which can be mapped into a hypercube of degree $i-1$ to tolerate any single fault, where $i \geq 2$.

We have $A(i)=2^{i-1}-2$, and $A(i+1)-A(i)=2^{i-1}$.

The proposed initial mapping reserves two consecutive feasible spare nodes, i.e., one is with **odd** parity and the other with **even** parity. This can be done by separating a 3-cube, and assigning the states of *six* consecutive loop vertices and two 0 states, to corner nodes, which is shown in Figure 4.4 (a) and (b). M_2 and M_2^* are **end** nodes.

We can generate a loop of length six by adding dashed arrow and transition 1 from M_2 to M_2^* (Figure 4.4(a)).

In general, when $2^i \leq n \leq 2^{i+1}-2$, for $i \geq 3$, and n is even, we can construct a new loop of length n from a loop of length $n-2$ by deleting the dashed line which labeled $n-2$, including two new nodes into the loop, $(M_{(n-2)/2}, M^*_{(n-2)/2})$, and adding one arrow between them. The transition graph is shown in Figure 4.5, where b_β depends on some b_i, $1 \leq i < (n-2)/2$.

Definition The ith **interval** contains codes and transition sequence from $M_{A(i)}$ to $M_{A(i+1)}$ (or from $M^*_{A(i+1)}$ to $M^*_{A(i)}$). The intersection between $(i-1)$th and (i)th interval is not empty, but includes $M_{A(i)}$. Hence the transition sequence, $b_{A(i)+1}$, $b_{A(i)+2}$, ... $b_{A(i+1)}$, belongs to the ith interval, where $b_{A(i)+1} = i+1$, for $i \geq 2$.

We propose the rules to decide the transitions, b_β :

$$b_1 = 3$$
$$b_2 = 2$$
$$b_{A(i)+1} = i+1, \quad i \geq 3$$
$$b_{A(i)+j} = b_{A(i)-j+2}, \quad 2 \leq j \leq 2^{i-2} + 1$$
$$b_{A(i)+j} = b_{j-2}, \quad 2^{i-2} + 2 \leq j \leq A(i+1) - A(i)$$

In summary, we copy the reverse transitions of the $(i-1)$th interval to that of the first half of the ith interval, and copy the transitions of the $(i-1)$th interval exclude the first transition, $b_{A(i-1)+1}$, to that of the second half of the ith interval.

Example Let $d = 5$, $M_0 = 00000$. Applying the proposed rules, we can obtain the transition graph as shown in Figure 4.6. $M_0 \rightarrow M_1 \rightarrow \ldots \rightarrow M_5 \rightarrow M_5^* \rightarrow M_4^* \rightarrow \ldots \rightarrow M_0^* \rightarrow M_0$ is a loop of length 12, and at least 4 bits are required. While $M_0 \rightarrow M_1 \rightarrow \ldots \rightarrow M_7 \rightarrow M_7^* \rightarrow M_6^* \rightarrow \ldots \rightarrow M_0^* \rightarrow M_0$ is a loop of length 16, and at least 5 bits are required.

We show that the proposed initial mapping indeed forms a loop in Theorem 4.1, and outline its single-fault reconfigurability in Theorem 4.2.

Lemma 4.2 for any possible i and j, $i \neq j$, we have

(1) $M_i \neq M_j^*$;

(2) $M_i \neq M_j$, $M_i^* \neq M_j^*$.

Theorem 4.1 M_i, M_i^* form a loop of length $2n+2$, for $0 \leq i \leq n$.

Proof Since $M_i \rightarrow M_{i+1}$, and $M_i^* \leftarrow M_{i+1}^*$, for $0 \leq i \leq n-1$; and $M_0 \leftarrow M_0^*$, $M_n \rightarrow M_n^*$, and for all distinct i, j, $M_i \neq M_j$, $M_i^* \neq M_j^*$, so no overlapping is possible. Hence $M_0 \rightarrow M_1 \rightarrow \ldots . M_n \rightarrow M_n^* \rightarrow M_{n-1}^* \rightarrow \ldots . M_0^* \rightarrow M_0$ is a loop of length $2n+2$.

□

Lemma 4.3 If i is even, $XOR(M_i, M_{i+1}, M_{i+2}) = M_{i-1}$. If i is odd, $XOR(M_i, M_{i+1}, M_{i+2}) = M_{i+3}$, for $i \geq 1$.

Lemma 4.4 If i is even, $XOR(M_i^*, M_{i+1}^*, M_{i+2}^*) = M_{i-1}^*$. If i is odd, $XOR(M_i^*, M_{i+1}^*, M_{i+2}^*) = M_{i+3}^*$, for $i \geq 1$.

Lemma 4.5 Let $M_A = XOR(M_0, M_1, M_2)$, and $M_A^* = XOR(M_0^*, M_1^*, M_2^*)$. We have $M_A \neq M_i$, $M_A \neq M_i^*$ $M_A^* \neq M_i$, $M_A^* \neq M_i^*$ for $i \geq 0$, and $i \neq A$. In other words, M_A and M_A^* are spare nodes.

Theorem 4.2 The system with the proposed initial mapping and algorithm can tolerate any single fault.

4.3.2. Mapping II

This section presents two initial mapping schemes which need only **one** step to reconfigure to a valid state in case of a fault.

Lemma 4.6 Let code $M_0 = x_1^0 \, x_2^0 \cdots x_d^0 = 000 \cdots 0$. And let code $M_i = x_1^i \, x_2^i \cdots x_d^i = x_2^{i-1} \, x_3^{i-1} \cdots x_d^{i-1} \, \hat{x}_1^{i-1}$, for $1 \leq i \leq 2d-1$, where $x_j^i = 0$ or 1, and $1 \leq j \leq d$. $\hat{x}$ is the complement of x. Then nodes M_0, M_1, ... , M_{2d-1} form a loop of length $2d$.

Lemma 4.7 Let code $M_0^* = y_1^0 \, y_2^0 \cdots y_d^0 = 100 \cdots 001$, where bit $y_j^1 = 0$ or 1 and $1 \leq j \leq d$. Codes $M_i^* = y_1^i \, y_2^i \cdots y_d^i = y_2^{i-1} \, y_3^{i-1} \cdots y_d^{i-1} \, \hat{y}_1^{i-1}$, for $1 \leq i \leq 2d-1$, where $y_j^i = 0$ or 1 and $1 \leq j \leq d$. $\hat{y}$ is the complement of y. We have $XOR(M_{i-1}, M_i, M_{i+1}) = M_i^*$, for $0 \leq i \leq 2d-1$, where addition and subtraction are modulo-$2d$.

Theorem 4.3 For a d-cube, we can map a loop of length $2d$, such that the system with the resulting mapping and the proposed strategy can reconfigure any single fault within 1 step.

Proof According to Lemma 4.6, M_i's form a loop of length $2d$. We can include arrows to define the parent-child relations.

$$M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow \ldots . \rightarrow M_{2d-2} \rightarrow M_{2d-1} \rightarrow M_0$$

So we know M_{i-1} is M_i's parent, and M_i is M_{i+1}'s parent. Since $XOR(M_{i-1}, M_i, M_{i+1}) = M_i^*$ (Lemma 4.7) and M_i^*'s are spare nodes, M_i^* can be used immediately when node M_{i-1} detects the fault $E(S(M_i))$.

□

As a matter of fact, we can start from any node instead of the nodes with only 1 or 2 blocks of 0's and 1's in their codes, e.g., 0000 $\cdots$ 0000, 0011 $\cdots$ 11 etc.

Theorem 4.4 For any node N_0 in a hypercube of degree d, let N_1, N_2, N_3, ... , N_{2d-1}, N_0 be generated by the transition sequence 1, 2, ... , d, 1, 2, ... , d. A loop of length $2d$ with image nodes N_0, N_1, N_2, ... , N_{2d-1} can tolerate any single fault within 1 step, for any possible starting node N_0.

4.3.3. Performance Analysis

Let $c(M_i)$ be the number of reconfiguration steps needed when node M_i becomes faulty. We can analyze the performance of Mapping I by computing the average number of reconfiguration steps as follows.

Let M_0 , M_1 , M_2 , ... , M_{n-1} , M_{n-1}^* , M_{n-2}^* , ... , M_0^* be the mapping sequence of a loop of length 2n. Then, the total number of reconfiguration steps summing from M_0 to M_{n-1} is:

Case I: n-1 is odd

$$\sum_{i=0}^{n-1} c(M_i) = [\, c(M_1) + c(M_3) + \ldots\ldots + c(M_{n-3})\,]$$
$$+ c(M_{n-1}) + c(M_0) + [\, c(M_2) + \ldots\ldots + c(M_{n-2})\,]$$
$$= (1 + 2 + \ldots\ldots + \frac{n-2}{2}) + 2 + 2 + (\frac{n-2}{2} + \frac{n-4}{2} + \ldots\ldots + 1)$$
$$= \frac{n^2-2n}{4} + 4 = O(n^2)$$

Case II: n-1 is even

$$\sum_{i=0}^{n-1} c(M_i) = [\, c(M_1) + c(M_3) + \ldots\ldots + c(M_{n-2})\,]$$
$$+ c(M_0) + [\, c(M_2) + c(M_4) + \ldots\ldots + c(M_{n-1})\,]$$
$$= (1 + 2 + \ldots\ldots + \frac{n-1}{2}) + 2 + [\,(\frac{n-1}{2} + \frac{n-1}{2})$$
$$+ (\frac{n-1}{2} + \frac{n-3}{2}) + \ldots\ldots + (\frac{n-1}{2} + 1)\,]$$
$$= \frac{n(n-1)}{2} + 2 = O(n^2)$$

Similarly, the result for summing from M_0^* to M_{n-1}^* is the same. Thus, we have $O(n)$ as the average number of reconfiguration steps for Mapping I.

The one-step reconfigurable property seems to make Mapping II a better choice than Mapping I in terms of the number of reconfiguration steps, especially when n is $O(2^d)$. However Mapping II allows only limited length of loops.

5. A Reconfiguration Algorithm for Embedded Multi-Dimensional Grids in Hypercubes

This section discusses the reconfiguration algorithm and initial mapping strategy for mapping a multi-dimensional grid of size $2^{m_1} \times 2^{m_2} \times \cdots \times 2^{m_n}$ into a hypercube of degree $d+1$, where $d = \sum_{i=1}^{n} m_i$.

5.1. State Assignment and Detection

We identify each vertex of a $2^{m_1} \times 2^{m_2} \times \cdots \times 2^{m_n}$-grid A by an n-tuple $A(i_1, i_2, \cdots, i_n)$, where $0 \leq i_j \leq \mu_j - 1$ and $\mu_j = 2^{m_j}$.

Definition The state of grid vertex $A(i_1, i_2, \cdots, i_n)$ is defined as follows.

$$\begin{aligned} S(A(i_1, i_2, \cdots, i_n)) &= i_1\mu_2\mu_3 \cdots \mu_n + i_2\mu_3\mu_4 \cdots \mu_n \\ &+ i_3\mu_4\mu_5 \cdots \mu_n + \cdots\cdots \\ &+ i_{n-1}\mu_n + i_n + 1 \\ &= \sum_{j=1}^{n} i_j a_j + 1 \end{aligned}$$

where $a_n = 1$, and $a_j = \prod_{k=j+1}^{n} \mu_k,\ 1 \leq j < n$.

Hence the system can have up to $\mu_1 \times \mu_2 \times \cdots \times \mu_n$ active states, one invalid state –1, and one spare state 0.

Definition A state missing due to the fault of a node changing its state to –1 is called an **original fault.** If the fault is induced by a state propagation, it is called a **propagated fault.** These two faults are distinguishable since the original fault will change the state to –1, while the propagated fault will switch the node to a state which is greater than 0.

Definition We define the new parent-child relationship as follows. Node $A(i_1, \cdots, i_j, i_{j+1}, \cdots, i_n)$ is responsible for detecting and recovering the fault of node $A(i_i, \cdots, i_j + 1, i_{j+1}, \cdots, i_n)$, where $1 \leq j \leq n$ and the addition is modulo-μ_j.

5.2. An Initial Mapping Strategy for Multi-dimensional Grids

Theorem 5.1 Let G_i^m denote the ith code in an m-bit Gray code sequence, where $G_0^m = 00 \cdots 0$, and $0 \leq i \leq 2^m - 1$. Let F_i^m denote the code of the ith loop state when we map a loop of length 2^{m-1} into a hypercube of degree m. For any vertex $W = A(i_1, i_2, \cdots, i_n)$, let $M_W = G_{i_1}^{m_1} \,||\, G_{i_2}^{m_2} \,||\, \cdots \,||\, G_{i_{n-1}}^{m_{n-1}} \,||\, F_{i_n+1}^{m_n+1}$, then these image nodes still maintain a multi-dimensional grid of size $2^{m_1} \times 2^{m_2} \times \cdots \times 2^{m_n}$.

Proof

(1) Two nodes of the hypercube are adjacent *iff* they differ by one bit.

(2) Two vertices $X = A(i_1, i_2, \cdots, i_n)$ and $Y = A(j_1, j_2, \cdots, j_n)$ of the grid are connected *iff* there exists exactly one index α, such that all the other entries are the same, and $i_\alpha = j_\alpha \pm 1$, i.e., $i_l = j_l$, for $1 \leq l \leq n$, and $l \neq \alpha$.

Case (a) : If $\alpha \neq n$:

$$M_X = G_{i_1}^{m_1} \,||\, G_{i_2}^{m_2} \,||\, \cdots \,||\, G_{i_\alpha}^{m_\alpha} \,||\, \cdots \,||\, G_{i_{n-1}}^{m_{n-1}} \,||\, F_{i_n+1}^{m_n+1},$$
$$M_Y = G_{i_1}^{m_1} \,||\, G_{i_2}^{m_2} \,||\, \cdots \,||\, G_{i_\alpha \pm 1}^{m_\alpha} \,||\, \cdots \,||\, G_{i_{n-1}}^{m_{n-1}} \,||\, F_{i_n+1}^{m_n+1},$$

Since $G_{i_\alpha \pm 1}^{m_\alpha}$ differs from $G_{i_\alpha}^{m_\alpha}$ by one bit, so M_X differs from M_Y by one bit. (Note that M_X and M_Y are in the same dimension α.)

Case (b) : If $\alpha = n$:

$$M_X = G_{i_1}^{m_1} \,||\, \cdots \,||\, G_{i_{n-1}}^{m_{n-1}} \,||\, F_{i_n+1}^{m_n+1},$$
$$M_Y = G_{i_1}^{m_1} \,||\, \cdots \,||\, G_{i_{n-1}}^{m_{n-1}} \,||\, F_{i_n}^{m_n+1}, \text{ (or } F_{i_n+2}^{m_n+1} \text{)}$$

According to the proposed loop construction, we know $F_{i_n}^{m_n+1}$ differs from $F_{i_n \pm 1}^{m_n+1}$ by one bit, so M_X differs from M_Y by one bit.

This completes the proof.

□

5.3. Recovery Strategy

Suppose node $y = A(i_1, i_2, \cdots, i_j, i_{j+1}, \cdots, i_n)$ generates a fault, either original or propagated, it will be detected by $x = A(i_1, i_2, \cdots, i_j - 1, i_{j+1}, \cdots, i_n)$, where $1 \leq j \leq n$. The recovery procedure for the parent x is as follows.

Reconfiguration Algorithm

Case (1): If $j = n$: performs loop reconfiguration along its nth dimension. In other words, either a spare node can immediately be available or a propagated fault is generated which will propagate along the nth dimension until it is absorbed.

Case (2): If $j \neq n$ and the error is original : changes its state $S(x)$ to -1, thus generates an original fault which can be detected by $A(i_1, i_2, \cdots, i_j - 2, i_{j+1}, \ldots, i_n)$ and $A(i_1, i_2, \cdots, i_k - 1, i_{k+1}, \cdots, i_j - 1, i_{j+1}, \cdots, i_n)$, for $1 \leq k \leq n$ and $j \neq k$.

Case (3): If $j \neq n$ and the error is a propagated fault : ignores this kind of fault to ensure the parallel propagations.

For each $X = A(i_1, i_2, \cdots, i_{n-1}, j_n)$, we have a loop of length 2^{m_n} along its nth dimension which has 2^{m_n+1} nodes. In other words, we can fix $i_1, i_2, \ldots, i_{n-1}$ and vary j_n to get a Gray code sequence of length 2^{m_n}.

Since we can vary $i_1, i_2, \ldots, i_{n-1}$ and do the same thing, we have a total of $\prod_{i=1}^{n-1} 2^{m_i}$ such "parallel" loops. Hence we have lemma 5.1 and Theorem 5.2.

Lemma 5.1 If $Y = A(j_1, j_2, \cdots, j_n)$ is faulty, then $2^{m_1} \times 2^{m_2} \times \cdots \times 2^{m_{n-1}}$ original faults, including $E(S(Y))$, will be generated by performing the proposed reconfiguration algorithm.

Theorem 5.2 The grid system with the proposed algorithm and initial mapping can tolerate any single fault within $O(\prod_{i=1}^{n} 2^{m_i})$ steps.

Example Figure 5.1 is an example of the above strategy. Suppose node 0010010 (in state 10) becomes faulty, say event (0), a sequence of events may occur. In this example, only original faults are generated to simplify the description, while faults which can not be recovered immediately should produce further propagated faults along the nth dimension, and eventually will be absorbed by a spare node.

6. Conclusion

In this paper, we have presented distributed reconfiguration algorithms for loops and multidimensional grids embedded in hypercubes. We have also proposed initial mapping algorithms to map loops and grids on hypercubes to facilitate reconfiguration.

We are currently investigating one-step reconfigurable schemes which are more efficient, in terms of the degree of hypercubes, than Mapping II. We also invstigate some engineering issues such as the effect of the reliability of each node on our mapping schemes.

7. Reference

[1] Agrawal, D. P., Janakiram, V.K., and Pathak, G. C., "Evaluating the Performance of Multicomputer Configurations," IEEE Computer, Vol. 19, No. 5, May 1986, pp. 23-37.

[2] Quinn, M.J., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, NY, 1987.

[3] Ullman, J.D., *Computational Aspects of VLSI*, Computer Science Press, Rockville, Md, 1984.

[4] Kuhl, J.G., and Reddy, S.M., "Distributed Fault-Tolerance for Large Multiprocessor Systems," Proc. of Sym. on Computer Architecture, 1980, pp. 23-30.

[5] Koren, I., "A Reconfigurable and Fault-Tolerant VLSI Multiprocessor Array," Proc. 8th Annu. Symp. Comput. Architecture, May 12-14, 1981.

[6] Clarke, E.M., and Nikolaou, C.N., "Distributed Reconfiguration Strategies for Fault-Tolerant Multiprocessor Systems," IEEE Trans. on Computers, Vol. C-31, No. 8, Aug. 1982, pp. 771-784.

[7] Pradhan, D.K., "Fault-Tolerant Processor Network Architectures," IEEE Trans. on Computers, Vol. C-34, No. 5, May 1985, pp. 434-447.

[8] Pradhan, D.K., "Fault-Tolerant Multiprocessor and VLSI-Based System Communication Architectures," in *Fault-Tolerant Computing: Theory and Techniques, Vol. 2*, Prentice Hall, Englewood Cliffs, New Jersey 07632, pp. 467-576.

[9] Kuhl, J.G., and Reddy, S.M., "Fault-Tolerance Considerations in Large Multiple-processor Systems," IEEE Computer, Vol. 19, No. 3, March 1986, pp. 56-67.

[10] Stone, H.S., "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Computers, Vol. C-20, No. 2, Feb. 1971, pp. 153-161.

[11] Thompson, C.D., and Kung, H.T., "Sorting on a Mesh-Connected Parallel Computer," CACM, Vol. 20, No. 4, Apr. 1977, pp. 263-271.

[12] Nassimi, D., and Sahni, S., "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Trans. on Computers, Vol. C-28, No. 1, Jan. 1979, pp. 2-7.

[13] Wiley, P., "A Parallel Architecture Comes of Age at Last," IEEE Spectrum, Vol. 24, No. 6, June 1987, pp.46-50.

[14] Moldovan, D.I., and Fortes. J.A.B., "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," IEEE Trans. on Computers, Vol. C-35, No. 1, Jan. 1986, pp. 1-12.

[15] *First International Conference on Hypercube Multiprocessors*, SIAM 1986

[16] Saad, Y., and Schultz, M.H., "Topological Properties of Hypercubes," Research Report YALEU/DCS/RR-389, June 1985.

[17] Wu, A.Y., "Embedding of Tree Networks into

Hypercubes," J. of Parallel and Distributed Computing 2, 1985, pp.238-249.

[18] Kwan, C.L., and Toida, S., "Optimal Fault-Tolerant Realizations of Hierarchical Tree Systems," Proc. Fault-Tolerant Computing, Oct. 1980, pp.176-178.

[19] Raghavendra, C.S., Avizienis, A., and Ercegovac, M.D., "Fault Tolerance in Binary Tree Architectures," IEEE Trans. on Computers, Vol. C-33, No. 6, June 1984, pp. 568-572.

[20] Raghavendra, C.S., Gerla, M., and Avizienis, A., "Reliable Loop Topologies for Large Local Computer Networks" IEEE Trans. on Computers, Vol. C-34, No. 1, Jan. 1985, pp. 46-55.

[21] Kartashev, S.P., and Kartashev, S.I., "Data Exchange Optimization in Reconfigurable Binary Trees," IEEE Trans. on Computers, Vol. C-35, No. 3, March 1986, pp. 257-273.

[22] Liang, C.T., Chen, S.K., and Tsai, W.T., "An Approach to Reconfigure a Fault-Tolerant Loop Systems" TR 86-58, Computer Science Department, Univ. of Minnesota, Minneapolis, Dec. 1986

[23] Yanney, R.M., and Hayes, J.P., "Distributed Recovery in Fault-Tolerant Multiprocessor Networks" IEEE Trans. on Computers, Vol. C-35, No. 10, Oct. 1986, pp. 871-879.

[24] Pease, M.C., "The Indirect Binary n-Cube Microprocessor Array," IEEE Trans. on Computers, Vol. C-26, 1977, pp. 458-473.

[25] Bhuyan, L.N., and Agrawal, D.P., "Generalized Hypercube and Hyperbus Structures for a Computer Network," IEEE Trans. on Computers, Vol. C-33, 1984, pp. 323-333.

[26] Saad, Y., and Schultz, M.H., "Data Communication in Hypercubes," Research Report YALEU/DCS/RR-428, Oct. 1985

[27] Seitz, C.L., "The Cosmic Cube," CACM, Vol. 28, Jan. 1985, pp. 22-33.

[28] Liu, M.T., "Distributed Loop Computer Networks," Advances in Computers, Vol. 17, M.C. Yovits, Ed. N.Y.: Academic, 1978, pp. 163-221.

[29] Meyer, G.G.L. and Masson, G.M., "An Efficient Fault Diagnosis Algorithm for Symmetric Multiprocessor Architectures," IEEE Trans. Comput., vol. C-27, pp. 1059-1063, Nov. 1978

[30] Armstrong, J.R. and Gray, F.G., "Fault Diagnosis in a Boolean n-cube Array of Microprocessors," *IEEE Trans. Comput.,* vol. C-30, pp. 587-590, Aug. 1981

[31] Bhat, K.V.S., "An Efficient Approach for Fault Diagnosis in a Boolean n-Cube Array of Microprocessors," *IEEE Trans. Comput.,* Vol. C-32 No. 11, Nov. 1983, pp. 1070-1071

[32] Chen, S.K., Liang, C.T., and Tsai, W.T., "Loops and Multi-Dimensional Grids on Hypercubes : Mapping and Reconfiguration Algorithms," TR 87-33, Computer Science Department, Univ. of Minnesota, Minneapolis, June 1987

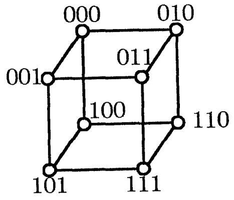


Figure 2.1 The fixed codes for a 3-cube

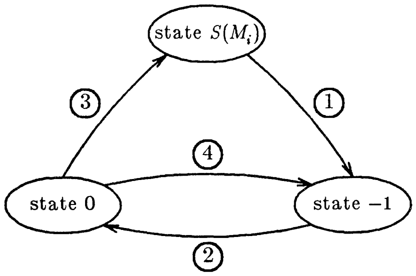


Figure 3.1 The state transition diagram of node M_i
Transition 1 --- Node M_i becomes faulty
Transition 2 --- Faulty node is repaired
Transition 3 --- Spare node is activated
Transition 4 --- Spare node becomes faulty

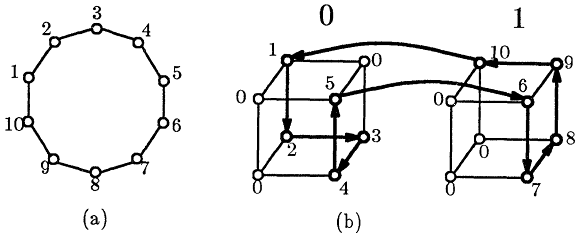


Figure 4.1 An example of an initial configuration for $n = 10,\ d = 4$
(a) Basic Graph for L_{10}
(b) The initial configuration of $C^4[L_{10}]$

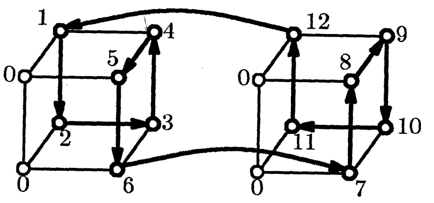


Figure 4.2 A counter example of the proposed strategy

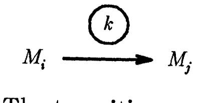


Figure 4.3 The transition graph from M_i to M_j.

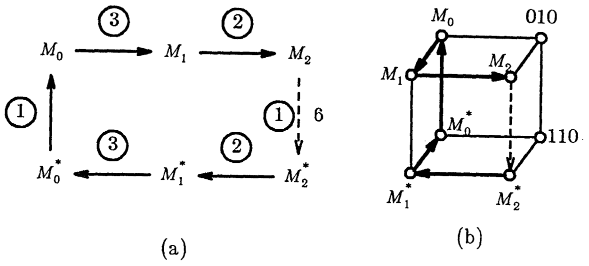


Figure 4.4 Reserve two consecutive nodes, 010 and 110, as spare nodes.

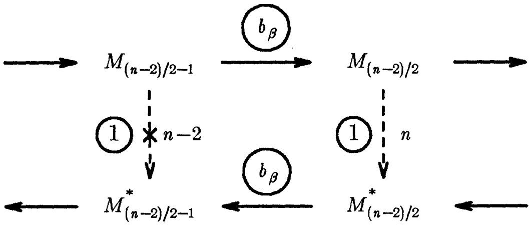


Figure 4.5 The attached transition graph

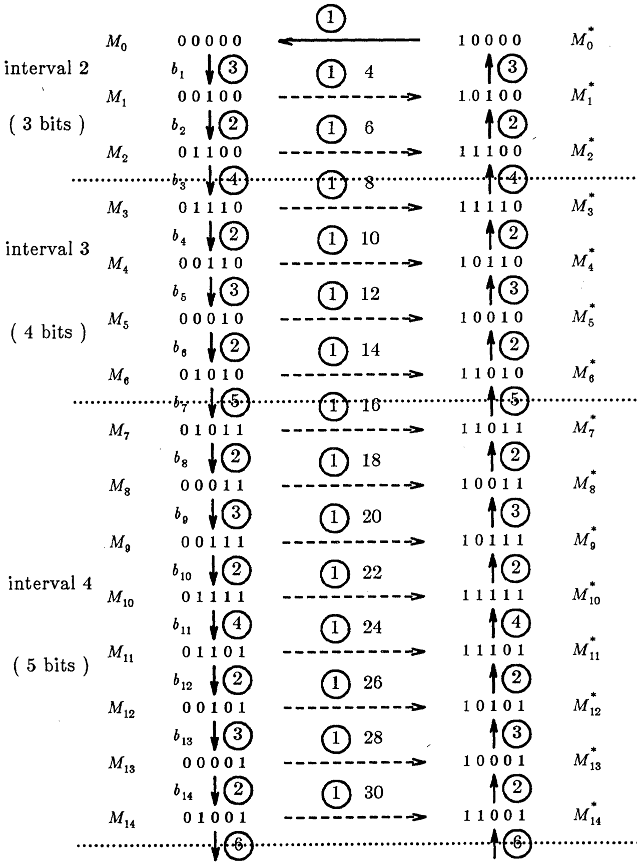


Figure 4.6 Transition Graph for d = 5, M_0 = 00000.

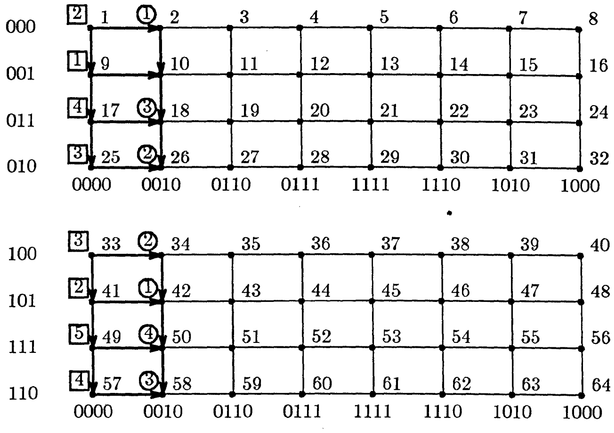


Figure 5.1 An example of how recovery protocol works when node in state 10 of a 2 × 4 × 8 grid becomes faulty

Dynamic Computational Geometry on Meshes and Hypercubes

Laurence Boxer *
Department of Computer and
Information Sciences
Niagara University
Niagara University, NY 14109, USA

Russ Miller †
Department of Computer Science
226 Bell Hall
State University of New York
Buffalo, New York, 14260, USA.

Abstract

Parallel algorithms are given for determining geometric properties of systems of *moving objects*. The properties investigated include nearest (farthest) neighbor, closest (farthest) pair, collision, convex hull, diameter, and containment. Several of these properties are investigated from both the dynamic and steady-state points of view. Efficient, and often optimal, implementations of these algorithms are given for the mesh and hypercube.

1 Introduction

Suppose n point-objects are moving in Euclidean space such that for each object, every coordinate of its motion is a polynomial of time. For such a system, we present parallel algorithms described in terms of abstract data movement operations to solve a variety of problems involving proximity, collision, containment, and convexity. We give solutions to these problems for the dynamic situation and for the steady-state situation (as time approaches infinity). We give implementations of these algorithms that are asymptotically optimal on the mesh and are efficient on the hypercube.

This paper was motivated by serial algorithms for dynamic computational geometry given in [Atal85].

2 Preliminaries

The notations O, Θ, and Ω will be used in this paper to mean, intuitively, "order at most," "order exactly," and "order at least," respectively (see, *e.g.*, [Mill88a]).

We will use the terms *processor* and *processing element (PE)* interchangeably.

2.1 Mesh-Connected Computer

A *mesh of size n* has n PEs arranged as an $n^{1/2} \times n^{1/2}$ lattice. The PEs of a mesh of size n are frequently numbered from 0 to $n-1$ so as to impose an order upon them. In this paper, we assume that the PEs are indexed via *proximity order* [Mill88a] (see Figure 1). The properties of proximity order that are useful to us are the following.

1. In a mesh of size n, if $0 \leq i < n-1$ then PE_i and PE_{i+1} are neighboring PEs.
2. A mesh may be recursively subdivided into sub-meshes such that each sub-mesh contains consecutively indexed PEs.

Let Σ be a nonempty subset of the processors of a mesh. We say Σ is an *interval* or a *string* of the mesh if and only if there are integers i_0 and i_1, $0 \leq i_0 \leq i_1 < n$, such that $\Sigma = \{PE_i | i_0 \leq i \leq i_1\}$. We will prefer the term *string* in this paper, as we will often use the term *interval* to refer to a subset of the real line.

0	1	14	15
3	2	13	12
4	7	8	11
5	6	9	10

Figure 1: Proximity order for a mesh of size 16.

2.2 Hypercube Computer

A *hypercube of size n*, where n is a nonnegative integral power of 2, has n PEs or *nodes* indexed by the integers $\{0, 1, \ldots, n-1\}$. If we view each integer in the index range as a $(\log_2 n)$-bit string, two PEs are connected by a bidirectional communication link if and only if their indices differ in exactly one bit.

It is useful to re-label the PEs of a hypercube so that consecutively labeled PEs are adjacent, and so that we may split the hypercube into subcubes such that the subcubes consist of consecutively labeled PEs. A commonly used method of ordering the PEs of a hypercube with these properties is the *binary reflected Gray code* [Rein77]. Throughout this paper, processors in a hypercube will be labeled *not* by node number, but according to a binary reflected Gray code ordering.

A *string* of processors in a hypercube will be a nonempty set of consecutive processors according to Gray code order, *i.e.*, a set Σ of PEs for which there are integers i_0 and i_1 such that $0 \leq i_0 \leq i_1 < n$ and $\Sigma = \{PE_j | i_0 \leq j \leq i_1\}$ according to a binary reflected Gray code ordering of the processors.

2.3 Pieces and the function λ

Input to problems in this paper consists of descriptions of real-valued, or more generally, Euclidean vector-valued functions $f_0(t)$, $f_1(t)$, $\ldots$, $f_{n-1}(t)$ defined on the interval $[0, \infty)$. We assume that at the start of a problem, no processor contains a description of more than one of the functions $f_0, \ldots, f_{n-1}$. For many problems, these functions describe the motion of point-objects $P_0, \ldots, P_{n-1}$, respectively, in Euclidean d-dimensional space. If every component of every function f_i is a polynomial of degree no greater than k, then the collective movement

*Research partially supported by a grant from the Niagara University Research Council.

†Research partially supported by National Science Foundation grant number DCR-8608640.

of the points is referred to as *k-motion*. For convenience, we assume that no pair of the points have the same initial position. That is, $f_i(0) \neq f_j(0)$ for $i \neq j, 0 \leq i, j < n$.

Given a set of real-valued functions $F = \{f_0, \ldots, f_{n-1}\}$ defined on $[0, \infty)$, it is often useful to describe the minimum function

$$h(t) = \min\{f_0(t), \ldots, f_{n-1}(t)\}. \tag{1}$$

Define a *piece of the minimum function generated by F* to consist of a description of some f_i and an interval $I \subset [0, \infty)$ such that $h = f_i$ identically on I and such that h is not identically equal to any f_j over any interval $J \subset [0, \infty)$ such that I is properly contained in J. A *piece of the maximum function generated by F* is defined similarly.

If $h_1(t)$ and $h_2(t)$ are real-valued functions defined on $[0, \infty)$ whose pieces are generated by a family of functions F, then a *piece of $h_1 - h_2$ generated by differences of members of F* consists of a description of a function g and an interval $I \subset [0, \infty)$ such that

1. there exist $f_1, f_2 \in F$ such that $g = f_1 - f_2$ identically on $[0, \infty)$,
2. $h_1 - h_2 = g$ identically on I, and
3. $h_1 - h_2$ is not identically equal to g on any interval $J \subset [0, \infty)$ such that I is a proper subset of J.

Many of our algorithms have processor requirements related to the number of pieces of a minimum function $h(t)$. Let $\lambda(n, s)$ be the maximum number of pieces of functions $h(t) = \min\{f_0(t), \ldots, f_{n-1}(t)\}$, where the maximum is taken over all sets $F = \{f_0, \ldots, f_{n-1}\}$ of continuous real-valued functions defined on $[0, \infty)$, no pair of which intersects more than s times.

To describe the behavior of $\lambda(n, s)$, we use the "inverse Ackermann function" $\alpha(n)$, a description of which is given in [Hart86]. Note that $\alpha(n)$ is a monotone nondecreasing function that grows to ∞ extremely slowly. For example, [Hart86] shows that

$$\alpha(n) \leq 4 \text{ for } n \leq 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}, \text{ (the number of 2's in the tower is 65536,)}$$

and that if we denote $\log^{(1)} n = \log n$, and more generally, $\log^{(k+1)} n = \log(\log^{(k)} n)$ for integer $k > 0$, then

$$\alpha(n) = O(\log^{(j)} n) \text{ for all integer } j > 0.$$

Theorem 2.1 *The following results concerning the function $\lambda(n, s)$ are known.*

1. $\lambda(n, 1) = n$ *and* $\lambda(n, 2) = 2n - 1$ [Dave65].

2. $\lambda(n, 3) = \Theta(n\,\alpha(n))$ [Hart86].

3. *For* $s \geq 3, \lambda(n, s) = \Omega(n\,\alpha(n))$ *(this follows from the previous result and the fact that λ is an increasing function of s), and*

4. *For* $s \geq 3$, $\lambda(n, s) = O(n\,[\alpha(n)]^{O(\alpha(n)^{s-3})})$ [Shar87].

■

For all problems considered in this paper that use the function $\lambda(n, s)$, the parameter s will be a bounded integer. Under such circumstances, the above implies that for "reasonable" values of n, $\lambda(n, s)$ is essentially $\Theta(n)$.

The next result gives a property that will be useful for bounding the number of processors in the algorithm associated with Theorem 3.2 for constructing the min function.

Lemma 2.2 [Boxe87a] *For all positive integers n and s, $2\lambda(n, s) \leq \lambda(2n, s)$.* ■

An interval is *nondegenerate* if and only if it contains more than one point. Two intervals have a *nondegenerate intersection* if and only if their intersection contains a nondegenerate interval. If p is a piece of a function f and q is a piece of a function g, we say p and q have nondegenerate intersection if and only if the interval of p and the interval of q have nondegenerate intersection. The next two results give useful bounds on the number of pieces in a "combined" function.

Lemma 2.3 [Boxe87a] *Let $f(t)$ and $g(t)$ be real-valued functions defined for all $t \geq 0$. Let m and n be positive integers. Suppose $f(t)$ has m pieces and $g(t)$ has n pieces. Then the pieces of $f(t)$ have, altogether, at most $m + n$ nondegenerate intersections with the pieces of $g(t)$.* ■

Lemma 2.4 [Boxe87a] *Let p and s be positive integers. Let $f(t)$ and $g(t)$ be real-valued functions defined for all $t \geq 0$. Suppose that for every piece of both $f(t)$ and $g(t)$, the function of the piece is a polynomial whose degree is at most s. Assume that the pieces of $f(t)$ have p nondegenerate intersections with the pieces of $g(t)$. Then the function $\min\{f(t), g(t)\}$ has no more than $p(s + 1)$ pieces.* ■

2.4 Data movement operations

Our algorithms are given in terms of machine independent fundamental data movement operations. We assume data values are distributed among the n PEs of a parallel machine so that no PE has more than $\Theta(1)$ elements. The operations are performed simultaneously within disjoint strings. (Notice that the entire machine corresponds to a single string.)

Operations not based on sorting include semigroup computation and broadcast. Each of these may be implemented on a mesh in $\Theta(n^{1/2})$ time and on a hypercube in $\Theta(\log n)$ time.

Sort-based operations include sorting, concurrent read, concurrent write, parallel prefix, grouping, and splitting the data evenly among the processors. Each of these may be implemented on a mesh in $\Theta(n^{1/2})$ time, on a hypercube in $\Theta(\log^2 n)$ time, and on a hypercube in expected $\Theta(\log n)$ time.

See [Mill88a] for descriptions of these operations and details concerning the implementations and proofs of the algorithms.

3 Constructing the MIN function

In this section, we show how a description of the minimum function may be constructed efficiently, under relatively mild restrictions, from descriptions of a set of real-valued functions.

The proof of Lemma 3.1 gives an algorithm to construct a description of the function $\min\{f(t), g(t)\}$. This algorithm can also be used to construct a description of the function that results from applying any of a variety of operations (*e.g.*, max, sum, product) to a pair of real-valued functions.

If I is a subset of the domain of the function $f(t)$, we denote by $f|_I$ the *restriction of f to I*. That is, $f|_I$ is the function whose domain is I such that $f|_I(t) = f(t)$ for all $t \in I$.

Lemma 3.1 *Let Φ be a family of real-valued functions defined on $[0,\infty)$. Let $f(t)$ and $g(t)$ be real-valued functions defined on $[0,\infty)$ by pieces generated by Φ. Let s be a positive integer. Suppose the function of every piece of $f(t)$ and of every piece of $g(t)$ has a $\Theta(1)$ storage description and can be evaluated for a given t in $\Theta(1)$ time by a single processor. Suppose that if I is the nondegenerate intersection of the intervals of a piece of $f(t)$ and a piece of $g(t)$, and F and G are members of Φ such that $f|_I = F$ identically and $g|_I = G$ identically, then there are at most s solutions to the equation $F(t) = G(t)$, and these solutions may be calculated by a single processor in $\Theta(1)$ time. Suppose m is a positive integer such that the total number of pieces of f and g is at most m. Suppose the pieces of f and the pieces of g are stored in disjoint strings of a mesh of size m or a hypercube of size m, at most one piece per PE. Then a description of the function $h(t) = \min\{f(t), g(t)\}$ can be constructed by the mesh in $O(m^{1/2})$ time; by the hypercube in $\Theta(\log^2 m)$ time; and by the hypercube in expected $\Theta(\log m)$ time.*

Proof: The general algorithm is given in 8 steps.

1. Since the pieces of f and the pieces of g are in disjoint strings, there is a PE_x such that, without loss of generality, pieces of f are stored in PEs whose labels are at most x and pieces of g are stored in PEs whose labels are greater than x. Broadcast x to all PEs.

2. In parallel, each PE containing a piece of f or a piece of g creates two sort-records, Left and Right, each containing the following information.

   - A tag whose value is "Left" in Left records, "Right" in Right-records.
   - A source field, whose value is the index of the PE.
   - A description of the piece.
   - An endpoint field, whose value is the left endpoint of the interval of the piece for the Left records, the right endpoint for the Right records.
   - An "other_piece" field, initially undefined.

3. Sort all of the Left and Right records together with respect to the endpoint field. Ties should be broken in favor of a Right record.

4. A *string of f* is a string whose first PE contains a Left record of f and whose last PE contains a Right record of f such that no intermediate PE contains a record of f. Records of f are recognized by having source $\leq x$. A *string of g* is defined analogously. Use a concurrent read so that each Left record of f (respectively, g) finds the PE-index of its corresponding Right record of f (respectively, g). In parallel, the first PE of each string of f broadcasts throughout its string a description of its piece of f, which is taken by each record of g in the string as its other_piece field. In parallel, the first PE of each string of g broadcasts throughout its string a description of its piece of g, which is taken by each record of f in the string as its other_piece field.

5. A concurrent read is performed based on the source field so that each PE gets back copies of the records it started with in Step 2, with all components now defined. Thus each PE containing a piece of f (respectively, g) now knows the left-most and right-most pieces of g (respectively, f) with which its piece has nondegenerate intersection.

6. We now construct the "subpieces" determined by nondegenerate intersections of a piece of f and a piece of g. All PEs act in parallel as follows.

   If PE_i contains a piece p_i of f, then PE_i handles the leftmost and rightmost nondegenerate intersections of p_i with pieces of g by performing the following three steps, once for the record with tag "Left" and once for the record with tag "Right."

   (a) Compute the intersection, I, of the intervals of p_i and the tag-record's other_piece.

   (b) Determine the (at most s) solutions to the equation $f|_I(t) = g|_I(t)$.

   (c) The roots found in (b) determine at most $s+1$ closed nondegenerate subintervals of I with disjoint interiors. For each such subinterval J, determine which of $f|_J$ and $g|_J$ is minimal by comparing $f(t_J)$ and $g(t_J)$, where t_J is any interior point of J.

   Let p be a piece of f and let q be a piece of g such that p and q have nondegenerate intersection and q is neither the leftmost nor the rightmost piece of g whose intersection with p is nondegenerate. Then the interval of q is contained in the interval of p. Hence, in the PE in which q is stored, the Left-record and Right-record have identical other_piece fields. Such processors PE_j perform the following two steps.

   (a) Determine the (at most s) solutions to the equation $f|_J(t) = g|_J(t)$, where J is the interval of the piece of g.

   (b) The roots found in (a) determine at most $s+1$ closed nondegenerate subintervals of J with disjoint interiors. For each such subinterval K, determine which of $f|_K$ and $g|_K$ is minimal by comparing $f(t_K)$ and $g(t_K)$, where t_K is any interior point of K.

   Suppose f has u pieces generated by Φ and g has v pieces generated by Φ. By Lemma 2.3, the intervals of the pieces of f and the intervals of the pieces of g have at most $u+v$ nondegenerate intersections. By Lemma 2.4 there are at most $(s+1)(u+v)$ subpieces determined in this step. Since the pieces of f and the pieces of g were stored one per PE, there are $\Theta(1)$ subpieces per PE.

7. Sort the subpieces (their intervals have disjoint interiors) from left to right so that the subpieces end up in a string such that each PE of the string has at least one and at most $s+1$ subpieces.

8. At this point, there may be adjacent subpieces with the same function $F(t)$. Such pairs should be joined into a single piece. *I.e.*, if there are subpieces of the form $(F(t), [a,b])$ and $(F(t), [b,c])$ or $(F(t), [b,\infty))$, they are joined as $(F(t), [a,c])$ (respectively, $(F(t), [a,\infty))$). Adjacent subpieces that have the same function may be joined by creating strings of such functions, broadcasting the first and last interval to all PEs in the string, letting the first PE in the string create a description of the combined subpiece, and using a parallel prefix to pack the final set of intervals.

For the mesh: Step 1 requires $O(m^{1/2})$ time. Steps 2 and 6 require $\Theta(1)$ time. Sorting (Steps 3 and 7) requires $O(m^{1/2})$ time. Concurrent reads (Steps 4 and 5) require $O(m^{1/2})$ time. The broadcasting in Step 4 requires $O(m^{1/2})$ time. Step 8 can be implemented by grouping and parallel prefix operations, both of which require $O(m^{1/2})$ time. Therefore, the running time of the algorithm is $O(m^{1/2})$.

For the hypercube: Step 1 requires $\Theta(\log m)$ time. Steps 2 and 6 require $\Theta(1)$ time. Sorting (Steps 3 and 7) and concurrent reads (Steps 4 and 5) require $\Theta(\log^2 m)$ time, expected $\Theta(\log m)$ time. The broadcasting in Step 4 takes $O(\log m)$ time. Step 8 can be implemented by grouping and parallel prefix operations, both of which require $\Theta(\log^2 m)$ time, expected $\Theta(\log m)$ time. Therefore, the running time of the algorithm is $\Theta(\log^2 m)$, expected $\Theta(\log m)$. ■

It should be noted that for some of our algorithms, running times for the mesh are given in O-notation, while all running times for the hypercube are in Θ-notation. This is because $\min\{f_0, \ldots, f_{n-1}\}$ may have less than $\lambda(n, k)$ pieces, in which case it may be possible to use a submesh and obtain asymptotically faster running times. The same is not true of the hypercube. Roughly, this is because $\lambda^{1/2}(n, k) \neq \Theta(n^{1/2})$, while $\log \lambda(n, k) = \Theta(\log n)$.

Constructing the minimum function for a pair of functions, as described above, is part of a recursive algorithm for describing the function $h(t)$ of Equation (1). An efficient description of $h(t)$ is obtained by means of the algorithm associated with Theorem 3.2.

Since the number of PEs in a mesh must be a power of 4, define

$$\lambda_m(n, s) = 4^{\lceil \log_4 \lambda(n,s) \rceil}.$$

Since the number of nodes in a hypercube must be a power of 2, define

$$\lambda_h(n, s) = 2^{\lceil \log_2 \lambda(n,s) \rceil}.$$

Note $\lambda_m(n, s) \geq \lambda(n, s)$, $\lambda_h(n, s) \geq \lambda(n, s)$, $\lambda_m(n, s) = \Theta(\lambda(n, s))$, and $\lambda_h(n, s) = \Theta(\lambda(n, s))$.

Theorem 3.2 *Let $f_0, \ldots, f_{n-1}$ be continuous real-valued functions defined on $[0, \infty)$, no distinct pair of which intersects more than s times. Assume a) each f_i has a $\Theta(1)$ storage description, b) each $f_i(t)$ may be calculated in $\Theta(1)$ time for a given t by a single processor, and c) for every distinct pair f_i and f_j, the (at most s) real solutions to $f_i(t) = f_j(t)$ can be computed in $\Theta(1)$ time by a single processor. Suppose descriptions of $f_0, \ldots, f_{n-1}$ are stored one per PE in a mesh with $\lambda_m(n, s)$ PEs or in a hypercube with $\lambda_h(n, s)$ PEs. Then the minimum function $h(t)$ can be constructed by the mesh in $O(\lambda^{1/2}(n, s))$ time; by the hypercube in $\Theta(\log^3 n)$ time; and by the hypercube in expected $\Theta(\log^2 n)$ time. At the end of the algorithm, the description of $h(t)$ is given with the pieces ordered by their intervals, one piece per PE.*

Proof: A general algorithm is given in 3 steps.

1. Split the descriptions of $\{f_0, f_1, \ldots, f_{n-1}\}$ evenly among the processors.

2. Recursively, and in parallel, have the string with $f_0, \ldots, f_{\lceil \frac{n-1}{2} \rceil}$ construct the ordered pieces $p_1, \ldots, p_u$ for

$$h_1(t) = \min\{f_0(t), \ldots, f_{\lceil \frac{n-1}{2} \rceil}(t)\}$$

generated by $\{f_0, \ldots, f_{\lceil \frac{n-1}{2} \rceil}\}$, while the string with the functions $f_{\lceil \frac{n-1}{2} \rceil + 1}, \ldots, f_{n-1}$ constructs the ordered pieces $q_1, \ldots, q_v$ representing

$$h_2(t) = \min\{f_{\lceil \frac{n-1}{2} \rceil + 1}(t), \ldots, f_{n-1}(t)\}$$

generated by $\{f_{\lceil \frac{n-1}{2} \rceil + 1}(t), \ldots, f_{n-1}(t)\}$. Since $u, v \leq \lambda(n/2, s)$, then from Lemma 2.2, each of the PEs is responsible for at most one piece of a minimum function.

At the end of this step, descriptions of the pieces $\{p_1, \ldots, p_u\}$ and $\{q_1, \ldots, q_v\}$ are ordered by their intervals in disjoint strings, each consisting of half of the PEs.

3. Describe $h(t) = \min\{h_1(t), h_2(t)\}$ by the algorithm of Lemma 3.1.

Let $T(n)$ be the running time of the algorithm. For the mesh we have the following analysis. Step 1 requires $O(\lambda^{1/2}(n, s))$ time. Step 2 is a recursive call. Since we have $u + v \leq 2\lambda(n/2, s) \leq \lambda(n, s)$, the latter inequality by Lemma 2.2, it follows from Lemma 3.1 that Step 3 requires $O(\lambda^{1/2}(n, s))$ time. Therefore, the running time of the algorithm satisfies the recurrence $T(n) = T(n/2) + O(\lambda^{1/2}(n, s))$. It follows from Lemma 2.2 that $T(n) = O(\lambda^{1/2}(n, s))$.

For the hypercube we have the following analysis. Step 1 requires $\Theta(\log^2 n)$ time, expected $\Theta(\log n)$ time. Step 2 is a recursive call. Step 3 requires $\Theta(\log^2 n)$ time, expected $\Theta(\log n)$ time, by Lemma 3.1. Therefore, the running time of the algorithm satisfies the recurrence $T(n) = T(n/2) + \Theta(\log^2 n)$, which is $\Theta(\log^3 n)$. The expected running time satisfies $T(n) = T(n/2) + \Theta(\log n)$, which is $\Theta(\log^2 n)$. ■

The function $f(t)$ has a *jump discontinuity at* u if both $\lim_{t \to u^+} f(t)$ and $\lim_{t \to u^-} f(t)$ exist, and $\lim_{t \to u^+} f(t) \neq \lim_{t \to u^-} f(t)$. The real-valued function $f(t)$ whose domain is a subset of $[0, \infty)$ has a *transition at* t_0 [Atal85] if and only if

- $t_0 > 0$, and
- there exists $\delta > 0$ such that either
  1. for all t such that $0 < t < \delta, f(t_0 - t)$ is defined and $f(t_0 + t)$ is undefined, or
  2. for all t such that $0 < t < \delta, f(t_0 - t)$ is undefined and $f(t_0 + t)$ is defined.

Lemma 3.3 [Boxe87a] *Let k be a positive integer. Let $f_0, \ldots, f_{n-1}$ be real-valued functions of time such that (a) every f_i is continuous except for at most p_i jump discontinuities, (b) every f_i has at most q_i transitions, where (c) $p_i + q_i \leq k$, and (d) no pair of distinct functions f_i and f_j intersect more than s times. Then $h(t) = \min\{f_0(t), \ldots, f_{n-1}(t)\}$ has no more than $\lambda(n, s+2k)$ pieces generated by $\{f_0, \ldots, f_{n-1}\}$.* ■

Theorem 3.4 *Let k be a positive integer and let $f_0, \ldots, f_{n-1}$ be as in Lemma 3.3. Assume also that the f_i satisfy a), b), and c) of Theorem 3.2. A description of the function $h(t) = \min\{f_0(t), \ldots, f_{n-1}(t)\}$ can be given in $O(\lambda^{1/2}(n, s+2k))$ time by a mesh of size $\lambda_m(n, s+2k)$ so that the pieces are ordered, one per PE. A description of the function $h(t)$ can be given in $\Theta(\log^3 n)$ time, and in expected $\Theta(\log^2 n)$ time, by a hypercube of size $\lambda_h(n, s + 2k)$ so that the pieces are ordered, one per PE.*

Proof: The assertion may be proved by an argument that is virtually identical to that given for Theorem 3.2. ■

4 Transient Behavior Computations

We apply the results of the previous section to dynamic systems of point-objects, showing how to determine geometric properties of the system.

4.1 Closest Points, Farthest Points, and Collision

Let S be a sequence of points closest, say, to P_0, listed in chronological order. That is, the first member of S is a closest point to P_0 at time $t = 0$, and the last member of S is a closest point to P_0 as t approaches infinity. Let S' be a sequence of "farthest" points from P_0, listed in chronological order.

Theorem 4.1 *For a system of n points in d-dimensional space with k-motion, each of S and S' can be constructed on a mesh of size $\lambda_m(n-1, 2k)$ in $O(\lambda^{1/2}(n-1, 2k))$ time; on a hypercube of size $\lambda_h(n-1, 2k)$ in $\Theta(\log^3 n)$ time; and on a hypercube of size $\lambda_h(n-1, 2k)$ in expected $\Theta(\log^2 n)$ time.*

Proof: Let $d_{0j}(t)$ be the Euclidean distance between points P_0 and P_j at time t. Then each $d_{0j}^2(t)$ is a polynomial of degree $\leq 2k$.

We give an algorithm in 3 steps for constructing S. A similar algorithm may be used to construct S'.

1. Broadcast a description of function f_0 so that, without loss of generality, PE_j has descriptions of the distinct pairs (f_0, f_j), $0 < j < n$.

2. In parallel, each processor PE_j constructs the function $d_{0j}^2(t)$ from f_0 and f_j.

3. Construct the min function $h(t)$ of the family of functions $d_{0j}^2(t)$ by the algorithm of Theorem 3.2. For each piece of $h(t)$, a pair of points that yielded the piece corresponds to an element of S.

For the mesh we have the following analysis. Step 1 is accomplish in $O(\lambda^{1/2}(n-1, 2k))$ time. Step 2 is accomplished in $\Theta(1)$ time. Step 3 requires $O(\lambda^{1/2}(n-1, 2k))$ time by Theorem 3.2. Thus, the running times for the mesh are as claimed.

For the hypercube we have the following analysis. Step 1 is accomplish in $\Theta(\log n)$ time. Step 2 is accomplished in $\Theta(1)$ time. Step 3 requires $\Theta(\log^3 n)$ time, and expected $\Theta(\log^2 n)$ time, by Theorem 3.2. Thus, the running times for the hypercube are as claimed. ■

Sometimes it is more important to determine whether or not two points collide rather than which pair is closest. Define points P_i and P_j to *collide* at time t if and only if $f_i(t) = f_j(t)$.

Theorem 4.2 *Assume that a system of n points in d-dimensional space with k-motion is given. Then a chronological list of times at which P_0 collides with any other point of the system can be created in $\Theta(n^{1/2})$ time on a mesh of size $4^{\lceil \log_4 n \rceil}$; in $\Theta(\log^2 n)$ time on a hypercube of size $2^{\lceil \log_2 n \rceil}$; and in expected $\Theta(\log n)$ time on such a hypercube.*

Proof: We observe that P_0 and P_j $(j > 0)$ collide if and only if $d_{0j}^2(t) = 0$ has a solution for $t > 0$. Without loss of generality, PE_i stores a description of f_i, $0 \leq i < n$.

The algorithm is given in 4 steps.

1. Broadcast to each PE a description of f_0.

2. In parallel, $PE_i, 0 < i < n$, determines $d_{0i}^2(t)$.

3. In parallel, $PE_i, 0 < i < n$, solves $d_{0i}^2(t) = 0$ for its at most $2k$ roots (since $d_{0i}^2(t)$ is a polynomial of degree at most $2k$) that are greater than 0.

4. Sort the roots to obtain the desired list.

Each solution in Step 3 represents a collision of P_0 with another point-object of the system.

For the mesh, we have the following analysis. Steps 1 and 4 each require $\Theta(n^{1/2})$ time. Steps 2 and 3 each require $\Theta(1)$ time. Thus, our algorithm has $\Theta(n^{1/2})$ running time.

For the hypercube, we have the following analysis. Step 1 requires $\Theta(\log n)$ time. Steps 2 and 3 each require $\Theta(1)$ time. Step 4 requires $\Theta(\log^2 n)$ time, expected $\Theta(\log n)$ time. Thus, the algorithm has $\Theta(\log^2 n)$ running time, and expected $\Theta(\log n)$ running time. ■

4.2 Convex Hull

The *convex hull* of a set of points $S = \{P_0, \ldots, P_{n-1}\}$, denoted *hull*$(S)$, is the smallest convex set containing S. A point $P_i \in S$ is an *extreme point* or *vertex* of hull(S) if $P_i \notin$ hull$(S - \{P_i\})$. In this section, we develop a general parallel algorithm to generate a description of the intervals of time over which a given point $P_0 \in S$ is an extreme point of hull(S). We also give efficient implementations of the algorithm for the mesh and hypercube.

Assume k-motion in the plane. Let $T_{ij}(t)$ be the angle made by rotating the positively oriented horizontal ray with endpoint P_i about P_i until the ray contains the line segment from P_i to P_j at time t. By convention, $-\pi < T_{ij}(t) \leq \pi$. Formally, if $x_i(t), x_j(t), y_i(t)$, and $y_j(t)$ are the x and y coordinates of the points P_i and P_j, respectively, at time t, then

$$T_{ij}(t) = \begin{cases} \pi/2 & \text{if } x_i(t) = x_j(t) \text{ and } y_i(t) < y_j(t) \\ -\pi/2 & \text{if } x_i(t) = x_j(t) \text{ and } y_i(t) > y_j(t) \\ \arctan\left(\frac{y_j(t)-y_i(t)}{x_j(t)-x_i(t)}\right) & \text{if } x_i(t) < x_j(t) \\ \arctan\left(\frac{y_j(t)-y_i(t)}{x_j(t)-x_i(t)}\right) + \pi & \text{if } x_i(t) > x_j(t) \text{ and } y_i(t) < y_j(t) \\ \arctan\left(\frac{y_j(t)-y_i(t)}{x_j(t)-x_i(t)}\right) - \pi & \text{if } x_i(t) > x_j(t) \text{ and } y_i(t) > y_j(t) \\ \text{undefined} & \text{if } x_i(t) = x_j(t) \text{ and } y_i(t) = y_j(t). \end{cases}$$

$$\text{Define } G_{ij}(t) = \begin{cases} T_{ij}(t) & \text{if } T_{ij}(t) \geq 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\text{Define } B_{ij}(t) = \begin{cases} T_{ij}(t) & \text{if } T_{ij}(t) < 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Define the functions a_i, b_i, c_i, and d_i as follows.

$$a_i(t) = \min\{G_{ij}(t) | 0 < j < n, i \neq j, G_{ij}(t) \text{ is defined}\}.$$

$$b_i(t) = \max\{G_{ij}(t) | 0 < j < n, i \neq j, G_{ij}(t) \text{ is defined}\}.$$

$$c_i(t) = \min\{B_{ij}(t) | 0 < j < n, i \neq j, B_{ij}(t) \text{ is defined}\}.$$

$$d_i(t) = \max\{B_{ij}(t) | 0 < j < n, i \neq j, B_{ij}(t) \text{ is defined}\}.$$

If at time t, $G_{ij}(t)$ is undefined (respectively, $B_{ij}(t)$ is undefined) for all j, then $a_i(t)$ and $b_i(t)$ (respectively, $c_i(t)$ and $d_i(t)$) are undefined.

$$\text{Define } T = \{T_{0j} | 0 < j < n\}.$$

Lemma 4.3 [Atal85], [Boxe87b] *For a system of n points with k-motion, each of the functions a_0, b_0, c_0, and d_0 has at most $\lambda(n, 4k)$ pieces generated by T.* ■

Lemma 4.4 [Atal85] *Given a set S of n points moving in the plane, a point P_0 is an extreme point of hull(S) at time t if and only if*

1. *$a_0(t) - d_0(t) \geq \pi$, or*
2. *$b_0(t) - c_0(t) \leq \pi$, or*
3. *$a_0(t)$ and $b_0(t)$ are undefined, or*
4. *$c_0(t)$ and $d_0(t)$ are undefined.* ■

Theorem 4.5 *Let $S = \{P_0, \ldots, P_{n-1}\}$ be a set of points in the plane with k-motion. Then the ordered intervals of time during which a given point P_0 is an extreme point of hull(S) can be determined in $O(\lambda^{1/2}(n, 4k))$ time on a mesh of size $\lambda_m(n, 4k)$; in $\Theta(\log^3 n)$ time on a hypercube of size $\lambda_h(n, 4k)$; and in expected $\Theta(\log^2 n)$ time on a hypercube of size $\lambda_h(n, 4k)$.*

Proof: For each j, $0 \leq j < n$, let $x_j(t)$ be the x-coordinate of P_j at time t and let $y_j(t)$ be the y-coordinate of P_j at time t. Observe that solving $T_{0j}(t) = T_{0m}(t)$ means finding instants at which the directed line segment from P_0 to P_j and the directed line segment from P_0 to P_m have equal slopes and are similarly oriented. Finding instants when the line segments have equal slopes can be determined by solving

$$[x_m(t) - x_0(t)][y_j(t) - y_0(t)] = [x_j(t) - x_0(t)][y_m(t) - y_0(t)],$$

a polynomial equation of degree at most $2k$, which we assume can be solved in $\Theta(1)$ time by a single PE. Further, determining whether or not two directed line segments with equal slopes are similarly oriented can be accomplished in $\Theta(1)$ serial time. It follows that $T_{0j}(t) = T_{0m}(t)$ can be solved by a single processor in $\Theta(1)$ time.

Define

$$A_0(t) = \begin{cases} 1 & \text{if } a_0(t) - d_0(t) \geq \pi \\ 0 & \text{otherwise,} \end{cases}$$

$$B_0(t) = \begin{cases} 1 & \text{if } b_0(t) - c_0(t) \leq \pi \\ 0 & \text{otherwise,} \end{cases}$$

$$C_0(t) = \begin{cases} 1 & \text{if both } a_0(t) \text{ and } b_0(t) \text{ are undefined} \\ 0 & \text{otherwise,} \end{cases}$$

and

$$D_0(t) = \begin{cases} 1 & \text{if both } c_0(t) \text{ and } d_0(t) \text{ are undefined} \\ 0 & \text{otherwise.} \end{cases}$$

Our general algorithm is given below.

1. It is shown in the proof of Lemma 4.3 [Boxe87b] that each G_{0j} (similarly, each B_{0j}) has at most k values of t that yield jump discontinuities or transitions. Construct the functions $a_0(t), b_0(t), c_0(t)$, and $d_0(t)$.

2. From Lemma 4.3 and Lemma 2.3, each of $a_0(t) - d_0(t)$ and $b_0(t) - c_0(t)$ has no more than $2\lambda(n, 4k) = O(\lambda(n, 4k))$ pieces generated by differences of members of T. Construct the ordered pieces of the functions $a_0(t) - d_0(t)$ and $b_0(t) - c_0(t)$. Similarly, construct the $O(\lambda(n, 4k))$ ordered maximal intervals on which $a_0(t)$ and $b_0(t)$ are both undefined (respectively, on which $c_0(t)$ and $d_0(t)$ are both undefined).

3. If I_1 and I_2 are intervals of pieces of a_0 and d_0, respectively, where $I = I_1 \cap I_2$ is nondegenerate, then $(a_0 - d_0)|_I(t) = \pi$ implies there are integers j and m determined by I_1 and I_2, respectively, such that $a_0|_I = T_{0j}, d_0|_I = T_{0m}$, and $T_{0j}(t) - T_{0m}(t) = \pi$. There are at most $2k$ such instants, and they may be determined in $\Theta(1)$ time. It follows from Lemma 2.4 that every piece of $a_0(t) - d_0(t)$ generated by differences of members of T yields at most $2k + 1$ pieces of $A_0(t)$ generated by the set of constant functions $\{0, 1\}$. Therefore, $A_0(t)$ has at most $(2k+1)\, 2\, \lambda(n, 4k) = O(\lambda(n, 4k))$ pieces generated by $\{0, 1\}$. Similarly, $B_0(t)$ has $O(\lambda(n, 4k))$ pieces generated by $\{0, 1\}$. Construct descriptions of the functions $A_0(t)$ and $B_0(t)$ by using the algorithm of Lemma 3.1. Similarly, construct the $O(\lambda(n, 4k))$ ordered pieces generated by $\{0, 1\}$ of each of the functions $C_0(t)$ and $D_0(t)$.

4. It follows from Lemma 2.4 that there are $O(\lambda(n, 4k))$ pieces generated by $\{0, 1\}$ of

$$H_0(t) = \max\{A_0(t), B_0(t), C_0(t), D_0(t)\}.$$

Describe $H_0(t)$ via a fixed number of applications of the algorithm of Lemma 3.1. Note that Lemma 4.4 implies P_0 is an extreme point at time t if and only if $H_0(t) = 1$.

5. Pack the intervals for which $H_0(t) = 1$ into a string by sorting in order to obtain the desired sequence of intervals.

For the mesh: Step 1 requires $O(\lambda^{1/2}(n, 4k))$ time, by Theorem 3.4. Steps 2, 3, and 4 each require $O(\lambda^{1/2}(n, 4k))$ time, by Lemma 3.1. Step 5 requires $O(\lambda^{1/2}(n, 4k))$ time. Hence the running time of the algorithm is $O(\lambda^{1/2}(n, 4k))$.

For the hypercube: Step 1 needs $\Theta(\log^3 n)$ time, expected $\Theta(\log^2 n)$ time, by Theorem 3.4. Steps 2, 3, and 4 each needs $\Theta(\log^2 n)$ time, by Lemma 3.1. Step 5 needs $\Theta(\log^2 n)$ time. Hence the running time of the algorithm is $\Theta(\log^3 n)$, expected $\Theta(\log^2 n)$. ■

4.3 Containment Problems

In this section, we address a variety of problems concerning shapes and sizes of containers into which a dynamic system of points will fit. We assume k-motion in d-dimensional space, for fixed k and d.

Let J be the ordered list of intervals of time during which the points $P_0, \ldots, P_{n-1}$ can be enclosed within a rectilinear, iso-oriented hyperrectangle (a d-dimensional analog of a box with sides parallel or perpendicular to each of the coordinate axes) of given fixed dimensions.

Theorem 4.6 *For a set of n points with k-motion in d-dimensional space, the ordered sequence J can be constructed on a mesh of size $\lambda_m(n, k)$ in $O(\lambda^{1/2}(n, k))$ time; on a hypercube of size $\lambda_h(n, k)$ in $\Theta(\log^3 n)$ time; on a hypercube of size $\lambda_h(n, k)$ in expected $\Theta(\log^2 n)$ time.*

Proof: For $1 \leq i \leq d$, let $p_i : \mathbb{R}^d \rightarrow \mathbb{R}$ be the i^{th} coordinate function. That is, for a point $X = (x_1, \ldots, x_d)$, $p_i(X) = x_i$. Note that for each i and j, a description of $p_i(f_j(t))$ is stored in the PE containing a description of f_j. Define

$$m_i(t) = \min\{p_i(f_0(t)), \ldots, p_i(f_{n-1}(t))\},$$

and

$$M_i(t) = \max\{p_i(f_0(t)), \ldots, p_i(f_{n-1}(t))\}.$$

Our algorithm is given below.

1. Describe the functions $m_1(t), \ldots, m_d(t)$, $M_1(t), \ldots, M_d(t)$ using the algorithm of Theorem 3.2, such that each of m_i and M_i, $1 \leq i \leq d$, has at most $\lambda(n,k)$ pieces generated by $F_i = \{p_i(f_0(t)), \ldots, p_i(f_{n-1}(t))\}$.
2. Construct descriptions of all the functions $D_i(t) = M_i(t) - m_i(t)$, $1 \leq i \leq d$, ($D_i(t)$ is the maximum separation in the i^{th} coordinate among the points $\{P_0, \ldots, P_{n-1}\}$ at time t), by the algorithm of Lemma 3.1. It follows from Lemma 2.3 that each $D_i(t)$, $1 \leq i \leq d$, has at most $2\lambda(n,k)$ pieces generated by differences of pairs of members of F_i.
3. Let X_i be the length of the hyperrectangle in the i^{th} coordinate, $1 \leq i \leq d$. Then for each i, $1 \leq i \leq d$, the function
$$W_i(t) = \begin{cases} 1 & \text{if } D_i(t) \leq X_i \\ 0 & \text{otherwise} \end{cases}$$
has at most $2(k+1)\lambda(n,k)$ pieces generated by the set of constant functions $\{0,1\}$, by Lemma 2.4. Describe $W_1(t), \ldots, W_d(t)$ by the algorithm of Lemma 3.1.
4. Let $C(t) = \min\{W_1(t), \ldots, W_d(t)\}$. Notice that $C(t) = 1$ if and only if $\{P_0, \ldots, P_{n-1}\}$ will fit inside a hyperrectangle of the given fixed dimensions at time t. Describe $C(t)$ from descriptions of the set of functions $\{W_1(t), \ldots, W_d(t)\}$. This is accomplished by performing $\Theta(\log d) = \Theta(1)$ stages of the algorithm of Lemma 3.1, where at each stage, pairs of functions are combined.
5. The (ordered) intervals of the pieces of $C(t)$ for which $C(t) = 1$ form the desired sequence J. Pack these intervals into a string via a sorting operation.

For the mesh: Step 1 requires $O(\lambda^{1/2}(n,k))$ time by Theorem 3.2. Steps 2, 3, and 4 require $O(\lambda^{1/2}(n,k))$ time, by Lemma 3.1. Step 5 requires $O(\lambda^{1/2}(n,k))$ time. Thus the algorithm requires $O(\lambda^{1/2}(n,k))$ time.

For the hypercube: Step 1 uses $\Theta(\log^3 n)$ time, expected $\Theta(\log^2 n)$ time. Steps 2, 3, and 4 use $\Theta(\log^2 n)$ time, by Lemma 3.1. Step 5 uses $\Theta(\log^2 n)$ time. Thus the algorithm requires $\Theta(\log^3 n)$ time, expected $\Theta(\log^2 n)$ time. ■

Define

$$\Psi = \{p_i(f_j(t)) | 1 \leq i \leq d, 0 \leq j < n\}.$$

Theorem 4.7 *Assume a system of points $S = \{P_0, \ldots, P_{n-1}\}$ has k-motion in d-dimensional space. The function $D(t)$, whose value at time t is the edgelength of the smallest iso-oriented rectilinear hypercube that will contain S, has $O(\lambda(n,k))$ pieces generated by differences of members of Ψ. A description of $D(t)$ can be constructed in an ordered fashion on a mesh of size $\lambda_m(n,k)$ in $O(\lambda^{1/2}(n,k))$ time; on a hypercube of size $\lambda_h(n,k)$ in $\Theta(\log^3 n)$ time; and on a hypercube of size $\lambda_h(n,k)$ in expected $\Theta(\log^2 n)$ time.*

Proof: We give a general algorithm of 2 steps.

1. Let $D_1(t), \ldots, D_d(t)$ be as in Theorem 4.6. Construct descriptions of $D_1(t), \ldots, D_d(t)$ via the algorithm of Theorem 4.6. It was shown in the proof of Theorem 4.6 that each of these functions has at most $2\lambda(n,k)$ pieces generated by differences of members of Ψ.
2. Since $D(t) = \max\{D_1(t), \ldots, D_d(t)\}$, observe $D(t)$ can be described from $D_1(t), \ldots, D_d(t)$ by performing $\Theta(\log d) = \Theta(1)$ stages of the algorithm of Lemma 3.1, where at each stage, ordered pieces of pairs of functions are combined. If each function being combined has no more than $c\lambda(n,k)$ pieces, c a constant, then the maximum of the two functions has no more than $2c(k+1)\lambda(n,k)$ pieces, by Lemma 2.3 and Lemma 2.4.

Since each of the $\Theta(1)$ combine steps increases the number of pieces by no more than a constant factor, the number of pieces of $D(t)$ is $O(\lambda(n,k))$.

Our claims concerning the running times follow from Theorem 4.6 and Lemma 3.1. ■

We now show how to determine a smallest iso-oriented rectilinear hypercube that can ever contain the set of points $S = \{P_0, \ldots, P_{n-1}\}$.

Theorem 4.8 *Let S be a system of n points with k-motion in Euclidean d-dimensional space. Let $D_{\min} = \min\{D(t) | t \geq 0\}$, where $D(t)$ is as in Theorem 4.7. Then $D_{\min}$ and a time $t_{\min}$ at which $D(t_{\min}) = D_{\min}$ can be computed in $O(\lambda^{1/2}(n,k))$ time on a mesh of size $\lambda_m(n,k)$; in $\Theta(\log^3 n)$ time on a hypercube of size $\lambda_h(n,k)$; and in expected $\Theta(\log^2 n)$ time on a hypercube of size $\lambda_h(n,k)$.*

Proof: We give a general algorithm of 4 steps.

1. Construct a description of $D(t)$ by Theorem 4.7. The function $D(t)$ has $O(\lambda(n,k))$ pieces generated by differences of members of Ψ, so that responsibility for the pieces is divided evenly among the PEs in that each PE is responsible for $\Theta(1)$ pieces of $D(t)$.
2. In parallel, each PE does the following. For each of the pieces of $D(t)$ for which the PE is responsible, compute the minimum value of $D(t)$ on the interval of the piece.
3. In parallel, each PE determines the minimum of its pieces' minima, and a time when the minimum occurs.
4. Compute the minimum of all the PEs' minima, $D_{\min}$, with a minimizing time being recorded.

For the mesh: Step 1 requires $O(\lambda^{1/2}(n,k))$ time, by Theorem 4.7. Step 2 may be done, using well-known principles of calculus, in $\Theta(1)$ time. Step 3 requires $\Theta(1)$ time. Step 4 requires $O(\lambda^{1/2}(n,k))$ time. Hence the algorithm requires $O(\lambda^{1/2}(n,k))$ time.

For the hypercube: Step 1 uses $\Theta(\log^3 n)$ time, expected $\Theta(\log^2 n)$ time, by Theorem 4.7. As with the mesh, Step 2 requires $\Theta(1)$ time. Step 3 requires $\Theta(1)$ time. Step 4 requires $\Theta(\log n)$ time. Hence the algorithm requires $\Theta(\log^3 n)$ and expected $\Theta(\log^2 n)$ time. ■

5 Steady-State Computations

We use the term "steady-state" to refer to conditions as t (time) approaches infinity. In this section, we give parallel algorithms for determining steady-state properties of dynamic systems, mostly in the plane. Due to space limitations, we omit the proofs.

The algorithms in this section are adapted from [Boxe87a, Mill86, Mill88a, Mill88b, Mill88c, Sanz87, Sham75].

Theorem 5.1 *Let k and d be fixed integers such that $k \geq 0$ and $d > 0$. Given a set of points $\{P_0, \ldots, P_{n-1}\}$ with k-motion in d-dimensional space, a steady-state nearest (farthest) neighbor to a given point P_0 can be determined on a mesh of size $4^{\lceil \log_4 n \rceil}$ in $\Theta(n^{1/2})$ time and on a hypercube of size $2^{\lceil \log_2 n \rceil}$ in $\Theta(\log n)$ time.* ■

Theorem 5.2 *For a system of n points with k-motion in the plane, a steady-state closest pair can be identified by a mesh of size $4^{\lceil \log_4 n \rceil}$ in $\Theta(n^{1/2})$ time; by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in $\Theta(\log^2 n)$ time; and by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in expected $\Theta(\log n)$ time.* ■

Theorem 5.3 *For a system S of n points with k-motion in the plane, the steady-state hull(S) can be constructed by a mesh of size $4^{\lceil \log_4 n \rceil}$ in $\Theta(n^{1/2})$ time; by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in $\Theta(\log^2 n)$ time; and by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in expected $\Theta(\log n)$ time.* ■

Theorem 5.4 *Let an integer $k \geq 0$ be fixed. Let $S = \{P_0, \ldots, P_{n-1}\}$ be a set of point-objects moving in the plane with k-motion such that when steady-state is reached, S is the set of distinct extreme points of a convex polygon C. The diameter function of C may be determined in $\Theta(n^{1/2})$ time by a mesh of size $4^{\lceil \log_4 n \rceil}$; in $\Theta(\log^2 n)$ time by a hypercube of size $2^{\lceil \log_2 n \rceil}$; and in expected $\Theta(\log n)$ time by a hypercube of size $2^{\lceil \log_2 n \rceil}$.* ■

Corollary 5.5 *For a set S of n points with k-motion in the plane, a steady-state farthest pair can be determined by a mesh of size $4^{\lceil \log_4 n \rceil}$ in $\Theta(n^{1/2})$ time, by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in $\Theta(\log^2 n)$ time, and by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in expected $\Theta(\log n)$ time.* ■

Theorem 5.6 *Let $S = \{P_0, P_1, \ldots, P_{n-1}\}$ be a system of point-objects in k-motion, where k is a fixed nonnegative integer, such that, in steady-state, S is the set of distinct extreme points of a convex polygon C. A rectangle of minimum area enclosing all the points of C may be determined by a mesh of size $4^{\lceil \log_4 n \rceil}$ in $\Theta(n^{1/2})$ time; by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in $\Theta(\log^2 n)$ time; and by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in expected $\Theta(\log n)$ time.* ■

Corollary 5.7 *Let k be a fixed nonnegative integer. Let S be a system of n point-objects in k-motion. Then a description of a steady-state minimal-area rectangle enclosing all the points of S can be given by a mesh of size $4^{\lceil \log_4 n \rceil}$ in $\Theta(n^{1/2})$ time; by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in $\Theta(\log^2 n)$ time; and by a hypercube of size $2^{\lceil \log_2 n \rceil}$ in expected $\Theta(\log n)$ time.* ■

References

[Atal85] M.J. Atallah, Some dynamic computational geometry problems, *Comp. and Maths. with Appls.* 11 (1985), 1171-1181.

[Batc68] K.E. Batcher, Sorting networks and their applications, *Proc. AFIPS Spring Joint Computer Conference*, vol. 32 (1968), AFIPS Press, New Jersey, 307-314.

[Boxe87a] L. Boxer and R. Miller, Parallel algorithms for dynamic systems with known trajectories, *Proc. IEEE 1987 Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, 37-43.

[Boxe87b] L. Boxer and R. Miller, Parallel dynamic computational geometry, *Tech. Rept.* 87-11, Dept. of Comp. Sci., SUNY - Buffalo, 1987.

[Dave65] H. Davenport and A. Schinzel, A combinatorial problem connected with differential equations, *Amer. J. Math.* 87 (1965), 684-694.

[Hart86] S. Hart and M. Sharir, Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes, *Combinatorica* 6 (1986), 151-177.

[Mill86] R. Miller and Q.F. Stout, Mesh computer algorithms for computational geometry, *Tech. Rept.* 86-18, Dept. of Comp. Sci., SUNY - Buffalo, 1986 (accepted for publication in *IEEE T. on Computers*).

[Mill88a] R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Architectures*, MIT Press, Cambridge, Mass., 1988, to appear.

[Mill88b] R. Miller and Q.F. Stout, Convexity algorithms for parallel machines, *Proceedings 1988 IEEE Conference on Computer Vision and Pattern Recognition*, to appear.

[Mill88c] R. Miller and Q.F. Stout, Computational geometry on hypercube computers, *Proc. of The Third Conference on Hypercube Concurrent Computers and Applications*, 1988, to appear.

[Reif87] J.H. Reif and L.G. Valiant, A logarithmic time sort for linear size networks, *JACM* 34 (1987), 60-76.

[Rein77] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms*, Prentice Hall, New York, 1977.

[Sanz87] J.L.C. Sanz and R.E. Cypher, Data reduction and fast routing: a strategy for efficient algorithms for parallel message-passing computers, *Tech. Rept.*, Computer Science Dept., IBM Almaden Research Center, 1987.

[Sham75] M.I. Shamos, Geometric complexity, *Proc. 7th Annual ACM Symposium on Theory of Computing* (1975), 224-233.

[Shar87] M. Sharir, Almost linear upper bounds on the length of general Davenport-Schinzel sequences, *Combinatorica* 7 (1987), 131-143.

[Yagl61] I.M. Yaglom and V.G. Boltyanskii, *Convex Figures*, Holt, Rinehart and Winston, New York, 1961.

I/O EMBEDDING IN HYPERCUBES*

A.L.Narasimha Reddy, P.Banerjee

Computer Systems Group
Coordinated Science Lab
University of Illinois
1101, W.Springfield Ave.
Urbana, IL 61801

Santosh G. Abraham

Dept. of Elec. Engg. & Comp. Sci.
University of Michigan
1301 Beal Ave.
Ann Arbor, MI 48109

ABSTRACT

Many multiprocessor systems based on the hypercube or binary n-cube topology have been built recently. In such systems, I/O processors are used to handle the data transfer between the processors and the outside world or the host. In this paper, we propose a method of embedding the I/O processors in such a system. The proposed method is shown to require far fewer links than the earlier methods. It is also shown that the new method achieves a higher I/O adjacency, and as a result higher degree of tolerance of I/O failures. Necessary and sufficient conditions are derived to obtain such an embedding. A generalization of the problem to k-regular networks is presented and necessary conditions are derived for I/O embedding in such a network. It is shown that embedding in a general k-regular network is NP-complete. An algorithm is presented for finding a minimal embedding in a k-regular network.

1. INTRODUCTION

A binary n-cube consists of 2^n processors interconnected in an n-dimensional binary cube topology. Each processor in such a system has its own memory and the processors communicate by message passing. A processor in an n-cube can be represented by an n-bit string, $p_1 p_2 \cdots p_n$. Each processor is adjacent to a processor along the n dimensions. Specifically, a processor $(p_1 p_2 \cdots p_n)$ is adjacent to $(\overline{p}_1 p_2 \cdots p_n)$, $(p_1 \overline{p}_2 \cdots p_n)$, ...,$(p_1 p_2 \cdots \overline{p}_n)$. Binary cubes are known to have several useful properties, namely a high degree of connectivity, fault tolerance, low diameter, etc. Several problems such as sorting, FFT are known to map well on to the hypercubes. Different interconnections such as a linear array, mesh are known to map easily onto the hypercube interconnection. Several hypercube systems are built recently [1,2,3,4]. One issue that has not been addressed effectively in the past is the support of efficient I/O operations in multiprocessors. The importance of balancing I/O bandwidth and computational power has been pointed out by Kung [5].

I/O processors are used for transferring data between the hypercube nodes and the outside world and the host. This is to be distinguished from the I/O hardware that is required for communication between the processors. I/O communication is required to distribute data and code to the processors before the computation and to receive the results after the computation has been completed. Each processor in the system is connected to an I/O processor and the I/O processor handles all the data transfer between that processor and the outside world. In this paper, we propose a method to connect the I/O processors and processors together to build a system with a higher degree of I/O adjacency and a higher degree of fault tolerance. The Intel iPSC system uses I/O hardware within each processor for I/O communication using the ethernet protocol[2]. In the NCUBE system, an I/O processor is connected to a subcube of 8 processors and the I/O processors are themselves interconnected partially [1]. Our method uses the system links efficiently for both the I/O and processor-to-processor communication. The proposed method requires no explicit processor-to-I/O processor links. It is also shown that our method achieves a higher processor-to-I/O processor adjacency and as a result higher tolerance of I/O failures. In a related recent work, the hypernet architecture has been proposed [6] for maintaining a constant node degree. The hypernet architecture provides a set of nodes explicitly meant for performing I/O operations in a concurrent manner. However, that work on I/O embedding is applicable only to the hypernet topology and the I/O embedding itself has not been evaluated. Our method can be directly used in the hypercubes with minor modifications to the architecture.

Embedding general graphs onto the hypercube topology is known to be an NP-complete problem [7]. Embedding arbitrary meshes onto hypercubes is reported in [8]. Several numerical computations have been mapped onto the hypercube topology [9, 10].

The new method of connecting I/O processors and processors is presented in Section 2. Necessary and sufficient conditions are derived for embedding I/O processors in such a manner. Section 3 looks at the advantages and disadvantages of the proposed method. In Section 4, we consider a generalization of the proposed method to a k-regular interconnection network and derive some necessary conditions.

* This research was supported in part by the National Science Foundation Presidential Young Investigator Award under Grant NSF MIP 86-57563 PYI, and in part by the Semiconductor Research Corporation under Contract SRC 86-12-109.

2. I/O EMBEDDING

DEFINITION 1: I/O embedding is the problem of mapping I/O processors onto a multiprocessor system such that each processor in the system is adjacent to at least one I/O processor.

Consider embedding I/O processors in a k-cube multiprocessor. The I/O processors are embedded in the system along with the processors. The system will then consist of two types of nodes 1) processor nodes, P-nodes and 2) I/O processor and processor nodes, I-nodes. The processor and I/O processor share the links to the I-node through a switch as shown in Fig.1. The switch can connect any of the system links to the processor or I/O processor and the processor to the I/O processor. The implications of sharing the links by the processor and the I/O processor are discussed in Section 3. But we briefly note that in a model of computation where each node receives the data it has to operate on and after carrying out the computation, sends the results back to the host, I/O communication and interprocessor communication do not overlap. Hence, sharing links does not result in congestion. Since each I-node also contains a processor, the cube topology of the multiprocessor is not disturbed and hence the proposed architecture will retain all the desirable properties of a binary cube such as ease of problem mapping. Each node in a k-cube has k neighbors. Hence each I-node is adjacent to k nodes. So an I/O processor in an I-node can serve as an I/O processor to its k neighbors and the processor in its node. A processor sends data along the system links with an appropriate tag to indicate whether the data is meant for the processor or the I/O processor at that node. At an I-node, the data is appropriately switched to the processor or the I/O processor depending on the tag. By embedding enough I-nodes through the hypercube, we can make sure that each processor is adjacent to at least one I/O processor. The advantage of embedding the I/O processors in such a way is that we do not require explicit processor-to-I/O processor links. As a result, we can construct a larger size hypercube given the same number of links for processor communication. For example, in the Intel iPSC, there are eight channels at each processing node, seven of which are used to connect 128 processors, the eighth to connect every node to the host. Using our scheme, we can connect a 256 processor cube using the same number of channels. Each node in the cube has a register associated with it which indicates along which dimension it is adjacent to an I/O processor. Whenever the processor has to transfer I/O data, the processor sends the data along this dimension with an appropriate tag to indicate whether it is I/O data or a message to the processor at that node. The hypernet architecture [6] also uses two types of nodes, but the I/O nodes in their architecture are only used for I/O communication and the architecture is concerned with maintaining a constant node degree.

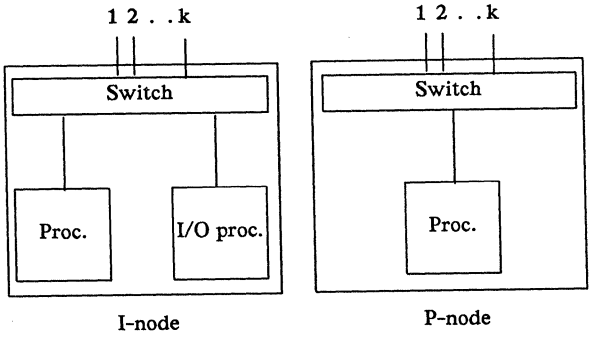


Fig.1. Architecture of the nodes.

DEFINITION 2: A network is said to have a *perfect embedding* if it is possible to embed the I-nodes in the network such that each processor in the system is adjacent to exactly one I/O processor.

In the remainder of the section we will characterize when a k-cube can have a perfect embedding.

LEMMA 1: For a k-cube to have perfect embedding, $k = 2^l - 1$, for some integer l.

PROOF: Each I-node in the system is adjacent to k processors along the k dimensions of the cube. Moreover, the I/O processor in an I-node also serves the processor in its node. Hence, each I/O processor serves $(k+1)$ processors in the cube. Then, if a perfect embedding exists, $(k+1)$ should divide the number of processors $n = 2^k$ in the system. Since the number of processors in a hypercube system is always a power of 2, the only factors of n are some smaller powers of 2. This implies that $(k+1) = 2^l$, for some l. □

From Lemma 1, for a perfect embedding to exist $k =$ 3, 7, 15,... A perfect embedding for a 3-cube is shown in Fig.2. A perfect embedding for a 7-cube, with processors numbered from 0 to 127, can be obtained by locating the I-nodes at 0, 7, 25, 30, 42, 45, 51, 52, 75, 76, 82, 85, 97, 102, 120, 127. The next question one would like to ask is, can we always find a perfect embedding in a k-cube with $k = 2^l-1$. In other words, is this condition sufficient? The answer to this question is yes and this leads us to the following lemma:

LEMMA 2: A perfect embedding exists in a k-cube when $k = 2^l-1$.

PROOF: Consider two I-nodes a and b in a perfect embedding. Then nodes a and b cannot be adjacent. If they are adjacent, then the processors in nodes a and b are each adjacent to two I/O processors. By a similar argument, the neighbors of a cannot be neighbors of b. This implies that any two I-nodes in a perfect embedding have to be at a Hamming distance of 3 or greater.

The result follows from the theory of single error correcting Hamming codes. Each code word is at a Hamming distance of 3 or greater from each other. Hamming codes are known to be perfect codes [11], i.e., they attain the upper bound of distance-3 codes that can be found in a given space. With k=2^l−1 bits, the number of distance-3 code words is bounded by $\frac{2^k}{(k+1)}$ and Hamming codes achieve this bound. In such a code space, each non-code word is adjacent to exactly one code word. Now the hypercube can be seen as a k-dimensional space and we can place I-nodes in the code word locations. Each non-code word can be a P-node and we have the required perfect embedding in the k-cube when k = 2^l−1. This completes the proof. □

THEOREM 1: In a k-cube, $k = 2^l-1$ is a necessary and sufficient condition to find a perfect embedding.

PROOF: From Lemmas 1 and 2. □

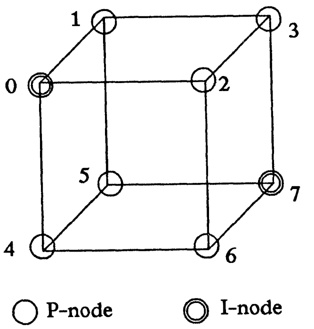


Fig.2. A perfect embedding in a 3-cube.

The constructive proof of Lemma 2 gives us a way of finding a perfect embedding in a k-cube. One could also use a method of sieves [12] to find the locations of the I-nodes. In such a method, we choose a node in the hypercube space and eliminate (sieve out) all its neighbors and distance-2 neighbors from the list. Then pick another node from the list and continue this process till the list is empty. This method gives a similar embedding that is obtained by the Hamming codes approach when a perfect embedding exists. Moreover, this method gives an I/O embedding for any k-cube, even when $k \neq 2^l-1$, the obtained embedding is perfect only if $k = 2^l-1$.

By shifting each I-node location in a fixed direction, we can obtain another perfect embedding of a given cube. For example, 0 and 7 is a perfect embedding in a 3-cube. If we shift the I-node along 1-dimension to location 1, by suitably shifting 7 to location 6, we have another embedding with 1 and 6. We can similarly shift the perfect embedding 1 and 6 to another perfect embedding such as 3 and 4. Hence, we can find a perfect embedding that contains a given node of a hypercube.

DEFINITION 3: If we specify that a particular node i has to be in the I/O embedding, we call such an embedding i*—specified embedding.*

If we get a perfect embedding only for a few sizes of the cube, how do we embed the I/O processors in cubes of other sizes? For example, consider a 4-cube. We can view the 4-cube as a union of two 3-cubes. Use perfect embeddings in each subcube to obtain an embedding in the 4-cube. Such an embedding for a 4-cube is shown in Fig.3. We notice from the figure that some of the nodes (2,5,8,15) in the 4-cube are adjacent to two I/O processors. Though we still have the same ratio of I/O processors (1 for every 4 processors), we obtained a higher I/O adjacency for some of the processors. This is due to the fourth dimension links in the 4-cube. Similarly, if we embed processors in a 5-cube, more processors will have an I/O adjacency of two. The higher I/O adjacency can be useful in several ways: in increasing the tolerance of the I/O failures and decreasing the possibility of congestion at an I-node etc. We can carry out this procedure for any size cube to obtain an I/O embedding. This leads us to the question: can we embed the I/O processors systematically to get an I/O adjacency of two for each processor? The answer to this question is yes and we give an algorithm to obtain an I/O adjacency of two.

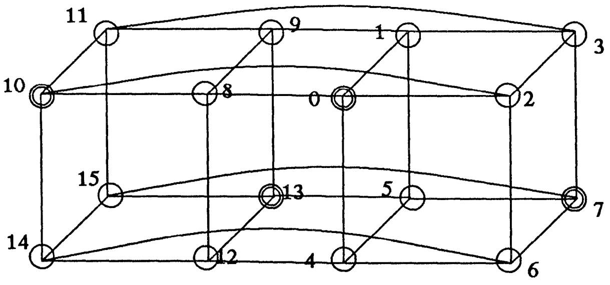


Fig. 3. An embedding in a 4-cube.

Consider I/O embedding in a $2k$+1-cube, where k = 2^l−1. A $2k$+1-cube consists of 2^{k+1} k-cubes. Let these subcubes be represented by $S_0, S_1, ..., S_{2^{k+1}-1}$. When k = 2^l−1, we can obtain a perfect embedding of a k-cube. Let the nodes in each subcube be numbered 0,1,2...2^k−1, with node i in subcube S_j being the node numbered $i+j*2^k$ in the $2k$+1-cube. Each subcube of size k can be embedded perfectly by Lemmas 1 and 2. To obtain an embedding in the $2k$+1-cube with an I/O adjacency of two, get an i-specified embedding of subcubes S_i and S_{i+2^k}, for i = 0,1,...2^k−1. Within each subcube, each node is adjacent to exactly one I-node. Now consider nodes within S_0 with nodes numbered from 0,1,...2^k. Node 1 is adjacent to an I-node in the subcube S_1, since it is 1-specified. Similarly each P-node in the subcube S_0 is adjacent to one I-node within the subcube and one I-node in some other subcube. Since the cube is symmetric, each subcube has a similar property and hence each P-node in the $2k$+1-cube is adjacent to two I-nodes. Since we have similar embedding in cubes S_i and S_{i+2^k}, each I-node is adjacent to another I-node. This completes the proof that the given algorithm generates an embedding with an I/O adjacency of two. For each node to have an I/O adjacency of two the size of the cube also has to be of the form 2^l-1. For example, in a 7-cube, each subcube of size 3 has I-nodes along a diagonal and this diagonal is oriented in different directions in different 3-cubes to get a symmetrical embedding with an I/O adjacency of two. A 0-specified embedding of S_0 gives (0,7), a 1-specified embedding of S_1 gives (9,14) and so on. By continuing a detailed enumeration, we obtain the following I/O embedding for the 7-cube with an I/O adjacency of two: (0,7), (9,14), (18,21), (27,28), (36, 35), (45,42), (54,49), (63,56) and (64,71), (73,78), (82,85), (91,92), (100,99), (109,106), (118,113), (127,120). We can carry this idea further to get embeddings with higher I/O adjacency.

Another simpler construction exists which gives an I/O adjacency of two. Consider a k-cube with $k = 2^l-1$. Find a perfect embedding of the k-cube. From Lemmas 1 and 2, such an embedding exists and we can find it by the method of sieves or the Hamming code method. Each node in the cube is adjacent to one of the I-nodes now. Obtain a new embedding by shifting one of the I-nodes. Each processor is again adjacent to one I-node in the second embedding as well. Hence, by overlapping two embeddings we can obtain an embedding where each node in the cube is adjacent to two I-nodes in the cube.

It is to be noted that the two methods described above obtain different embeddings, both with an I/O adjacency of two. The difference in the two methods of construction is that the first method starts building from a subcube, whereas the second method treats the cube as a whole. The first method is well suited for expanding an existing system. It is also to be noted that with one I/O processor for every four processors, a 3-cube has an I/O adjacency of one and a 7-cube has an I/O adjacency of two. If one interconnects I/O processors and processors in the way as it is done in NCUBE, the adjacency remains the same even in higher dimensional cubes.

3. PRACTICAL CONSIDERATIONS

By placing the I/O processor along with the processor in a node, we do not require explicit I/O processor-to-processor links. Thus saved links can be utilized in building a larger size cube. For a given number of links per processor, the proposed method enables connecting a larger number of nodes together. As we observed, because of utilizing the system links, as we go to higher dimensions we can achieve a higher I/O adjacency. Higher adjacency implies higher tolerance of I/O failures. Each I/O processor now needs links only for interconnecting with other I/O processors. Since the I/O processors are connected to the processors by the system links, we would need fewer links per I/O processor compared to other designs. The links on an I/O processor are only needed for inter-I/O processor communication. In other words, with the same number of links per I/O processor, we can achieve a higher connectivity between the I/O processors. Since each processor is adjacent to at least one I/O processor, an I/O operation would involve a single message transfer. By appropriately interconnecting the I/O processors, we can reduce the inter I/O processor traffic. This design can be seen as an integrated system design consisting of the processors and the I/O processors. This is to be contrasted with designing the multiprocessor system separately and then connecting some I/O processors to it.

Utilizing the system links for I/O transfer requires some consideration. We might create congestion along the links when I/O and interprocessor communication have to take place along the same link at the same time. There are two reasons to believe that sharing the links for I/O and interprocessor communication does not lead to congestion or a bottleneck. Most problems are solved on a multiprocessor system in the following manner: 1) Distribute the data and code to each processor (2) carry out the computation in a cooperative manner and (3) combine the results together. Steps 1 and 3 are I/O communication and Step 2 requires computation and interprocessor communication. Within such a model of solving a problem, we can see that the I/O communication and interprocessor communication do not overlap in time. And this leads us to conclude that the system links can be efficiently shared for both I/O communication and interprocessor communication. Besides, the I/O bandwidth required for most problems is about an order of magnitude smaller than that of interprocessor communication. Hence, even when I/O communication may overlap with interprocessor communication, this may not lead to a severe strain on the resources. Another solution can be put forward for this problem. Give interprocessor communication priority over I/O communication such that the computation may not be delayed even in the presence of I/O communication. When the computation is completed, all the processors need to send the data to an I/O processor. Since each processor uses a distinct link for I/O communication, the links will not be congested. However, the I/O processor may not be able to receive all the computed values from all its neighbors at the same time. This bottleneck is unavoidable and exists in other designs as well [1,2].

A perfect embedding of a 3-cube gives one I/O processor for every four processors in the system and similarly a perfect embedding of a 7-cube gives one I/O processor for every eight processors. Higher ratios can be easily obtained by superimposing two or more perfect embeddings as explained in Section 2. The ratio of I/O processors to processors in other size cubes depends on the subcube size (3 or 7) chosen for embedding such a cube. Smaller ratios than 1 out of 4 cannot be obtained for cubes of size smaller than 2^7.

Since each subcube of size 3 or 7, depending on the system size, looks like any other subcube, the design can be made uniform by designing one board for a subcube. The boards can be connected together to form higher size cubes. The required design effort may be more than that of the other designs because of the non-uniformity among the nodes. The switch hardware needed in an I-node is an overhead for this organization.

A comparison of different I/O embeddings in a k-cube is made in Table 1. In the NCUBE design, there exists one I/O processor for every eight nodes. The number of links in a k-cube = $k2^{k-1}$, which is also the number of links in the proposed method. In NCUBE and iPSC designs, there is an extra link per processor for I/O and hence the extra 2^k links. The iPSC system uses a bus to connect all the I/O processors and hence its I/O bandwidth is a constant, whereas the NCUBE design and the proposed method enable connecting multiple I/O devices to each I/O processor and hence the bandwidth is equal to the number of I/O processors. It is seen from the table that the proposed method achieves same or higher bandwidth with fewer links. This is achieved by efficient utilization of the links.

Table 1. Comparison of Different I/O embeddings

Method	Node Degree	# of links	# of I/O proc.	I/O BW
iPSC	$k+1$	$k2^{k-1}+2^k$	1	1
NCUBE	$k+1$	$k2^{k-1}+2^k$	2^{k-3}	2^{k-3}
Proposed	k	$k2^{k-1}$	2^{k-2}	2^{k-2}

4. POSSIBLE PERFORMANCE IMPROVEMENTS

In this Section, we will consider how the proposed I/O embedding may affect the performance of the system. The performance improvements depend whether the problem is I/O bound or computation bound. Clearly, the performance improvements will be more pronounced in an I/O bound problem. We consider an I/O bound problem, matrix-vector multiplication to show the possible performance improvements by using concurrent I/O.

Consider multiplying a matrix A_{nxn} by a vector B_n to generate C_n. Algorithm mapping depends on the number of processors in the system, size of available memory at each node and the size of the problem. Assume that the size of the matrix and the system are such that, p rows of A and the vector B are mapped to each node. Assume that the maximum message size is such that we can send at most k rows as a single message. Then the total number of messages sent by the host is given by $l = \lfloor p/k \rfloor * m$, where m is the number of processors in the system. The number p is a function of various parameters, available memory size at each node, number of processors in the system, size of the problem etc. and has to be chosen carefully to minimize the execution time for the problem. If memory at the nodes is not a problem, then we can make $p = n/m$. Then, the data distribution time is given by, $O(l)$. The computation time is given by, $O(n^2/m)$. Once the computation is complete, in a single host system, the data needs to be collected at one site and this again incurs an $O(l)$ cost. If $m = n$, then $l = n$. If the relative cost of a message transmission is a compared to a unit of computation (an addition and a multiplication here), then the total cost of the algorithm is given by, n + a (2n). When we use an I/O embedding as described in the paper, the second term in the above cost can be reduced significantly. Since, we have $m/4 = n/4$ I/O devices, the effective cost of the algorithm will be n + a (2 * 4). This is based on the assumption that the data was initially distributed among the I/O devices such that all the I/O devices can distribute data in parallel to their neighboring processors. The speedup obtained by concurrent I/O is then given by $\frac{n+a(2n)}{n+a(8)}$. With $n = 128$, $a = 8$, we get a speedup of 17 * 128 /192 = 11.33. This speedup factor is an optimistic measure of improvement because of concurrent I/O. However, this measure does reflect the importance of using concurrent I/O.

It is to be noted that, in the above calculation, we assumed that the data is initially stored on the I-nodes in the desired way. This raises the important question of data organization in such a system, with possibly a disk at each I-node. The pattern of data access during the algorithm dictates the way the data is to be stored on the disks. The algorithm, in turn, depends on the way the data is organized in the system. Hence, the algorithm design and data distribution problems need to be considered together. Sometimes, the data needs to be organized in different ways during different phases of a computation. It is possible to build a system where the I/O nodes can carry out data reorganization in parallel while the processors carry out the computation.

An embedding in a 4-cube that enables parallel I/O operations is shown in Fig.8. We add an extra link between the two I-nodes in a 3-cube. Thus the two I/O processors in a 3-cube are directly connected and can carry out any data transfer between the two I-nodes, while the nodes can communicate in parallel over the system links. In the Section 2, we aimed at obtaining an embedding that utilized minimum number of I/O nodes such that each processor is adjacent to at least one I/O processor. As a result, the I/O processors are required to be at a hamming distance of three or greater from one another. With an extra link between the two I-nodes in a

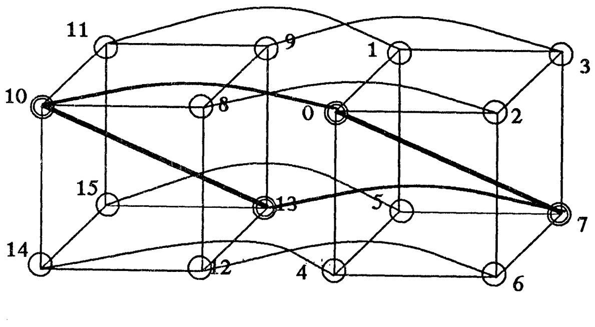


Fig.8. An embedding in a 4-cube

3-cube as shown in Fig.8, the I-nodes can communicate in an efficient manner. The I-nodes in a 3-cube are seen to be connected in a 1-cube fashion. When two such 3-cubes are connected together to form a 4-cube, the I-nodes are connected in a 2-cube. In general, if an n-cube is built out of the basic 3-cube, shown in Fig.8, the I-nodes are connected as a (n-2)-cube. The advantage of such a configuration is that any algorithms developed for efficient data movement over the cube can be used for data reorganization among the I-nodes since they are now connected in the form of a cube themselves. The data reorganization issues are discussed in greater detail elsewhere [13]. If the basic 3-cube is initially balanced with respect to I/O and computation power, any resulting larger configuration is seen to have the same desirable property.

5. GENERALIZATION TO REGULAR INTERCONNECTION NETWORKS

In this section, we consider generalizing the I/O embedding idea to a general interconnection network. An interconnection network is normally regular and we can represent such a network by a k-regular graph. A k-regular graph is a graph in which each node has a degree k. Let the number of nodes in such a graph be n. It is to be noted that $n \neq 2^k$ in a general k-regular graph and no such simple relation exists between n and k. What are the necessary conditions for a given k-regular graph to have a perfect embedding?

LEMMA 3: For a given k-regular graph on n nodes to have a perfect embedding, $(k+1)$ should divide n and $\frac{nk(k-1)}{(k+1)}$ has to be even.

PROOF: Each I-node in the graph serves as an I/O processor for (k+1) nodes. If the graph has a perfect embedding, each node is adjacent to exactly one I-node and this implies that $(k+1)$ should divide n. Now consider a perfect embedding in a given graph. Since each P-node is adjacent to exactly one I-node, and the graph is k-regular, the partition containing only P-nodes has to be (k-1) regular. There are $n - \frac{n}{(k+1)} = \frac{kn}{(k+1)}$ nodes in this partition. Now counting the degrees on each node, we require $\frac{nk(k-1)}{(k+1)}$ to be even which is equal to twice the number of edges within this partition. □

For example, consider the cube-connected cycles [14] in 3-dimensions. The network has 3*8 = 24 nodes and each node has degree of 3 i.e., $n = 24$ and $k = 3$. The numbers n and k satisfy the given necessary conditions and a perfect embedding for this network is shown in Fig.4. Are these necessary conditions sufficient? No. We give a simple counter example in Fig.5. where $n = 12$ and $k = 3$. Though this graph satisfies the necessary conditions, no perfect embedding exists.

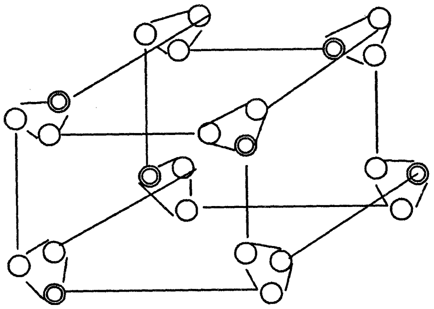


Fig. 4. A perfect embedding in a cube-connected cycles network.

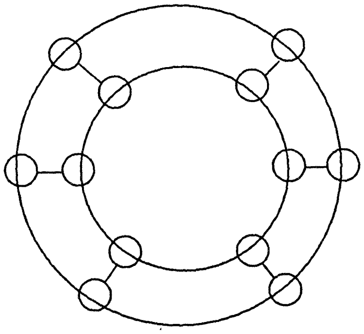


Fig.5. A Counterexample with necessary conditions.

LEMMA 4: The problem of finding a perfect embedding in a k-regular graph is NP-complete.

PROOF: Given a k-regular graph on n nodes, the problem of finding a perfect embedding is same as that of finding a dominating set in that graph. A dominating set $V' \subseteq V$ is a set of nodes such that every node $v \in V - V'$ is adjacent to at least one node belonging to V'. A perfect embedding requires that every node in the graph be adjacent to exactly one I-node and hence the set of I-nodes forms a dominating set. The perfect embedding problem now can be seen as the problem of finding a dominating set V' with $|V'| = \frac{n}{(k+1)}$. A perfect embedding requires that V' be both a dominating set and an independent set and finding such a set is known to be NP-complete [15]. Hence, the problem of finding a perfect embedding in a k-regular graph is NP-complete. □

However, some of the commonly used networks such as ring, star, binary tree, and completely connected networks have simple characterizations for a perfect embedding. Networks that do not have simple characterizations for a perfect embedding include meshes and toroids.

DEFINITION 4: A minimal I/O embedding is defined as an I/O embedding from which we cannot discard an I-node and still obtain an I/O embedding of the graph.

It is noted that a minimal embedding may not be minimum or perfect. A minimal embedding requires that the set of vertices V be partitioned into two sets T, an independent set and the set of its neighbors $N(T)$ such that $T+N(T) = V$. An independent set is a set of nodes with no edges between them.

LEMMA 5: A graph can always be partitioned into two sets T, an independent set and $N(T)$, the set of its neighbors such that $T+N(T) = V$.

PROOF: If we can partition the graph in such a manner, then we can make T the set of I-nodes and $N(T)$ the set of P-nodes to obtain a minimal embedding of the network. Proof by contradiction: assume that it is not possible to partition a graph in such a manner. Consider a maximal independent set T. If $T+N(T) \neq V$, then there must exist a third set of vertices S such that $T+N(T)+S = V$. The vertices in S are not adjacent to any vertex in T, otherwise they would be in $N(T)$. Then we can choose a vertex u from S and augment T with it to obtain a larger independent set contradicting that T is a maximal independent set. Hence, by contradiction, we can always find such a partition that gives a minimal embedding of a given network. □

The proof of Lemma 3 gives us the following algorithm to find a minimal embedding in a given graph. Choose a node from the set of nodes V. Put this in the set T and call the set of its neighbors $N(T)$ as shown in Fig.6. Let $S = V-T-N(T)$. If possible, choose a node from S that is not in N(N(T)). If not, choose any node in S. Put this in T and continue as above till the set S is empty. This algorithm gives a minimal embedding, not a minimum or perfect embedding. For example, consider a 2-regular bipartite graph on 12 vertices as shown in Fig.7.

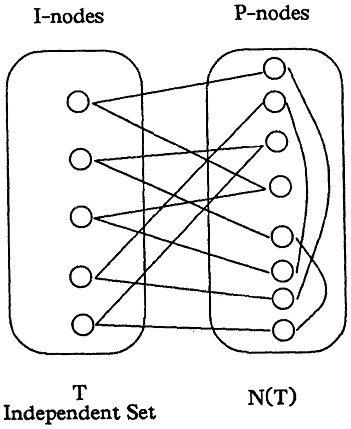


Fig.6. A minimal embedding of a graph.

Start with node 0 in T, then we have $N(T) = \{6,7\}$ and $S = \{1,2,3,4,5,8.9.10,11\}$. Choose a node, say 2, from S that is not in $N(N(T)$ (since it is possible). Now $T = \{0,2\}$, $N(T) = \{6,7,8,9\}$ and $S = \{1,3,4,5,10,11\}$. Continuing like this, we may get an embedding $T = \{0,2,10,5,1\}$, which is minimal but not minimum or perfect. A perfect embedding exists in this graph and it is given by $\{0,8,3,11\}$.

By carrying out all the choices, one could find all the minimal embeddings and choose the minimum embedding from them. The given graph may have a number of minimal embeddings and hence considerable effort may be needed to find the minimum embedding this way. A perfect embedding in a k-regular graph requires that there exist an independent set T such that $T+N(T) = V$ and $|N(T)| = k|T|$, where $|N(T)|$ and $|T|$ are the cardinalities of the two sets N(T) and T respectively. One could use this condition to check if a minimal embedding is a perfect embedding.

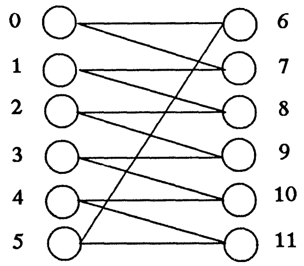


Fig.7. An example network for finding a minimal embedding.

6. CONCLUSIONS

In this paper, we presented a new method of connecting I/O processors and processors in a hypercube multiprocessor system. Necessary and sufficient conditions are derived to obtain a perfect embedding. It is shown that, the proposed method achieves a higher degree of I/O adjacency and higher degree of fault tolerance with the same number of I/O processors. The practical implications of this method are discussed. The problem is generalized to a k-regular interconnection network and it is shown that the conditions derived for a hypercube are necessary but not sufficient. It is shown that finding a perfect embedding in a k-regular interconnection network is NP-complete. We also presented an algorithm to find a minimal embedding in a general interconnection network. It would be interesting to see how concurrent I/O may improve the performance of the numerous algorithms developed for the hypercube.

References

[1] J.P.Hayes et al., "Architecture of a Hypercube Supercomputer," *Proc. of ICPP*, pp. 653-660, 1986.

[2] "Intel IPSC System Product Summary," Intel, Oregon.

[3] J.Tuazon, J.Peterson, M.Pniel, and D.Liberman, "Caltech/JPL Mark II Hypercube Concurrent Processor," *Proc. of ICPP*, pp. 666-673, 1985.

[4] J.C.Peterson et al., "The Mark III Hypercube-Ensemble Concurrent Computer," *Proc. of ICPP*, pp. 71-73, 1985.

[5] H.T.Kung, "Memory Requirements for Balanced Computer Architectures," *Proc. of 13th Annu. Int. Symposium on Computer Architecture*, pp. 49-54, 1986.

[6] Kai Hwang and J.Ghosh, "Hypernet: A Communication-Efficient Architecture for Constructing Massively Parallel Computers," *IEEE Trans. on Computers*, vol. C-36, pp. 1450-1466, Dec. 1987.

[7] D.W.Krumme, K.N.Venkataraman, and G.Cybenko, "Hypercube Embedding is NP-Complete," *Proc. of 1st. Conf. on Hypercube Multiprocessors*, pp. 148-157, Aug. 1985.

[8] C.T.Ho and S.L.Johnsson, "On the Embedding of Arbitrary Meshes in Boolean Cubes with Expansion Two Dilation Two," *Proc. of ICPP*, pp. 188-191, 1987.

[9] T.F.Chan, Y.Saad, and M.H.Schultz, "Solving Elliptic Partial Differential Equations on Hypercubes," *Proc. of 1st Conf. on Hypercube Multiprocessors*, pp. 196-210, Aug. 1985.

[10] G.C.Fox and S.W.Otto, "Algorithms for Concurrent Processors," *Physics Today*, pp. 13-20, May 1984.

[11] J. Wakerly, in *Error Detecting Codes, Self-Checking Circuits and Applications*. New York, New York: Elsevier North Holland Inc., 1978.

[12] E.M.Reingold, J.Nievergelt, and N.Deo, *Combinatorial Algorithms: Thoery and Practice*. New Jersey: Prentice-Hall Inc., 1977.

[13] A.L.Narasimha Reddy and P.Banerjee, "I/O Embedding in Hypercubes," in *CSG Technical Report, in preparaion*, Univ. of Illinois, Urbana.

[14] F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network For Parallel Computation," *Commun. Ass. Comput. Mach.*, vol. 24, pp. 300-309, May 1981.

[15] M.R.Garey and D.S.Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H.Freeman & co., 1979.

A Unified Approach to Designing Fault-Tolerant Processor Ensembles
(Extended Abstract)

S. Chakravarty [1]
Dept. of Computer Science
State University of New York
Buffalo, NY 14260

S. J. Upadhyaya [2]
Dept. of Electrical and Computer Engineering
State University of New York
Buffalo, NY 14260

Abstract – **Processor ensembles** (abbrev. PEN) form part of parallel processing systems. We present a unified approach to designing fault-tolerant PENs. Our approach is illustrated by presenting fault-tolerant schemes for several commonly used interconnection topologies. Our fault-tolerance scheme is shown to be "area-efficient", unlike another fault-tolerance scheme viz. the Diogenes approach[7]. Unlike the reliability analysis of fault-tolerant PENs that have appeared in the literature our reliability analysis takes into account switch failures along with processor and link failures.

1. Introduction

Processor ensembles (**PEN**) form part of parallel processing systems. The parallel processing system could be a parallel machine or a special purpose VLSI chip. We assume that parallel processing systems using PENs consist of a control unit (CU) and a PEN with N processing elements (**PE**). The N PEs in the PEN communicate with each other through a set of communication links. The interconnection topology of a PEN is represented by a graph where the nodes of the graph represent the PEs and the edges represent the communication links between the PEs. There exists an edge between nodes I and J if and only if there exists a communication link between the corresponding pairs of processors.

The PEs could be very complex in which case each PE is integrated on a seperate chip and are interconnected to form a PEN. In this case, processor or link failures lead to low system availability. On the other hand, the PEs could be very simple in which case the PEN can be integrated on a single chip. In this case processor or link failures lead to low yield. This motivates the need to design PENs that can tolerate link and/or processor failures. In this paper we address the problem of designing fault-tolerant PENs. It is assumed that the CU can diagnose the faulty processors and it is capable of reconfiguring the PEN by setting appropriate control signals.

We present a unified approach to designing fault-tolerant PENs. Our approach is illustrated by presenting fault-tolerant schemes for a number of commonly occurring interconnection topologies like the Binary Tree, X-Tree, Mesh, Hypercube, Pyramid etc. These PENs can be recursively defined and can be constructed by interconnecting a number of copies of a **basic module**(abrrev. **BM**). There could be a number of different BMs for each PEN and a number of copies of any one of them could be suitably interconnected to construct the PEN.

The proposed **fault-tolerance scheme** is based on the above principle. We first determine a BM from which the given PEN can be constructed. Based on the BM so determined we design a **fault-tolerant basic module** (abbrev. **FTBM**). The **fault-tolerant PEN** is then constructed by suitably interconnecting the FTBMs.

The problem of designing fault-tolerant Binary Trees has been extensively studied[3,6,7,9,10]. We show that the fault-tolerance scheme for Binary Trees resulting from our approach is the same as the fault-tolerance scheme for Binary Trees proposed in [3].

An attractive feature of the scheme proposed here is that it is **area-efficient**. Let M be a PEN having N PEs; and Q be the corresponding fault-tolerant PEN derived from M using a specific fault-tolerance scheme. The fault-tolerance scheme is said to be area-efficient if there exists a layout of size $O(p(n))$ for Q given that there exists a layout of size $O(p(n))$ for M.

Not all fault-tolerance schemes are area efficient as illustrated by the fault-tolerance scheme for Binary Trees proposed in [6]. It is known that there exists a layout of size $O(N)$ for a Binary Tree with N nodes[4]. But, the fault-tolerant N node Binary Tree resulting from the scheme proposed in [6] requires area equal to $O(NLog(N))$[3]. The Diogenes scheme[7] is also not an area efficient fault tolerant scheme. For certain N node PENs the area could increase by a factor of $O(\sqrt{N})$ if the Diogenes approach is used.

We also present a different approach to analyzing the reliability of fault-tolerant PENs. In our analysis we take into account the failure of the switches required for reconfiguring the system, along with link and processor failures. This is in contrast to the analysis in [3,5,6,8,9] where switches are assumed to be fault-free. We believe that our approach to reliability analysis of PENs is more accurate than the approach in [3,5,6,8,9].

2. Fault-Tolerant Basic Modules

In our approach to designing fault-tolerant PENs we first define a **basic module** (**BM**) for the PEN of interest. The BM should be such that a number of copies of the BM could be interconnected to construct the PEN. Figure 2.1(a) shows a BM for Meshes having even number of rows and an even number of columns. The corresponding FTBM is shown in Figure 2.1(b).

Given a BM we construct an FTBM as follows. The number n of **corners** of the FTBM equals the number of PEs in the BM. The corners of the FTBM are named $C_1, \ldots, C_n$. Every FTBM with n corners has $(n+1)$ PEs, $PE_1, \ldots, PE_n$, and SPE. SPE is a spare PE which is used to replace a faulty PE within the FTBM. Since our scheme uses only one spare within an FTBM, our scheme can tolerate only one processor failure within an FTBM. One can easily extend the scheme to tolerate multiple failures within an FTBM by inserting more than one spare processor per FTBM.

Given a PEN the number of neighbors of a processor in the PEN i.e. the number k of **ports** per processor is known. For example, all processors in a Mesh are connected to 4 other processors; so the number of **ports** per processor in a Mesh is 4. Every corner C_i consists of k **cornerpoints** $P_{i,1}, \ldots, P_{i,k}$. For all $1 \leq i \leq n$, processor PE_i has k ports $L_{i,1}, \ldots, L_{i,k}$. The SPE has k ports named $SL_1, \ldots, SL_k$.

FTBM also contains a number of switches which are used for reconfiguring the FTBM. Each switch can be either open or close. The arrangement of the switches in FTBM is described below.

If PE_i is fault free then it is *placed* in *corner* C_i. For all $1 \leq i \leq n$, associated with PE_i are a set of switches $S_{i,1}, \ldots, S_{i,k}$. To place PE_i in *corner* C_i port $L_{i,t}$ is connected to the *cornerpoint* $P_{i,t}$ by closing switch $S_{i,t}$, for all $1 \leq t \leq k$. Note that in order to connect a port to a cornerpoint we require a switch; and a link that connects the port to the cornerpoint. In our discussion we consider such links to be part of the switch itself. If PE_i is faulty then PE_i is *removed* from *corner* C_i by opening all the switches associated with PE_i.

SPE has associated with it $n \times k$ switches. These $n \times k$ switches are divided into n groups $G_1, \ldots, G_n$ corresponding to the n corners of the FTBM. Each group consist of k switches and for each port of the SPE we have one switch in each of the groups. The switches in group G_i are named $SW_{i,1}, \ldots, SW_{i,k}$. For all $1 \leq t \leq k$, switch $SW_{i,t}$ is used for connecting/disconnecting port SL_t to/from the *cornerpoint* $P_{i,t}$. If none of the PEs are faulty then all the switches associated with the SPE are open and the SPE is not used. If there exists an i

[1] Research Supported by Grant No. RDF 150-9755-H from the Research Foundation of the State University of New York

[2] Research Supported by Grant No. 220003-07 from NYS/UUP of the State University of New York

such that PE_i is faulty then all the switches in group G_i are closed and the switches in all the other groups are opened. This places the SPE in *corner* C_i.

Figure 2.1(b) shows the naming of the various components of the FTBM for the corner C_1. Observe that for each FTBM we can use n control signals, $E_1, \ldots, E_n$, for reconfiguring the FTBM as discussed below. The switches that belong to group G_i and the switches associated with PE_i are ganged. If E_i is 1 (respectively 0) then all the switches associated with PE_i are closed (respectively open) and all the switches in G_i are open (respectively closed).

The interconnection topology within an FTBM is specified by providing a graph G(V,E). V is the set of cornerpoints and there exists an edge between two nodes in the graph if and only if the two corner points are connected. In the sequel, we refer to these links as **intra-FTBM** links. The interconnection topology for an FTBM can be derived from the BM of interest.

Note that FTBMs also have some **external links** which are used to interconnect the copies of the FTBM while constructing the fault-tolerant system. We refer to these external links as **inter-FTBM links**. The inter-FTBM links for an FTBM can be derived from the architecture of interest and the BM being used. In fact, the inter-FTBM links are identical to the inter-BM links used for interconnecting the BMs.

We next derive a set of identities which are used in the reliability analysis to follow. Switches could be either stuck-close, stuck-open or normal and that switch-failures are independent. Let m be the number of intra-FTBM links; p be the probability that a processor is fault free; p_n be the probability that a switch is normal; p_o be the probability that a switch is stuck-open; p_c be the probability that a switch is stuck-close; p_L be the probability that a link is fault-free; and P_M be the probability that we have a working FTBM.

Observe that we have a working FTBM if and only if one of the following $(n+1)$ disjoint events occur. (i) $PE_1, \ldots, PE_n$ are fault-free; all the switches associated with $PE_1, \ldots, PE_n$ are closed; and all the switches associated with SPE are open. (ii) For each $1 \leq i \leq n$ PE_i is faulty; all switches associated with PE_i are open; all switches associated with PE_j, $i \neq j$, are closed; the switches in group G_i are closed; and all switches in group G_j, $i \neq j$, are open. From this we can derive the following expression for P_M.

$$\begin{aligned} P_M &= \left[p^n \times \left[(p_n+p_c)^k \times (p_n+p_o)^k\right]^n\right] \times p_L^m \\ &+ n \times (1-p) \times p^n \times \left[(p_n+p_c)^k \times (p_n+p_o)^k\right]^n \times p_L^m \\ P_M &= p^n \times \left[(p_n+p_c)^k \times (p_n+p_o)^k\right]^n \times [1+n(1-p)] \times p_L^m (1) \end{aligned}$$

Let K be the number of FTBMs required and T the number of inter-FTBM links required to construct the PEN. The reliability R of the fault-tolerant PEN is given by the following equation.

$$R = P_M^K \times p_L^T \qquad (2)$$

The values of K and T are computed for the PENs under consideration and are dependent on the BMs used. It should be clear from the above discussion that the values of n, k, m, K and T along with equations (1) and (2) are sufficient for reliability analysis of the fault-tolerant PEN.

3. Fault-Tolerant Pyramids

Pyramids have found extensive use in image-processing[8]. A Pyramid of size 16 is shown in Figure 3.1. Each processor is connected to as many as 9 neighbors. A PE in *level i* is connected to one PE in *level (i+1)* using the *up* link, provided *level (i+1)* exists; to four PEs in *level i* using the *north*, *south*, *east* and *west* links; and to four PEs in *level (i-1)* using the *downlinks.* Note that every level, except the level containing the apex, has an even number of PEs which is also a perfect square.

Our fault-tolerant Pyramid uses the FTBM derived from the BM of Figure 3.2, as discussed in Section 2. For this FTBM we have $n=4$; $m=4$; and $k=9$. The inter-FTBM links are identical to the links of the BM shown in Figure 3.2. N_1 and N_2 are the two **north links.** E_2 and E_3 are the two **east links.** S_3 and S_4 are its two **south links.** W_1 and W_4 are the two **west links.** Each FTBM has one *up-neighbor.* All corners of the FTBM are connected to exactly one corner of its up-neighbor. Accordingly, each corner C_i of the FTBM has one **up link** U_i which is used for connecting the corner C_i of the FTBM to its *up-neighbor.* Each corner C_i has four *down* links, $D_1^i, D_2^i, D_3^i, D_4^i$, which are used for connecting the corner C_i of the FTBM to its four *down neighbors.*

Figure 3.1 shows how the FTBM of Figure 3.2 can be used to construct the Pyramid. Note that the level containing the apex of the pyramid contains only one processor. In our discussion we assume that the root is also made up of one FTBM and only one of its corners is used. In Figure 3.1 the FTBMs are demarcated using dashes.

We next present a sketch of the derivation of the reliability expression of the fault-tolerant PEN resulting from this scheme. For a complete derivation refer to [2]. Consider a pyramid with $L+1$ levels. Let N_i be the number of FTBMs at level i; and K be the total number of FTBMs. Then, for all $0 \leq i \leq L-1$, $N_i = 4^{L-i-1}$. Therefore, $K = 1 + \sum_{i=0}^{L-1} N_i = \frac{4^L+2}{3}$.

To compute the number T of inter-FTBM links the inter-FTBM links are grouped into G_1, G_2. G_1 consists of all the inter-level links and G_2 consist of all the intra-level links. Let $T_1 = |G_1|$; and $T_2 = |G_2|$. $T_1 = 4 \times \frac{4^L-1}{3}$.

There are no intra-level links for the level containing the root. For all $0 \leq i \leq L-1$, let T_2^i be the number of intra-level links at level i; T_H^i be the number of horizontal links between the columns of FTBM of level i; and T_V^i be the number of links between the rows of FTBM of level i. We have, $T_H^i = 2 \times \sqrt{N_i} \times (\sqrt{N_i} - 1)$; and $T_2^i = T_H^i + T_V^i = 2 \times T_H^i$. Therefore, $T_2^i = 4 \times 4^{\frac{L-i-1}{2}} \times \left[4^{\frac{L-i-1}{2}} - 1\right]$. Also, $T_2 = \sum_{i=0}^{L-1} T_2^i = \frac{4\times(2^L-1)\times(2^L-2)}{3}$. Therefore, $T = T_1 + T_2 = \frac{4}{3} \times (2^L-1)(2^{L+1}-1)$.

For this FTBM, n = 4, k = 9 and m = 4. The reliability expression for the fault-tolerant PEN can now be computed using equation (2) of Section 2.

4. Other Common Topologies

We discuss briefly the fault-tolerant schemes for Binary Trees, Meshes and Hypercubes derived using our approach. The scheme for X-Tree is very similiar to the Binary Tree and is discussed in [2].

The **Binary Tree** structure is well known. A BM for the Binary Tree is shown in Figure 4.1(a). The corresponding FTBM for the BM of Figure 4.1(a) can be derived as discussed in Section 2. Note that the FTBM so derived is the module used in [3] for designing the reconfigurable Binary Tree. Figure 4.1(b) shows how the FTBM can be interconnected to form the Binary Tree.

For this FTBM we have $n=3$; $m=2$; and $k=3$. Let K be the number of FTBMs; 2M be the number of levels in the tree; and T be the number of inter-FTBM. Therefore, as shown in [2], $K = \frac{4^M-1}{3}$ and $T = \frac{4^M-4}{3}$. From the values of n, m and k the expression for P_M can be derived using equation (1) of Section 2. From the values of P_M, T and K computed above, we can derive the reliability expression for the Binary Tree using equation (2) of Section 2.

Meshes, as an interconnection topology, have been used in the ILLIAC IV and studied in [1]. Figure 4.2(a) depicts a mesh whose sides are of size n and we assume that $n = 2M$. Figure 2.1(a) depicts a BM for the mesh. For the corresponding FTBM we have $n=4$; $k=4$; and $m=4$. Figure 4.2(b) shows how the BM of Figure 2.1(a) can be interconnected to form a mesh whose sides are an even number. We have $T = M^2$; and $K = 4^{(M-1)}$[2]. From this and the model for FTBM in Section 2 we can derive the reliability expressions for the mesh.

The **hypercube** of size N, where $N = 2^q$, is defined as follows. Each PE is assumed to have a $q(=log(N))$ bit address. Two PEs are connected if and only if their address differs in exactly one bit position. A BM for the hypercube is shown in Figure 4.3(a) and a method for constructing the hypercube from the BM of Figure 4.3(a) is shown in Figure 4.3(b). We can view the BMs to be made up of the four processors that have the $q-2$ most significant bits to be identical. Accordingly, we can assign a $q-2$ bit address to each of the FTBMs used. Then, two FTBMs have an inter-FTBM link between them if and only if their $q-2$ bit address differ in atmost one bit position. Therefore, $K = 2^{q-2}$; and $T = (q-2) \times 2^{q-1}$[2].

5. Comparison with the Diogenes Approach

The Diogenes approach[7] has been shown to be applicable to a variety of PENs. Here we compare our approach with the Diogenes approach.

The **first figure of merit** we use is the **area required** by the fault-tolerant PENs designed using the two approaches. We had noted earlier that the fault-tolerant Binary Tree resulting from our approach is the same as the fault-tolerant Binary Tree of [3]. In [3] it was shown that the fault-tolerant Binary Tree with N nodes has a layout of size $O(N)$. We next present a layout strategy for our fault-tolerant PENs to shows that our scheme is area efficeint.

Define the **corner layout graph (CLG)** of the fault-tolerant PEN as follows. For each FTBM used in constructing the PEN, CLG has n nodes, where n is the number of corners of the FTBM. There exists an edge between two nodes in the CLG if and only if there exists either an inter-FTBM link or an intra-FTBM link between the cornerpoints of the two corresponding corners.

Define the **FTBM layout graph (FLG)** of the fault-tolerant PEN as follows. For each FTBM used in constructing the PEN, FLG has one node. There exists an edge between two nodes in the FLG if and only if there exists an inter-FTBM link between the corresponding FTBMs. The following observation follows from the definition of FLG. *The FLG of the fault-tolerant Binary-Tree, X-Tree, Mesh, Pyramid and Hypercubes presented in this discussion are respectively Binary-Tree, X-Tree, Mesh, Pyramid and Hypercube.*

The CLG can be derived from the FLG by replacing each node in FLG by the **corner graph** of the FTBM which is defined as follows. For each corner of the FTBM the CG contains a node. There exists an edge between two nodes in the CG if and only if there exists an intra-FTBM link between the cornerpoints of the corresponding corners. The CGs of the different FTBMs used in this discussion are shown in Figure 5.1. We next describe the basic steps of our layout scheme.

1. Construct the FLG H of the fault-tolerant PEN.
2. Layout H using the layout algorithm for the PEN.
3. Expand each node of H as follows. Let H_i be the corner graph of the FTBM in question. H_i is said to be hamiltonian if and only if there exists an acyclic path p in H_i such that p traverses each of the nodes of H_i. An inspection of Figure 5.1 shows that the CGs of each of the FTBMs used in this discussion are hamiltonian. In Figure 5.1 the hamiltonian paths are shown using dashed lines.

The node in H_i corresponding to the corner C_j is henceforth referred to as node C_j. Without loss of generality let $C_1, \ldots, C_n$ be the hamiltonian path. Then, C_1 contains the spare processor SPE; for all j, C_j contains the switches $S_{j,1}, \ldots, S_{j,k}$ and the processor PE_j; for all j, C_j contains the switches in group G_j; and for all j, C_j contains the cornerpoints of the corner C_j.

The edges between the nodes of the CG are expanded to include the intra-FTBM links between the cornerpoints. The connection between the ports of the spare and the switches in the different groups are run parallel to the hamiltonian path. For each switch there is a link to it from the appropriate port of the SPE. Only $n \times k$ such links between two adjacent nodes in the hamiltonian path are needed.

Asymptotically, the area required by the resultant layout will be the same as the area required by the FLG constructed in the first step provided the number of neighbors of a PE in a PEN is a constant. This is because each of the nodes will be replaced by a constant number of switches and processors and each edge will be replaced by a constant number of edges. From this it follows that the fault-tolerant Binary-Tree, X-Tree, Mesh, Pyramids are all area efficient.

The above layout strategy does not give an area efficient fault-tolerant Hypercube because the number of neighbors of processors in the Hypercube is not a constant. But, for the Hypercube we use a different layout strategy which for area efficient layout and is discussed in [2].

Unlike our fault-tolerant scheme, the Diogenes scheme is not area-efficient. This is because, as stated in [7], the Diogenes approach uses collinear layouts[4]. Collinear layouts suffer from the drawback that they do not lead to optimal area. For example, there exists a layout viz. the H-Layout[4] for the Binary Tree which requires area $O(N)$ but the area required by the collinear layout of a tree is $O(N Log(N))$[4]. Similiarly, the collinear layout for the Mesh requires area $O(N\sqrt{N})$[4] whereas, there exist layouts for the Mesh that require area $O(N)$[4]. Thus we see that the fault-tolerant PENs derived using the Diogenes approach have an area overhead that is larger than the area overhead of the fault-tolerant PENs derived using our approach. The ratio of the areas required by the Diogenes approach and our approach could be as high as $O(\sqrt{N})$.

The **second figure of merit** we use is the **fault tolerance** of the fault-tolerant PENs. It was stated in Section 2 that our scheme can tolerate only one processor failure per FTBM. The Diogenes approach is a globally fault-tolerant scheme in that it has the characteristic that if S spares are used it can recover from any S processor failures. This is often referred to as 100% spare utilization. Therefore, the Diogenes approach has greater fault-tolerance than our approach.

6. Discussion

We presented an area efficient, locally redundant fault-tolerance scheme for PENs and showed it to be applicable to a variety of PENs. The Diogenes approach which is a global redundancy scheme having 100% spare utilization is not area efficient. These two schemes represent the two extremes of the spectrum of fault-tolerance schemes for PENs. For the Binary Tree a number of other fault-tolerant schemes [5,6,10] has also been proposed. These schemes are either a combination of local and global schemes[5,6] or local schemes using a varying number of spares[10].

A new approach to analyzing the reliability of PENs was presented. Our reliability analysis takes into account failure of switches which are used for reconfiguring the system. This makes our reliability analysis more accurate than the reliability analysis of fault tolerant PENs that had appeared in the literature which does not take into account switch failures.

References

1. M. J. Atallah and S. R. Kosaraju, "Graph Problems on a mesh-connected processor array", *JACM* 31(1984), pp. 649-667.
2. S. Chakravarty and S. J. Upadhyaya, "A Unified Approach to Designing Fault-Tolerant Processor Ensembles", *Tech. Rep. No. 87-18*, Dept. of Computer Science, State University of New York, Buffalo.
3. A. S. M. Hassan and V. K. Agrawal, "A Fault-Tolerant Modular Architecture for Binary Trees", *IEEE Trans. on Comput.*, Vol. C-35, No. 4, pp. 356 - 361, April 1986.
4. C. E. Leiserson, "Area-Efficient VLSI Computation". *PhD Dissertation, Dept. of Computer Science, CMU, Pittsburgh, PA*, 1981.
5. M. B. Lowrie and W. K. Fuchs, "Reconfigurable Tree Architectures Using Subtree Oriented Fault Tolerance", *IEEE Trans. on Comput.*, Vol. C-36, No. 10, 1986, pp. 1172 - 1182.
6. C. S. Raghavendra, A. Avizienis and M. D. Ercegovae, "Fault-Tolerance in Binary-Tree Architectures", *IEEE Trans. on Comput.*, Vol. C-33, No. 6, June 1984.
7. A. Rosenberg, "The Diogenes Approach to Testable Fault-Tolerant Network of Processors", *IEEE Trans. on Comput.*, Vol. C-32, pp. 902 - 910, Oct. 1983.
8. A. Rosenfeld, "Quadtrees and pyramids for pattern recognization and image processing", in *Proc.. Fifth Int. Conf. Pattern Recognition*, 1981, pp. 802 - 806.
9. A. D. Singh, "An Area Efficient Redundancy Scheme for Wafer Scale Processor Arrays", in *Proc. 1985 ICCD*, Oct. 1985, pp. 505-509.
10. A. D. Singh, "A Reconfigurable Modular Fault-Tolerant Binary Tree Architecture", in *Proc. 17th Fault-Tolerant Computing Symposium*, 1987, pp. 298 - 304.

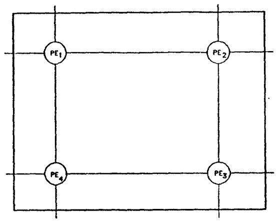


Figure 2.1 (a): A Basic Module for Mesh.

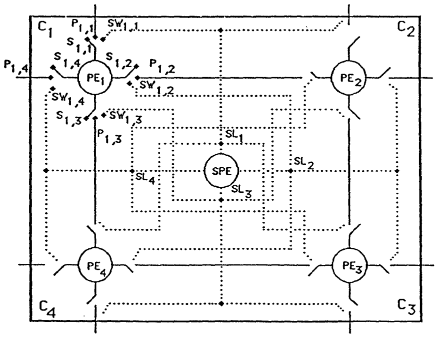


Figure 2.1(b): A Fault Tolerant Basic Module with One Spare

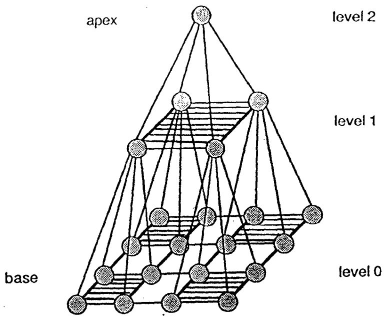


Figure 3.1: A Pyramid of Size 16.

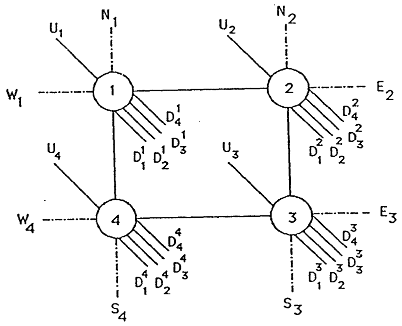


Figure 3.2 A Basic Module for Pyramid

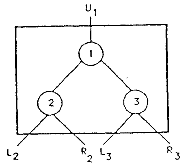


Figure 4.1(a): A Basic Module For Binary Tree.

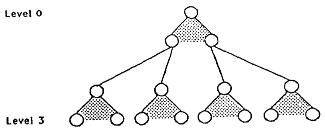


Figure 4.1 (b): A 4-Level Binary Tree From Basic Modules

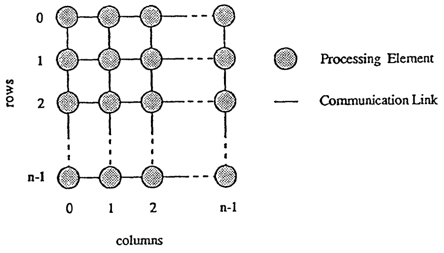


Figure 4.2.(a): A Mesh of Size n.

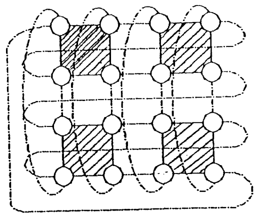


Figure 4.2.(b): A Mesh of Size 4 From The Basic Modules.

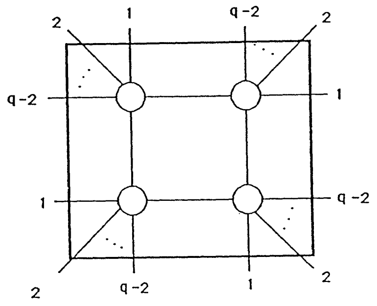


Figure 4.3 (a): A Basic Module for Hypercubes

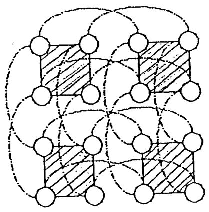


Figure 4.3 (b): A Hypercube of Size 16 from The Basic Modules.

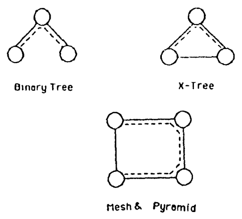


Figure 5.1: Corner Graphs.

FAULT-TOLERANT SCHEDULING OF INDEPENDENT TASKS AND CONCURRENT FAULT-DIAGNOSIS IN MULTIPLE PROCESSOR SYSTEMS

by

Seyed H. Hosseini
Department of Electrical Engineering and Computer Science
P.O. Box 784
University of Wisconsin--Milwaukee
Milwaukee, Wisconsin 53201

Abstract--The reliable execution of the critical tasks on computing systems where faulty responses can jeopardize human life or can cause a vast loss of money are important design issues.

This work studies the reliable execution of the tasks in an environment where processors and interprocessor communication channels are subject to failure. Every task is assigned to a group of the processors for execution. The processors of a group compare their outputs to obtain the group output.

Performance is improved if the number of concurrent tasks (i.e., the number of groups) is maximized. In order to achieve that, a system is modeled with the use of a graph where a node and a link of the graph represent a processor and a communication channel between two processors in the system, respectively. A new concept called group maximum matching, which is an extension to the classical maximum matching, is introduced. In a group maximum matching the nodes of a graph are grouped such that no two groups share the same node, the nodes of a group comprise a connected subgraph of the graph, and the number of groups is maximum. A heuristic algorithm is proposed to obtain a group maximum matching.

A fault-tolerant scheduling algorithm based on the group maximum matching is developed which ensures the error-free execution of the tasks. Furthermore, the proposed algorithm has the capability for the on-line fault-diagnosis of the faulty processors and interprocessor communication channels. Fault-diagnosis is achieved as system runs its normal tasks; hence, a diagnosis program is not needed for that purpose.

I. Introduction

Reliable execution of critical tasks on computing systems where faulty responses can jeopardize human life or can cause a vast loss of money are important design issues. The most stringent requirement for reliability is in the real time control systems where repair is not possible and recovery time from faults must be short. The design of fast, available, reliable, and dependable computers has become possible with the use of multiple processor systems.

Non-fault-tolerant scheduling of the tasks are considered in [1]-[5], where it is assumed that processors and interprocessor communication channels are fault-free.

System level fault diagnosis was introduced in [6], where processors test each other for the detection and location of the faults, and generalized in [7]. The above techniques, in general, require first diagnosis of the faults, then recovery from the faults. However, testing a processor for the detection of all kinds of faults is debatable and is not an easy job; and also subsequent error recovery takes time and is generally slow. Therefore, this has motivated the introduction of another model called comparison model where at least two processors are assigned to execute the same task and their outputs are compared.

Comparison model is employed in [8]-[16]. However, works in [8]-[14] are geared only toward fault-diagnosis of the processors. As a result the assignment of the jobs to the processors is not based on any efficient job allocation scheme which maximizes the system performance (i.e., the system throughput) while executing the jobs reliably. For instance, in [8] in order to check the status of every processor a minimum covering algorithm for single fault-diagnosis is proposed. Furthermore, in [8-10], it is assumed that a faulty processor produces an incorrect output. This does not hold true all the time unless the programs running on the processors are diagnostic programs rather than normal system or user programs. The works in [11]-[13] do not consider assignment of jobs to the processors based on any particular allocation scheme. The work in [14] assign jobs to the processors in rounds based on a permutation scheme in which processors with short jobs finish early and remain idle until the end of the current round of the execution.

In [8-14] only processor failure is considered, we consider both processor and interprocessor communication channel failures.

We consider a homogeneous system in which every task can be assigned to any processor. Examples of such systems are Intel IPSC System [17], and several other systems [18]-[20]. We assume that tasks are independent tasks, do not communicate with each other, and are either independent subtasks of a job or independent jobs. We further assume a non-preemptive scheduling scheme where as soon as a task is scheduled to run on a processor, it runs to the completion of the task. We also assume that a faulty processor does not have Byzantine malicious behavior and its faulty behavior is solely caused by the faults affecting its hardware circuitry. Finally, we assume that when running a task on a faulty processor, faults in that processor may or may not affect the output of the task, which depend on the task and the nature and place of the faults in the processor. Thus, we consider the tasks running on the processors are normal user programs but not diagnostic programs which are carefully written to force the faulty processor to generate incorrect output.

In this work, we are only concerned with fault-tolerant scheduling of the tasks and concurrent diagnosis of the faults. But, the proposed algorithm may be combined with other scheduling policies [21] such as first-come, first-served, shortest job first, round robin, etc., under any type of constraint such as priority and deadline of the tasks. Hence, in our work we do not specify how the next job from the job ready queue is selected for the execution.

In the next section II, we propose an algorithm for the effective pairing or grouping of the processors. Then, in the section III, we propose our fault-tolerant scheduling algorithm based on the first algorithm.

A. System Model

A system is modeled by a graph G(V,E), where V and E are the node set and the edge set of the graph G, respectively. A node represents a processor with its local memory, while an edge between two nodes represents a communication channel between their corresponding processors in the system.

B. Fault-Model

A comparison fault model is defined as follows: When two processors P_i and P_j which are assigned to execute the same task T_k, then they will do: First, each executes T_k independently, then they exchange their outputs, next $P_i(P_j)$ will obtain its test outcome $a_{ij}(a_{ji})$ with respect to T_k as follows:

1. If $P_i(P_j)$ agrees with $P_j(P_i)$ then $a_{ij} = 0$ $(a_{ji} = 0)$
2. Else $a_{ij} = 1$ $(a_{ji} = 1)$
   Notice that:
   a) a_{ij} is not necessarily the same as a_{ji}.
   b) Both a faulty processor or a faulty communication channel could be the source of the disagreement.
   c) P_i and P_j may produce the same output and agree with each other even if one or both of them are faulty, which depends on whether faults in the processors and communication channel between them affect their outputs or not. For instance, a fault could be in a register of one of the processors, but that fault cannot affect the output of the T_k if that register is not used in executing T_k.

II. Matching Concept

In the classical graph theory the matching concept is defined as follows:

Definition 1 [22]: Given a graph G(V,E), a matching is a subset of edges $F \subseteq E$ such that no two edges of F are adjacent.

A matching which covers every node is a perfect matching.

A matching is maximum if there is no other matching which has more edges. A perfect matching is always a maximum matching, while the converse is not necessarily true.

The concept of the classical matching problem is used to pair or to group the nodes of a graph into 2-node disjoint groups. A generalization to that concept is to group the nodes into (t+1)-node disjoint groups. We refer to the generalization matching problem as the group matching problem. In the classical maximum matching problem, nodes are paired such that the number of pairs is maximum. Similarly, we define group maximum matching as a problem where nodes are grouped such that no two groups share the same node, the nodes of every group comprise a connected subgraph of the graph, and the number of groups is maximum.

A. Group Maximum Matching Algorithm

In the following we develop an algorithm to find a group maximum matching for the graphs. This algorithm which is a greedy heuristic algorithm attempts to avoid the isolation of the nodes such that they can be included in a group. This is achieved by first including the nodes with the lower degrees in the groups, then the ones with the higher degrees. In this algorithm every group is intended to have (t + 1) nodes.

```
1.        Initialize i = 1, $g_1 = \phi$, $g_2 = \phi$, and so
          on
2.        While system graph is non-empty do
2.1       Find the node with the lowest degree.
          In case of a tie, choose one of them
          randomly. Call the node with the mini-
          mum degree the current node.
2.2       Add the current-node to $g_i$
2.3       Sort the neighbors of the current node
          with respect to their degrees in an in-
          creasing order.
2.4       If the current-node(s) have some neigh-
          bors then
2.4.1     If the current-node(s) have $t-|g_i| + 1$
          or more neighbors then
2.4.1.1   Choose the first $t-|g_i| + 1$ sorted
          neighbors and add them to $g_i$
2.4.1.2   Delete the nodes in $g_i$ from the system
          graph
2.4.1.3   i = i + 1
2.4.1.4   Go to 2
2.4.2     Else
2.4.2.1   Add all the neighbors of the current-
          node(s) to $g_i$
2.4.2.2   Sort the neighbors of the nodes in $g_i$
          with respect to their degrees in an in-
          creasing order. Call the nodes in $g_i$
          the current nodes.
2.4.2.3   Go to 2.4
2.5       Else
2.5.1     Delete the nodes in $g_i$ from the system
          graph
2.5.2     i = i + 1
2.5.3     Go to 2
2.6       End while do
```

<u>Example 1</u>: Figure 1.a shows an example with $t = 2$. Initially, every node is of degree 3. Nodes are sorted with respect to their degrees. Let the node P_1 be listeded on the top of the list. Then, P_1 is elected as the current node and added to g_1. Let the neighbors of P_1 be sorted as $\{P_2, P_{12}, P_8\}$. Then the first two neighbors are added to g_1 and $g_1 = \{P_1, P_2\ P_{12}\}$. Next the nodes in g_1 are deleted from the system graph and the graph shown in Fig. 1.b is obtained. Then, the nodes in this graph are sorted and the node P_3 with the lowest degree of 1 will be on the top of the list. P_3 is elected as the current node and added to g_2. Then, its only neighbor P_4 is added to g_2. The sorted neighbors of the nodes in g_2 are listed as $\{P_{11}, P_5\}$. Then, P_{11} is added to g_2 and $g_2 = \{P_3, P_4, P_{11}\}$. Nodes in g_2 are deleted from the graph and the graph shown in Fig. 1.C is obtained. Let, after sorting the nodes in this graph, P_5 with the degree of 2 be on the top of the list. Then, P_5 is elected as the current node and added to g_3. Next, the sorted neighbors of P_5 are listed as $\{P_{10}, P_6\}$, both are added to g_3, and $g_3 = \{P_5, P_6, P_{10}\}$. Nodes in g_3 are deleted from the graph and the graph shown in Fig. 1.d is obtained. Subsequently, $g_4 = \{P_7, P_8, P_9\}$ and algorithm terminates.

A group matching which groups the nodes randomly, generally turns out to generate a group matching with less number of groups than the proposed algorithm. For instance, Fig. 2 shows a matching with only three groups, g_1, g_2, and g_3 for the same graph of Fig. 1 where, $g_1 = \{P_1, P_2, P_{12}\}$, $g_2 = \{P_4, P_{10}, P_{11}\}$, and $g_3 = \{P_6, P_8, P_9\}$. In this group non-maximum matching nodes P_3, P_5 and P_7 are isolated because of the random grouping of the nodes and not using any state knowledge to group them effectively. For instance, after the selection of the nodes of group g_1, the nodes of the group g_2 are selected randomly without the regard to the status of the node P_3 which can be grouped only with the node P_4. A better strategy is first to try to include P_3 in a group which has fewer neighbors to group with, and then try to include other nodes which have more neighbors to group with. This strategy is used in the proposed heuristic algorithm.

B. <u>Time Complexity of the Algorithm</u>

Assuming that sorting a list by the heap sort takes n log n time, then the total time is obtained as follows. The while-do loop and the step 2.4 are repeated at most n and t times, respectively. Thus, the total time is in the order of $n.(t.(n \log n))$ or $n^2.t.\log n$.

III. <u>Fault-Tolerant Scheduling Algorithm</u>

This algorithm is devised to execute tasks reliably and also to achieve on-line fault-diagnosis in the environment where processors and communication channels are subject to failure. For the reliable execution of the tasks each task is assigned to a group of processors. Processors are grouped with the use of group maximum matching algorithm. A task is released if at least $(t + 1)$ processors produce the same output with the assumption that no more than t faulty processors or communication channels exist.

In order to achieve on-line fault-diagnosis the notion of a disagreement graph $G(D,M)$ with the node set D and the edge set M is introduced as follows.

A. <u>Disagreement Graph</u>

A disagreement graph $G(D,M)$ with respect to a task T is obtained for a group of the nodes which are assigned to execute the task T as follows: Every node D_i of the disagreement graph contains those nodes of the group such that for every node $P_j \in D_i$ and $P_k \in D_i$, P_j and P_k agree with each other on the output for the task T, i.e., $a_{jk} = a_{kj} = 0$ if they are neighbors in the system graph. An edge exists between two nodes D_i and D_j of the disagreement graph if there exists a node $P_x \in D_i$ and a node $P_y \in D_j$ such that P_x and P_y are neighbors in the system graph and they disagree with each other on the output for the task T, i.e., either $a_{xy} = 1$ or $a_{yx} = 1$.

In the following the formal description of the fault-tolerant task scheduling algorithm is given.

B. <u>Fault-Tolerant Scheduling Algorithm</u>

```
1.        Run the maximum matching algorithm for $t = 1$ or the group maximum matching
          algorithm for $t > 1$ to group the nodes of the system into groups $g_1$, $g_2$, ..., and so on.
2.        Assign every task $T_i$ to a group $g_h$ for the execution by all the nodes in that group,
          where $|g_h| > 1$ and all the nodes in $g_h$ are free.
3.        Upon the completion of every task $T_i$ by all the nodes assigned $T_i$ do
3.1       Ask the nodes which are assigned $T_i$ to exchange and compare their outputs if they are
          neighbors in the system graph, and then to obtain their test outcomes with respect to the task $T_i$.
3.2       Obtain the disagreement graph $G(D,M)$ for the task $T_i$. Let $D_1$, $D_2$, ..., and so on
          be the nodes of the disagreement graph with respect to $T_i$
3.3       For every $D_i$ with $0 < |D_i| \leq t$ do
3.3.1     For every node $P_j \in D_i$ do
3.3.1.1   If the number of nodes $P_j$ disagreed with in this round of execution of $T_i$ was more
          than t, then consider $P_j$ faulty, add it to the faulty set $S_F$, and set $D_i = D_i - \{P_j\}$
3.3.1.2   End for do
3.3.2     End for do
3.4       Find the first $D_i$ with $0 < |D_i| \leq t$ such that the nodes in $D_i$ have some
          neighbors $P_j$ which are neither in the faulty set $S_F$ nor in $D_i$
3.4.1     If there exists at least a $D_i$ then
3.4.1.1   If among the above neighbors in 3.4 there are some nodes which are idle (i.e., not
          included in any group $g_k$ with $|g_k| > t$) then select them
```

```
3.4.1.2     Else
3.4.1.2.1   Select a node $P_j$ among the above neighbors in 3.4 which is running a task $T_j$ with the lowest priority, in case of a tie, choose one of the neighbors randomly
3.4.1.2.2   Abort the task $T_j$ assigned to $P_j$ and to all of the nodes in its group. $T_j$ must be rescheduled for execution by some other nodes as is done in the step 2.
3.4.1.2.3   Select the node $P_j$ and all of the nodes in the group of $P_j$
3.4.1.3     Assign the task $T_i$ to all of the selected nodes in 3.4.1.1 or 3.4.1.2.3 and also again to all of the nodes in the current disagreement graph with respect to $T_i$.
3.4.1.4     Go to 3.
3.5         If there are any $|D_i| > t$ during the above process then release the output of $T_i$ executed by one node in $D_i$.
3.6         Else abort $T_i$ because can not find at least t+1 nodes to agree with each other over $T_i$.
3.7         The link between two fault-free processors is considered faulty if they disagree with each other.
3.8         If no node or link has failed or there are still some disagreements with respect to some other tasks not yet resolved then go to 2
3.9         Else
3.9.1       Finish the tasks already in progress with the use of the old group maximum matching and concurrently go to 1 to get a new group maximum matching after the deletion of the faulty nodes and links from the system graph.
3.9.2       Assign the new tasks based on the new group maximum matching.
```

Example 2: Consider the system graph given in Fig. 3.a. Assume that t = 1 and a maximum matching, obtained by running the step-1 of the algorithm, pairs the nodes into the groups $g_1 = \{P_1, P_2\}$, $g_2 = \{P_3, P_4\}$, $g_3 = \{P_5, P_6\}$, $g_4 = \{P_7, P_8\}$ and $g_5 = \{P_9, P_{10}\}$. Assume that task T_i is assigned to the nodes in g_i for i = 1, 2, 3, 4, and 5. Furthermore, assume that the node P_2 and the link $P_5 - P_6$ are faulty. Upon the completion of T_1, the disagreement graph with respect to T_1 shown in Fig. 3.b is obtained. Assuming that the priority of T_5 is greater than T_2, then the task T_2 is aborted and the nodes in $g_1 \cup g_2$ are assigned to perform T_1, see Fig. 3.c. Then, assume that nodes in g_3 finish executing T_3. Figure 3.e shows the disagreement graph with respect to T_3. Assuming that the priority of T_1 is greater than that of T_4, then the task T_4 is aborted and the task T_3 is assigned to the nodes in $g_3 \cup g_4$, see Fig. 3.f. Next, assume that T_5 finishes. Figure 3.g shows the disagreement graph with respect to T_5. Since $|D_1| > t = 1$, T_5 is released and a new task T_6 is assigned to the nodes in g_5, see Fig. 3.h. Upon the completion of T_1, the disagreement graph with respect to T_1 is obtained, see Fig. 3.i. Since $|D_1| = 1 \leq t$, the lower priority task T_6 is aborted and the nodes in $g_1 \cup g_2 \cup g_5$ are assigned to execute T_1, see Fig. 3.j. Then, assume that T_1 finishes and the disagreement graph with respect to T_1 shown in Fig. 3.k is obtained. Since the number of nodes P_2 has disagreed with is greater than t = 1; P_2 is concluded to be faulty. Then, task T_1 is released because there is at least a D_i (i.e., either D_1 or D_3) with $|D_i| \geq t = 1$. Thereafter; since there still exists disagreement with respect to the task T_3, tasks T_2 and T_6 are assigned to the freed nodes in g_2 and g_5, respectively; see Fig. 3.l. Next, assume that T_3 completes and the disagreement graph shown in Fig. 3. m is obtained. Since $|D_2| > t = 1$ and $|D_1| \leq t = 1$, the lower priority task T_2 is aborted, and task T_3 is assigned to the nodes in $g_2 \cup g_3 \cup g_4$; see Fig. 3.n. Upon the completion of T_3, the disagreement graph in Fig. 3.p is obtained. Since $|D_1| = |D_2| > t = 1$, all the nodes in $g_2 \cup g_3 \cup g_4$ are fault free, but the link $P_5 - P_6$ is faulty. Then, task T_3 is released. At this time no disagreement exists. Hence, a new maximum matching pairs the nodes into the new pairs $g_1 = \{P_1, P_{10}\}$, $g_2 = \{P_8, P_9\}$, $g_3 = \{P_6, P_7\}$, and $g_4 = \{P_4, P_5\}$. Task T_6 continue to complete based on the old maximum matching by the nodes in the old group $g_5 = \{P_9, P_{10}\}$. Tasks T_2 and T_4 are assigned based on the new maximum matching to the nodes in the new group g_3 and new g_4, respectively; see Fig. 3.r. Next, assume that T_6 completes and the disagreement graph shown in Fig. 5.s is obtained. Since $|D_1| > t = 1$, T_6 is released and the use of the old maximum matching terminates, and the new tasks T_7 and T_8 are assigned, based on the new maximum matching, to the new groups g_1 and g_2 as shown in Fig. 3.t.

Example 3: Consider the system shown in Fig. 4.a and assume that t = 2. Running the step-1 of the algorithm partitions the nodes into the groups: $g_1 = \{P_1, P_2, P_{12}\}$, $g_2 = \{P_3, P_4, P_{11}\}$, $g_3 = \{P_5, P_6, P_{10}\}$, and $g_4 = \{P_7, P_8, P_9\}$. Assume that tasks T_1, T_2, T_3, and T_4 are assigned to g_1, g_2, g_3, and g_4; respectively. Furthermore, assume that the node P_2 and the link $P_1 - P_{12}$ are faulty. Upon the completion of T_1, the disagreement graph shown in Fig. 4.b is obtained. Assume that T_2, because of low priority, is aborted and nodes in $g_1 \cup g_2$ are assigned T_1, see Fig. 4.c. Upon the completion of T_1, the disagreement graph shown in Fig. 4.d is obtained. Then, assume that task T_4, because of low priority is aborted and all the nodes in $g_1 \cup g_2 \cup g_4$ are assigned T_1, see Fig. 4.e. Then disagreement graph shown in Fig. 4.f is obtained. Since the number of nodes P_2 has disagreed with is greater than t = 2, P_2 is concluded to be faulty. Also since $|D_1| = |D_3| = 4 > t = 2$, P_1 and P_{12} are concluded to be fault-free, but the link between P_1 and P_{12} is faulty. Then, task T_1 is released. Next a new group maximum matching is obtained which partitions the nodes into the new groups $g_1 = \{P_1, P_7, P_8\}$, $g_2 = \{P_3, P_4, P_{12}\}$, and $g_3 = \{P_9, P_{10}, P_{11}\}$, see Fig. 4.g. Task T_3 continues to complete by the nodes in the old group g_3 based on the old group maximum matching. Tasks T_2 and T_4 are assigned to the new groups g_2 and g_1, respectively, based on the new group maximum matching, see Fig. 4.g. Upon the completion of T_3, the use of old group maximum matching terminates, and task T_5 is assigned to the nodes of the new group g_3, see Fig. 4.i, based on the new group maximum matching.

Theorem 1: Every task T_i is executed error-free if the proposed algorithm is employed and the number of faculty processors and faulty communication channels (both temporary and permanent faults) does not exceed an upper bound t during any round of execution of the task T_i.

Proof: Every task T_i is released when at least (t + 1) or more processors produce the same output and agree with each other. Thus, as long as for every round of the execution of the task T_i their exists not more than t faults, then when T_i is released and it is error-free. Q.E.D.

Theorem 2: The status of every processor or communication channel is identified correctly if a) the proposed algorithm is employed, b) the number of faulty processors and communication channels does not exceed t with respect to every round of the execution of each task, and c) every processor is in a connected subgraph of the system graph with at least t + 1 fault-free processors which are connected to each other through some fault-free links.

Proof: A processor is declared faulty if it disagrees with at least t + 1 other processors during the execution of a task T_i. As long as the condition (b) holds true, a fault-free processor will never be declared faulty. A faulty processor will eventually be assigned a task T_i to run for which it will produce an incorrect output. Then, as long as the conditions (b) and (c) hold true, the faulty processor will disagree with at least t + 1 fault-free processors in a connected subgraph of the system graph. Hence, it will be declared faulty. The status of a faulty communication channel between two fault-free nodes is automatically considered faulty because of their disagreement with each other over that channel. Q.E.D.

Corollary 2: The status of every processor and communication channel is identified correctly if the conditions (a) and (b) of the Theorem 2 hold true, the system graph is at least (t + 1) connected, and $n \geq 2t + 1$ (where n is the number of processors in the system).

Proof: Deleting t faulty nodes and links from the system graph will keep it still connected with at least t + 1 fault-free nodes. This satisfies the condition (c) of Theorem 2. Q.E.D.

C. Discussion:

Faults are of two types--permanent and temporary. Permanent faults describe permanent damage to the system components. Temporary faults are further subdivided into two classes--intermittent and transient. Intermittent faults describe faults that are only occasionally present due to unstable hardware and are caused by factors such as loose connection, component aging, poor design, chip contamination, etc. Transient faults describe faults which are present due to undesired environmental disturbances such as radiation, humidity, temperature variation, power supply fluctuation, physical vibration, etc. Spillman [25] has studied the nature of the temporary faults, their detection techniques, and modeling their behavior.

The proposed algorithm identifies the faulty processors and communication channels without specifying the type of the faults. But it is important to be able to differentiate permanent and intermittent faults from the transient faults. Because if the faults are permanent or intermittent, then the faulty components should not be reused any longer. We propose the following supplementary, off-line and modified on-line testing techniques.

Supplementary off-line testing can be carried on as follows: Every time a processor P_i is declared faulty with respect to a task T_j by the algorithm, then task T_j must be recorded. Then, at some later time when the system can run T_j in a clean and nice environment, the processor P_i must be asked to run T_j again. If it produced an incorrect output again, then it is either permanently or intermittently faulty; otherwise, the sources of failure are either intermittent or transient faults.

The modified on-line testing technique is based on the use of time redundancy. That is, in the occasions when some processors disagree with each other in executing a particular task T_j, before expanding the disagreement graph by asking some more processors to execute T_j, the same processors should be asked to run T_j again. If the same disagreement graph was obtained, then expand the disagreement graph with respect to T_j by asking more processors to join in executing T_j. Otherwise, do not expand the disagreement graph and ask the same group of processors to run T_j again.

IV. Conclusion

A fault-tolerant scheduling algorithm for error-free execution of the reliability critical programs was proposed. In that algorithm every program was assigned to a group of neighboring processors for execution. The conditions under which programs are executed error-free were given. A new concept called group maximum matching was introduced. This concept was used to maximize the system performance, i.e., to maximize the number of concurrent groups or programs running in the system. A heuristic algorithm for finding a group maximum matching was given. It is important to notice that every system is fault-free most of the time; hence, its average performance is dominated by its fault-free performance. The proposed group maximum matching attempts to maximize the system performance.

The proposed fault-tolerant scheduling algorithm is geared toward fault-free execution of the tasks while it attempts to achieve on-line fault-diagnosis of the faulty processors or interprocessor communication channels as system runs its normal user programs. This is an interesting feature and it frees the system designers from the troubles of writing diagnostic programs which can detect all kinds of faults in the processors in an acceptable amount of time.

References

[1] E.G. Coffman, Jr., and R.L. Graham, "Optimal Scheduling for Two Processor Systems," Acta Information 1, 1972, pp. 200-213.

[2] M.J. Gonzalez, Jr., "Deterministic Processor Scheduling," Computing Surveys, Vol. 9, No. 3, Sept. 1977, pp. 173-204.

[3] G. Dobson, "Scheduling Independent Tasks on Uniform Processors," SIAM J. Computing, Vol. 13, No. 4, 1984, pp. 705-716.

[4] C.P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," IEEE Trans. on Software Engr., Vol. SE-11, No. 10, Oct. 1985, pp. 1001-1016.

[5] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," IEEE Trans. on Computers, Vol. C-33, No. 11, Nov. 1984, pp. 1023-1029.

[6] F.P. Preparata, G. Metze and R.T. Chien, "On the Connection Assignment Problem of Diagnosable Systems," IEEE Trans. Electron Comput., Vol. EC-18, Dec. 1967, pp.848-854.

[7] S.L. Hakimi and A.T. Amin, "Characterization of the Connection Assignment of Diagnosable Systems," IEEE Trans. Comput., Vol. C-23, Jan. 1974, pp. 86-88.

[8] M. Malek, "A Comparison Connection Assignment for Diagnosis of Multiprocessor Systems," The 7th Annual Symposium on Computer Architecture, 1980, pp. 31-36.

[9] J. Maeng and M. Malek, "A Comparison Connection Assignment for Self-Diagnosis of Multiprocessor Systems," Fault-Tolerant Computing Systems, FTCS-11, 1981, pp. 173-175.

[10] M. Malek and J. Maeng, "Partitioning of Large Multicomputer Systems," 12th Annual International Symposium on Fault-Tolerant Computing, FTCS-12, 1982, pp. 341-348.

[11] A.T. Dahbura and G.M. Masson, "Greedy Diagnosis as the Basis of an Intermittent-Fault/Transient-Upset Tolerant System Design," IEEE Tran. on Computers, Vol. C-32, No. 10, Oct. 1983, pp. 953-957.

[12] A.T. Dahbura, K.K. Sabiani and L.L. King, "The Comparison Approach to Multiprocessor Fault-Diagnosis," The Fifteenth Annual International Symposium on FaultTolerant Computing, FTCS-15, 1985, pp. 260-265.

[13] C.L. Yang and G.M. Masson, "An Efficient Algorithm for Multiprocessor Fault Diagnosis Using the Comparison Approach," The Annual International Symposium on Fault-Tolerant Computing Systems, FTCS-16, 1986, pp. 238-243.

[14] K.Y. Chwa and S.L. Hakimi, "Schemes for Fault-Tolerant Computing: A Comparison of Modularly Redundant and t-Diagnosable Systems," Information and Control, 49, 1981, pp. 212-238.

[15] C.M. Krishna and K.G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," IEEE Trans. on Computers, Vol. C-35, No. 5, May 1986, pp. 448-455.

[16] D.P. Siewiorek et al., "C. vmp: A Voted Multiprocessor," in the Theory and Practice of Reliable Systems Design, by D.P. Siewiorek and R.S. Swarz, Digital Press, 1982.

[17] "IPSC System," Intel Scientific Computers.

[18] K. Hwang, "Advanced Parallel Processing with Supercomputer Architectures," in Proc. of IEEE, Oct. 1987, pp. 1348-1379.

[19] W.J. Karplus, Ed., Multiprocessors and Array Processors. San Diego, CA: Simulation Councils, Inc., Jan. 1987.

[20] K. Hwang and K.A. Briggs, Computer Architecture and Parallel Processing, McGraw Hill, 1984.

[21] H.M. Deitel, An Introduction to Operating Systems, Addison Wesley, 1984.

[22] M.N.S. Swamy and K. Thulasiraman, Graphs, Networks, and Algorithms, John Wiley and Sons, 1981.

[23] Z. Galil, "Efficient Algorithms for Finding Maximum Matching in Graphs," Computing Surveys, Vol. 18, No. 1, March 1986, pp. 23-38.

[24] S. Micali and V.V. Vazirani, "An O($\sqrt{|V|}.|E|$) Algorithm for Finding Maximum Matching in General Graphs," 21st Annual Symp. on Foundations of Computer Science, 1980, pp. 17-27.

[25] R.J. Spillman, "A Continuous Time Model of Multiple Intermittent Faults in Digital Systems," Comput. and Elec. Engng., Vol. 8, 1981, pp. 27-40.

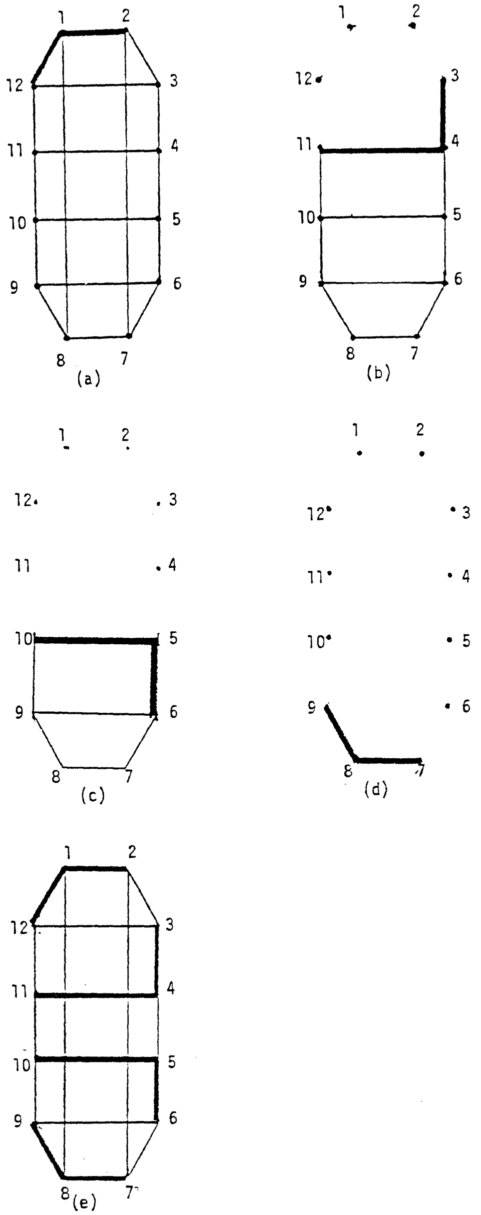


Figure 1

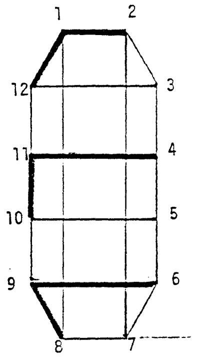
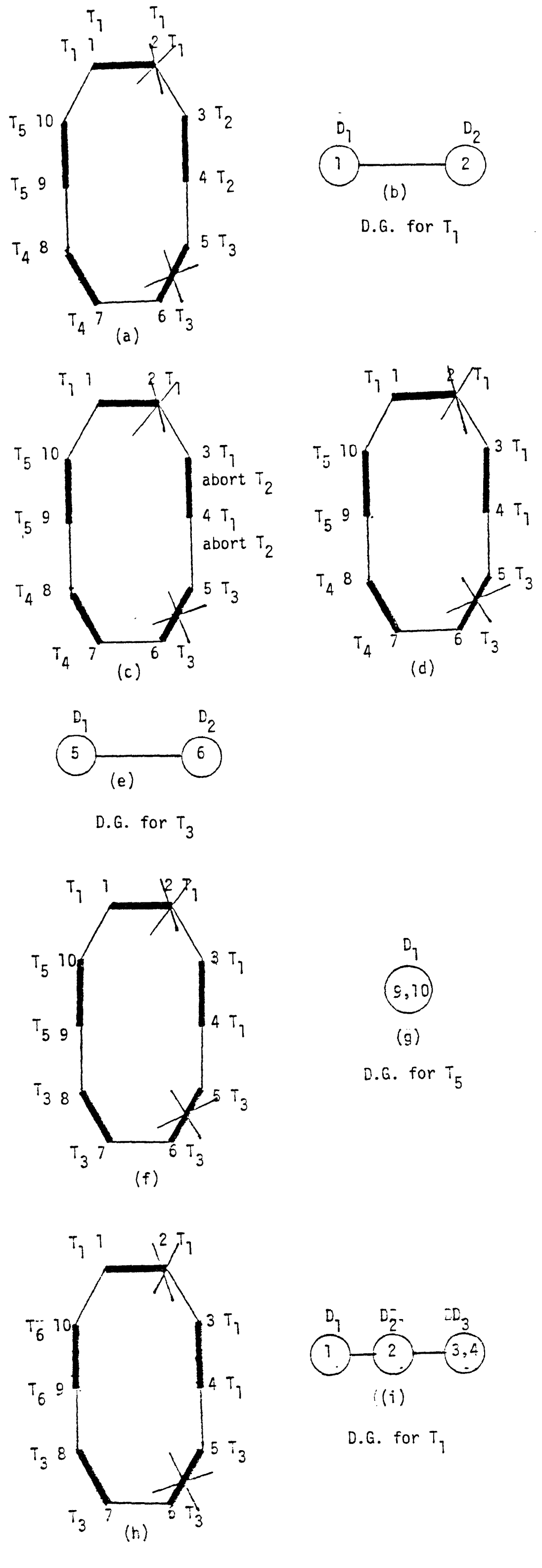


Figure 2



(b) D.G. for T_1

(e) D.G. for T_3

(g) D.G. for T_5

(i) D.G. for T_1

Figure 3

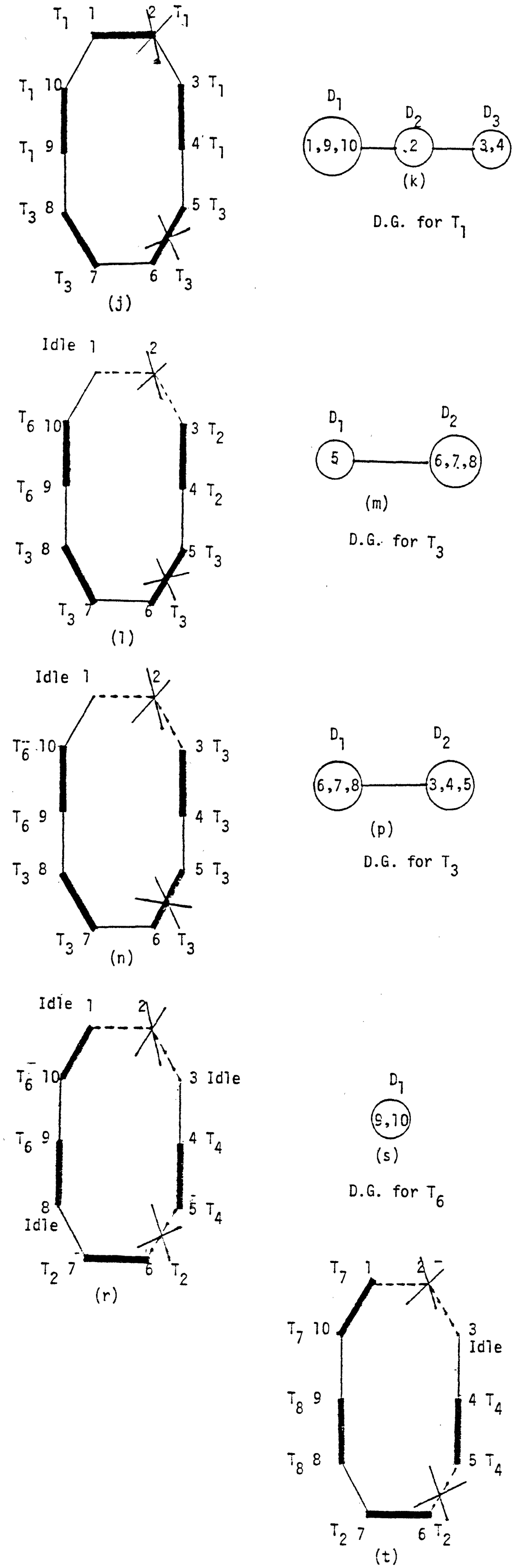


Figure 3

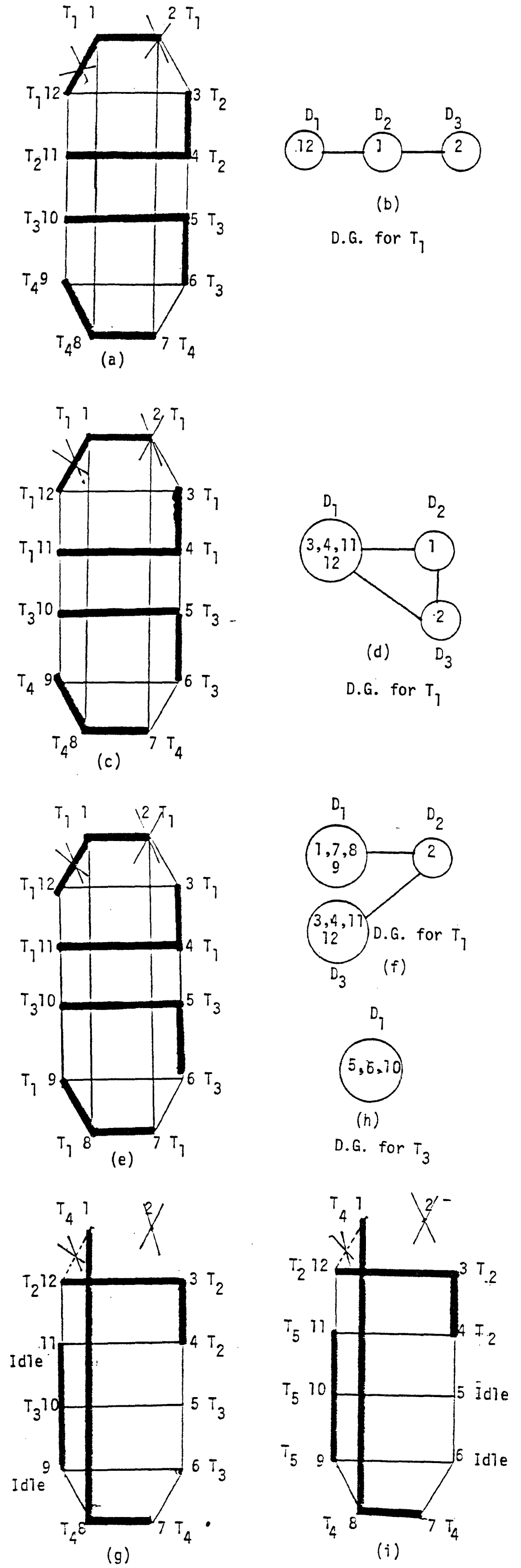


Figure 4

THE RESILIENCY TRIPLE IN MULTIPROCESSOR SYSTEMS

Miroslaw Malek and Kitty H. Yau

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712

Abstract: A multiprocessor system is represented by an architecture graph G, where the nodes correspond to processors (or computers) and the edges represent communication links among them. A job executed on a system G is represented by a computation graph H, which is a subgraph of G, where the nodes correspond to one or more tasks assigned to a particular processor and the edges represent communications among tasks that are allocated on different processors. In this paper we define three important parameters, multiplicity, robustness, and configurability, called the resiliency triple, pertinent to the fault tolerance in multiprocessor systems. We will discuss how each parameter is related to fault tolerance and fault recovery and how it is determined for a given G and H. We present solutions for H being a path and G being either a hypercube or a mesh.

Key Words: Fault tolerance, multiplicity, robustness, configurability, multiprocessor systems, hypercube, mesh.

1. INTRODUCTION

The proliferation of ever more powerful and complex multiprocessor systems has made fault tolerance a necessity in today's computer design. Although a large amount of related research work has been reported in the literature, and considerable efforts are still being made by many researchers to perfect the art of multiprocessor fault tolerance, there is very little analytical work done in the area of fault recovery. The existing research on fault recovery is rather fragmented and application specific. Moreover, techniques requiring imposed component redundancy have been widely proposed, while the intrinsic redundancy associated with multiprocessor systems has been overlooked. Since mapping application programs precisely onto a system architecture is very difficult, a multiprocessor system is often not fully utilized at all times. This means that some processors in the system are often left idle at one time or another. This inherent component redundancy should enable fault recovery to be achieved by mapping programs around faulty processors. Recently, Harary and Malek have developed a graph theoretic framework for fault recovery in multiprocessor systems [1]. In their work, existing graph theoretic models for system architecture and program structure are referred to as the architecture graph (G) and the computation graph (H) respectively, and are used to formalize the studies of fault recovery. Several parameters that affect the effectiveness of a fault recovery technique in various ways are introduced to allow easier comparison of different methodologies and to quantify the optimization of fault recovery. Also introduced in their work is a set of three parameters called the resiliency triple. These include the multiplicity, the robustness, and the configurability, collectively denoted by (m, r, c). These parameters play an important role in the better utilization of a multiprocessor system, its resiliency to faults, and its suitability for various fault recovery strategies. This paper presents methods developed to determine these parameters for an important computation graph, the path (pipeline), on two well known architecture graphs: the hypercube and the mesh. Without loss of generality, we shall assume an *sxs* square mesh and denote it by M_s, where s is the number of nodes on each side. We will also use Q_n to denote an n-dimensional hypercube and P_k to denote a path that consists of k nodes.

In the next section, the fault recovery model is briefly described. The resiliency triple is defined in Section 3, and its impact on task allocation and fault recovery is discussed. Section 4 presents the methods for determining each parameter in the resiliency triple for a pipeline computation structure (H) on two important classes of multiprocessor systems: the hypercube and the mesh (G). Section 5 gives the conclusions.

2. THE FAULT RECOVERY MODEL

In order to show how the parameters in the resiliency triple are related to fault recovery, we would like to briefly introduce the fault recovery model proposed by Harary and Malek [1].

In general, fault recovery models can be used in system synthesis or analysis. The synthesis involves the construction of an appropriate architecture with redundant components in order to meet a set of required conditions. An excellent example of the synthesis for fault recovery of cycles and binary trees can be found in [2]. In the analysis, a prescribed architecture graph such as a mesh or a hypercube is given and all fault recovery measures must be taken within this framework. In order to take advantage of the inherent component redundancy offered by a multiprocessor system, we decided to concentrate on the latter, which requires no imposed hardware redundancy. The fault recovery model is described as follows.

Let an architecture graph G represent the physical architecture of a multiprocessor system. Nodes in this graph represent processors (or computers) and interface communication modules while edges indicate the actual point-to-point communication links. Each node in this graph can be extended to include memory, input/output channels, and other devices. Fig. 1 shows the architecture graph of an 8-processor hypercube. Let a computation graph H represent an actual computation (job) where each node corresponds to a task and each edge indicates the inter-task communications. The dark line in Fig. 1a shows a computation graph of a 4-node path mapped onto an architecture graph of an 8-processor hypercube. Since the computation graph has to be mapped onto the architecture graph, H is a subgraph of G. A faulty link leads to the removal of an edge from G and a faulty processor results in the removal of a node and the incident edges. When one or both of these cases occur, there are two possibilities: Either the resulting graph G' contains another subgraph H' that is isomorphic to H or it does not. If it does not, then the system G is called non-recoverable with respect to H and the particular fault(s). On the other hand, when G' does contain a subgraph

This work was supported in part by DARPA under Grant No. N00039-86-C-0167 and by ONR under Grant No. N0014-86-K-0554 under the SDIO/IST Program.

H' isomorphic to H, and there are two or more such subgraphs, then the one yielding the minimum cost (such as some function of distance, time, or other parameters introduced in [1]) will result in the most efficient fault recovery. Fig. 1 shows the recovery of a job on a hypercube system.

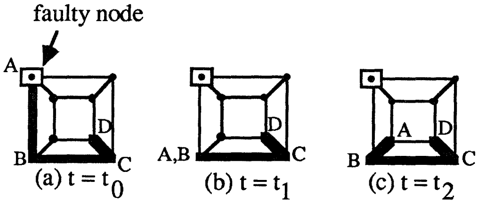


Fig. 1. The recovery of P_4 on a hypercube

We observe that in general case a computation graph H is a digraph representing a task graph and one-to-one mapping of H onto G may not be possible. In such cases we resort to a concept of dilation which allows mapping of nodes that are adjacent in H onto G in such a way that the distance between nodes in G corresponding to adjacent nodes in H is equal to or longer than that in H. The general recoverability problem is NP-complete. In this paper we restrict G to be a mesh or a hypercube and H to be a path (pipeline).

3. THE RESILIENCY TRIPLE

The resiliency triple (m, r, c) consists of the following three parameters: multiplicity (m), robustness (r), and configurability (c).

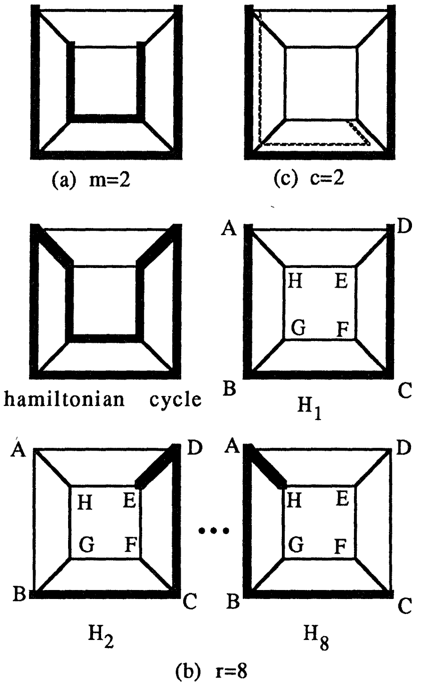


Fig. 2. The multiplicity, robustness, and configurability for P_4 on Q_3.

Multiplicity (Fig. 2a) is the maximum number of node-disjoint embeddings of H onto G, denoted by *m(G, H)*. In graph theory [3], this is known as the node-disjoint packing number $pac_0(G, H)$ as introduced in [4]. When m=2, two identical jobs can be run simultaneously on the system (with some additional hardware) to allow single fault detection. When m>2, any single fault can be masked by voting the outputs of multiple copies of the same job. In other words, it allows N Modular Redundancy (NMR) in space. Usually, we are only interested in knowing whether *m* is equal to or greater than a chosen number in the range of 2 to 9. Multiplicity is also an indication of a system's fault tolerance. Given the necessary hardware, a system can be (m-1)-fault-tolerant. Higher multiplicity also allows more homogeneous jobs to be run on the system simultaneously to achieve better system utilization. Futhermore, testing can be performed by comparing results of the same job executed on different subsets of processors.

Robustness, denoted by *r(G, H),* is the number of embeddings of a graph H onto a labeled graph G such that each node of H is at a different label of G in each embedding (Fig. 2b). When r>1, fault recovery can be achieved through time redundancy by executing each stage of the computation (systolic array or pipeline) on two or more different processors at a time [5]. This corresponds to duplex or NMR in time. Again, we are usually concerned about whether *r* is equal to or bigger than a chosen number within the range of 2 to 9. Since multiplicity and robustness correspond to redundancy in space and time, they are also useful in system diagnosis.

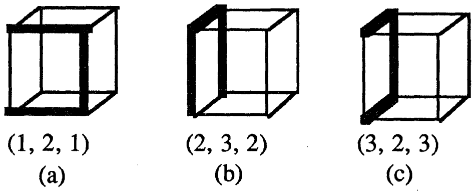
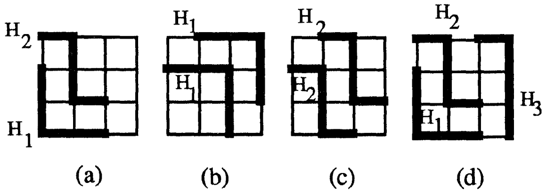


Fig. 3. Equivalent configurations (fixed-labeling).

Configurability (Fig. 2c) is the number of ways in which a particular job H can be configured on a system G. If H is a proper subgraph of G, there may be many ways to map H onto G. Each particular mapping, represented by a graph H_c (c is a positive integer), is called a configuration. Since all configurations of H on G are isomorphic, the computation at hand can be performed using any of them. However, some of these isomorphic graphs are equivalent. Although isomorphism among a collection of configurations is itself an equivalence relation, we have, for reasons that will become obvious later, defined equivalence in a stricter sense. If all configurations are considered as "rigid" graphs, then two isomorphic configurations may not possess the same properties such as dimensionality and space occupancy. Two configurations H_1 and H_2 are equivalent if, after some necessary rotation and/or translation, H_2 either coincides with, or becomes a mirror image of H_1. Fig. 3 shows some equivalent configurations of a 4-node path P_4 on an 8-node hypercube Q_3. The number of non-equivalent configurations of H on G is defined as the *configurability* of H on G and is denoted by *c(G, H).* Notice that each set of equivalent configurations is counted as one in deriving the configurability of a given computation graph on an architecture graph. The parameter, configurability, is a measure of several aspects of a multiprocessor system. Higher configurability generally allows a better system utilization, a greater resiliency to faults, and a higher efficiency in fault recovery. The following examples demonstrate the importance of this parameter in a fault-tolerant multiprocessor system.

Example 1:

Consider the computation graph H to be a 5-node path P_5 and the architecture graph G to be a 16-node mesh M_4. Fig. 4a shows two nonequivalent configurations (H_1 and H_2) of H on G. If these are the only configurations available, then at most two copies of H can be run simultaneously on the system G, as shown in Figs. 4a, 4b, and 4c. However, if we include the configuration H_3 shown in Fig. 4d, then three copies of H can be mapped onto G as depicted. This means that either more jobs can be scheduled on the system to achieve better utilization or more copies of the same job can be run concurrently to obtain a higher degree of fault tolerance. In the above example, configurations shown in Figs. 4a, 4b, and 4c allow two copies of the same job to be run simultaneously so that any single fault can be detected. However, the situation shown in Fig. 4d allows three copies of the job to be executed simultaneously and therefore, any single fault can be masked. Observably, higher configurability results in higher multiplicity which allows the system to be more efficiently utilized.



(a) (b) (c) (d)

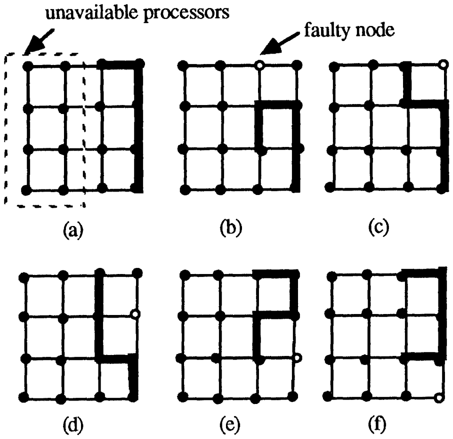
Fig. 4. Mappings of P_5 on M_4.

Example 2:

Consider the same computation and architecture graphs used in Example 1. If the two columns of processors on the left half of the system G are unavailable and the job H must be scheduled on the remaining processors, then H may be mapped on G as shown in Fig. 5a. Since the left half of the system is busy, fault recovery must be accomplished by reconfiguring the job around faulty components while using only the right half of the system. Fortunately, due to the existence of various configurations, the job H can be reconfigured to bypass any single faulty node. Figs. 5b through 5f show the possible configurations when the faulty nodes are as indicated. Clearly, higher configurability signifies a bigger chance of successful job reconfiguration and is thus an indication of the system's greater resiliency to faults.

Example 3:

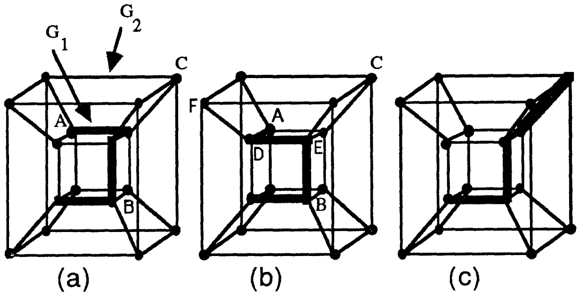
Consider the same computation graph H (in previous examples) embedded onto the architecture graphs G_1 and G_2, which are a 3- and a 4-dimensional hypercube, respectively. The configurability of H on G_1 is two ($c(G_1, H) = 2$) as shown in Figs. 6a and 6b, and that of H on G_2 is three ($c(G_2,H) = 3$) as shown in Figs. 6a, 6b, and 6c. In Fig. 6a, if node A is faulty on G_1, then the task executed there can be transferred to node B. All other nodes can remain stationary for the job to continue. Any other reconfiguration will result in the disturbance of more nodes. However, due to the existence of an additional configuration of H on G_2, if this node is faulty on G_2, then the task assigned to that node can be transferred to either node B or node C, depending on which node is available at the time. This example shows that higher configurability also indicates a more flexible system for reconfiguration.



(a) (b) (c)

(d) (e) (f)

Fig. 5. Reconfiguration of P_5 on M_4 with different faulty nodes.



(a) (b) (c)

Fig. 6. Non-equivalent configurations of P_5 on Q_3 and Q_4.

Example 4:

Consider the same computation graph H and architecture graphs G_1 and G_2 used in Example 3. If node A in Fig. 6b becomes faulty on G_1, then not only does node A have to be transferred (to node B), but node D has to be moved (to node E) also in order to maintain the 5-node pipeline. However, if this node becomes faulty on G_2, only that node has to be moved (to node F) to complete the reconfiguration. Therefore, higher configurability may also increase the efficiency of fault recovery.

Since each parameter in the resiliency triple has some impact on the fault tolerance of a multiprocessor system, the study of these parameters is not merely of theoretical interest, but it is also useful in solving practical problems. Next, we present the methods developed for determining each of these parameters for a computation graph H on an architecture graph G.

4. DETERMINING THE RESILIENCY TRIPLE

The resiliency triple is dependent on both the computation and the architecture graphs. Since the pipeline is a very widely used computation structure, we have decided to start with the pipeline (path) as the computation graph under consideration. The binary n-cube (hypercube) and the mesh have both received much research and commercial attention and are useful for a wide range of problems. We therefore choose these two systems as the architecture graphs.

4.1. Multiplicity

A systematic way to map multiple node-disjoint copies of a path P_k on a mesh M_s or a hypercube Q_n is to concatenate as many P_k's as possible along a hamiltonian path. If a path P_6 is mapped onto a mesh M_5 in such a manner, four node-disjoint copies will result. The multiplicity is, therefore, equal to four. Since a hamiltonian path exists in a hypercube or a mesh of any size, the multiplicity m(G, H), where H represents a path computation graph P_k and G represents either a mesh M_s or a hypercube Q_n system, is given by the following expression.

$$\mathbf{m(G, H)} = \lfloor N/k \rfloor$$

Notice that $\lfloor x \rfloor$ is the largest integer smaller than or equal to x, N is the number of processors in the system, and k is the number of tasks in the computation (pipeline). $N=2^n$ for Q_n and $N=s^2$ for M_s. For a P_6 on an M_5, N=25 and k=6. Therefore, m= $\lfloor 25/6 \rfloor$ = 4. Furthermore, we may easily generalize and observe that for any architecture graph G with N nodes that has a hamiltonian path, the multiplicity is given by $\mathbf{m(G, P_k)} = \lfloor N/k \rfloor$.

4.2. Robustness

The maximum number of mappings of a path P_k on a mesh M_s or a hypercube Q_n, such that each node in a P_k is assigned to a different node in an M_s or Q_n, can be obtained in the following manner. Starting with a mapping H_1 ($H_1= P_k$), the next mapping H_2 can be obtained by moving each node in H_1 to an adjacent node in the same direction along the hamiltonian cycle. The different mappings, H_i's ($1 \le i \le r$, where r is the robustness) are obtained by sliding P_k along a hamiltonian cycle in M_s or Q_n, one node at a time, until we return to the original mapping, H_1. Fig. 2b shows such mappings of P_4 on Q_3. We observe that robustness is equal to the number of nodes N in any system graph G if a hamiltonian cycle exists. Since a hamiltonian cycle exists in all hypercubes and all meshes with an even number of nodes, the robustness is given by **r(G, H) = N** for a P_k on a Q_n or an M_s where s is even. However, if there are an odd number of nodes in a mesh (s is odd), then a hamiltonian cycle does not exist. In this case, we can choose one of the following alternatives which are easy to implement.

(1) Find the largest cycle in M_s and use it to generate the mappings as described above.

(2) Slide P_k along a hamiltonian path, one node at a time, from one end to another.

If we choose option (2), then robustness will be given by **r(G, H) = N-k+1.** Clearly, if we can find a cycle in M_s such that N'>N-k+1, where N' is the number of nodes in this cycle, then option (1) will give a better result.

We may observe that the largest cycle in M_s with N nodes, where s is odd, contains N-1 nodes. Fig. 7 shows how such a cycle is constructed on an arbitrary M_s (s is odd). Consequently, using option (1), the robustness is given by **r(G, H) = N-1** for P_k on M_s when s is odd. This is optimum (for k > 2) since a hamiltonian cycle does not exist.

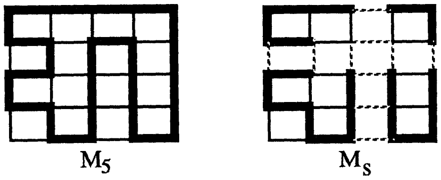


Fig. 7. The largest cycle on M_s when s is odd.

4.3. Configurability

4.3.1. Configurability of a Path on a Hypercube

In order to determine configurability, we need to generate various non-equivalent configurations. Before presenting an algorithm to accomplish this, we shall describe a scheme which is suitable to represent a configuration of a path P_k on a hypercube Q_n. Since P_k has k-1 edges, a logical way to represent the path is by a vector of k-1 positive integers. Each integer indicates the dimension in which the corresponding edge resides. Since non-equivalent configurations are not distinguished by their positions or orientations in the system, and the hypercube is a symmetric graph, we need not adopt a fixed coordinate system. Furthermore, it is more advantageous not to assign a fixed integer to each dimension. This can be demonstrated by the following example.

Example 5:

In a Q_3 system, if the dimensions are labeled such that the horizontal dimension (x-coordinate) is denoted by 1, the vertical dimension (y-coordinate) is denoted by 2, and the remaining dimension (z-coordinate) is denoted by 3, then the configuration in Fig. 3a will be represented by the vector (1, 2, 1), and those shown in Figs. 3b and 3c will be represented by the vectors (2, 3, 2) and (3, 2, 3) respectively (for easier reference, we have chosen the lower left node as the starting point of the path). Obviously, all three configurations are equivalent and should be counted as one. But many such equivalent configurations will be generated as different vectors if each dimension is assigned a fixed integer. However, if we label every dimension dynamically, according to the order in which they are traversed by the path, then all the above three configurations will be represented by the vector (1, 2, 1). Fig. 8 shows all the non-equivalent configurations of P_6 on Q_3 using this representation. Since there are four non-equivalent configurations, the configurability of P_6 on Q_3, $c(Q_3, P_6)$, is equal to four.

The configurability of H on G (P_k on Q_n here) can be obtained by enumerating all the non-equivalent configurations of H on G. But since the distinct configurations themselves are also very useful for task allocation and fault recovery, we want to generate and save the vectors which represent them. When there are many non-equivalent configurations of H on G, we may decide to save only a chosen number, say *x*, of them in order to save time and memory space. In this case, we are only interested in knowing the exact number when configurability is smaller than *x*.

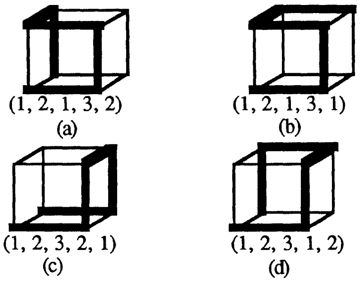
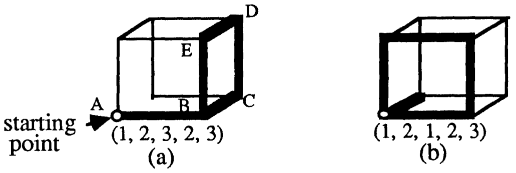

(1, 2, 1, 3, 2)
(a)
(1, 2, 1, 3, 1)
(b)
(1, 2, 3, 2, 1)
(c)
(1, 2, 3, 1, 2)
(d)

Fig. 8. Path representation using dynamic labeling.

Before presenting the algorithm for generating non-equivalent configurations of P_k on Q_n, let us discuss some related issues. Firstly, we observe that when k=2 or k=3, there is only one configuration, represented by the vectors (1) and (1, 2) respectively. As a result, to find a vector that represents a configuration of P_k (k-1 edges) on Q_n, where k>3, we only need to generate k-3 integers to be appended to the vector (1,2). Secondly, for any path mapped on a hypercube, no two adjacent edges can lie in the same dimension. This means that in the vector representing a configuration of P_k on Q_n, adjacent integers cannot be equal. Consequently, given an edge represented by an integer i, the next edge in the path can only be represented by an integer j such that $1 \leq j \leq n$ and $j \neq i$. In other words, we can only choose from n-1 integers to represent this edge. Having observed this, it is clear that we may generate up to $(n-1)^{k-3}$ vectors to be candidates for the configurations of P_k on Q_n. Many of these vectors represent configurations that contain cycles and must thus be eliminated. Other vectors may represent equivalent configurations and must thus be counted as one. Fig. 9 shows two configurations with cycles. Observably, these configurations are also equivalent. If we have to test each of the $(n-1)^{k-3}$ vectors for cycles and equivalence, the $O((\log N)^{k-3})$ computing time may be excessive ($N=2^n$ is the number of processors in Q_n). For a typical case of n=10 and k=10, the number of operations becomes 9^7=4,782,969. However, if we take another approach by building the configurations of P_k from those of P_{k-1} on Q_n, then only $2(n-1)c(Q_n, P_{k-1})$ vectors will be generated. This is because we can build a path of k nodes by appending a node either to the front or to the end of a (k-1)-node path. Knowing that the configuration of P_3 is (1, 2), we can extend the path one edge at a time until we reach P_k. As a result, only $\sum_{i=4}^{k} 2(n-1)c(Q_n, P_{i-1})$ vectors need to be generated to obtain all the configurations of P_k on Q_n. If we put an upper bound, x, on the $c(Q_n, P_{i-1})$'s, O(klogN) computation time is required for the whole operation. The latter approach is also more efficient in terms of memory requirement. O(k) memory space is required instead of $O((\log N)^{k-3})$, which is necessary for the former approach.



(1, 2, 3, 2, 3)
(a)
(1, 2, 1, 2, 3)
(b)

Fig. 9. Configurations with cycles on a Q_3.

After generating the configuration vectors mentioned above, we need to perform the eligibility test. This consists of testing for the existence of cycles and equivalent configurations. In order to detect cycles, let us observe that any cycle on Q_n is represented by a vector in which every integer appears for an even number of times. Any vector that corresponds to a configuration which contains a cycle must have a subvector that exhibits the above characteristic. Let us scan a vector from the left to the right and keep an n-bit binary number as a parity indicator, in which the i-th bit (b_i) indicates whether the integer i has appeared for an even number of times. If it has, b_i is set to 0; otherwise b_i is set to 1. For example, if we consider the vector (1, 2, 1, 2, 3) in Fig. 9b, the 3-bit parity indicator ($P=b_1b_2b_3$) will be updated as follows when we scan the vector from the left to the right one bit at a time:

step 1:	100	(1 appeared once)
step 2:	110	(2 appeared once)
step 3:	010	(1 appeared twice)
step 4:	000	(2 appeared twice)

After scanning the fourth integer in the vector, the parity indicator becomes zero, indicating the detection of a cycle. Notice that if the first edge in the configuration is part of a cycle, as shown in Fig. 9b, then the parity indicator would become zero as soon as a cycle is detected, even though there is still another edge attached to the cycle. In this case, there is no need to scan the rest of the vector. However, if the first edge is not part of the cycle, as shown in Fig. 9a, then the parity indicator would not go to zero if the vector is scanned as a whole. In this case, the cycle will be detected when the subvector (2, 3, 2, 3) is scanned. Thus, cycle detection requires scanning the configuration vector and its subvectors while updating and checking the parity indicator. This requires $O(k^2)$ computations for P_k (O(k) if all the subvectors are checked in parallel).

The way to test for equivalence is by observing that any configuration can be traced from either end. Consider the configuration in Fig. 9a. If the configuration is traced in the order ABCDEB, , then we get the vector (1, 2, 3, 2, 3). But if we traverse in the opposite direction starting with node B, then the vector obtained would be (1, 2, 1, 2, 3). How do we derive this vector from (1, 2, 3, 2, 3) and thus detect the equivalence? The answer is by inverting the vector (i.e. listing a given vector by starting with the last element) and renumbering the resulting vector as follows.

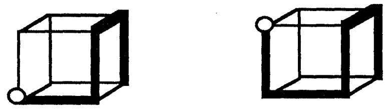
$$H_1 = (1, 2, 3, 2, 3)$$
$$H_1' = \text{INVERT}\ (H_1) = (3, 2, 3, 2, 1)$$
$$H_2 = \text{RENUMBER}(H_1') = (1, 2, 1, 2, 3)$$
$$\therefore H_1 \equiv H_2$$

Observably, the operation RENUMBER performs the following mapping:

$$3 \rightarrow 1, \quad 2 \rightarrow 2, \quad 1 \rightarrow 3.$$

However, it may perform a different mapping in a different situation. Its job is to scan the current vector and relabel each integer according to the order in which it appears in the vector. In H_1', the integer 3 is the first label to appear in the vector, and is thus given a new label 1. Similarly, the integer 2 is the second label and 1 the third appearing in H_1', they are therefore reassigned the new labels 2 and 3 respectively. Since we have not assigned a fixed number to any particular dimension in Q_n, inverting and then renumbering a vector would not produce a new configuration, and is equivalent to tracing the path from the opposite end. Renumbering is also necessary when a configuration vector for P_k is generated by appending an interger to the front of a vector of P_{k-1}. For example, if we want to obtain a configuration for P_6 by appending an edge to the front of the P_5 shown in Fig. 10a, we can append an integer (either 2 or 3 in this case) to the front of the vector that represents the P_5. If we choose to append a 2, the following vector is obtained: $H'(P_6) = (2, 1, 2, 3, 2)$. The corresponding configuration is shown in Fig. 10b. $H'(P_6)$ needs to be renumberred as follows:

$$2 \rightarrow 1, \quad 1 \rightarrow 2, \quad 3 \rightarrow 3,$$
$$H(P_6) = \text{RENUMBER}(\ H'(P_6)) = (1, 2, 1, 3, 1).$$



$H(P_5) = (1, 2, 3, 2)$ $H(P_6) = (1, 2, 1, 3, 1)$

(a) (b)

Fig. 10. Extension of a 5-node path.

Having discussed the various issues involved in generating the configurations of P_k on Q_n, we are now ready to present the algorithm which enumerates up to x non-equivalent configurations of P_k on Q_n. The final configuration vectors are stored in an x by (k-1) array, **H**(1:x, 1:k-1), which contains up to x vectors of k-1 integers, each of which represents a unique configuration. Then **H**(*m*, 1:k-1) would correspond to the (k-1)-integer vector representing the *m*-th configuration enumerated. An array **T**(1:x, 1:k-1) is used to save the intermediary vectors (for P_{i-1}'s). The algorithm is as follows:

```
Algorithm 1:

Input: n, k, x.
1. If k=2, exit with H(1)=(1), c=1.
2. If k=3, exit with H(1)=(1, 2), c=1.
3. If k>3, set H(1,1)=1 and H(1,2)=2: h=2.
   \ h keeps track of the largest integer in the vector and h≤n \

4. Until H contains vectors of k-1 elements, set T = H, erase
   H, and do the following:
A. For every vector v in T, do the following:
   a. If h ≠ n, set h=h+1.
   b. For every integer i such that  1≤i≤h  and i ≠ j, do the
        following:            \ j is the last integer in v \
   i. Append i to the end of v.
   ii. Perform cycle detection. If positive, go to 4b.
   iii. Invert and renumber the vector; check if the resulting
        vector has been saved in H. If positive, go to 4b.
   iv. Save the resulting vector in H. If |H| = x, go to 4;
        otherwise go to 4b.
           \ |H| is the number of vectors in H \
   c. For every integer i such that  2≤i≤h, do the following:
   i. Append i to the front of v and renumber the vector.
   ii. Check if vector exists in H. If positive, go to 4c.
   iii. Perform cycle detection. If positive, go to 4c.
   iv. Invert and renumber the vector; check if it exists in H.
        If positive, go to 4c.
   v. Save the resulting vector in H. If |H| = x, go to 4;
        otherwise go to 4c.
5. If |H| < x, set c= |H| and output("c is equal to", c);
   otherwise, output("c is at least", x).
```

In the above algorithm, Step 4 is executed k-3 times. For each iteration of Step 4, Step 4A is invoked at most x (a constant) times, each of which causes Steps 4b and 4c to be performed h times. Steps 4b and 4c perform cycle detection and vector renumbering, which require $O(k^2)$ computing time. Consequently, the total time requirement for Algorithm 1 is $O(k^3h)$. By observing that h is O(k) if $k \le n+1$ and is O(n) if $k \ge n+1$, we conclude that the computation time is either $O(k^4)$ or $O(k^3 \log N)$. If we assume an upper bound on the size of the pipeline so that $k \le K$, where K is a chosen constant, then the computation can be accomplished either in a constant (O(1)) time or O(logN) time, depending on the values of k and n. Since two x by (k-1) arrays are used to store the final and the intermediary results, the memory requirement for Algorithm 1 is O(k). Again, for $k \le K$, this means O(1) storage space.

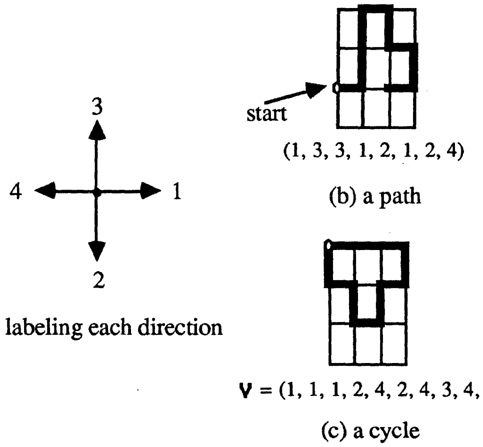
4.3.2. Configurability of a Path on a Mesh

Before discussing the method of representation for a path P_k on a square mesh system (M_s), it is helpful to observe the following differences between a square mesh and a hypercube (Q_n):

1. The Q_n is a regular graph in which every node has the same degree, n. But the M_s has four corner nodes, which are of degree 2, and 4s - 8 boundary nodes, which are of degree 3. The remaining nodes all have a degree 4.

2. A configuration of P_k in any particular position on Q_n can be rotated n-1 times before returning to its original orientation. A configuration of P_k on M_s has only three rotations besides itself. It also has two mirror images, one along the x-axis (horizontal) and the other along the y-axis (vertical).

We have decided to ignore the boundary cases on M_s in order to simplify the discussion. This requires that $k \leq s$, which is usually satisfied. However, if $k > s$, then the number of edges traversed in each direction must be counted so as not to exceed s. Since each node under consideration has a degree of 4 regardless of s, and given an edge in a path, the next edge to be traversed can only be oriented in one of three directions (two adjacent edges cannot be traversed in opposite directions on a mesh), we have chosen to assign a fixed integer to each of the four directions, as shown in Fig. 11a. Then a configuration of P_k on M_s can be represented by a (k-1)-element vector as shown in Fig. 11b. We may require that the first integer be 1 (first edge always heads to the right) for easier reference. When k=2, there is only one configuration, which is represented by (1). When k>2, we need to find an additional k-2 integers to complete the (k-1)-edge path. Because each integer can assume one of three values as mentioned earlier, up to 3^{k-2} vectors may be generated. These include many cycle-bearing or equivalent configurations. For a typical case of k=10, $3^8=6561$ operations are required. Although this may be acceptable, we can improve the time efficiency by using a method similar to the one described in Section 4.1. A configuration for P_k can be obtained from that of P_2 by appending an edge to either end of the latter, step by step until k-1 edges are accumulated. To obtain P_i from P_{i-1}, $6c(M_s, P_{i-1})$ vectors are generated. Thus, $\sum_{i=3}^{k} 6c(M_s, P_{i-1})$ operations are required for the complete process. If we again put an upper bound on the number of non-equivalent configurations for each P_i, where $2 < i \leq k$, then O(k) computing time is needed (O(1) if $k \leq K$).



(a) labeling each direction

(b) a path

(c) a cycle

Fig. 11. Representation of a path and a cycle on a mesh.

The eligibility test for configurations of P_k on M_s also consists of cycle detection and equivalence test. However, the particular methods to accomplish these are different from those presented in Section 4.1. The following theorem can be used for cycle detection.

Theorem 1: Given the labeling scheme in Fig. 11a and a vector v representing a configuration on a mesh M_s, if we let *sum* denote the sum of all the integers in v and *length* denote the number of integers in v, then the configuration is a cycle *iff* ***sum* = 2.5 x *length*.**

Proof: If a cycle on a mesh is traversed starting from an arbitrary node, each edge in the cycle would belong to a pair of edges pointing in opposite directions. A cycle of *length* edges consists of *length*/2 such pairs. Since in the introduced labeling scheme (Fig. 11a) each direction is numbered in such a way that integers representing opposite directions add to 5, each pair of these edges are denoted by integers adding to 5. Since there are *length*/2 pairs, the sum of all the integers, each representing an edge in the cycle, is 5x*length*/2, or **2.5x*length*.** □

Fig. 11c shows an example in which v=(1,1,1,2,4,2,4,3,4,3). From this we get *sum* =1+1+1+2+4+2+4+3+4+3=25, *length*=10. Applying Theorem 1, a cycle is detected. A vector corresponding to a cycle-bearing configuration would have a subvector that demonstrates the above characteristic.

We have made the following observations on equivalent configurations of P_k on M_s: Consider the example shown in Fig. 12a. If the path is traced starting from node A, then the following vector is obtained: H_1 = (1, 2, 2, 4, 4, 3, 1). But if node B is the starting point, then we would get H_1' = (4, 2, 1, 1, 3, 3, 4). We know that H_1 and H_1' are equivalent, but how do we detect the equivalence? Since we have chosen 1 to be the first integer in all vectors, H_1' needs to be renumbered. In order to convert 4 to 1, we realize that the edges heading left must be forced to head right. Since mirror images are equivalent, we can convert H_1' to its mirror image along the y-axis, causing the horizontal edges to exchange directions. As a result, the integers 1 and 4 are interchanged, giving the vector $H_1'' \equiv H_2$=(1, 2, 4, 4, 3, 3, 1). Fig. 12b shows the corresponding configuration. Similarly, the mirror image of a configuration along the x-axis causes the vertical edges to exchange directions, resulting in the interchanging of 2 and 3 in the corresponding vector. Fig 12c shows such a mirror image (of the path in Fig.12a). Fig. 12d shows the path in 12a reflected twice, once along the x-axis and once along the y-axis. The corresponding vector is obtained by interchanging 1 and 4 as well as 2 and 3. This is equivalent to subtracting each integer in the original vector from 5. Clearly, interchanging 1 and 4 or 2 and 3 in a vector does not result in a new (non-equivalent) configuration. Finally, let us observe the configurations in Figs. 12e and 12f. These are both 90° rotations of Fig. 12a, one clockwise and the other counterclockwise. When a configuration is rotated 90° clockwise, integers in the original vector must be renumbered according to the following:

$$1 \rightarrow 2, \quad 2 \rightarrow 4, \quad 3 \rightarrow 1, \quad 4 \rightarrow 3 \qquad (2)$$

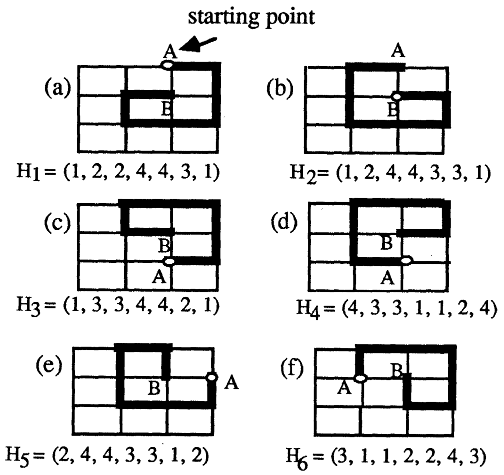
When a configuration is rotated 90° counterclockwise, the vector must be renumbered as follows:

$$1 \rightarrow 3, \quad 2 \rightarrow 1, \quad 3 \rightarrow 4, \quad 4 \rightarrow 2 \qquad (3)$$

Thus, renumbering a vector according to (2) or (3) would not alter the configuration (all resulting configurations are equivalent). After observing the above, it is readily seen that whenever we have a vector whose first element (i) is not 1, the vector can be renumbered (to begin with 1) as follows:

1. If i=2, then reassign integers according to (3).
2. If i=3, then reassign integers according to (2).
3. If i=4, then interchange 1 and 4.

As mentioned earlier, two adjacent edges cannot be in opposite directions. Therefore, when appending an edge to the front of an existing path (represented by a vector that begins with 1), only three choices are available. The edge may be represented by one of the following three integers: 1, 2, or 3. Whenever a vector is inverted or extended at the front, renumbering may be required. A configuration (H_1 in Fig. 12a) and its mirror image (H_3 in Fig. 12c) along the x-axis both have vectors that begin with 1. It is therefore necessary to check for these equivalent configurations.



$H_1 = (1, 2, 2, 4, 4, 3, 1)$ $H_2 = (1, 2, 4, 4, 3, 3, 1)$

$H_3 = (1, 3, 3, 4, 4, 2, 1)$ $H_4 = (4, 3, 3, 1, 1, 2, 4)$

$H_5 = (2, 4, 4, 3, 3, 1, 2)$ $H_6 = (3, 1, 1, 2, 2, 4, 3)$

Fig. 12. Mirror images and rotations of a path on a mesh.

Now we are ready to present the algorithm for enumerating up to x (a chosen constant) non-equivalent configurations of P_k on M_s. As in Algorithm 1, two arrays, **H**(1:x, 1:k-1) and **T**(1:x, 1:k-1), are used to store the final and the intermediary vectors, respectively. The algorithm is as follows.

Algorithm 2:

```
Input: s, k, x.
1. If k=2, exit with H(1)=1, c=1.
2. If k>2, set H(1, 1)=1.
3. Until H contains vectors of k-1 elements, set T=H, erase
   H, and do the following:
A. For every vector v in T, do the following:
   a. For every integer i such that 1≤i≤4 and i+j≠5, do the
      following:              \j is the last integer in v\
    i.   Append i to the end of v.
    ii.  Perform cycle detection. If positive, go to 3a.
    iii. Invert and renumber the vector; check if it exists in H.
         If positive, go to 3a.
    iv.  Interchange 2 and 3 in the vector; check if the resulting
         vector exists in H. If positive, go to 3a.
    v.   Invert and renumber the vector; check if it exists in H.
         If positive, go to 3a.
    vi.  Save the vector in H. If |H| = x, go to 3; otherwise, go
         to 3a.
   b. For every integer i such that 1≤i≤3, do the following:
    i.   Append i to the front of v and renumber the resulting
         vector. Check if it exists in H. If positive, go to 3b.
    ii.  Perform cycle detection. If positive, go to 3b.
    iii. Invert and renumber the vector; check if it exists in H.
         If positive, go to 3b.
    iv.  Interchange 2 and 3 in the vector; check if the resulting
         vector exists in H. If positive, go to 3b.
    v.   Invert and renumber the vector; check if it exists in H.
         If positive, go to 3b.
    vi.  Save the vector in H. If |H| = x, go to 3; otherwise, go
         to 3b.
4. If |H| < x, set c = |H|, output("c is equal to", c).
   Otherwise, output("c is at least", x).
```

In Algorithm 2, Step 3 is repeated k-2 times. Each iteration of Step 3 causes Step 3A to be executed up to x times, each of which in turn performs Steps 3a and 3b three times. Steps 3a and 3b each requires $O(k^2)$ computing time. As a result, the total time requirement for Algorithm 2 is $O(k^3)$. If $k \le K$, where K is a constant upper bound on k, this reduces to O(1). Like Algorithm 1, the memory requirement is O(k), or O(1) if k is bounded.

5. CONCLUSIONS

We have defined three parameters (the resiliency triple) and discussed their importance in the fault tolerance and diagnosis of multiprocessor systems. We have also presented the solutions for determining the first two parameters, multiplicity and robustness, and two algorithms which determine the configurability and enumerate various non-equivalent configurations for a path computation graph mapped onto a hypercube or a mesh architecture graph. The resiliency triple is a good measure of a multiprocessor system's resiliency to faults and its flexibility for job reconfiguration. It is also related to the system utilization and fault recovery. The configurations generated by the algorithms proposed in Section 4.3 are useful for efficient task allocation as well as effective fault recovery. The efficiencies of these algorithms have been improved by carefully choosing a path representation scheme in each case, and by selecting an effective approach to generating the configurations. For bounded k, Algorithm 1 has O(logN) computing time, and requires O(1) storage space, and for Algorithm 2, both time and memory requirements are constant.

REFERENCES

[1] F. Harary and M. Malek, "Fault recovery in multiprocessor systems: a graph theoretic approach," Technical Report, Department of Electrical and Computer Engineering, the University of Texas at Austin, 1987.

[2] R. M. Yanney and J, P. Hayes, "Distributed recovery in fault-tolerant multiprocessor networks," IEEE Trans. on Computers, vol. C-35, no. 10, Oct. 1986, pp. 871-879.

[3] F. Harary, "Graph Theory," Reading, Mass., Addison-Wesley, 1969.

[4] F, Harary, "Covering and packing in graphs," I. Annals N. Y. Acad. Sci. 175, pp. 195-208, 1970.

[5] Y. H. Choi and M. Malek, "A fault-tolerant FFT processor," IEEE Trans. on Computers, vol. 37, no. 5, May 1988, pp. 617-621.

FAULT-TOLERANT ALGORITHMS AND ARCHITECTURES FOR REAL TIME SIGNAL PROCESSING

Jing-Yang Jou
*Jacob A. Abraham***

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Abstract— An encoding technique, the weighted checksum code (WCC), is proposed to achieve concurrent error detection in matrix arithmetic and signal processing on highly concurrent VLSI structures. In order not to increase the roundoff errors when we incorporate the WCC into the computation, a simple roundoff error analysis is used to guide the construction of the WCC. A new data retry technique is then proposed to locate the faulty processors and identify the correct outputs. Such an approach provides rapid error detection with low hardware overhead while system performance is not significantly degraded for the sake of fault tolerance.

1. Introduction

Many algorithms for digital signal and image processing, such as Fast Fourier Transform (FFT), Finite Impulse Response filters (FIR), 1-D convolution, 2-D convolution [1], and feature extraction and pattern classification [2], require large-scale matrix or vector computations in their solutions. Fast matrix algorithms for solving large-scale matrix computations have been proposed by Kant and Kimura [3], Sameh and Kuck [4], Hwang and Cheng [5], and many other researchers. Also, many existing architectures consisting of array-structured machines, such as ILLIAC IV, MPP (Massively Parallel Processor) [6], and systolic array processors [1,7] have been proposed to solve these problems effectively. A major difficulty with a high degree of integration is that a single flaw in a chip can render an entire computing system useless. It is, therefore, desirable to have a high-performance system which can also tolerates physical failures in the system by providing correct results, or one which can at least detect the error, restructure the system, and retry the computation.

An encoding technique, the weighted checksum code (WCC), was proposed in [8] to achieve both error detection and correction for matrix operations using highly concurrent VLSI computing structures. This technique is very cost-effective and valid when fixed-point number systems are employed. Since roundoff errors may destroy the error correction capability of WCC, it may be difficult to apply this technique alone on floating-point number systems. In this paper, the Weighted Checksum Code (WCC) will be used to perform concurrent error detection (CED) fast and cost-effectively. A simple roundoff error analysis is used to guide the construction of the WCC such that the roundoff errors will not increase due to the incorporation of the WCC into the computation. A new data retry technique is then proposed to locate the faulty processors and identify the correct outputs. Such an approach provides rapid error detection with low hardware and time overhead compared with previous attempts at using hardware for error correction [8]. Once an error is detected (a relatively rare event in practice), additional time steps are used for fault location. Thus, system performance is not significantly degraded for the sake of fault tolerance. Large roundoff errors are detected and treated in the same manner as functional errors. However, the data retry technique can also distinguish between the roundoff errors and functional errors which are caused by some physical failures. The proposed scheme, error detection by hardware redundancy method and error correction by time redundancy method, is thus cost-effective and valid for both fixed-point and floating-point number systems.

For simplicity of treatment, this discussion will be based on linear array architectures which are believed to hold the most promise in VLSI computing structures for their flexibility, low cost, and applicability to most of the interesting algorithms. A similar discussion clearly holds for two-dimensional array architectures as well.

In Section 2, a module-level fault model applicable to VLSI is described. In Section 3, the matrix encoding technique is reviewed. Section 4 discusses the effect on the word length and the roundoff error analysis. In Section 5, a concurrent error detection scheme using the weighted checksum is proposed. Section 6 describes the faulty processor identification and error correction procedures. In Section 7, a procedure for obtaining correct data and identifying faulty processors is described for systems with multiple faulty processors.

2. The Fault Model

In this paper, we allow a module (such as a processor or computation unit in a multiple processor system) to produce any arbitrary logical errors under failures. We also assume that, at most, one module is faulty within a given period of time, which will be relatively short compared to the mean time between failures. In Section 7, systems with multiple faulty processors are also discussed.

Since effective error correcting schemes, such as Hamming codes [9] and Alternate-data retry [10], exist for communication lines and memories, we will assume that failures in the communication lines and memories are detected and corrected by those methods. In this paper, we will, therefore, focus on the fault tolerance of the processor array.

3. The Weighted Checksum Encoding Scheme

Let us denote a matrix H, the WCC-matrix, as

$$H = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} & -1 & 0 & \cdots & 0 \\ w_{21} & w_{22} & \cdots & w_{2n} & 0 & -1 & \cdots & 0 \\ . & . & \cdots & . & . & . & \cdots & . \\ w_{t1} & w_{t2} & \cdots & w_{tn} & 0 & 0 & \cdots & -1 \end{bmatrix}.$$

Using this, a compact description for the Weighted Checksum Code (WCC) can be given in terms of matrices. Readers can refer to [8] for detail.

Definition 1: Let H be a t–by–$(n+t)$ matrix of numbers. Then the set of $(n+t)$-element vectors that satisfy the matrix equation $HA = \mathbf{0}$ is called the code space of H, where A is a column vector.

Theorem 1: The code vector of a WCC-matrix H has at least d nonzero elements (or is a distance-d code) if and only if every combination of $d-1$ or fewer columns of H are linearly independent.

From Theorem 1 and the construction principle of the WCC code, we can construct a WCC with a suitable capability by suitably assigning the weights of the matrix H. Since a single module level fault model applicable to VLSI has been assumed, a WCC-matrix H, where $H = [\ 1\ 1\ ..\ 1\ -1\ ]$, will be used to demonstrate the encoding technique and develop the theory. This specific distance-2 code which is a subset of WCC and is simply called the checksum code, will be used to efficiently achieve concurrent error detection (CED). However, the fact holds for the general distance-$t+1$ Weighted Checksum Encoding Matrices whose WCC-matrices satisfy the requirement of Theorem 1.

* This research was supported by the Semiconductor Research Corporation under Contract SRC-RSCH84-06-049.

** Professor Abraham is with Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

Assume in the following discussion that A is an n-by-m matrix. Since n can be 1 or m can be 1, vectors are defined in the same way as matrices. Let us define:

$$e^T = [\,1\;1\;\cdots\;1\,] \quad \text{and} \quad f^T = [\,1\;1\;\cdots\;1\,],$$

where e is a n-by-1 column vector and f is a m-by-1 column vector.

Definition 2: The column, row and full checksum matrix A_c, A_r, and A_f of the matrix A are defined as:

$$A_c = \begin{bmatrix} A \\ e^T A \end{bmatrix} \quad A_r = [\,A\ Af\,], \quad A_f = \begin{bmatrix} A & Af \\ e^T A & e^T Af \end{bmatrix}.$$

It can be seen that five matrix operations exist which preserve the checksum property: addition, multiplication, LU-decomposition, transpose, and product of a matrix with a scalar. They are given in the following theorems without proofs.

Theorem 2: If $B_1 \,..\, B_n A = C$, then $B_1 \,..\, B_n A_r = C_r$.

Corollary 1: $A_c B_1 \cdots B_n = C_c$.

Corollary 2: $A_c B_r = C_f$.

Theorem 3: If $A + B = C$, then $A_r + B_r = C_r$, $A_c + B_c = C_c$, $A_f + B_f = C_f$.

Theorem 4: If $sA = C$, where s is a scalar, then $sA_r = C_r$, $sA_c = C_c$, $sA_f = C_f$.

Theorem 5: If $A^T = C$, then $A_r^T = C_c$, $A_c^T = C_r$, $A_f^T = C_f$.

Theorem 6: When the matrix A is LU decomposable, the full checksum matrix of A, A_f, can be decomposed into a column checksum lower matrix and a row checksum upper matrix such that $A_f = L_c U_r$.

4. *Effect On The Word Length*

When processing the matrix with fixed-point systems, the definitions of the summation elements may be modified to use residue arithmetic and, thus, avoid very large checksums [8,11]. Then,

$$a_{i,\,m+1} = \sum_{j=1}^{j=m} a_{i,j} \quad mod\ M \quad \text{for } 1 \le i \le n$$

$$a_{n+1,j} = \sum_{i=1}^{i=n} a_{i,j} \quad mod\ M \quad \text{for } 1 \le j \le m,$$

where $M = 2^0$. We assume that all the numbers lie in the range $-1 \le a \le 1$.

In floating-point number systems, the modified definitions are:

$$a_{i,\,m+1} = 2^{-\lceil \log_2 m \rceil} \sum_{j=1}^{j=m} a_{i,\,j} \quad \text{for } 1 \le i \le n$$

$$a_{n+1,\,j} = 2^{-\lceil \log_2 n \rceil} \sum_{i=1}^{i=n} a_{i,\,j} \quad \text{for } 1 \le j \le m.$$

The reason for the modified definitions of floating-point number systems will be discussed later in this Section.

In floating-point arithmetic, each number x is represented in the form $x = m2^e$, where m is called the mantissa and e the exponent. We assume that $1/2 \le |m| < 1$ for normalized floating-point arithmetic operations, where $|m|$ is the absolute value of m. We denote l as the number of binary digits allocated both to a fixed-point number and the mantissa of a floating-point number, and u as the machine-dependent unit roundoff. For example, $u = 2^{-(l+1)}$ in a l-digit fixed-point system.

Following [12], we will use $fi(.)$ to denote the computed fixed-point result of the argument and $fl(.)$ to denote the computed floating-point result of the argument. The equivalence sign will be used to emphasize that rounding errors have been taken into account. We have

$$fi(x \pm y) \equiv x \pm y, \quad fl(x \pm y) \equiv x(1+\varepsilon) \pm y(1+\varepsilon),$$
$$fi(xy) \equiv xy + \delta, \quad fl(xy) \equiv xy(1+\delta).$$

where $|\varepsilon|$, $|\delta| \le u$. In the computations, we assume that the computed results do not lie outside of the permitted range.

Let's discuss the case of fixed-point number systems first. Assume that the equation we want to calculate is $A_{n+1,\,m} X_{m,\,1} = B_{n+1,\,1}$.

$$\begin{aligned} b_i &= fi(a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,\,m}x_m) \\ &\equiv a_{i,1}x_1 + a_{i,2}x_2 + .. + a_{i,\,m}x_m + \delta_1 + \delta_2 + .. + \delta_m \end{aligned}$$

If we do the checksum verification for the vector B, the error bound E is

$$E = \left| \sum_{i=1}^{n} b_i - b_{n+1} \right| \le (n+1)mu.$$

The error bounds of the fixed-point systems only depend on the size of the problems.

Let's discuss the case of floating-point number systems.

$$\begin{aligned} b_i &= fl(a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,\,m}x_m) \\ &\equiv a_{i,1}x_1(1+\delta_1)(1+\varepsilon_1)(1+\varepsilon_2) \,..\, (1+\varepsilon_{m-1}) \\ &+ a_{i,2}x_2(1+\delta_2)(1+\varepsilon_1)(1+\varepsilon_2) \,..\, (1+\varepsilon_{m-1}) \\ &+ \cdots + a_{i,\,m}x_m(1+\delta_m)(1+\varepsilon_{m-1}). \end{aligned}$$

The error bound E of vector B is

$$E = \left| \sum_{i=1}^{n} b_i - b_{n+1} \right| \le (n+1)mu\, ||\mathbf{x}||_2\, ||\mathbf{a}_{\max}||_2.$$

Thus, in floating-point systems, the error bound is affected by $||\mathbf{a}_i||_2$. In order not to increase the roundoff error bound when we incorporate checksum techniques into the computations, we can use

$$a_{n+1,\,j} = 2^{-\lceil \log_2 n \rceil} \sum_{i=1}^{n} a_{i,\,j}$$

because

$$||\mathbf{a}_{n+1}||_2 = 2^{-\lceil \log_2 n \rceil}\, ||\sum_{i=1}^{n} \mathbf{a}_i||_2 \le ||\mathbf{a}_{\max}||_2.$$

5. *Concurrent Error Detection*

We have seen in Section 3 that checksum matrix operations produce code outputs which provide some degree of error-detecting or correcting capability. However, a faulty module may cause more than one element of the result to be erroneous if it is used repeatedly during a simple matrix operation. This problem is solved by using multiple processors and scheduling each processor to calculate only a few data elements. The errors caused by a faulty processor are then confined either to a few data elements or to only one element. In this manner, the checksum technique incorporated into matrix operations can detect or correct errors caused by a faulty module.

In this paper, weighted checksum technique is only used to achieve concurrent error detection capability. An new data retry technique, which will be described in Sections 6 and 7, will thus be used to achieve error correction and faulty processor identification.

In the following discussion, the matrix-vector multiplication will be used as an example to demonstrate the concurrent error detection scheme. It is obvious that the concurrent error detection scheme can apply to a variety of matrix operations and signal processing algorithm as well.

When matrix operations are performed on a computer system or with special-purpose hardware, roundoff errors due to finite word length are hard to avoid whenever fixed-point or floating-point arithmetic is used. A small difference, η, which can be decided by simulated results or analytic error bounds, as discussed in the previous section, must be allowed for when checking for equality. If the analytic error bounds are used as η, then any functional errors which affect the outputs of the computations more than the analytic error bounds will be detected. In practice, η should be chosen between zero and the analytic error bounds. Thus, large roundoff errors can also be detected and treated in the same manner as a computing unit with a functional fault. The functional errors which affect the outputs of the computations much less than η will, of course, not be detected, but these will not affect the results significantly. The best choice of η may depend on the applications and will not be discussed in this paper. Thus, in the following discussion, we will only outline the concurrent error detection scheme.

Many signal and image processing algorithms such as FIR and DFT [1] involving a "multiply-and-accumulate" type of expression can be formulated as matrix-vector multiplication problems. Figure 1 shows a linear processor array; the input data streams show the multiplication of a 5-by-4 column checksum matrix with a 4-by-1 vector. The operation of the array is described as follows: we want to calculate the equation $A_{n+1,\, n}\, X_{n,\, 1} = B_{n+1,\, 1}$. The matrix element $a_{i,j}$ is stored in the local memory of the *ith* processor at the *jth* time step, and x_j is broadcast to each processor at the *jth* time step, as shown in Figure 1. Each processor multiplies n pairs of $a_{i,j}$ and x_j and accumulates the products in a register. Each processor thus calculates one element of the result vector and the faulty processor affects only one element. Any error can, therefore, be detected by using checksum scheme.

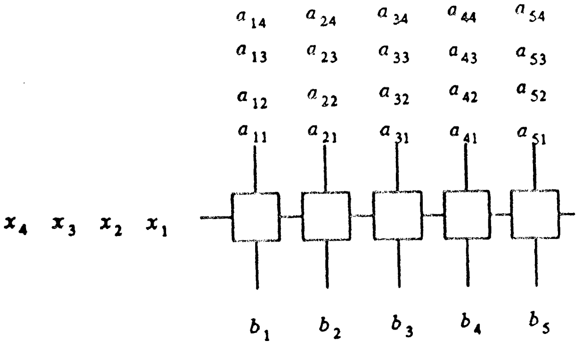

a_{14} a_{24} a_{34} a_{44} a_{54}
a_{13} a_{23} a_{33} a_{43} a_{53}
a_{12} a_{22} a_{32} a_{42} a_{52}
a_{11} a_{21} a_{31} a_{41} a_{51}
x_4 x_3 x_2 x_1
b_1 b_2 b_3 b_4 b_5

Figure 1. Checksum matrix-vector multiplication

After the result vector is obtained, one n-operand adder can be used to calculate the checksum. Define the *overhead ratio* as the ratio of the time, hardware, or delay overhead required by the CED technique to the time, hardware, or delay complexity of the original system without CED. Since the whole computation, including the checking, is pipelined, there is no time overhead in terms of performance. However, the delay overhead ratio is $O(\left\lceil \frac{n}{l} \right\rceil / n)$, and the hardware overhead ratio is $O((n/\ l+1)/\ n))$, where l is the word length. If $n = l$, then the delay overhead ratio is O(1/n) and the hardware overhead ratio is O(2/n). In floating-point systems, since the execution time of addition is comparable to that of multiplication, the delay overhead ratio becomes $O(\left\lceil \log_2 n \right\rceil / n)$.

6. *Error Correction By Data Retry*

In this section, we will propose a technique for the correction of erroneous results using time redundancy. This technique also enables us to distinguish between functional errors and large roundoff errors. Assume we want to calculate the equation $A_{n,n}\, X_{n,1} = B_{n,1}$. Our approach is that on the second try, we assign the computation step of each element of the output vector B to a processor which is different from the one used before. For example, in the first run, the computation of b_1 is assigned to processor one, b_2 to processor two . . . and b_n to processor n. In the second try, the computation of b_1 is assigned to processor 2, b_2 to processor 3 . . . and b_n to processor 1. (In order to simplify the notation, we will use $i+1$ to represent $i \ mod\ n+1$ in the following discussion.) After two tries, each element of the vector has two results (not necessarily different). We then compare the two results. If the two results of one element from two tries are different, we say that this element has inconsistent results.

(1) If there are two elements which have inconsistent results, for example b_i and b_{i+1}, we know that processor $i+1$ is faulty. The correct result of b_i can be obtained from the first try and the correct result of b_{i+1} can be obtained from the second try.

(2) If there is only one element which has an inconsistent result, for example b_i, then either the processor i produced a transient error in the first try or the processor $i+1$ produced a transient error in the second try. The correct result of b_i can be obtained (i) by subtracting the difference of the computed sum of elements of the vector and the checksum to the erroneous element in the information part, or (ii) by replacing the checksum by the computed sum of the information elements in the summation vector, in the case where the checksum is incorrect. This correction procedure can be based on the data either from the first try or the second try.

(3) If there is no element which has an inconsistent result, then large roundoff errors have been detected in the first try.

7. *Error Correction For Multiple Faults*

From Theorem 1 and the construction principle of the WCC, we can construct a WCC with a distance $(t+1)$ such that the code can detect up to t errors. A time redundancy method can then be used to perform the error correction and identification of faulty units. The general theory of the WCC has been reviewed in Section 3. In this section we will concentrate on the error correction by using time redundancy.

There is assumed to be a set of jobs $J = \{\, j_1, j_2, \ldots \}$ to be performed, and a set of identical units $U = \{\, u_1, u_2, \ldots \}$ available to perform them. For example, we want to calculate the equation $A_{n,\, n}\, X_{n,\, 1} = B_{n,\, 1}$ in a linear array with n processors. The computation of each element of the result vector will be thought as a job. Once we detect any errors with the WCC, each job will be reassigned to a unit which is different from the units which have been assigned to do this job during the previous computation. When the jobs have been completed by the units, the results are compared to the previous results. The outcomes of such comparisons are the basis for identifying faulty units and obtaining correct data.

A system under a t-fault assumption refers to one in which up to t faulty processors are permitted. It will be assumed that when two faulty units perform the same job, they do not produce identical, incorrect results [13]. This is also shown in Figure 2. The outcome "pass" indicates that both units at this computation are fault-free, or the output data calculated by these two units are reliable. The outcome "fail" indicates that at least one of the units is faulty.

Unit 1	Unit 2	Comparison outcome
fault-free	fault-free	0 (pass)
fault-free	fault	1 (fail)
fault	fault-free	1 (fail)
fault	fault	1 (fail)

Figure 2. The outcomes of a comparison of a pair of units

It is also assumed that there is a host computer which collects the information on comparisons and, thus, derives the state of the whole system. Here, we require that the diagnosis never be incorrect in the sense that a fault-free unit is diagnosed as faulty.

Theorem 7: For a t-fault system, at most $t+2$ tries are required to identify the correct data and the faulty processors.

In real situations, we may usually use only a small number of tries to identify the correct data and faulty processors. For example, we have a system with $t = 4$. Figure 3 shows an example with four faulty processors and a hypothetic set of outcomes. The jobs which are incorrectly performed are marked with *. The maximum number of tries to locate the correct data and the faulty processors is 6. From

Figure 3, we obtain the correct results of j_4 and j_6 after two tries. But we do not know the correct results of j_1, j_2, j_3, j_5 and j_7. No units can be identified as faulty. After the third try, we obtain the correct results of j_1, j_2 and j_7, and we know that u_1 and u_3 are faulty. But we still do not know the correct results of j_3 and j_5. After the fourth try, we obtain the correct result of j_3 and identify that u_5 is faulty. After the fifth try, we obtain the correct result of j_5 and identify that u_6 is faulty. Thus, instead of six tries, we have only used five tries to identify the correct results and the faulty processors. If we can replace or repair the faulty processors right after they are identified, the number of tries might be reduced further. In this particular example, if we replace u_1 and u_3 after the third try, we can get the correct results of both j_3 and j_5 and identify the faulty processors u_5 and u_6 right after the fourth try. Thus, instead of five tries, four tries are enough to locate the correct results and identify the faulty processors.

tries	u_1	u_2	u_3	u_4	u_5	u_6	u_7
1	j_1^*	j_2	j_3^*	j_4	j_5^*	j_6	j_7
2	j_7^*	j_1	j_2^*	j_3	j_4	j_5^*	j_6
3	j_6	j_7	j_1	j_2	j_3^*	j_4	j_5
4	j_5^*	j_6	j_7^*	j_1	j_2	j_3	j_4
5	j_4^*	j_5	j_6^*	j_7	j_1^*	j_2^*	j_3

Figure 3. An example of job assignments and the results

The algorithm for determining correct data and identifying faulty processors is described below:

(1) Assign each job to a unit which is different from the ones have been assigned to execute this job in the previous tries.

(2) Check the results of each job with its previous results. If there is an outcome "pass", then the correct output of that job is identified. All the processors which produce the erroneous outputs of that job are identified as faulty.
(Optional -- replace or repair the faulty processors)

(3) If there is any job for which we still do not know the correct output, go to step (1). Else, exit.

If we obtain correct outputs of all jobs in the second try, we know that large roundoff errors have been detected in the first try.

8. Conclusion

In this paper we proposed a concurrent error detection scheme using the WCC with low hardware overhead for matrix algebra and signal processing with highly concurrent VLSI structures. A simple roundoff error analysis is used to guide construction of the WCC. A new data retry technique is used to locate the faulty processors, obtain the correct results, and distinguish between roundoff errors and functional errors. Such an approach provides rapid error detection with low hardware overhead and solve the roundoff error problem in floating-point number systems. System performance is also not significantly degraded for the sake of fault tolerance.

References

[1] C. Mead and L. Conway, *Introduction to VLSI Systems,* Addison-Wesley, 1980.

[2] F. A. Briggs, K. S. Fu, and B. W. Wah, "PUMPS: Architecture for Pattern Analysis and Image Database Management," *IEEE Transactions on Computers,* vol. C-31, pp. 969-983, October 1982.

[3] R. M. Kant and T. Kimura, "Decentralized Parallel Algorithms for Matrix Computations," *Proceeding 5th Annual Symposium on Computer Architecture,* Palo Alto, California, pp. 96-100, April 1978.

[4] A. Sameh and D. Kuck, "On Stable Parallel Linear System Solvers," *Journal of the Association for Computing Machinery,* vol. 25, pp. 81-91, January 1978.

[5] Kai Hwang and Yeng-Heng Cheng, "Partitioned Matrix Algorithms for VLSI Arithmetic Systems," *IEEE Transactions on Computers,* vol. C-31, pp. 1215-1224, December 1982.

[6] Kenneth E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers,* vol. C-29, pp. 836-840, September 1980.

[7] A. V. Kulkarni and D. W. L. Yen, "Systolic Processing and an Implementation for Signal and Image Processing," *IEEE Transactions on Computers,* vol. C-31, no. 10, pp. 1000-1009, October 1982.

[8] Jing-Yang Jou and J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of IEEE,* May, 1986.

[9] R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technology Journal,* vol. 29, no. 1, pp. 147-160, January 1950.

[10] J. J. Shedletsky, "Error Correction by Alternate-Data Retry," *IEEE Transactions on Computers,* vol. C-27, pp. 106-114, February 1978.

[11] K. H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers,* vol. C-33, pp. 518-528, June 1984.

[12] J. H. Wilkinson, *The Algebraic Eigenvalue Problem,* Clarendon Press, Oxford, 1965.

[13] F. Barsi, F. Grandoni and P. Maestrini, "A Theory of Diagnosability in Digital Systems," *IEEE Transactions on Computers,* vol. C-25, pp. 585-593, June 1976.

EFFICIENT DESIGNS OF PRIORITY QUEUE

Kam Hoi Cheng
Computer Science Department
University of Houston
Houston, TX 77004

ABSTRACT

VLSI designs are examined for the priority queue problem. We develop designs with superior performance to earlier designs.

Keywords and Phrases
VLSI architectures, systolic systems, priority queue

1. Introduction

VLSI architectures for a variety of problems have been proposed by several authors. A bibliography of over 150 research papers dealing with this subject appears in [8]. In this paper, we are concerned solely with the priority queue problem. Many applications require the ability to insert records into a set and to retrieve from the set the record having the smallest key according to some ordering. A data structure that provides such services is called a priority queue.

In evaluating our designs, we assume that the VLSI system will be attached to the host processor using a bus. The evaluation of a VLSI design should take the following into account:

1. Processors --- how many processors are used in the VLSI system? This figure is denoted by P.
2. Bus bandwidth --- the maximum amount of data to be transmitted between the host and the VLSI system in any cycle. This figure is denoted by B.
3. Speed --- how much time does the VLSI system need to complete its task and be ready to accept the next operation? This time may be decomposed into two non-overlapping times T_C (time for computations) and T_D (time for data transmissions both within the VLSI system and between the host and the VLSI system).

One may expect that by using a very high bandwidth B and a large number of processors P, we can make T_C and T_D quite small. So, T_C and T_D are not in themselves a very good measure of the effectiveness with which the resources B and P have been used. Let D denote the total amount of data that needs to be transmitted between the host and VLSI system. The ratio

$$R_D = B * T_D / D$$

measures the effectiveness with which the bandwidth B has been used. Clearly, $R_D \geq 1$ for every VLSI design.

Let C denote the time spent for computation by a single processor algorithm. The ratio

$$R_C = P * T_C / C$$

measures the effectiveness of processor utilization. Once again, we see that $R_C \geq 1$ for every VLSI design.

In evaluating a VLSI design, we shall be concerned with T_C and T_D and also with R_C and R_D. We would like R_C and R_D to be close to 1. Finally, we may combine the two efficiency ratios R_C and R_D into the single ratio $R = R_C * R_D$. A design that makes effective use of the available bandwidth and processors will have R close to 1.

The efficiency measure R as defined here is the same as that used in [1]-[4] to evaluate VLSI designs for matrix multiplication, finite impulse response filter, recursive filter and back substitution. This measure is also quite similar to that proposed in [6]. In fact, the two measures become identical when $T_C = T_D$.

For each of the designs considered in this paper, we compute R_C, R_D and R. In all cases, our designs have improved efficiency ratios than all earlier designs using the same model. In comparing different architectures for the same problem, one must be wary about over emphasizing the importance of R_C, R_D and R. Clearly, using $P = 1$ and $B = 1$, we can get $R_C = R_D = R = 1$ and no speed up at all. So, we are really interested in minimizing T_C and T_D while keeping R close to 1.

For the priority queue problem, the single processor algorithm uses the heap data structure. When n records are already in the heap, both insertion and delete-min operations take $3 \log n$ comparisons including the test for the end of heap. So, for a single operation, $C = 3 \log n$ and $D = 1$. For comparison among different designs, the parameter n is used where n is the maximum number of values that the designs can handle.

VLSI designs for this problem have been proposed earlier in [5], [7] and [9]. All designs use a linear bidirectional chain of PEs. The design of [5] permits an insert/delete-min operation in every four cycles. In each cycle, at least two comparisons have to be made, one to determine whether neighboring elements are out of order and the other to check the status of its three neighboring PEs, two left and one right. The performance figures of [5] is $P = 3n$, $B = 2$, $T_C = 8$, $T_D = 8$, $R_C = 8n/\log n$, $R_D = 16$ and $R = 128n/\log n$. The design of [9] is ready to receive an operation in every 2 cycle with each cycle requiring 1 data move and 3 key comparisons to order 3 numbers. Since PEs work in alternate cycle, the number of PEs can be reduced by half. So $P = n/2$, $B = 2$, $T_C = 6$, $T_D = 2$, $R_C = n/\log n$, $R_D = 4$ and $R = 4n/\log n$. The design of [7] is ready to receive an operation in every cycle but each cycle requires 6 key comparisons to order 5 numbers in a special order. Their design has $P = n/2$, $B = 3$, $T_C = 6$, $T_D = 1$, $R_C = n/\log n$, $R_D = 3$ and $R = 3n/\log n$.

In between two priority queue operations, application program usually performs some processing which is likely to take times much longer than $T_C + T_D$. Since these hardware designs operate continuously, an no-op operation is required when neither an insertion nor a delete-min is necessary. None of the above designs handle no-op explicitly. However both the designs of [7] and [9] can perform no-op by the input of $(\infty, -\infty)$, while the design of [5] accomplish this by the input of ∞. In all our designs, an input operation (insert, delete-min, no-op) will be performed when the designs are ready to accept a new operation. PEs are numbered from left to right with PE 1 being the leftmost PE.

2. New Designs

In our first design, a linear array of n PEs are required. Each PE has three registers, a, s and l. Register l contains the value input from its left neighboring PE in the current cycle. For the leftmost PE, this is the input to the design. Register s is the status variable. When $s = 0$, the last operation performed is an insert and the value of register a in the PE is to be used directly. However when $s = 1$, the last operation performed is a delete-min. The value in register a has already been moved to its left neighbor in the previous cycle and it will be replaced by the value coming in from its right neighbor. Initially for all PEs, $s = 0$ and the contents of registers a and l are α ($= \infty - 1$). Inserting a new value is simply done by the input of the value to the leftmost PE. Delete-min operation is done by the input of a special largest value ∞. No-op can be achieved by the input of α. An example sequence of operations are shown in Figure 1. For each PE, the value of register s is shown above the value of register a. The even and odd PEs execute alternately as in the design of [9]. The exact workings for PE i, when active, are formally described in Algorithm 1. From Algorithm 1 and Figure 1, the performance figures of the first design are $P = n$, $B = 1$, $T_C = 2$, $T_D = 2$, $R_C = 2n/(3\log n)$, $R_D = 2$ and $R = 4n/(3\log n)$. Since odd and even PEs execute alternately, the number of PEs can be reduced to $n/2$ with $R_C = n/(3\log n)$ and $R = 2n/(3\log n)$.

```
loop
    do in parallel
        $l_i \leftarrow l_{i-1}$
        $a_i \leftarrow a_{i+1}$        if $s_i = 1$
    end
    do in parallel
        $a_i \leftarrow min(l_i, a_i)$; $l_i \leftarrow max(l_i, a_i)$
        if $l_i = \infty$ then $s_i \leftarrow 1$
                  else $s_i \leftarrow 0$
    end
forever
```

Algorithm 1

The second design makes use of a linear chain of $o(n/3)$ PEs. In each PE, four registers a, b, c and d are required. Register a is the value kept in the PE, registers b and c are the values sent from its left neighbor and satisfies the relation $b \leq c$. Register d is the value just moved in from its right neighbor. In each cycle, after values have been moved in from its left and right neighbors, each PE will rearrange the contents of registers a, b, c and d in such a way that $d \leq a \leq b \leq c$. Since $b \leq c$ originally, the above rearrangement process only requires 4 comparisons. To insert a value, simply input the tuple ($-\infty$,value) to the leftmost PE. Delete-min is performed by the input of the tuple (∞,∞) and no-op is to input the tuple ($-\infty$,∞). An example sequence of operations are shown in Figure 2. The exact workings for PE i are formally described in Algorithm 2. The functions max2 and min2 will find the second largest and second smallest values in the given list respectively. From Algorithm 2 and Figure 2, we see that the performance figures for the second design are $P = \lceil (n+2)/3 \rceil$, $B = 3$, $T_C = 4$, $T_D = 1$, $R_C = 4n/(9\log n)$, $R_D = 3$ and $R = 4n/(3\log n)$.

```
loop   $1 \leq i \leq p$
    do in parallel
        $b_i \leftarrow b_{i-1}$      {$b_0$ and $c_0$ are input}
        $c_i \leftarrow c_{i-1}$      {where $b_{i-1} \leq c_{i-1}$}
        $d_i \leftarrow d_{i+1}$      $0 \leq i < p$   {$d_0$ is output}
    end
    $d_i \leftarrow min(a_i, b_i, c_i, d_i)$
    $a_i \leftarrow min2(a_i, b_i, c_i, d_i)$
    $b_i \leftarrow max2(a_i, b_i, c_i, d_i)$
    $c_i \leftarrow max(a_i, b_i, c_i, d_i)$
forever
```

Algorithm 2

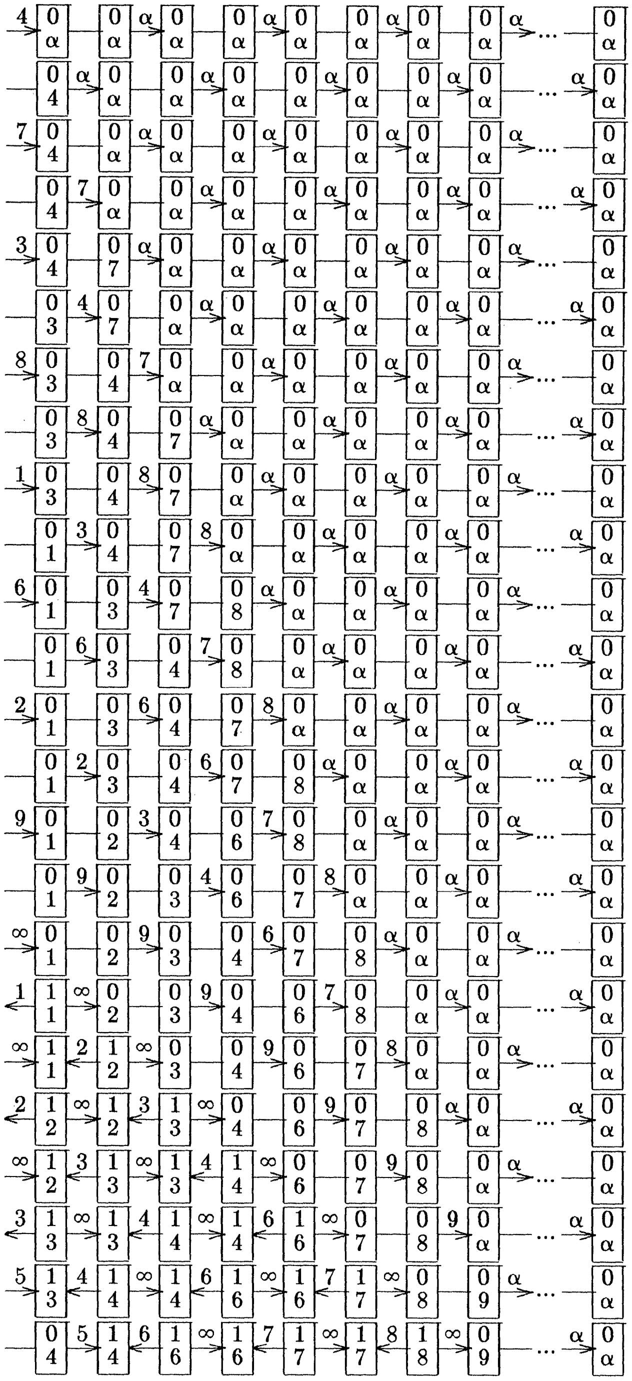


Figure 1

All the previous designs have a R value of $O(n/\log n)$. The third design improves the ratio R to $O(1)$ by using a chain of only $\log n$ PEs. A fictitious PE, PE 0, is assumed to handle the input and output of the design. As in our first design, the even and odd PEs execute alternately. This design tries to simulate the action of a min-heap which is a complete binary tree with the property that the value of a node is not greater than its two sons. A min-heap with n elements has $\lceil \log n \rceil$ levels. Each PE in the chain will therefore be responsible to maintain a level in the min-heap. Quinn [10] have shown

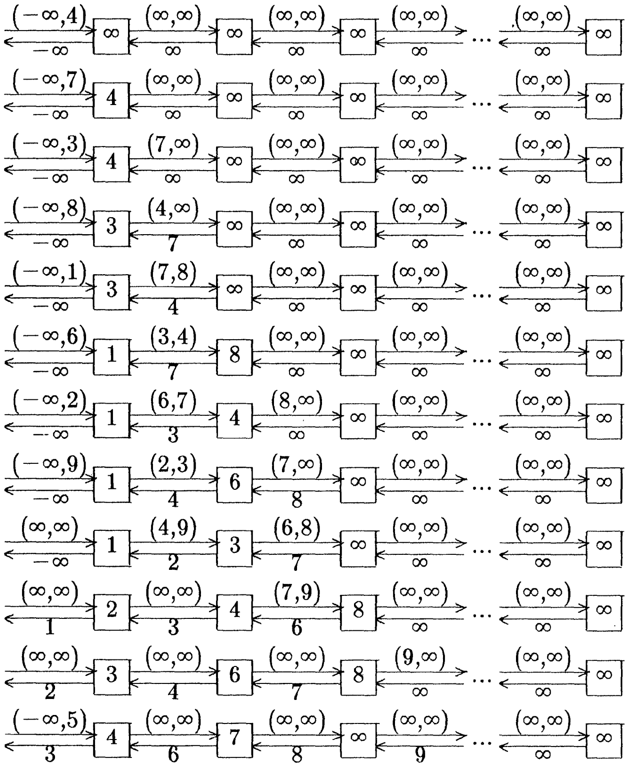


Figure 2

that a tightly coupled shared memory multiprocessor with $\lceil \log n \rceil$ processors can remove an element from an n-element heap in constant time. However, besides using shared memory, their design also requires all insertions to be done before all deletions which fails to handle the case of random insertions and deletions as required by priority queue.

A single processor delete-min operation will delete the minimum element from the root of the min-heap, reinserted the last element into the root and then reheapify. However, in any design using a chain of processors, this element may be in the process of getting to the last processor and hence is not known immediately. Therefore, a modification to the single processor algorithm is required. Now suppose that the element which is moving up from the next level to replace the deleted element is chosen to be the minimum of its two sons. Repeating this process all the way to the lowest level of the heap may create empty locations in the data structure. One major drawback of this is that the number of processors required to handle n elements will be greater than $\lceil \log n \rceil$. Our design is based on the observation on how to maintain a min-heap in a linearly connected chain of processors. When an element is deleted in the previous level, the element that is moving up to replace it is chosen to be the minimum of the following three numbers: its two sons and the last (rightmost) element at the same level of the two sons in the min-heap. The above process repeats with the two sons of this minimum being used at the next level.

Let the elements of the min-heap be stored in the a arrays of each PE. PE i will require a memory of size 2^i. Since the amount of memory required in the worst case is approximately $n/2$ for the last PE, so instead of using registers to store these a values, random access memory will be used because its cost is cheap and it is readily available. The access time of random access memory will only be a constant multiple of the access time when registers are used. Besides the storage for the a values, registers are required for the following variables: s, c, l, r, t, e, pi, pn, pp, vl and vr. Register s has similar meaning here as in our first design. When $s = 0$, the last operation performed is an insert. However, when $s = 1$, the last operation performed is a delete-min. Register l contains the maximum number of memory locations in this PE, so for PE i, $l = 2^i$. Register c indicates how many locations of the a array are currently being used, so $1 \leq c \leq l$. Register r, t and e are responsible to keep track of which path the next insertion is to go down the PE chain. The element $a[pi]$ is used to compare with the element input from the previous PE on its left. As for register pn and pp, if the element $a[pp]$ of PE $i-1$ is being deleted and sent back to PE $i-2$, the index of its right son, $pn = 2 * pp$, will be sent to PE i. The elements that PE i sent back to PE $i-1$ will then be placed in $a[pp]$ to replace the deleted element. Register vl and vr contains respectively the values sent from its left and right neighboring PEs. As for initial configuration, the initial values for the a arrays are α. The initial values for other registers in PE i are as follows: $s = c = 0$, $l = e = 2^i$, $pi = r = t = 1$ and register pn of PE 0 will always be set to 2. Finally, the operation of inserting a value is simply done by the input of the value to the leftmost PE. Delete-min operation is done by the input of ∞ and no-op can be done by the input of α.

An example sequence of operations for the third design are shown in Figure 3. Here, only the contents of the a arrays, vl and vr are shown. The exact workings of

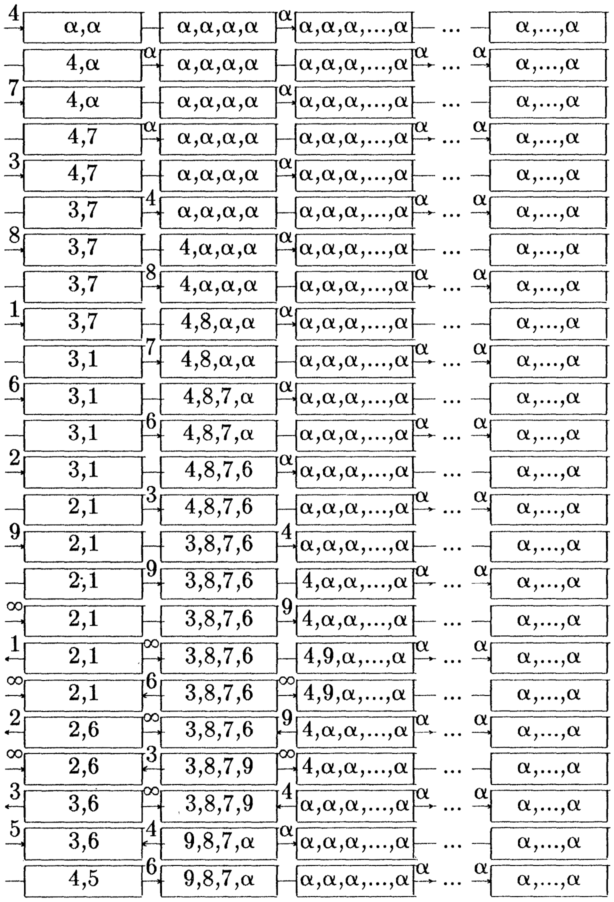

4 α,α α,α,α,α α α,α,α,...,α ... α,...,α
4,α α α,α,α,α α,α,α,...,α α ... α α,...,α
7 4,α α,α,α,α α α,α,α,...,α ... α,...,α
4,7 α α,α,α,α α,α,α,...,α α ... α α,...,α
3 4,7 α,α,α,α α α,α,α,...,α ... α,...,α
3,7 4 α,α,α,α α,α,α,...,α α ... α α,...,α
8 3,7 4,α,α,α α α,α,α,...,α ... α,...,α
3,7 8 4,α,α,α α,α,α,...,α α ... α α,...,α
1 3,7 4,8,α,α α α,α,α,...,α ... α,...,α
3,1 7 4,8,α,α α,α,α,...,α α ... α α,...,α
6 3,1 4,8,7,α α α,α,α,...,α ... α,...,α
3,1 6 4,8,7,α α,α,α,...,α α ... α α,...,α
2 3,1 4,8,7,6 α α,α,α,...,α ... α,...,α
2,1 3 4,8,7,6 α,α,α,...,α α ... α α,...,α
9 2,1 3,8,7,6 4 α,α,α,...,α ... α,...,α
2;1 9 3,8,7,6 4,α,α,...,α α ... α α,...,α
∞ 2,1 3,8,7,6 9 4,α,α,...,α ... α,...,α
1 2,1 ∞ 3,8,7,6 4,9,α,...,α α ... α α,...,α
∞ 2,1 6 3,8,7,6 ∞ 4,9,α,...,α ... α,...,α
2 2,6 ∞ 3,8,7,6 9 4,α,α,...,α α ... α α,...,α
∞ 2,6 3 3,8,7,9 ∞ 4,α,α,...,α ... α,...,α
3 3,6 ∞ 3,8,7,9 4 α,α,α,...,α α ... α α,...,α
5 3,6 4 9,8,7,α α α,α,α,...,α ... α,...,α
4,5 6 9,8,7,α α,α,α,...,α α ... α α,...,α

Figure 3

PE i during its active cycles are formally described in Algorithm 3 where reg_i is the reg register of PE i and its subscript is dropped if it is understood to be of PE i. From Algorithm 3 and Figure 3, the performance figures of this design are $P = O(\log n)$, $B = O(1)$, $T_C = O(1)$, $T_D = O(1)$, $R_C = O(1)$, $R_D = O(1)$ and $R = O(1)$.

```
do in parallel
    $pn_i \leftarrow pn_{i-1}$                        $\{pn_0 = 2\}$
    $vl_i \leftarrow vl_{i-1}$                        $\{vl_0$ is input$\}$
    $a[pp] \leftarrow vr_{i+1}$   if $s_i = 1$   $\{i = 0$ is output$\}$
end
if $vl_i < \alpha$ then          {insert}
    do in parallel
        $s_i \leftarrow 0$; $t \leftarrow (t + r - 2)$ $mod$ $r + 1$
        if $c < l$ then
            do in parallel
                $a[pi] \leftarrow vl_i$; $c \leftarrow c + 1$
            end
        else
            do in parallel
                $a[pi] \leftarrow min(a[pi], vl_i)$
                $vl_i \leftarrow max(a[pi], vl_i)$
            end
        endif
    end
    if $t = r$ then
        do in parallel
            $pi \leftarrow pi$ $mod$ $l + 1$
            $e \leftarrow (e + l - 2)$ $mod$ $l + 1$
        end
        if $e = l$ then
            do in parallel
                $t \leftarrow 2t$; $r \leftarrow 2r$
            end
        endif
    endif
elseif $vl_i = \alpha$ then          {no-op}
    $s_i \leftarrow 0$
else                                   {delete}
    do in parallel
        Let $a[pp] = min(a[pn-1], a[pn], a[c])$
        $vr_i \leftarrow a[pp]$; $pn \leftarrow pp * 2$
        $s_i \leftarrow 1$; $t \leftarrow t$ $mod$ $r + 1$
    end
    if $vr_i \neq \alpha$ then
        if $t = 1$ then
            do in parallel
                $pi \leftarrow (pi + l - 2)$ $mod$ $l + 1$
                $e \leftarrow e$ $mod$ $l + 1$
            end
            if $e = 1$ then $r \leftarrow r/2$
        endif
        if $r = 1$ then               {rightmost active PE}
            do in parallel
                $c \leftarrow c - 1$; $a[pp] \leftarrow a[c]$
                $s_i \leftarrow 0$; $vl_i \leftarrow \alpha$
            end
            $a[c+1] \leftarrow \alpha$   $\{$to avoid conflict when $pp = c\}$
        endif
    else
        do in parallel
            $s_i \leftarrow 0$; $vl_i \leftarrow \alpha$; $t \leftarrow 1$
        end
    endif
endif
```

Algorithm 3

3. Summary

The performance figures of the various VLSI architectures for the priority queue problem are summarized in Table 1. As can be seen, all our designs represent an improvement over earlier designs. Our third design is the first VLSI system that has attained an R value of O(1).

Finally, we note that the comparisons among the different designs are not entirely fair as our third design requires different and considerable amount of memory for each of the $\log n$ PEs. However, the total amount of memory used in all designs are the same, namely $O(n)$.

Perf	Architecture					
	Bidirectional Chain					
	[5]	[9]	[7]	Our		
				I	II	III
P	$3n$	$n/2$	$n/2$	$n/2$	$n/3$	$O(\log n)$
B	2	2	3	1	3	O(1)
T_C	8	6	6	2	4	O(1)
T_D	8	2	1	2	1	O(1)
R_C	$\frac{8n}{\log n}$	$\frac{n}{\log n}$	$\frac{n}{\log n}$	$\frac{n}{3\log n}$	$\frac{4n}{9\log n}$	O(1)
R_D	16	4	3	2	3	O(1)
R	$\frac{128n}{\log n}$	$\frac{4n}{\log n}$	$\frac{3n}{\log n}$	$\frac{2n}{3\log n}$	$\frac{4n}{3\log n}$	O(1)

$C = 3\log n$, $D = 1$

Table 1

4. References

[1] K.H. Cheng and S. Sahni, *VLSI Systems For Matrix Multiplication,* Springer-Verlag Lecture Notes in Computer Science, **1985,** pp. 428-456.

[2] K.H. Cheng and S. Sahni, *VLSI Architectures for the Finite Impulse Response Filter,* IEEE Journal on Selected Areas in Communications, **January 1986,** pp. 92-100.

[3] K.H. Cheng and S. Sahni, *A New VLSI System For Adaptive Recursive Filtering,* International Conference on Parallel Processing (ICPP), **August 1986.**

[4] K.H. Cheng and S. Sahni, *VLSI Architectures For Back Substitution,* International Federation for Information Processing Conference, **September 1986.**

[5] L.J. Guibas and F.M. Liang *Systolic Stacks, Queues, and Counters,* Conference on Advanced Research in VLSI, M.I.T., **1982,** pp. 155-164.

[6] K.H. Huang and J.A. Abraham, *Efficient parallel algorithms for processor arrays,* ICPP, **1982,** pp. 271-279.

[7] O.H. Ibarra, M.A. Palis and S.M. Kim, *Designing Systolic Algorithms Using Sequential Machines,* IEEE Transactions on Computers, **June 1986,** pp. 531-542.

[8] H.T. Kung, *A Listing of Systolic Papers,* Comp. Sci. Dept., Carnegie-Mellon University, **May 1984.**

[9] C.E. Leiserson, *Systolic Priority Queues,* Caltech Conference on VLSI, **January 1979,** pp. 200-214.

[10]M.J. Quinn and Y.B. Yoo, *Data Structures for the Efficient Solution of Graph Theoretic Problems on tightly-coupled MIMD computers,* ICPP, **August 1984,** pp. 431-438.

ALGORITHMS FOR HIGH SPEED MULTI-DIMENSIONAL ARITHMETIC AND DSP SYSTOLIC ARRAYS

Nam Ling and Magdy A. Bayoumi

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504, U.S.A.

Abstract - With the advent of 3-D VLSI and the essentialness of CAD tool in design, the demand for high speed computation in several arithmetic and digital signal processing (DSP) applications can be met by having a systematic technique for transforming algorithms to specific forms for mapping onto multi-dimensional systolic arrays. This paper presents such a technique (called STAMS). The resulting multi-dimensional systolic arrays derived from the technique give significant improvements in computation time compared to their 1-D counterparts, yet maintaining the same number of processing cells. Two examples are illustrated in the paper: the matrix-vector multiplication algorithm and the k-point Discrete Fourier Transform (DFT) algorithm. The technique can also be applied to other problems such as the FIR filter algorithm and the 1-D convolution algorithm. An example of an entire systematic transformation and mapping procedure that can be incorporated into an integrated CAD package suitable for user-friendly interactive design is also given.

1. INTRODUCTION

Systolic arrays have been developed for the implementation of many arithmetic and digital signal processing (DSP) algorithms in the past decade. With the rising demand for high-speed computations in these applications and the recognizing of three-dimensional (3-D) VLSI chips [1,2], the need to speed up algorithm computation by going beyond 1-D (and sometimes even 2-D) systolic networks has increased [3,4]. Many throughput improvements have been shown by higher-dimensional systolic array implementation. For example, significant throughput improvements have been shown by a 3-D systolic array implementation of matrix-matrix multiplication [5] and simultaneous triple matrix multiplication [6]. However, many of the existing methods in mapping algorithms onto multi-dimensional networks are ad-hoc, which take long design time and cannot be developed as part of the CAD tool. The benefits of 3-D VLSI technology and the improvements in timing by higher-dimensional structures can be fully exploited only if we can devise a systematic algorithm transformation and mapping technique, which is the scope of this paper.

With the advent of silicon-on-insulator (SOI) technologies, 3-D circuitry are being realized using techniques such as laser recrystallization of polysilicon, which allows fabrication of active devices stacked in two or more layers. Some laboratories have already succeeded in producing 3-D circuit cells [7,8,9]. With 3-D VLSI, wire routing becomes easier, more systematic and shorter, due to the use of the third dimension. The interconnect wire length increases at a much slower rate than planar ones. Moreover, the gain in circuit density, which results in saving of materials, has permitted much larger networks to be implemented. The increase in packing density, together with the improvements in wire routing, lead to a decrease of parasitic capacitances in circuit, and hence to an increase in speed. Besides these benefits, design time can also be minimized as wire-routing is easier and more systematic.

The benefits and the reality of 3-D VLSI necessitate having efficient and systematic (automatic) implementation techniques. These technological advances and opportunities, together with the essentialness of CAD tools in array design and the need for high speed computations, represent a challenge in systematic mapping of arithmetic and DSP algorithms onto multi-dimensional arrays [10].

2. REVIEW OF PRIOR ART

The design of systolic arrays requires a fundamental understanding of application, algorithm, and architecture. A survey of literature with respect to systematic methods of mapping and transforming algorithms onto systolic arrays reveals many stimulating and efficient ideas. For example, S.Y.Kung [11,12] proposes a mapping technique based on data dependence graph, signal flow graph and its systolization. Moldovan [13,14] develops a mapping procedure for cyclic loop algorithms based on mathematical linear transformation of index sets and data dependence vectors. Capello [15,16] presents geometric transformation and linear space-time transformation techniques in array design and representation. This provides an insightful look into how several systolic designs of the same algorithm relate to each other. Leiserson [17] provides a systolization scheme for minimizing the number of delay elements. Quinton [18] produces a systematic method for mapping algorithms that can be expressed by a set of uniform recurrence equations. The method uses a timing function and an allocation function to map these equations onto a finite architecture.

Many of these authors have proposed procedures for systematically mapping an iterative algorithm defined over a multi-dimensional index-space onto a lower-dimensional array of processors, using linear transformations. They restrict their attention to one-dimensional projection so that if the index-space is N-dimensional, then the systolic array is (N-1)-dimensional, with one dimension for time.

In this paper, an attempt is made to increase the dimension of the index-space for certain class of iterative algorithms in a systematic way in order to achieve higher parallelism without increasing area (silicon chip area) complexity. The method first transforms the algorithm by increasing the index-space from N-dimension to M-dimension (M>N). Mapping of the algorithm with M-dimensional index-space can be obtained by combining the systolic array for the same algorithm with N-dimensional index-space, or can be realized by many of the linear transforming techniques mentioned. The technique is particularly suitable for many DSP and arithmetic algorithms. The resulting systolic network is (M-1)-dimensional (> N-1) usually. By doing so, the computation time, which can be defined as the time interval between loading the first input and unloading the last output of a problem instance into/from the array, and its order of complexity can be significantly improved while keeping the number of processing cells (which is the silicon chip area complexity in many cases) constant. The price to be paid is the small amount of additional circuitry (usually in the form of adders and interconnection wires) required for inter-row or inter-plane communications. The multi-dimensional systolic network can be laid out by either 2-D or 3-D VLSI chip [1,2].

Most of the currently available methods that do implement algorithms by high-dimensional systolic networks to achieve higher parallelism are based on ad-hoc procedures. Having a systematic mapping algorithm for multi-dimensional network is not an easy task and it can be an NP-complete problem due to the diverse and several factors and constraints controlling the mapping process. However, it has the benefits of reducing design time and producing efficient mappings. Moreover, it can also be incorporated into an integrated CAD tool (an array compiler [12], for example) for automated array design. A systematic method to transform and map a class of algorithms to high-dimensional network, called STAMS (Systematic Transformation of Algorithms for Multi-dimensional Systolic arrays), is presented in this paper. STAMS technique is presented in Section 3. Two application examples to illustrate this transformation technique are presented in Sections 4 and 5.

3. STAMS: SYSTEMATIC TRANSFORMATION OF ALGORITHMS FOR MULTI-DIMENSIONAL SYSTOLIC ARRAYS

The kind of algorithms that is considered for STAMS technique is especially common in many arithmetic and DSP applications. It is of the form

$$y_r = \sum_{i=0}^{k-1} f(w,i)*g(x,r,i)*... \qquad (1)$$

where "*" indicates multiplication, f(w,i) represents a function f with a variable w and an index i of w (i can form the subscript or the power of w), and g(x,r,i) represents a function g with a variable x and indices r and i associated with x (r and i can form the subscript or the power of x). The index r is also the subscript of y. Examples of such algorithms are:

(1) Matrix-vector multiplication: $y_r = \sum_{i=0}^{k-1} a_{r,i+1}*b_{i+1}$

(2) 1-D convolution: $y_r = \sum_{i=0}^{k-1} w_{i+1}*x_{r+i}$

(3) k-point Discrete Fourier Transform (DFT):

$$y_r = \sum_{i=0}^{k-1} x_i*w^{r*i}$$

(4) k-tap finite impulse response (FIR) filter:

$$y_r = \sum_{i=0}^{k-1} w_i*x_{r-i}$$

Computation of these algorithms are conventionally carried out by 1-D systolic arrays of k cells, as shown in Fig.1. These 1-D systolic arrays can be obtained by many linear mapping procedures listed in Section 2. The 1-D systolic array realization of the matrix-vector multiplication, the 1-D convolution, the k-point DFT and the k-tap FIR filter problems are given in [12,19], [12,20], [21] and [21], respectively. STAMS technique tc obtain the corresponding multi-dimensional systolic arrays is described in the next three subsections.

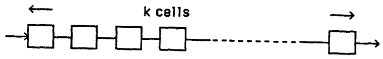


Fig. 1 A 1-D systolic array of k cells

3.1 Derivation of 2-D Arrays using STAMS

In Eq.(1), if k is not a prime number, it can then be expressed as a product of two integers p and q (i.e. k = p*q). Let ij = q*i+j and rt = p*r+t, Eq.(1) can then be rewritten (transformed) as

$$y_{rt} = \sum_{i=0}^{p-1}\sum_{j=0}^{q-1} f(w,ij)*g(x,rt,ij)*... \qquad (2)$$

The index space is hence increased. If the original problem using Eq.(1) requires the computations of y_r, r = 0,1,...,k-1 (i.e. y_{rt}, rt = 0,1,...,k-1), the same problem using Eq.(2) will require the computations of y_{rt} for r = 0,1,...,q-1, and for each r, t = 0,1,...,p-1. Different mathematical expressions of Eq.(2) can be exploited to select a suitable or efficient expression for implementation. Step by step sequential algorithm for computing Eq.(2) can then be developed and mapped onto a 2-D systolic array, usually of p rows, each with q cells, as shown in Fig.2a. The position of each cell is indicated by ij (ith row and jth column, starting from 0), and its corresponding position in the 1-D array is indicated by q*i+j. In the 2-D array, computations in the rows and/or the columns can be carried out in parallel to improve computation speed. The 2-D array consists of p*q (=k) cells and therefore the area complexity is not increased compared to that of the 1-D array, except that a small amount of inter-row communication circuits are added.

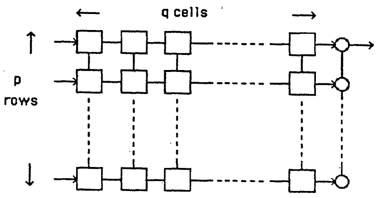


Fig. 2a A 2-D systolic array of k (= p*q) cells

3.2 Derivation of 3-D Arrays using STAMS

If k can be expressed as a product of many integers, the algorithm can then be directly mapped onto a higher-dimensional systolic network. For example, if k = p*q*s, let ijm = q*s*i+s*j+m and rtu = q*p*r+p*t+u, Eq.(1) can be transformed to

$$y_{rtu} = \sum_{i=0}^{p-1}\sum_{j=0}^{q-1}\sum_{m=0}^{s-1} f(w,ijm)*g(x,rtu,ijm)*... \qquad (3)$$

The index-space is thus increased further. If the original problem using Eq.(1) requires the computations of y_r, r = 0,1,...,k-1 (i.e. y_{rt}, rt = 0,1,...,k-1), the same problem using Eq.(3) will require the computations of y_{rtu} for r = 0,1,...,s-1, and for each r, t = 0,1,...,q-1, and for each t, u = 0,1,...,p-1. Different mathematical expressions of Eq.(3) can be exploited to select a suitable or efficient expression for implementation. Step by step sequential algorithm for computing Eq.(3) can then be developed and mapped onto a 3-D systolic array of p planes, each with q rows of s cells each, as shown in Fig.2b. The position of each cell is indicated by ijm (ith plane, jth row and mth column) with its corresponding position in the 1-D array indicated by q*s*i+s*j+m. Higher speed can be achieved by parallel computations in the planes and in the rows. The 3-D array consists of p*q*s (=k) cells and therefore the area complexity is not increased compared to that of the 1-D array.

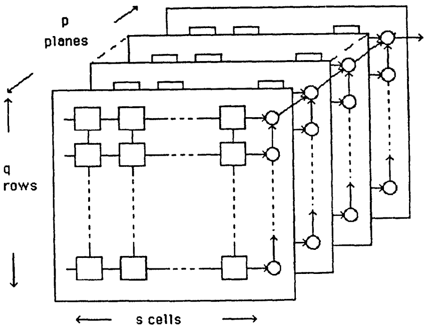


Fig. 2b A 3-D systolic array of k (=p*q*s) cells

3.3 Other Considerations

4-D or higher-D systolic networks can also be mapped by the similar extensions in STAMS technique and by further increasing the index space dimension. In general, systolic network with higher dimension produces higher computation speed at the expense of more communication circuitry. Moreover, the number of processing cells will not be a good measure of area complexity since laying out a 4-D or higher-D systolic network on 2-D or 3-D VLSI will have an area complexity higher than the number of processing cells due to additional interconnections required. An optimal trade off among area, time and layout complexity is thus necessary to achieve an efficient network.

Besides these, different values of p, q, s, ..., can also be used to achieve the best trade-off. On the implementation-level, efficient techniques for laying out multi-dimensional systolic arrays onto 2-D or 3-D VLSI chips must be devised so that the length of interconnection wires, propagation delays, and synchronization problems, can be kept to the minimum.

Combining the STAMS technique just discussed with a standard mapping methodology (for example, the canonical mapping method developed by S.Y.Kung [11,12]), a complete procedure for mapping algorithms (of Eq.(1) form) onto multi-dimensional systolic arrays can be developed. An example is shown briefly in Fig.3. The entire procedure is systematic and can be incorporated into an integrated software CAD package (array compiler) for user-friendly interactive design.

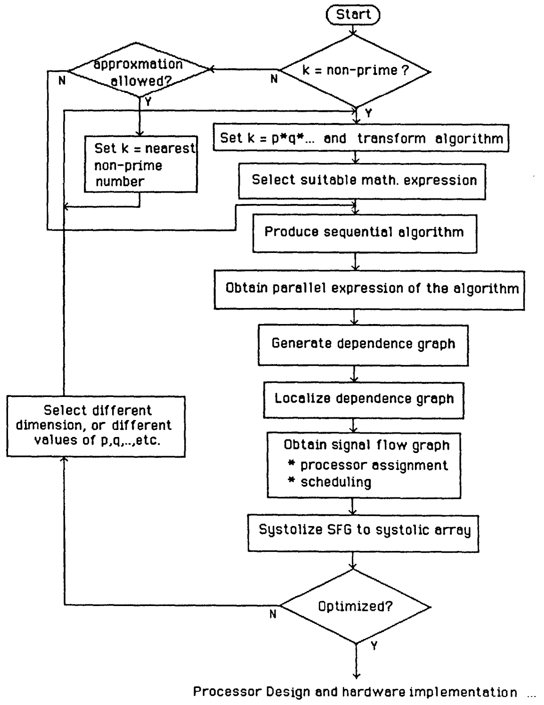

Start
approxmation allowed?
k = non-prime ?
N
N
Y
Y
Set k = nearest non-prime number
Set k = p*q*... and transform algorithm
Select suitable math. expression
Produce sequential algorithm
Obtain parallel expression of the algorithm
Generate dependence graph
Localize dependence graph
Select different dimension, or different values of p,q,..,etc.
Obtain signal flow graph
* processor assignment
* scheduling
Systolize SFG to systolic array
Optimized?
N
Y
Processor Design and hardware implementation ...

Fig. 3 Procedure for mapping algorithms onto multi-dimensional systolic arrays

In this paper, the STAMS technique is applied to the matrix-vector multiplication problem and the k-point DFT problem. It can also be applied to many other problems such as the 1-D convolution problem and the FIR filter problem. In all cases, computational times are improved without increasing the number of processing cells.

4. MATRIX-VECTOR MULTIPLICATION

The matrix-vector multiplication problem A * b for matrix A and vector b is defined as follows:

Given $b_1,b_2, \ldots , b_k$ and

$a_{11},a_{12}, \ldots , a_{1k},a_{21},a_{22}, \ldots , a_{kk};$

compute $y_1,y_2, \ldots , y_k$

defined by $y_r = \sum_{i=0}^{k-1} a_{r,i+1}*b_{i+1}$ (4)

The above algorithm has a 2-dimensional index space (r and i). A linear 1-D systolic network for implementing matrix-vector multiplication can be produced as given in Fig.4 [12]. All y_r's are initialized to zeros and all b_i's are preloaded into the cells. a's inputs are skewed as shown in the figure. Computations are pipelined and the results start to appear at the output of cell b_k k cycles after the first input, followed by a new output every cycle. The last result appears 2*k-1 cycles after the first input, giving the total computation time of 2*k-1 cycles.

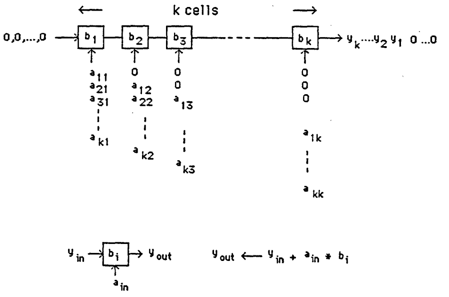


Fig. 4 1-D systolic array for matrix-vector multiplication and its cell definition

4.1 Derivation of 2-D Array using STAMS

If k is not a prime number and can be expressed as k = p*q, using STAMS technique, we let ij = q*i+j and rt = p*r+t and transform Eq.(4) to

$$y_{rt} = \sum_{i=0}^{p-1}\sum_{j=0}^{q-1} a_{rt,ij+1} * b_{ij+1} \qquad (5)$$

It now has a 3-dimensional index space (rt, i, and j). The result y_{rt} can be computed by the following algorithm:

(1) Compute $y_{rti} = \sum_{j=0}^{q-1} a_{rt,ij+1} * b_{ij+1}$;

which is the matrix-vector multiplication of size q.

(2) Compute $y_{rt} = \sum_{i=0}^{p-1} y_{rti}$;

which is the sum of (1) for i = 0,1,...,p-1.

Each y_{rti} can be computed by a linear 1-D systolic array, same as that of Fig.4, except for a smaller size q. The results are then summed up by adders to produce y_{rt}. Thus a 2-D systolic array consists of p rows, each with q columns, as shown in Fig.5 is obtained for executing the algorithm. The position of each cell is given by ij (ith row and jth column, starting from 0).

Computations in the rows are carried out in parallel to improve computation speed. The first result will appear q cycles after the first input. The last result will appear q+k-1 cycles after the first input, giving a computation time of q+k-1 cycles. Compared to the original 1-D array of Fig.4, the 2-D array consists of the same number of cells (k cells), i.e. same area complexity of O(k), whereas the computation time is improved by (k-q) cycles. The price paid is the small amount of area and time overhead of the additional adders. To improve the efficiency further, efficient multi-operand carry save adders, for instance, can be used. We are currently in the process of laying out such a 1-D and the corresponding 2-D systolic array for matrix-vector multiplication using NORA CMOS bit-parallel logic structure and CMOS p-well process technology.

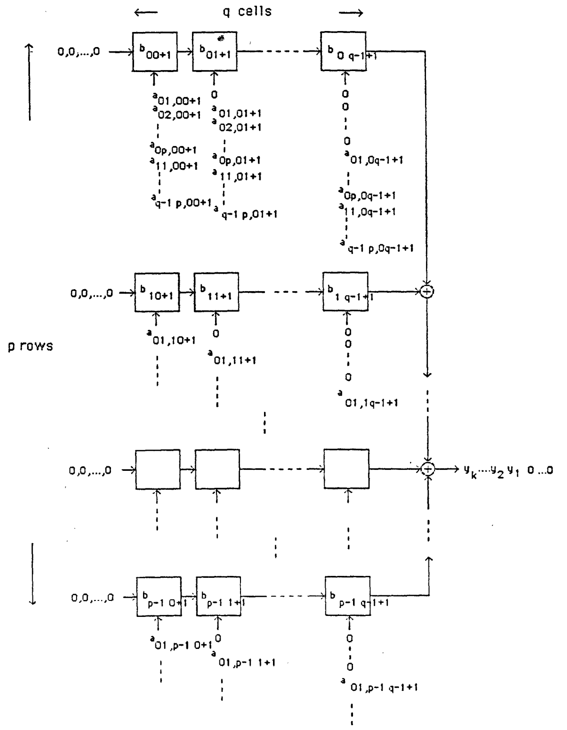


Fig. 5 2-D systolic array for matrix-vector multiplication

4.2 Derivation of 3-D Array using STAMS

If k = p*q*s, using STAMS technique, we let ijm = q*s*i + s*j + m and rtu = q*p*r + p*t + u, and transform Eq.(4) to

$$y_{rtu} = \sum_{i=0}^{p-1}\sum_{j=0}^{q-1}\sum_{m=0}^{s-1} a_{rtu,ijm+1} * b_{ijm+1} \qquad (6)$$

This has a 4-dimensional index space (rtu, i, j, and m). The result y_{rtu} can be computed by the following algorithm:

(1) Compute $y_{rtuij} = \sum_{m=0}^{s-1} a_{rtu,ijm+1} * b_{ijm+1}$;

which is the matrix-vector multiplication of size s.

(2) Compute $y_{rtui} = \sum_{j=0}^{q-1} y_{rtuij}$;

which is the sum of (1) for j = 0,1,...,q-1.

(3) Compute $y_{rtu} = \sum_{i=0}^{p-1} y_{rtui}$;

which is the sum of (2) for i = 0,1,...,p-1.

Each y_{rtuij} can be computed by a 1-D systolic array of size s. The results are summed by adders in parallel to produce y_{rtui}'s, which are further summed to produce y_{rtu}. Thus a 3-D systolic array of p planes, each with q rows of s cells each, as shown in Fig.6, can be obtained to execute the algorithm. The position of each cell is given by ijm (ith plane, jth row and mth column).

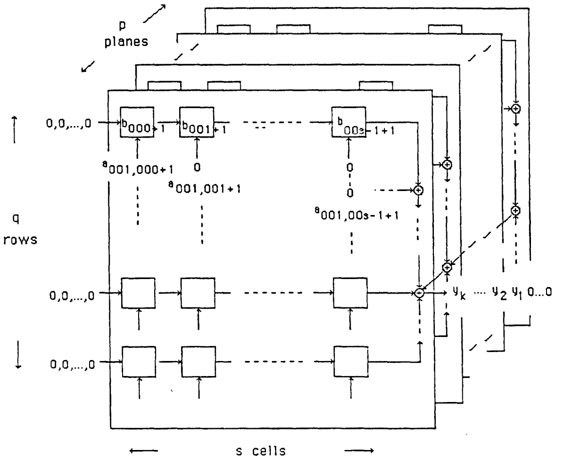


Fig. 6 3-D systolic array for matrix-vector multiplication

Computations in the planes and the rows are carried out in parallel. The summing processes in the planes are also carried out in parallel. These parallelisms improve the computation speed. The resulting outputs start to appear s cycles after the first input. The last output appears s+k-1 cycles after the first input, giving a computation time of s+k-1 cycles. Compared to the original 1-D array of Fig.4, the 3-D array has the same area complexity (k cells) of O(k) with computation time improvement of (k-s) cycles. This is also faster than the 2-D array case, in general, with more communication circuitry required.

4.3 Derivation of Higher-D Arrays using STAMS

Higher-D systolic networks can be similarly obtained by the application of STAMS technique, which will give further improvements in computation time at the expense of more communication circuitry. The final structure of the network can be the interconnections of modules of 3-D or higher-D array. However, laying out a higher-D systolic network on 2-D or 3-D VLSI will give an area complexity higher than O(k) due to additional interconnections, which may be undesirable.

5. DISCRETE FOURIER TRANSFORM

The k-point Discrete Fourier Transform (DFT) problem is defined as follows:

Given $x_0, x_1, \dots, x_{k-1}$;

compute $y_0, y_1, \dots, y_{k-1}$

defined by $y_r = \sum_{i=0}^{k-1} x_i * w^{r*i}$ (7)

where w is an nth root of unity.

The k-point DFT can be viewed as that of evaluating the polynomial

$$x_{k-1}*w^{k-1} + x_{k-2}*w^{k-2} + \cdots + x_1*w + x_0$$

by Horner's rule:

$$(...((x_{k-1}*w + x_{k-2})*w + x_{k-3})*w + \cdots x_1)*w + x_0$$

The computations of y_0, y_1, y_2, ..., y_{k-1} are carried out using the above formula with w replaced by 1, w, w^2,..., w^{k-1} respectively. A linear 1-D systolic network to implement k-point DFT is shown in Fig.7 [21]. The 1-D network consists of k-1 cells (area complexity = O(k)). Computations are pipelined and the results start to appear k-1 cycles after the first input, followed by a new output every cycle. The total computation time is 2*k-2 cycles or O(k).

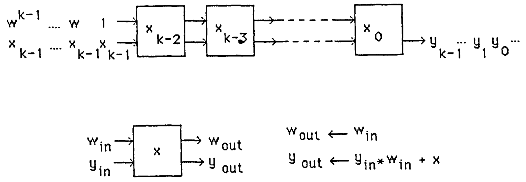

w^{k-1} .... w 1
x_{k-1} .... x_{k-1} x_{k-1}
x_{k-2}
x_{k-3}
x_0
y_{k-1} ... y_1 y_0 ...
w_{in}
y_{in}
x
w_{out}
y_{out}
$w_{out} \leftarrow w_{in}$
$y_{out} \leftarrow y_{in} * w_{in} + x$

Fig. 7 1-D systolic array for k-point DFT and its cell definition

5.1 Derivation of 2-D Array using STAMS

If k=p*q, using STAMS technique, we let ij=q*i+j and rt=p*r+t and transform Eq.(7) into

$$y_{rt} = \sum_{i=0}^{p-1}\sum_{j=0}^{q-1} x_{ij}*w^{(p*r+t)*(q*i+j)}$$

Different mathematical expressions of the above equation can be exploited to produce Eq.(8) for implementation, as shown below:

$$y_{rt} = \sum_{i=0}^{p-1}\sum_{j=0}^{q-1} x_{ij}*w^{p*q*r*i}*w^{q*t*i}*w^{(p*r+t)*j}$$

$$= \sum_{i=0}^{p-1}\sum_{j=0}^{q-1} x_{ij}*w^{q*t*i}*w^{(p*r+t)*j} \qquad \text{(since } w^{p*q}=w^k=1\text{)}$$

$$= \sum_{j=0}^{q-1}\sum_{i=0}^{p-1} x_{ij}*w^{q*t*i}*w^{(p*r+t)*j}$$

$$= \sum_{j=0}^{q-1} w^{(p*r+t)*j} * \sum_{i=0}^{p-1} x_{ij}*w^{q*t*i}$$

$$= \sum_{j=0}^{q-1} w^{p*r*j} * (w^{t*j} * (\sum_{i=0}^{p-1} x_{ij}*w^{q*t*i})) \qquad (8)$$

The result y_{rt} can be computed by the following algorithm, express informally as:

(1) Compute $v_{tj} = \sum_{i=0}^{p-1} x_{ij}*w^{q*t*i}$; a p-point DFT.

(2) Compute $z_{tj} = w^{t*j}*v_{tj}$; a multiplication.

(3) Compute $y_{rt} = \sum_{j=0}^{q-1} z_{tj}*w^{p*r*j}$; a q-point DFT.

A 2-D systolic array of p rows and q columns is thus obtained to compute y_{rt}'s. Each column corresponds to a linear 1-D systolic array of size p, with similar structure as that of Fig.7, for computing the p-point DFT. The multiplication is achieved through multipliers. Registers are used for storing the resulting z_{tj}'s. Each row of the 2-D network corresponds to a linear 1-D systolic array of size q, with similar structure as that of Fig.7, for computing the q-point DFT.

The resulting 2-D systolic array of p rows and q columns is shown in Fig.8. All communications are local. The position of each cell is given by ij (ith row and jth column, starting from 0). The cell definition is given in Fig.9. The operation of the cell is controlled by the control input C_{in}. It can perform the same function as that of the 1-D array cell in both the vertical and horizontal directions. Initially all C_{in}'s are set to 1's, the DFT of the columns are first computed in parallel by the left half of the cells, which give the results v_{tj}'s. Multiplications by w^{t*j}'s to form z_{tj}'s are then performed at the bottom row and the results are fed back and stored in the right half of the cells. Finally, all C_{in}'s are set to 0's and the DFT of the rows are computed in parallel to obtain y_{rt}'s.

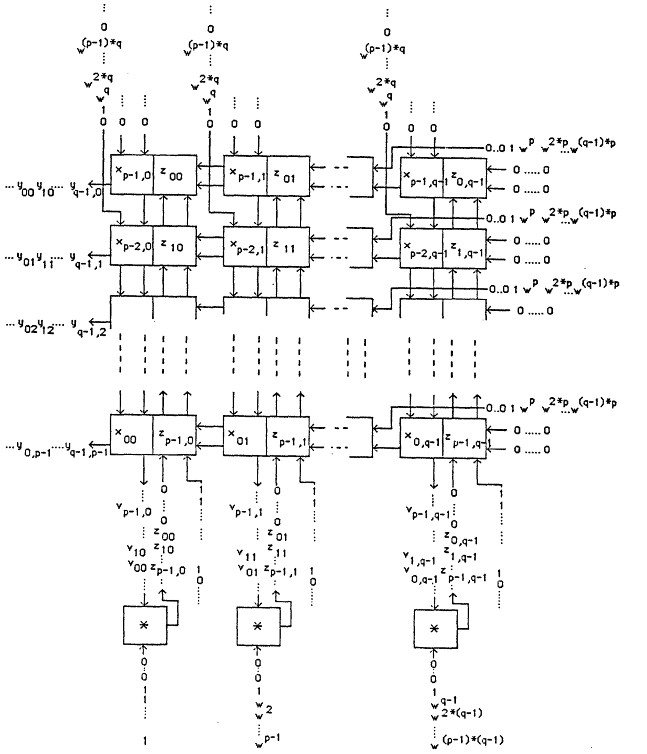
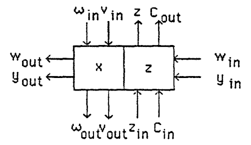


Fig. 8 2-D systolic array for k-point DFT

The 2-D array consists of p*q=k cells with area complexity O(k), which is the same as that of the original 1-D array. However, the computation time has been significantly improved. The first results y_{0t}'s appear after 2*p+q+1 cycles, an improvement of k-(2*p+q)-2 cycles. More importantly, total computation time is improved to 2*(p+q) cycles, which is O(p) or O(q), whichever is larger. Compared to the O(k) (= O(p*q)) computation time of 1-D array, this is a significant improvement. If $p=q=k^{1/2}$, computation time is improved from O(k) to $O(k^{1/2})$.



$C_{in}= 1$:
$v_{out} \leftarrow v_{in} * \omega_{in} + x$;
$\omega_{out} \leftarrow \omega_{in}$;
$C_{out} \leftarrow C_{in}$;
$z \leftarrow z_{in}$;
$y_{out} \leftarrow 0$;
$w_{out} \leftarrow w_{in}$

$C_{in} = 0$:
$y_{out} \leftarrow y_{in} * w_{in} + z$;
$w_{out} \leftarrow w_{in}$;
$C_{out} \leftarrow C_{in}$;
$z \leftarrow z$;
$v_{out} \leftarrow 0$;
$\omega_{out} \leftarrow \omega_{in}$;

Fig. 9 Cell definition for 2-D systolic array implementation of k-point DFT

5.2 Derivation of 3-D Array using STAMS

If k = p*q*s, using STAMS technique, we let ijm = q*s*i + s*j + m and rtu = q*p*r + p*t + u, and transform Eq.(7) to

$$y_{rtu} = \sum_{i=0}^{p-1}\sum_{j=0}^{q-1}\sum_{m=0}^{s-1} x_{ijm}*w^{(q*p*r+p*t+u)*(q*s*i+s*j+m)}$$

This can be further expressed as:

$$= \sum_{i=0}^{p-1}\sum_{j=0}^{q-1}\sum_{m=0}^{s-1} x_{ijm}*w^{q*s*u*i+p*s*t*j+s*u*j+p*q*r*m+p*t*m+u*m}$$

(since $w^{p*q*s} = w^{k} = 1$)

$$= \sum_{m=0}^{s-1}\sum_{j=0}^{q-1}\sum_{i=0}^{p-1} x_{ijm}*w^{q*s*u*i+p*s*t*j+s*u*j+p*q*r*m+p*t*m+u*m}$$

$$= \sum_{m=0}^{s-1} w^{p*q*r*m}*(w^{p*t*m}*w^{u*m}*$$

$$(\sum_{j=0}^{q-1} w^{p*s*t*j}*(w^{s*u*j}*(\sum_{i=0}^{p-1} x_{ijm}*w^{q*s*u*i})))) \qquad (9)$$

The result y_{rtu} can be computed by the following algorithm expressed informally as:

(1) Compute $a_{ujm} = \sum_{i=0}^{p-1} x_{ijm}*w^{q*s*u*i}$; a p-point DFT.

(2) Compute $b_{ujm} = w^{s*u*j}*a_{ujm}$; a multiplication.

(3) Compute $v_{tum} = \sum_{j=0}^{q-1} b_{ujm}*w^{p*s*t*j}$; a q-point DFT.

(4) Compute $z_{tum} = w^{p*t*m}*w^{u*m}*v_{tum}$; a multiplication.

(5) Compute $y_{rtu} = \sum_{m=0}^{s-1} z_{tum}*w^{p*q*r*m}$; an s-point DFT.

Similar to the 2-D systolic array shown, a 3-D systolic array of p planes, each with q rows and s columns can thus be obtained to compute y_{rtu}'s. The 3-D array consists of several 2-D arrays with similar structure as that of Fig.8, for computing the p-point DFTs, the q-point DFTs, the s-point DFTs and the multiplications.

The resulting 3-D systolic array of p planes, q rows and s columns is shown in Fig.10a, Fig.10b and Fig.10c. All communications are local. The position of each cell is given by ijm (ith plane, jth row and mth column, starting from 0). The cell definition is given in Fig.11. The operations of each cell are controlled by the control inputs $C1_{in}C2_{in}$. Each cell can perform the same function as that of the 1-D array cell in all the three directions. The operation of the array is a similar extension of the 2-D case. Initially, all $C1_{in}C2_{in}$'s are set to 01's, the DFTs along the direction perpendicular to the planes are first computed in parallel by the left half of the cells, which give the results a_{ujm}'s. Multiplications by w^{s*u*j}'s to form b_{ujm}'s are then performed and the results are fed back and stored in the right half of the cells. These are shown in Fig.10b. $C1_{in}C2_{in}$'s are then set to 10's and the DFT of the columns are then computed in parallel by the right half of the cells to give v_{tum}'s. Multiplications by w^{p*t*m}'s and w^{u*m}'s to form z_{tum}'s are then performed at the top of the network and the results are fed back and stored in the left half of the cells. Finally, $C1_{in}C2_{in}$'s are set to 11's, the DFT along the rows are computed in parallel by the left half of the cells to obtain y_{rtu}'s. These are shown in Fig.10c.

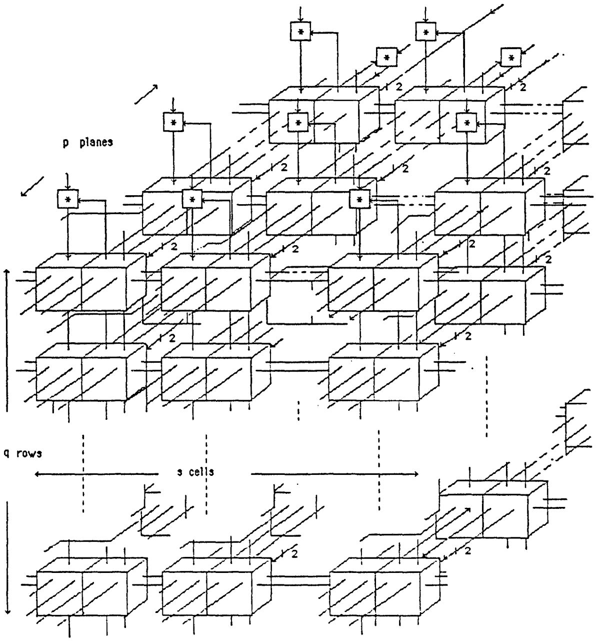


Fig. 10a 3-D systolic array for k-point DFT
(showing the structure)

The 3-D array consists of p*q*s=k cells with area complexity O(k), which is the same as that of the 1-D array. However, the computation time has been significantly improved. The first parallel outputs y_{0tu}'s appear after 2*p+2*q+s+2 cycles, an improvement of k-(2*p+2*q+s)-3 cycles. More importantly, total computation time is improved to 2*(p+q+s)+1 cycles, which is O(p), or O(q), or O(s), whichever is the largest. Compared to the O(k) (=O(p*q*s)) computation time of 1-D array, this is a significant improvement. If $p=q=s=k^{1/3}$, the computation time complexity is improved from O(k) to O($k^{1/3}$). This is also faster than implementing the algorithm by 2-D network, but more communication circuitry are required.

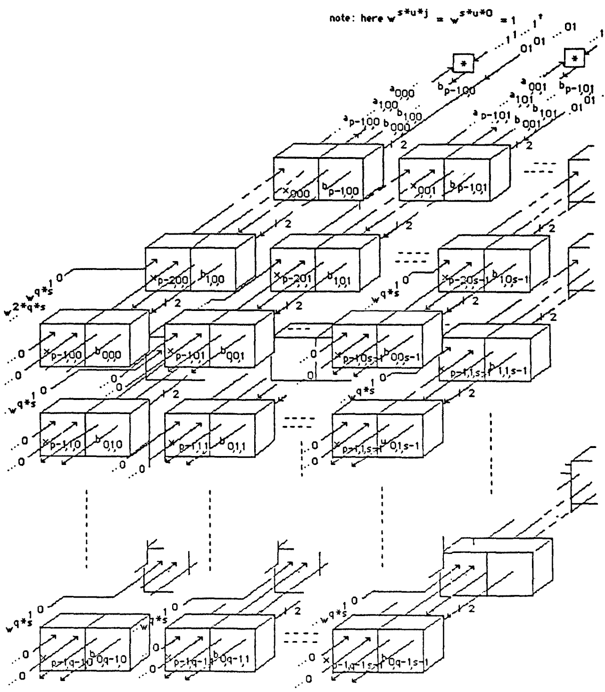


Fig. 10b 3-D systolic array for k-point DFT
(showing for T= 0 to 2*p+1 cycles)

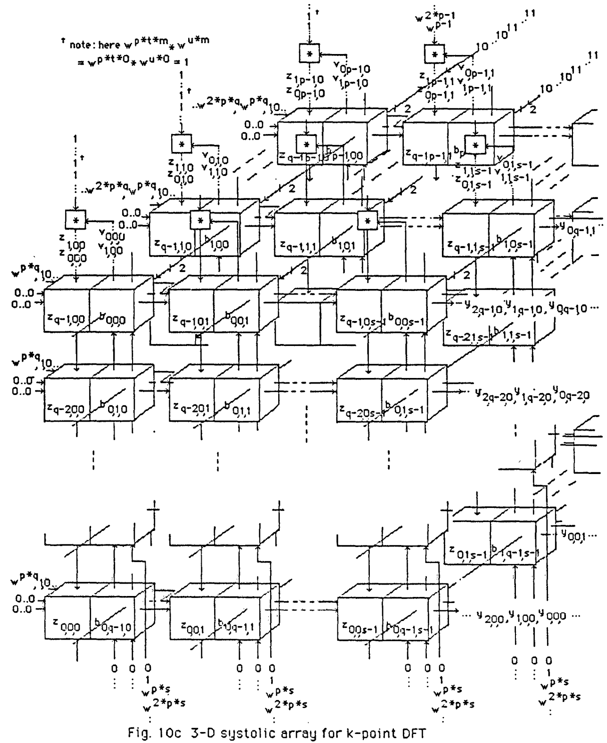
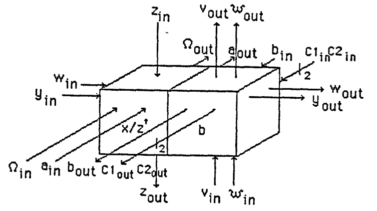


Fig. 10c 3-D systolic array for k-point DFT
(showing for T = 2*p+2 to 2*(p+q+s)+1 cycles)

5.3 Derivation of Higher-D Arrays using STAMS

4-D or higher-D systolic networks can be similarly obtained by the application of STAMS technique, which give further improvements in computation time at the expense of more communication circuitry. However, laying out these networks on 2-D or 3-D VLSI will give area complexities higher than O(k) due to additional interconnections, which may be undesirable.



† cell storage denoted by x when $C1_{in}C2_{in} = 01$; denoted by z otherwise.

$C1_{in}C2_{in} = 00$: No Operation

$C1_{in}C2_{in} = 01$:
$a_{out} \leftarrow a_{in} * \Omega_{in} + x$;
$\Omega_{out} \leftarrow \Omega_{in}$;
$b \leftarrow b_{in}$;
$C1_{out}C2_{out} \leftarrow C1_{in}C2_{in}$;
no operation on other signals;

$C1_{in}C2_{in} = 10$:
$v_{out} \leftarrow v_{in} * w_{in} + b$;
$w_{out} \leftarrow w_{in}$;
$z \leftarrow z_{in}$;
$C1_{out}C2_{out} \leftarrow C1_{in}C2_{in}$;
no operation on other signals;

$C1_{in}C2_{in} = 11$:
$y_{out} \leftarrow y_{in} * w_{in} + z$;
$w_{out} \leftarrow w_{in}$;
$C1_{out}C2_{out} \leftarrow C1_{in}C2_{in}$;
no operation on other signals;

Fig. 11 Cell definition for 3-D systolic array implementation of k-point DFT

6. CONCLUSIONS

In general, there is no optimal interconnection topology for all algorithms, but, it depends on several factors such as the application, the data flow and the available layout technology. Having a systematic mapping for algorithms onto multi-dimensional arrays is an important asset for an efficient implementation and can be developed as part of a CAD tool. The STAMS technique presented in this paper produces systolic arrays with significant improvements in computation time and its order of complexity while keeping the number of processing cells constant.

The technique is useful in many arithmetic and DSP applications. Two examples, the matrix-vector multiplication problem and the k-point DFT problem, have been given to demonstrate the proposed transformation and mapping procedure. Significant improvements in computation time are achieved. This procedure can also be applied to many other algorithms such as the 1-D convolution and the FIR filter.

Even though better computation time can be produced by increasing the dimension of the network to go beyond 3-D, the additional circuitry required for interplane communications and the length of the interconnection wires due to layout on a lower dimensional VLSI are also increased. An optimal trade-off (the method of which is beyond the scope of this paper) among area, time and layout complexity will be helpful in producing reliable and efficient implementations.

References

[1] A.L.Rosenberg, "Three-Dimensional VLSI: A Case Study," *Journal of the ACM,*, July 1983.

[2] A.Terao and F.V.d.Wiele, "Purposes of Three-Dimensional Circuits," *IEEE Circuits and Devices Mag.,* Nov. 1987.

[3] R.P.Preparata and J.Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Comm. of ACM,* May 1981.

[4] G.R.Nudd et al., "Three-Dimensional VLSI Architecture for Image Understanding," *Journal of Parallel and Distributed Computing,* vol.2, 1985.

[5] R.W.Linderman and W.H.Ku, "A Three Dimensional Systolic Array Architecture for Fast Matrix Multiplication," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing,* 1984.

[6] G.Panneerselvam, "Three Dimensional Systolic Cubic Architecture for Simultaneous Triple Matrix Multiplication," *Proc. of 1st Int. Conf. on Supercomputing Systems,* Dec. 1985.

[7] S.Akiyama et al., "Multilayer CMOS Device Fabricated on Laser Recrystallized Silicon Islands," *IEDM Tech. Dig.,* Dec 1983.

[8] Y.Inoue et al., "A Three-Dimensional Static RAM," *IEEE Electron Device Lett.,* May 1986.

[9] K.Mitsuhashi, "Etch Back Planarization Technique and Its Application to Multilayer Devices," *Proc. 4th FED Symp.,* July 1985.

[10] R.D.Etchelles et al., "Development of a Three-Dimensional Circuit Integration Technology and Computer Architecture," *Proc. SPIE,* April 1981.

[11] S.Y.Kung, "VLSI Array Processors," *IEEE ASSP Mag.,* July 1985.

[12] S.Y.Kung, *VLSI Array Processors,* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[13] D.I.Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," *Proc. of the IEEE,* Jan. 1983.

[14] D.I.Moldovan, "ADVIS: A Software Package for the Design of Systolic Arrays," *Proc. of the IEEE Int. Conf. on Computer Design,* 1984.

[15] P.R.Capello and K.Steiglitz, "Unifying VLSI Array Design with Geometric Transformations," *Proc. of the IEEE Int. Conf. on Parallel Processing,* 1983.

[16] P.R.Capello and K.Steiglitz, "Unifying VLSI Array Design with Linear Transformations of Space-Time," *Advances in Computing Research,* vol.2, 1984.

[17] C.E.Leiserson, F.M.Rose, and J.B.Saxe, "Optimizing Synchronous Circuitry by Retiming," *Proc. of the 3rd Caltech Conf. on VLSI,* 1983.

[18] P.Quinton, "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations," *Proc. of the Annual Symp. on Comp. Arch.,* 1984.

[19] C.Mead and L.Conway, *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass., 1980.

[20] H.T.Kung, "Why Systolic Architectures?" *IEEE Computer Mag.,* Jan. 1982.

[21] H.T.Kung, "Special Purpose Devices for Signal and Image Processing: An Opportunity in Very Large Scale Integration (VLSI)," *Proc. of the SPIE,* Real-Time Signal Processing III, July 1980.

A HIGHLY EFFICIENT DESIGN FOR RECONFIGURING THE PROCESSOR ARRAY IN VLSI

Hee Yong Youn and Adit D. Singh
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003

Abstract – *Wafer Scale Integration* of processor arrays for the parallel computation offers important advantages, specifically high performance, low power consumption and high reliability. However, low yield due to the large silicon area is a major problem that remains to be solved. This paper presents a *highly efficient* design for reconfiguring both the *rectangular* and *tree* architecture of processor array when a significant number of processors in the host array are faulty. The proposed scheme always allows the reconfiguration of the maximum size of array with short maximum reconfigured edge length. It also works consistently well even for clustered faulty processors. Comparisons of the proposed design with others in the literature reveal that the improvements are quite substantial. The reconfiguration overhead is also found to be very small.

1 Introduction

Parallel processing using processor arrays is being widely investigated to overcome the performance limitations of traditional uniprocessor computer systems. Some inherent problems with the traditional board level implementation of these systems are separate packaging cost, assembly cost on the printed circuit boards, and low reliability due to the complex pin to pin interconnections on the boards. While all these problems are important, especially significant is the large signal *propagation delay* in MOS VLSI technology required to drive signals off chip. Wafer Scale Integration(WSI)[1] promises a solution to this problem by integrating the entire processor array and the interconnection structure on a single wafer. Thus WSI makes it possible to eliminate the off chip signal drivers within the processor chips, and the complex board level interconnections among the processors. As a result, signal delays can be substantially reduced. In addition, system reliability may also be improved because of elimination of the mechanical and electrical failures frequently observed at the pins and interconnections in traditional designs.

Although WSI has many attractive features, the low yield problem due to the large chip area [2] must be overcome before such circuits can become practical. In conventional VLSI designs, the entire circuit is discarded if it contains even a single defect that is capable of causing a logical fault. For large area circuits, which have a high likelihood of containing at least one defect, this leads to extremely low yield. To overcome this problem, an on-chip fault tolerance scheme, employing redundant components and a reconfigurable interconnection structure is required. Such a scheme can allow proper operation even in the presence of defects. This will increase the yield of 'good' circuits at the manufacturing stage, and can perhaps also be used to increase the reliability of the system in the operating stage.

A number of fault tolerance schemes for VLSI processor arrays[3-9] have been proposed by other researchers. The objective is to reconfigure the failure free processors in the physical array into a desired specific logical computational topology to best match a given parallel algorithm. The effectiveness of such a fault tolerance scheme is generally evaluated on following three criteria.

- Processor utilization - defined by the ratio of the number of processors actually utilized as the computing nodes in the reconfigured array, to the total number of failure free processors actually realized in the physical array. When each processor is relatively large, almost all the chip area is taken up by processors. Then this ratio also reflects the chip area utilization. Because the cost of a VLSI circuit is significantly influenced by the chip area, this factor evaluates the cost effectiveness of the fault tolerance scheme.
- Maximum reconfigured edge length - defined by the maximum distance between any pair of two communicating nodes after the reconfiguration. This factor limits the execution speed of the system, and is particularly critical in systolic designs where the processors operate in tight synchronizations and the system clock must be slowed to accommodate the longest delay. Since the most important benefit obtained from wafer scale integration is perhaps increased performance (because of the elimination of the off chip delay), the maximum reconfigured edge length should be as short as possible.
- Reconfiguration overhead - defined by the overhead required for the reconfiguration such as channel width, number of switches and their complexity.

In this paper, we present a new fault tolerance scheme which enables the efficient embedding of two important computational topologies, namely the rectangular array and complete binary tree on a host array of processors. The proposed scheme maximizes the utilization of the failure free processors in the host array with short maximum reconfigured edge length. It also works consistently well even when the faulty processors in the host array are severely clustered. The reconfiguration overhead is found to be small considering the high efficiency.

The rest of the paper is organized as follows. In Section 2, we present a scheme for reconfiguring the rectangular array when a significant number of processors in the host array are faulty. Also, the interconnection structure of the host array realizing the reconfiguration is presented. The proposed scheme is compared with other designs in Section 3. Section 4 shows how our scheme can also be used to reconfigure the complete binary tree architecture. Section 5 concludes the paper.

2 Design for Rectangular Arrays

In this section, a reconfiguration scheme for embedding the rectangular array on the 2-dimensional processor array with faulty elements is proposed. Also, the interconnection structure of the host array of processors which realizes the reconfiguration is presented.

2.1 Reconfiguration Scheme

In reconfiguring a rectangular array, let us assume that the host array is an N × N rectangular array and a rectangular array with equal sides is desired to be reconfigured. The reconfiguration scheme is now presented for two cases of the number of the faulty processors(F) in the host array; i) F $\leq 2N-1$, ii) F $> 2N$.

2.1.1 Case 1: F $\leq 2N-1$ When the number of faulty processors(F) in the host array is not greater than $2N-1$, maximally $(N-1) \times (N-1)$ processor array can be reconfigured because there exist at least $N^2-(2N-1)=(N-1)^2$ failure free processors. To obtain the desired $(N-1) \times (N-1)$ rectangular array with optimum maximum restructured edge length, the bipartite matching[10] algorithm is employed. For the matching, it is regarded that an $(N-1) \times (N-1)$ logical grid is overlayed on the original host array. Figure 1 shows that a 4×4 logical grid is overlayed on the 5×5 host processor array. It is also assumed that only the four surrounding processors of each grid point can be matched to it. The maximum bipartite graph matching is then sought between the $(N-1)^2$ logical grid points and their neighboring failure free processors in the host array. When the complete matching is accomplished, the $(N-1)^2$ failure free processors in the host array are assigned the logical indices. The desired $(N-1) \times (N-1)$ processor array can be obtained actually by realizing the interconnections of the logically neighboring processors through the interconnection circuitry. The interconnection structure of the host array which realizes the logical reconfiguration will be presented in Section 2.2.

The best known bipartite matching algorithm finds the matching in $O(|V|^{1/2} \bullet |E|)$ time. Here $|V|$ and $|E|$ is the number of vertices and edges in the bipartite matching graph. Because each processor in the host array can be matched to four surrounding logical points and each logical point has four processors assignable to it, the matching is highly flexible. Consequently, it is highly likely that a complete match can be achieved. Computer simulations reveal that the likelyhood of the complete matching (for reconfiguring an 8×8 array) is 98.7% when 10 processors are faulty in the 9×9 host array. It is 93.8% and 28% when 13 and 17 processors are faulty, respectively. Figure 2 demonstrates that a 4×4 array is reconfigured out of a 5×5 host array where 9 processors are faulty. In the figure, the boxes marked as 'X' denote the faulty processors. Two digit number (ij) in each box indicates the row(i) and column(j) of the logical node to which the processor is matched. Note that the maximum reconfigured edge length is very short and bounded to be the length of one side of a processor when the complete matching is possible because the logically neighboring nodes are matched to the failure free processors which are physically neighboring.

The complete matching is not possible when all four processors surrounding a logical grid point are faulty, or the faulty processors are clustered in such a way that all logical grid points cannot be assigned even though there exist enough number of failure free processors, as shown in Figure 3(a). Three logical grid points - (2,2), (2,3) and (2,4) - are not matched the failure free processor due to such fault clustering. Observe that three failure free processors in the fifth column of processors in the host array are remained unused. When complete matching is not possible, the desired size of rectangular array can be reconfigured by increasing the maximum reconfigured edge length as 2 (two times of the length of one side of a processor). Here, the unmatched grid point can be matched a failure free processor by borrowing it from a neighboring logical grid point which was matched a failure free processor. The neighboring logical grid point is now required to borrow other one to make up the processor lent to its neighbor. This process of borrowing is propagated to the grid point which can now be matched the unused failure free processor. We call this process as ***assignment borrowing.*** When multiple assignment borrowings are required due to multiple unmatched logical grid points, then the borrowing paths (horizontal and vertical) between the logical grid points and the unused failure free processors are selected with the constraint that they do not cross each other. Note that the number of unused failure free processors is always greater or equal to that of the unmatched grid points. Then, it is heuristically clear that such non crossing paths between these two sets of points always exist. Figure 3(b) shows the assignment borrowings for three unmatched grid points in Figure 3(a). Observe that the borrowing paths always pass through physically neighboring processors and the faulty processors which do not need to be connected. This ensures that the assignment borrowing can always be realized by a fixed interconnection structure which will be presented in Section 2.2. A reconfigured 4×4 processor array is shown in Figure 3(c).

As can be seen in the examples, the proposed scheme for reconfiguring the rectangular array based on bipartite matching and assignment borrowing always enables us to reconfigure the maximum size of rectangular array with short and bounded maximum reconfigured edges when the number of faulty processors in the host array does not exceed $2N-1$.

2.1.2 Case 2: F $> 2N-1$ When the number of faulty processors(F) exceeds $2N-1$, the maximum size of array that can be reconfigured from the host array can be easily seen to be M × M where M is $\lfloor\sqrt{N^2-F}\rfloor$. The desired M × M rectangular array is proposed to be reconfigured by overlaying M rows and columns of logical grid appropriately on the N × N host array and applying the same bipartite matching and assignment borrowing algorithm as for the Case 1. Recall that an $N \times N$ array is regarded that it contains $N-1$ rows and columns of logical grid points. Therefore, selecting M rows and columns of logical grid points to be matched out of $(N-1)$ row and columns of logical grid points in the host array is equivalent to selecting $(N-1)-M$ rows and columns. The rows and columns to be selected and excluded from the matching are determined by the number of the faulty processors along with each row and column of logical grid points. Thus, for each column of logical grid points, the faulty processors in the column of processors to the left and right of it are counted up. Similarly, for each row, the number is obtained by scanning both the upper and lower row of processors. Then, the row or column of logical grid points with the largest number of faulty processors is first excluded. The faulty processors along with the excluded row or column is now assumed to be failure free because the faults have already been reflected by the exclusion. Next, for each row and column except the excluded one, the number of faulty processors is counted again and the row or column of the largest number is excluded. This procedure is repeated until $(N-1)-M$ rows and columns of logical grid points are excluded in the array. Figure 4(a) and (b) show such exclusions when 16 processors are faulty in a 5×5 host array. In this example, the largest reconfigurable rectangular array is 3×3, and one row and column are required to be excluded.

After the exclusions, the bipartite matching is applied between the selected M × M grid points and their neighboring failure free processors. Also, the assignment borrowing is employed if the complete matching is not possible. Recall that the assignment borrowings are always possible as long as more unused failure free processors than the unmatched logical grid points reside in the host array. This condition has already been guaranteed by excluding $(N-1)-M$ rows and columns of logical grid points in the matching. A 3×3 rectangular array being reconfigured through the matching and borrowing algorithm, after the exclusion of Figure 4(a) and (b), is illustrated in Figure 4(c) and (d). Here the logical grid point (33) is matched a failure free processor by assignment borrowing through the logical grid points (23) and (13). The maximum reconfigured edge length, when some exclusions are necessary, is bounded by the maximum size of consecutive exclusions of row or column of logical grid points.

Another example of reconfiguration, where the faults are clustered severely is shown in Figure 5. Here all 16 processors in the half bottom of the 5×5 host array are faulty. This example shows that the proposed scheme can allow us to reconfigure the maximum size of rectangular array even for such extreme clustering of faulty processors.

We have discussed the reconfiguration of the rectangular array when some processors in the host array are faulty. As shown in both cases of considerations, the proposed reconfiguration scheme can always reconfigure the maximum size of array. The efficiency of the proposed scheme is not influenced by the distribution of the faulty processors and this is one of the most important characteristics of the proposed design.

Next we present the interconnection structure of the host array which realizes the desired rectangular array.

2.2 Structure of Host Array

Each processor in the rectangular array requires four ports for the four directional communications such as North(N), East(E), South(S) and West(W). We put each port at each corner of the processor block as shown in Figure 6. Figure 6 shows the structure of a processor block where an interconnection bus(dotted line) is implemented around it. Recall that a processor can be matched to one of the four neighboring logical grid points reside at upper-right, upper-left, lower-right and lower-left of it. Figure 6(a) shows the interconnection pattern when a processor is matched to the logical grid point at upper-right. The other three types of interconnection pattern are illustrated in Figure 6(b),(c) and (d).

The actual interconnection for reconfiguring the rectangular array is achieved by two steps. First, in each site of processor which has been matched to a logical grid point, appropriate interconnection pattern is realized according to the type of matching. Then, the logically neighboring processors are interconnected each other by connecting two proper ends of connections which are made in the previous step. As shown in Figure 6, a channel width of two is enough to realize almost all kinds of interconnection patterns. One extra bus is required only when two physically and horizontally(vertically) neighboring processors are matched to two vertically(horizontally) neighboring logical grid points between them as shown in Figure 7. The matching of two physically and horizontally neighboring nodes which are matched to the logical grid points (31) and (41), or (14) and (24) in Figure 2 are examples of those requiring an extra vertical bus between processors. However, the matching which requires an extra bus occurs only when the fault distribution does not allow us to avoid such a pattern. Therefore, a channel width of three is always sufficient for realizing all interconnections. Figure 8-11 show the actual interconnections realized for the reconfigured rectangular arrays shown in Figure 2-5, respectively.

In the next section, we compare the efficiency of our design with other classical designs presented in the literature.

3 Comparisons with Other Designs

In this section, the proposed scheme for reconfiguring the rectangular array on the 2-dimensional processor array is evaluated and compared with the other classical designs such as hierarchical scheme[5], column redundant scheme[6] and fault stealing scheme[7]. These schemes are compared on the processor utilization, maximum reconfigured edge length and the reconfiguration overhead.

3.1 Comparison of Processor Utilization

In carrying out the comparison, the objective is to obtain a computational array of fixed desired size$(M \times M)$. Therefore, for each scheme, the optimum size of host array which gives the best processor utilization are found for the given processor yield(P; probability that each processor is failure free) from 0.4 to 0.9 in steps of 0.1.

3.1.1 Proposed Scheme As discussed in the previous section, the proposed scheme can always reconfigure the desired size$(M \times M)$ of array whenever at least M^2 processors in the host array are failure free. Therefore, the yield of an $M \times M$ array out of a $R \times C$ host array is readily seen to be

$$yield = \sum_{i=M^2}^{R\times C} \binom{R \times C}{i} P^i (1-P)^{R\times C-i} \qquad (1)$$

The processor utilization(PU) is then obtained by

$$PU = \frac{M^2 \times yield}{R \times C \times P} \qquad (2)$$

The optimum size of host array, here R and C, which gives the best processor utilization(PU) for the given P(processor yield) is found by the equation (1) and (2).

3.1.2 Hierarchical Scheme This scheme uses redundant submodules[5] for extracting the desired rectangular array. The objective is to ensure, through redundancy, a very high probability that a failure free 2 × 2 submodule can be reconfigured at each submodule site, so that row and column exclusion[4], employed at a higher level to protect against such failures is very rarely needed. Here we need to find the optimum size of submodule ($R \times C$) that guarantees a 2 × 2 failure free processor array. The yield and PU can be shown to be given by

$$yield = (\sum_{i=4}^{R\times C} \binom{R \times C}{i} P^i (1-P)^{R\times C - i})^{(M/2)^2} \qquad (3)$$

$$PU = \frac{4 \times yield}{R \times C \times P} \qquad (4)$$

3.1.3 Column Redundant Scheme An $M \times C(C > M)$ array is used for the reconfiguration of an $M \times M$ rectangular array in this scheme. A failure free linear array of size M is reconfigured out of C processors in each row, and then the desired rectangular array is finally obtained by connecting the M failure free linear arrays vertically through the interconnection channels between each row of processors. The yield and PU can be seen to be given by

$$yield = (\sum_{i=M}^{C} \binom{C}{i} P^i (1-P)^{C-i})^M \qquad (5)$$

$$PU = \frac{M \times yield}{C \times P} \qquad (6)$$

Similarly, the optimum size of column(C) is found using these two equations.

3.1.4 Comparisons Table I(a) and (b) show that the size of host array which gives the best processor utilization for reconfiguring a desired size of array of 8 × 8 and 16 × 16, respectively. Observe that the size of the host array of the proposed scheme is always much smaller than that of the other schemes. Therefore, it can be expected that the processor utilization is much better. Figure 12(a) and (b) show the plots of the processor utilization for both sizes of array, respectively. As expected, the processor utilization of the proposed scheme is much better than that of other designs for the whole range of processor yield and the size of array. It is about 30% and 40% more efficient than the column redundant scheme and the hierarchical scheme, respectively. Because the proposed scheme always reconfigure the maximum size of array, the processor utilization can be said to be near optimal.

The fault stealing schemes proposed in [7] borrow failure free processors from the upper row of processors to replace the faulty processors which cannot be replaced by the redundant processors in the same row. Simulation data from [7] demonstrate that the yield for reconfiguring a 20 × 20 processor array out of a 21 × 21 array is less than 10% when more than 30 processors are faulty. Recall that the yield of proposed scheme is always 100% as far as enough number of processors are failure free (400 in this example).

3.2 Comparison of Maximum Edge Length

As we can see from Table I, the proposed scheme requires the smallest size of the host array to reconfigure a desired size of rectangular array. This means that the maximum reconfigured edge length of our design is smaller than that of other designs because each processor, which is matched to a logical grid point (overlayedon the host array regularly), is the physical neighbor to it and thus they can be regarded to be layed out regularly on the host array. Note that the maximum edge length of a rectangular array in a fixed area is minimal when nodes are equally separated from each other. Also recall that the borrowing process does not affect the reconfigured edge length substantially because the borrowing occurs between two neighboring grid points.

3.3 Comparison of Reconfiguration Overhead

For the proposed scheme, a channel width of three is enough for reconfiguring the rectangular array irrespective of the number of faulty elements in the host array and its distribution. Note that the processor utilization of the other schemes was obtained with the assumption that the sufficient channel was provided for the reconfiguration. For example, from Table I, the column redundant scheme requires an 8 × 18 array to reconfigure an 8 × 8 array most efficiently when processor yield is 0.7. Here, the channel width of at most 10 is required to realize all patterns of the reconfiguration. If the channel width is limited to three, then the yield and the processor utilization decreases significantly.

Also, it can be argued that the channel width of three is relatively small considering the high efficiency of the proposed scheme. Even the simple bypassing scheme[4] requires one bus around each processor (equivalent to a channel width of two) even though its efficiency is known to be very poor. It can also be expected that the efficiency of the proposed scheme will not degrade significantly even with a channel width restricted two.

From the comparisons, we can see that the proposed scheme can always reconfigure the desired size of rectangular array from the smaller host array with high yield (efficiency) and short maximum reconfigured edge length. Also the reconfiguration overhead, measured in terms of channel width, is relatively small. We next present a highly efficient fault tolerant tree embedding scheme that uses the same reconfiguration algorithm and interconnection structure that we have presented for the rectangular array.

4 Design for Tree Architecture

4.1 Tree embedding on Good Processors

A near optimal scheme for embedding a complete binary tree in an array of failure free processors with planar interconnections was proposed in [12], which substantially improves the efficiency of [13]. This scheme adopts a hierarchical strategy such that any required size of tree larger than four levels is embedded by connecting an appropriate *number* and *type* of basic modules shown in Figure 13. Observe that all nodes at adjacent levels (except for two nodes in basic module M2) are physical neighbors and can be connected with short links. 15 out of the 16 processors in the basic module are utilized as the tree nodes in each four level leaf subtree. The remaining unused processor in each basic module is used as a tree node at some higher level, when the basic modules are connected together to build a larger tree. Figure 14 shows a 9 level tree embedded using the basic modules. Because only one processor is always left unutilized in the implementation for any size of tree, the area efficiency quickly converges to 100% as the size of the embedded tree grows large. The propagation delay between the root node and leaf node is also very short. It converges to the *theoretical*lower bound as the size of tree grows large, as proven in [12].

4.2 Fault Tolerant Tree Embedding

A tree architecture can be efficiently embedded in processor array with faulty nodes when the reconfiguration scheme proposed for rectangular array is combined with the tree embedding scheme on failure free processors introduced in the previous subsection. As shown in Figure 14, the tree architecture is constructed using a number of basic modules of a 4 × 4 processor array containing a 4 level subtree. Here, we propose to construct a tree of desired size by reconfiguring each basic module of a 4 level tree efficiently and then connecting them appropriately. To obtain the desired tree architecture, first, a 4 × 4 processor array is reconfigured from the host submodule using the scheme proposed for the rectangular array. Then, the interconnection pattern for each type of basic modules are realized in each submodule. Figure 15 is an example that a basic module of type 2 (M2) can be embedded in a 5 × 5 processor array of host submodule assuming the same fault distribution and the reconfiguration as in Figure 2. In the figure, for every processor in the host submodule, the I/O port for the connection to the parent node is located at the upper-left corner of each processor site. Also, two other ports for children are located at the lower-left and upper-right corner, respectively. Note that this position needs to be shifted 90°, 180° or 270° all together according to the orientation of the basic modules inside of the host array to prevent the interconnections crossing over each other. For instance, the basic module M1 at the upper-left corner of the host array of Figure 14 has 180° shifted I/O ports such that the port for the parent node is located at the lower-right corner of the processor module. The main reason why the hierarchical scheme using submodules with different orientation of ports is employed here is that some interconnections can cross over each other if the three positions of I/O port are same for all the processors in the host array.

4.3 Comparison of Efficiency

Let's denote Y_k as the yield of a k level tree. Then the yield and PU assuming same notations as used in the previous section can be shown to be given by

$$Y_k = \Big(\sum_{i=16}^{R\times C} \binom{R\times C}{i} P^i (1-P)^{R\times C-i}\Big)^{2^{k-4}} \qquad (7)$$

$$PU = \frac{2^k \times Y_k}{R\times C\times 2^{k-4}\times P} \qquad (8)$$

The optimum size of submodule which gives the best PU is found from the above equations. Figure 16 shows the comparison of the processor utilization for embedding an eight level tree with other two designs presented in [14] and [15], respectively. The row exclusion scheme[14] excludes the entire row of processors containing the faulty processor, where the host array is CHiP[9] architecture. In the modular scheme[15], each module which contains a spare processor for the replacement of a faulty processor in that module constructs the whole tree architecture. We can see that, from the figure, the improvement is quite significant. The efficiency of the proposed scheme has also been found to be better than that of other designs such as SOFT[16] and Cluster Proof[17] design. The maximum reconfigured edge length and the reconfiguraion overhead is expected to be smaller than those of other designs due to the high processor utilization and the compact layout of the processor arrays.

5 Conclusion

A highly efficient design for reconfiguring the rectangular array and the binary tree architecture in the presence of a significant number of faults is presented. By employing bipartite graph matching with an assignment borrowing algorithm, the proposed scheme always allows reconfiguration of the maximum possible size of array. Also the maximum reconfigured edge is inherently short. The proposed scheme can reconfigure the desired structure successfully even when the faulty processors in the host array are severely clustered as might be realistically expected.

A heuristic strategy which excludes some row and column of logical grid points on the host array is suggested. Also, multiple paths for the assignment borrowing algorithm are suggested based on planarity considerations. Reconfiguring other important computational topologies using the algorithms proposed in this paper is also under investigation.

References

[1] J.F. McDonald et al., "The Trials of WSI," IEEE Spectrum, pp. 32-39, Oct., 1984.

[2] C.H. Stapper et al., "Integrated Circuit Yield Statistics," in Proc. IEEE, vol. 71, April 1983.

[3] T. Leighton and C.E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," IEEE Trans. Comput. vol. c-34, pp. 448-461, May 1985.

[4] I. Koren, "A Reconfigurable and Fault-tolerant VLSI Multiprocessor Array," in Proc. 8th Annu. Symp. Comput. Arch., pp. 425-431, May 1981.

[5] K.S. Hedlund, "Wafer Scale Integration of Parallel Processors," Tech. Rep. CSD-TR-422, Purdue Univ., 1982.

[6] J.W. Greene, "Configuration of VLSI Arrays in the Presence of Defects," Dissert. for Ph.D. submit. at Stanford Univ., Dec. 1983.

[7] M. Sami and R. Stefanelli, "Reconfigurable Architecture for VLSI Processing Arrays," Proc. IEEE, vol. 74, pp.712-722, May 1986.

[8] A.L. Rosenberg, "The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors," IEEE Trans. Comput., vol. C-32, pp. 902-910, 1983.

[9] L. Snyder, "Introduction to the Configurable Highly Parallel Computer," IEEE Computer, vol. 15, pp. 47-56, Jan. 1982.

[10] C.H. Papadimitriou and K. Steiglitz, "Combinational Optimization: Algorithms and Complexity", chap. 10, Prentice Hall, NJ, 1982.

[11] J.I. Raffel et al., "A Wafer-Scale Digital Integrator Using Restructurable VLSI," IEEE J. of Solid-State Circuits, vol. sc-20, pp. 399-406, Feb. 1985.

[12] H.Y. Youn and A.D. Singh, "Near Optimal Embedding of Binary Tree Architectures In VLSI," to appear in Proc. of the 8th Int'l Conf. Distributed Computing System, June 1988.

[13] H.Y. Youn and A.D. Singh, "On Area Efficient and Fault Tolerant Tree Embedding In VLSI," in Proc. Int'l Conf. Parallel Processing, pp. 171-178, 1987.

[14] H. Mizrahi and I. Koren, "Evaluating the Cost-Effectiveness of Switches in Processor Array Architectures," in Proc. Int'l Conf. Parallel Processing, pp. 480-487, Aug. 1985.

[15] A.S. Mahmudul Hassan and V.K. Agarwal, "A Fault-Tolerant Modular Architecture for Binary Trees," IEEE Trans. Comput., vol. C-35, pp. 356-361, April 1986

[16] Matthew B. Lowrie and W.Kent Fuchs, "Reconfigurable Tree Architectures Using Subtree Oriented Fault Tolerance," IEEE Transaction on Computers, Vol. C-36, pp. 1172-1182, Oct. 1987

[17] M.C. Howells and V.K. Agarwal, "Yield and Reliability Enhancement of Large Area Binary Tree Architectures," Proc. of the 15th Annual Symposium on Fault Tolerant Computing, pp. 290-295, June 1987.

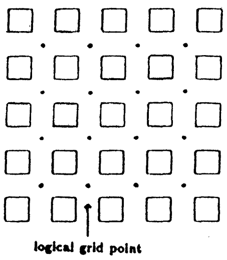


Figure 1. Overlaying a 4 × 4 logical grid on a 5 × 5 host array.

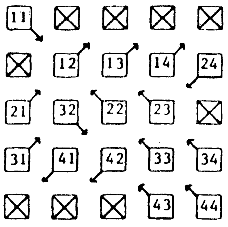


Figure 2. Reconfiguration of a 4 × 4 array using bipartite matching.

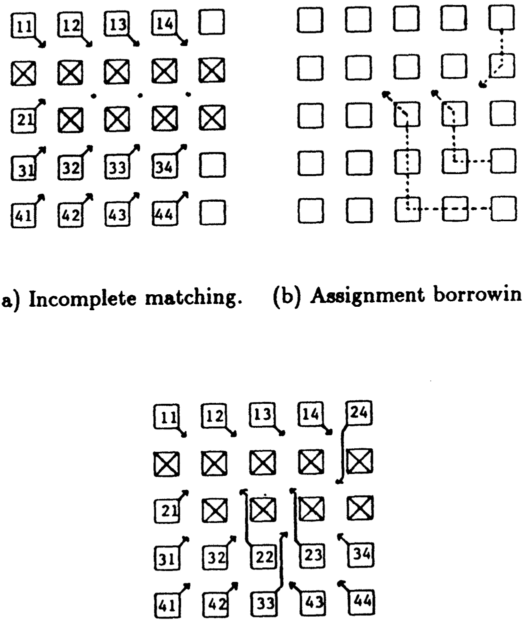
Processor yield	Proposed scheme	Hierarchical scheme	Column redundant scheme
0.9	9 × 9(=81)	8 × 12(=96)	8 × 12(=96)
0.8	9 × 10(=90)	12 × 12(=144)	8 × 14(=112)
0.7	10 × 10(=100)	12 × 12(=144)	8 × 18(=144)
0.6	11 × 11(=121)	12 × 16(=192)	8 × 21(=164)
0.5	12 × 12(=144)	16 × 16(=256)	8 × 27(=216)
0.4	13 × 14(=182)	16 × 20(=320)	8 × 35(=280)

(a). For an 8 × 8 array.

Processor yield	Proposed scheme	Hierarchical scheme	Column redundant scheme
0.9	17 × 18(=306)	24 × 24(=576)	16 × 23(=368)
0.8	18 × 19(=342)	24 × 24(=576)	16 × 27(=432)
0.7	20 × 20(=400)	24 × 32(=768)	16 × 33(=528)
0.6	21 × 22(=462)	32 × 32(=1024)	16 × 40(=640)
0.5	24 × 24(=576)	32 × 40(=1280)	16 × 50(=800)
0.4	27 × 27(=729)	40 × 40(=1600)	16 × 64(=1024)

(b). For a 16 × 16 array.

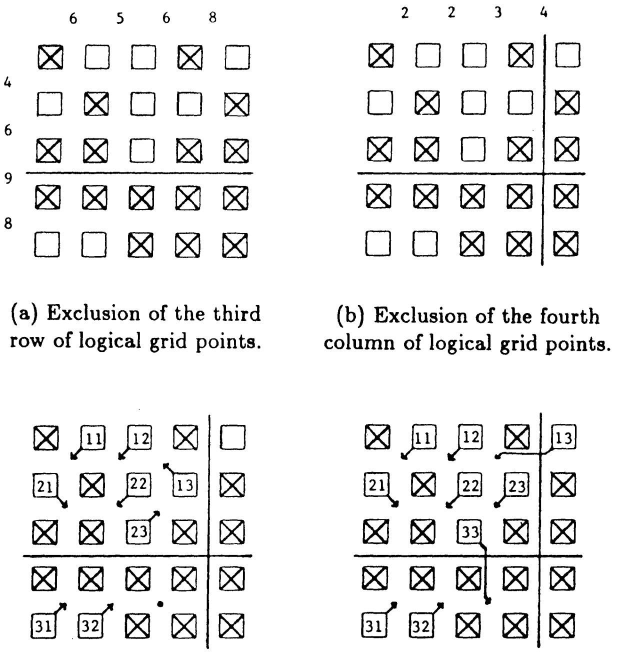
Table I. Size of host array which gives the best processor utilization for reconfiguring a rectangular array.



(a) Incomplete matching. (b) Assignment borrowing.

(c) Final reconfiguration.

Figure 3. Reconfiguration of a 4 × 4 array when the complete matching is not possible.



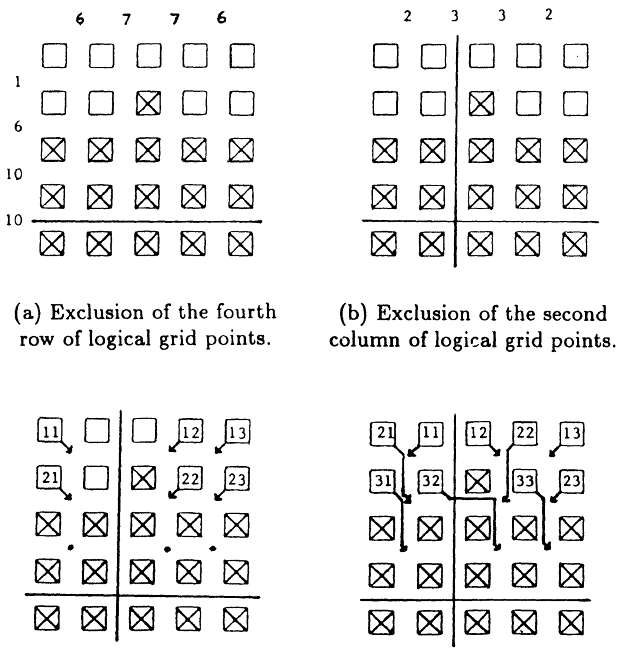
(a) Exclusion of the third row of logical grid points.

(b) Exclusion of the fourth column of logical grid points.

(c) Bipartite matching.

(d) Assignment borrowing.

Figure 4. Reconfiguration of a 3 × 3 array out of a 5 × 5 host array using all the failure free processors.



(a) Exclusion of the fourth row of logical grid points.

(b) Exclusion of the second column of logical grid points.

(c) Bipartite matching.

(d) Assignment borrowing.

Figure 5. Reconfiguration of a 3 × 3 array when all the processors in the bottom half of the host array are faulty.

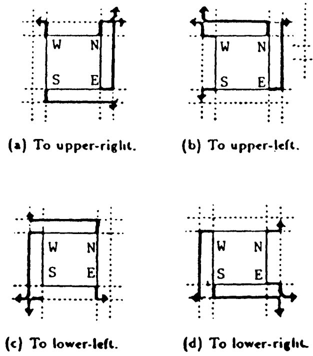


Figure 6. Four patterns of interconnection realization.

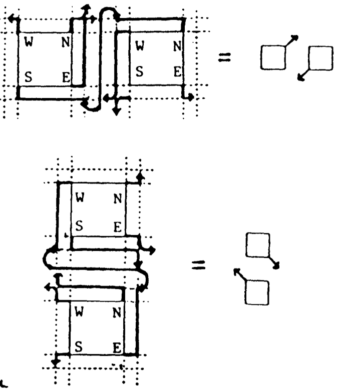


Figure 7. Matchings requiring one extra channel.

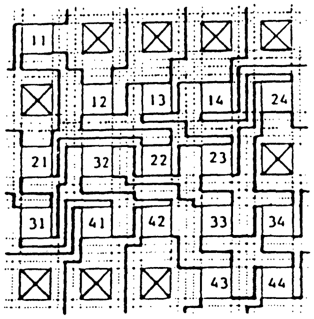


Figure 8. Interconnection for Figure 2.

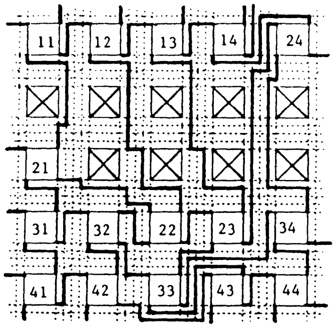


Figure 9. Interconnection for Figure 3.

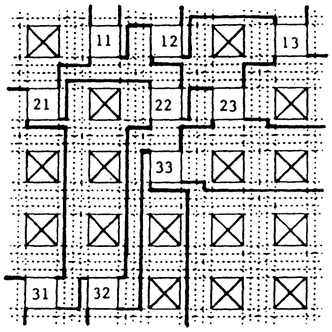


Figure 10. Interconnection for Figure 4.

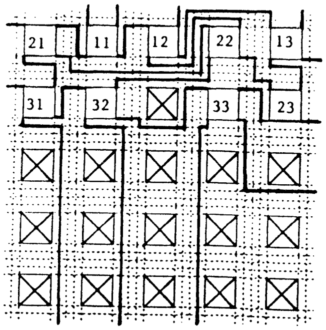
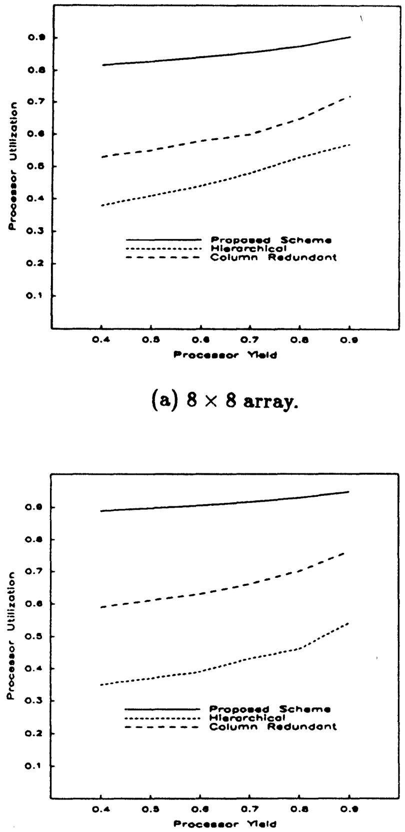


Figure 11. Interconnection for Figure 5.



(a) 8 × 8 array.

(b) 16 × 16 array.

Figure 12. Comparison of Processor Utilization.

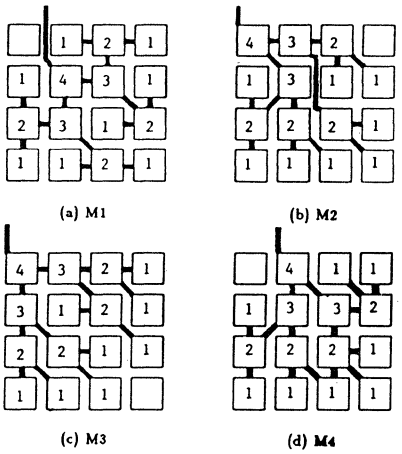


Figure 13. Four types of basic module.

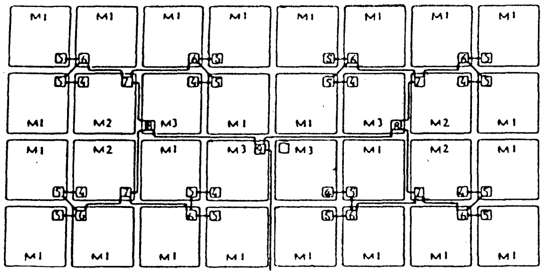


Figure 14. 9 level tree embedding using basic modules.

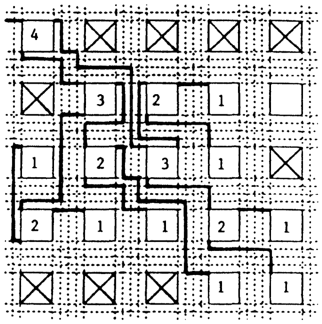


Figure 15. Reconfiguration of a four level tree for the same distribution of faults as in Figure 2.

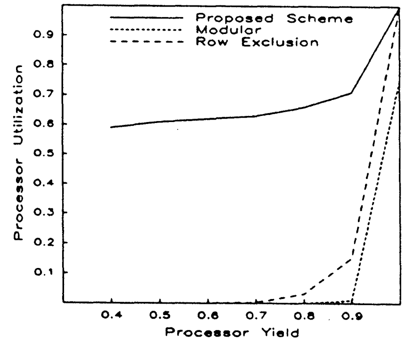


Figure 16. Comparison of Processor Utilization for reconfiguring an eight level tree.

A Parallel Processing Architecture for an Integrated Vision System

Alok N. Choudhary and Janak H. Patel

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Avenue
Urbana, IL 61801

Abstract-

Computation requirements for an integrated vision system are tremendous and thus a need for parallel processing. There are several tasks which must be performed in a sequence repeatedly. Each of these tasks have a great potential for spatial and temporal parallelism. In general, the degree of exploitable parallelism is high but dynamically variable. Therefore, efficient utilization of resources in a multiprocessor vision system requires the system to be highly flexible and modular. In this paper we consider an architecture for an integrated vision system. Then we illustrate how various steps involved in an integrated vision system which consist of low level, high level and hybrid algorithms can be efficiently mapped in an integrated environment. In particular, we consider stereo vision algorithm to extract 3-D object description from a set of 2-D images. The emphasis is on using small number of powerful processors concentrated in clusters and connected via flexible, reconfigurable and programmable crossbar. The issues considered are mapping algorithms independent of problem size, minimize communication, efficient pipelining of tasks and load balancing to evenly distribute the computation. We argue why the architecture is efficient as an integrated vision system. Furthermore, we show how various steps of the algorithm can be mapped onto the architecture with a brief description of each step of the algorithm.

I. Introduction

Computer vision has been regarded as a very complex problem. Image analysis and understanding procedures employ a very broad spectrum of techniques from several areas such as signal processing, advanced mathematics, graph theory, and artificial intelligence. These algorithms are, in general, characterized by massive parallelism. For low level processing, spatial decomposition of an image provides a natural way of generating parallel tasks. For higher level analysis operations, parallelization may also be based on other image characteristics. The multi-dimensional divide-and-conquer paradigm [1] is an attractive mechanism for providing parallelism in both of the above cases. In [2], Ahuja and Swamy proposed multiprocessor pyramid architecture as a straight forward implementation of the divide-and-conquer based approach. Such pyramids are natural candidates for executing divide-and-conquer algorithms as they most closely mirror the flow of information in these algorithms. However, design of an integrated vision system requires a greater flexibility, partitionability, and reconfigurability than is offered by regular array or pyramid structures[3].

Many multiprocessor architectures and parallel algorithms have been proposed to solve the problem of image understanding [4,5,6,7,8]. Most architectures such as pyramid, array processors, and mesh have limited capabilities to implement an integrated system for image processing due to several reasons. First, they are mostly suitable for SIMD type of algorithms which only constitute low level vision operations. Second, the architectures are inflexible due to the rigid interconnections between processors and processors and memory. Third, the number of processors needed to solve a problem of reasonable size is hundreds or thousands. Such a large number of processors is not only cost prohibitive, but the processors themselves cannot be very powerful and can have only limited features due to technological limitations. Fourth, it is normally assumed that the problem size exactly matches the number of processors available. Most of the time it is not clear how to adapt algorithms so that problems of different sizes can be solved on the same number of processors efficiently. Finally, the problem of input-output of data is rarely addressed in any of these architectures. It is important to note that no matter how fast or powerful a particular architecture is, its utilization can be limited by the bandwidth of the I/O. Significant research is being carried out in developing architectures and algorithms for image processing which are practically feasible. One good example is the CMU Warp processor [9,10,11,12]. The machine has a programmable systolic array of linearly connected cells, each capable of 10 MFLOPS. The array can efficiently perform local operations, in which each output depends on a small corresponding area of the input, since the connections between the cells are neighbor connections. It is also claimed that Warp is also suited for global-image operations.

An integrated vision application contains several algorithms in a sequence with input of one dependent on the the output of the previous algorithm and externally supplied parameters. Some of the algorithms are suited for SIMD architectures, some for MIMD and systolic architectures. Several issues such as efficient parallel mapping of individual algorithms, communication between tasks, data transfer, scheduling tasks etc. must be addressed. For example, stereo vision algorithms to obtain 3-D surface information from 2-D images is one such application[13, 14, 15] consisting of tasks such as edge detection, matching, hough transform, fitting, surface interpolation. There is a scope of considerable parallelism within each task and of pipelining tasks.

In this paper we consider an integrated vision architecture and describe its features. Then we argue why the architecture is efficient as an integrated vision system. Furthermore, we show how various steps of the algorithm can be mapped onto the architecture with a brief description of each step of the algorithm.

This paper is organized as follows. Section 2 presents the architecture. In Section 3, the mapping of the various steps of the stereo vision algorithms is described. Finally, summary and a few remarks about future work are presented.

II. Architecture

Figure 1 shows an architecture (called NETRA) for a large high performance multiprogrammed multiprocessor for image analysis and understanding systems. The architecture consists of the following components :-

(1) A large number (100 - 10000) of *Processing Elements (PEs)*, organized into clusters of, say, 16 to 64 PEs each.

(2) A tree of *Distributing-and-Scheduling-Processors (DSPs)* that make up the task distribution and control structure of the multiprocessor.

(3) A parallel pipelined shared *Global Memory*.

(4) An *Interconnection Network* that links the PEs and DSPs to the Global Memory.

The system is illustrated with a block diagram in Fig. 1.

A. Processor Clusters

The clusters consist of, say, 16 to 64 PEs, each with its own program and data memory. They form a layer below the DSP-tree, with a leaf DSP associated with each cluster. PEs within a cluster also share a common data memory. The PEs, the DSP associated with the cluster, and the shared memory are connected together with a crossbar switch. The crossbar switch permits point-to-point communications as well as selective broadcast by the DSP or any of the PEs.

Clusters can operate in an SIMD mode, a systolic mode, or an MIMD mode. Each PE is a general purpose processor with a high speed floating point capability. In an SIMD mode, PEs in a cluster execute identical instruction streams from private memories in a lock-step fashion. In the systolic mode, PEs repetitively execute an instruction or set of instruction on data streams from one or more PEs. In both cases,

This research was supported by the National Aeronautics and Space Administration under Contract NASA NAG-1-613.

communication between PEs is synchronous. In the MIMD mode PEs asynchronously execute instruction streams resident in their private memories. The streams may not be identical.

B. The DSP Hierarchy

The DSP-tree is an n-tree with nodes corresponding to DSPs and edges to bi-directional communication links. Each DSP node is composed of a processor, a buffer memory, and a corresponding controller.

The tree structure has two primary functions. First it represents the control hierarchy for the multiprocessor. A DSP serves as a controller for the subtree structure under it. Each task starts at a node on an appropriate level in the tree, and is recursively distributed at each level of the sub-tree under the node. At the bottom of the tree, the sub-tasks are executed on a processor cluster in the desired mode (SIMD or MIMD) and under the supervision of the leaf DSP.

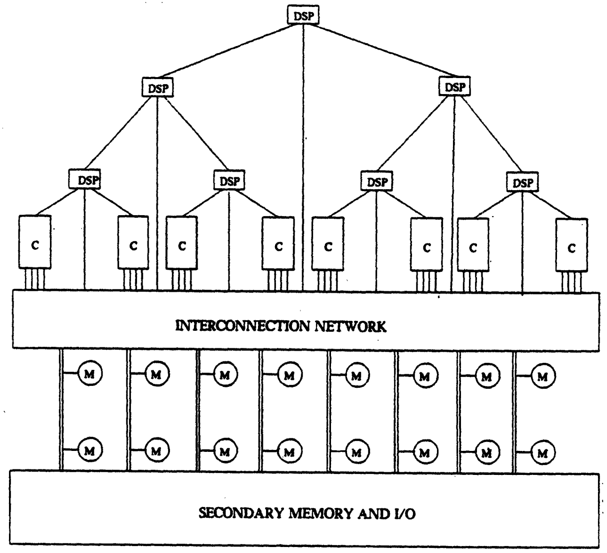
The second function is that of distributing the programs to leaf DSPs and the PEs. Vision algorithms are characterized by a large number of identical parallel processes operating on different data sets. It would be highly wasteful if each PE issued a separate request for its copy of the program block to the global memory because it would result in large unnecessary traffic through the interconnection network. Under the DSP-hierarchy approach, one copy of the program is fetched by the controlling DSP (the DSP at the root of the task subtree) and then broadcast down the subtree to the selected PEs.

C. Global Memory

The multiport global memory is a parallel-pipelined structure. Given a memory(chip)-access-time of *T* processor-cycles, each line has *T* memory modules. It accepts a request in each cycle and responds after a delay of *T* cycles. Since an *L*-port memory has *L* lines, the memory can support a bandwidth of *L* words per cycle.

Data and programs are organized in memory in *blocks*. Blocks correspond to "units" of data and programs. For example, in the case of a graph matching algorithm for a symbolic-matching task, each block may be a record containing all information corresponding to one node of the graph. The size of a block is, hence, variable and is determined by the size of a record for a task. A large number of blocks may together constitute an entire program or an entire image. Memory requests are made for blocks. The PEs and DSPs are connected to the Global Memory with a packet- switching multistage interconnection network.

The global memory is capable of queuing requests made for blocks that have not yet been written into. Each line (or port) has a Memory-line Controller (MLC) which maintains a list of read requests to the line and services them when the block arrives. It maintains a table of *tokens* corresponding to blocks on the line, together with their length, virtual address and *full/empty* status. The MLC is also responsible for virtual memory management functions.



DSP : Distributing-and-Scheduling Processor
C : Processor Cluster
M : Memory Module

Fig 1 : Organization of NETRA

III. Mapping the Stereo Vision Algorithm

Assume that I is an input to an image processing task **f** and the output is O. That is,

$$\mathbf{O = f(I)}$$

For example f may be edge detection, filtering, Fourier transform or object recognition. Therefore, f has several characteristics for parallel implementation. First, an identical, data independent, local operation is performed throughout an entire image on small quadrant of windows. This spatial characteristics implies that an image may be divided into a set of subimages which can be processed in parallel. Second, often several such functions are applied in a sequence to an image. For example, stereo vision for 3-D object extraction which uses convolution, matching, Hough transform, fitting etc. These temporal characteristics suggest the use of a pipeline environment to improve the processing rate. In summary, an overall processing function can be partitioned into several subfunctions which are pipelined yielding advantages of both spatial as well as temporal parallelism.

Architecture and the Model

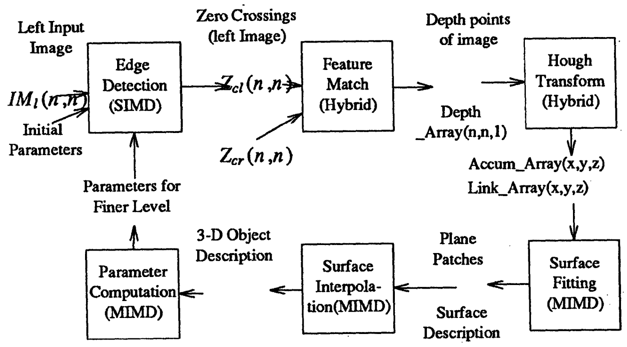
(1) **Processor Allocation :** Each stage of the image processing function can be performed on one or more clusters of processor. Spatial parallelism within each stage of the pipeline can be exploited using the flexible interconnect of the cluster and local DSP for scheduling.

(2) **Pipelining :** Pipelining between various stages can be achieved using the macro dataflow feature of the architecture. Once an output data block is produced by the previous task, it is sent to the global memory with appropriate address of the next cluster needing this data (details are given in[3]

(3) **Task Scheduling :** Scheduling of tasks can be performed locally in a cluster for MIMD mode by all processors sharing the load information in the common data memory. If more than one clusters are involved then an appropriate DSP can schedule tasks and dynamically allocate processors to various subtasks if the computation is heavily input data dependent and unevenly distributed.

We now illustrate how the various algorithms that are part of stereo vision can be implemented in parallel on the proposed architecture. We discuss the computation, communication and scheduling issues along with the data structure and dataflow requirements. We consider various algorithms individually and describe their possible parallel version on NETRA and suggest how they can be integrated and pipelined.

Figure 2 shows the data flow and main data structures for the integrated stereo vision algorithm. The figure only illustrates the computation and communication needs for the left input image. Exactly the same computation and communication is also done for the right input image. The type of algorithm suited for each step (such as SIMD, MIMD or a mix of SIMD and MIMD) is also indicated with the task blocks. The following is a description of the tasks shown in Figure 2.



SIMD, MIMD and Hybrid indicate type of algorithm suitable for the step

Exactly the same Computation is done for the right image in parallel

Data flow and Data Structures

Fig.2 Computation, Communication and Data flow for Stereo Vision

(1) Data Compression and Edge Detection : For Coarse-to-Fine processing, it is required to compress the input image data two or more times depending on resolution of the image. Data compression involves computing the weighted average. Each pixel in the output (compressed) image is the weighted average of a certain size neighborhood in the original (uncompressed) image. Compression is a variation of 2-D convolution (except for the fact that unlike convolution, output is not taken at each point, but every n^{th} point, where n depends on how much compression is needed). Both edge detection and data compression involve a variation of 2-D convolution, therefore, we describe a convolution algorithm. A 2-D convolution is expressed as follows:

$$G(i,j) = w_{i,j} * I(i,j)$$

where, I(i,j) is the image, $w_{i,j}$ is the convolution window and G(i,j) is the output of the convolution operation.

Edges are used as features to be matched in left and right images. The matched edges will then result in depth points. This algorithms uses the zero crossings of the convolution of the image with the $\nabla^2 G$ operator to determine the edge location in the image. There are numerous parallel algorithms available for 2-D convolution on various architectures [16]. Furthermore, there are special purpose integrated circuits available to perform convolution. We describe a parallel algorithm for edge detection on the processor clusters on NETRA.

The approach is to reduce 2-D convolution to a 1-D convolution without incurring additional steps. Each pixel is logically mapped onto a separate processor (as if there were as many processors available as there are pixels). Actually the image is folded and multiple pixels are mapped onto one processor. The image is folded in two dimensions in a wrap around fashion, both left to right, and top to bottom. If the image size is n x n, and number of processors is p x p then each processor will have n^2/p^2 pixels in its local memory. In general, pixel (i,j) ; $0 \le i \le n-1$, $0 \le j \le n-1$ will be mapped to processor ((i mod p), (j mod p)). Therefore, this mapping preserves the adjacency of any two pixels even though the image is folded.

Assume that the window (or neighborhood) size is w x w and the convolution mask is stored in each PE's memory. A small window is embedded in a larger one and therefore, same connections can be used for a larger window size with the addition of new connections for extra steps. The algorithm performs the convolution by each processor distributing its pixel values to the neighborhood in a pipelined manner.

In the following algorithm North, South, East and West Neighbors are defined in wrapped around fashion. At any step all the processors have the same neighbor connection. All the processors will follow exactly the same pattern. It should be noted that the data values at each processor are stored in a linear array and subscript (i,j) means the data value i in the connection number j. For a processor (i,j), N,S,E,W neighbors are defined as follows.

N = ((i-1),j), if (i-j) < 0, then N = ((i-1 + p), j)
S = ((i+1) mod p, j)
E = (i, (j+1) mod p)
W = (i, (j-1)), if (j-1) < 0, then W = (i, (j-1+p))

Assuming that each processor has m pixels in its local memory, the algorithm works as follows. For an image of size 256 x 256 and a processor cluster of size 64, each processor has 1K pixels. For a window size of 3 x 3, each processor performs 9K multiply-add operations. The interconnection needs to be reconfigured only **eight** times. It is important to note that the number of times the interconnection needs to be reconfigured only depends on the neighborhood window size. Also note that the algorithm can be easily adopted to any problem size and any processor cluster size. Once the convolution with the laplacian is performed, each processor stores the zero crossings by storing its (x,y) position and its orientation.

The above algorithm illustrates that SIMD algorithms can be mapped efficiently on to the processor clusters using the flexibility and programmability of the interconnection. Furthermore, the mapping is such that the interconnection reconfiguration is independent of the input image size. Following algorithms illustrate how MIMD and hybrid algorithms (algorithms needing both SIMD and MIMD type of computation such as Hough Transform) can be mapped and how pipelining of tasks can be achieved. It should be noted that each of these algorithms are executed on one or more different clusters. Also, there are two parallel stereo algorithms being executed at all time, one for left and one for right input image.

(2) Feature Matching : Feature matching is required in order to compute the depth points in the image. There are several sub tasks within this task. Firstly, the output data of the previous step (i.e. edges) needs to be properly organized. Secondly, since matching needs corresponding edges from the left(right) images, there is a need to be data transfer between processor clusters. Then there is a task of matching the properties of the edges within certain window along the epipolar lines. Epipolar lines are normally assumed to be horizontal and therefore, the search is limited to one dimension. Therefore, the goal is to map data efficiently onto the processors such that the communication is minimized. We suggest the following.

```
ALGORITHM CONVOLUTION;
Input : IM(n,n) , Output : Matrix of zero crossings $Z_c(n,n)$
  All the processors work in SIMD lock-step fashion.
  Set up Connection_array of size $wxw$ by choosing
  first $wxw$ connections from the set
  {N,E,S,S,W,W,N,N,N,E,E,E,S,S,S,W,W,W,W,N,N,N,N,E,..}.
      $m := \left\lceil \frac{n^2}{p^2} \right\rceil$
   For i = 1 to m do (in parallel)
        Result(i) := $w_{i,j} * data(i)$
   End_For

   For j = 1 to $wxw$ do (in parallel)
    Set up appropriate connections on the crossbar as follows.
      connection(j) := connection_array(j)
      For i = 1 to m do (in parallel)
        Send data (pixels) on the output port to
        the connected neighbor.
        At the same time receive data from its input port.
        Result(i) := $Result(i) + w_{i,j} * data(i,j)$
      End_for
   End_for
END_CONVOLUTION;
```

Each processor $P_{i,j}$ accumulates rows of zero crossing in such a way that communication is needed only in one direction without any need to change the interconnection. Therefore, the processors are organized into horizontal linear arrays in a wrapped around fashion as shown in Figure 3. The data is then pumped into one direction in systolic fashion and each processor accumulates appropriate rows (of edges) according to the following mapping. Processor $P_{i,j}$ receives the rows

$$(i \bmod j) + jp + kp^2 \text{ for } k = 0,1....$$

and accumulates at most n/p^2 rows.

The second subtask is that of data transfer between two clusters of processors which are working on the left and the right image respectively. This transfer is achieved by macro data flow using the global memory between corresponding processors in two clusters. Note that this can be performed at the same time when rows are being accumulated. Therefore, this is an example of how pipelining of tasks can be achieved. Once each processor accumulates appropriate rows, the next sub task is to find edges which have a match in the other image. This task is highly data dependent because the computational needs depend on the image, and how many edges are there to be matched. It is possible that some parts of the image result in lot of edges where as other parts have relatively very few edges. Therefore, the processors that work on those rows having comparatively large number of edges may be heavily loaded and others may be under utilized. Therefore, there is a need for efficient task scheduling and

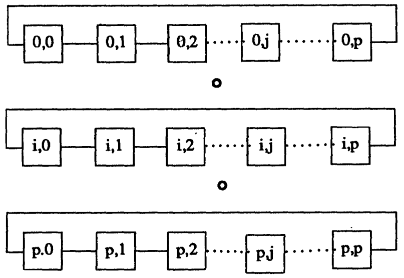


Fig. 3 : Reorganization of Processors for Feature Matching

dynamic load balancing. The above mapping of rows onto processors ensures that no processor has adjacent rows of edges, and in fact, no processor has p^2 adjacent rows of edges. Thus the mapping itself tends to evenly distribute the computation on the processors. The following algorithm sketches how the processors perform the task of feature matching in parallel.

Each processor first works on rows allocated to it initially. Then if it finishes before others do, it selects rows allocated to other processors and performs the feature matching algorithms on them. Therefore, load balancing is achieved. Once the feature matching is finished, depth map of the image is available in both left and right image's coordinate system. The next task is to perform surface fitting.

(3) Surface Fitting : The feature matching tasks provides part of the depth points in each processor's local memory. First subtask is to transfer the Depth_array to each processor's local memory. This can be done in macro data flow mode by connecting the processors in a circular array as before.

Once each processor has the Depth_array, surface fitting can be done in parallel in MIMD mode. The synchronization, task scheduling and load balancing can be achieved using the shared common data memory, or the global memory if more than one cluster of processors are involved in this computation. The algorithm uses hough transform method to fit planes onto a set of points.

The input to the following algorithm is a two dimensional array of points (e.g, zero crossings) in which line segments are to be detected. We will show how a parallel algorithm can be implemented to compute these line segments using hough transform method. The computation is done in the (r,θ) parameter space. If there exists a line whose normal distance from the origin is r, the normal makes an angle θ with the x-axis then if the point (x,y) lies on that line than the following equation is satisfied.

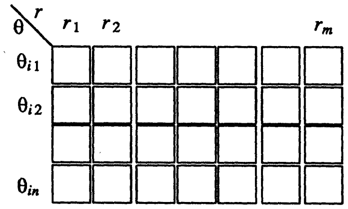
$$r = x\cos\theta + y\sin\theta$$

First of all r, θ are quantized. The quantization depends on how much accuracy is required in the final result. Let's assume that maximum value of r br $r_{\max}$ maximum value of θ be $\theta_{\max}$ (generally π or 2π). Then if r_{res}, θ_{res} are the resolutions used for quantization then total number of accumulator cells in the computation are $r_{\max}.\theta_{\max}/r_{res}.\theta_{res}$. The number of rows and columns in the accumulator array being $R = \theta_{\max}/\theta_{res}$ and $C = r_{\max}/r_{res}$ respectively. The mapping is as follows. Each processor computes all r values for its share of θ values. If there are p^2 processors then each processor gets $n = R/p^2\theta$ values to work on. Therefore, processor i gets to work on n θ values where, $1 \le i \le p^2$. Figure 4 shows the accumulator array for processor i. The main resons for such a mapping are that when looking for peaks later no two processors need to communicate thereby reducing the communication overhead. Further, the processor can store sinθ, cosθ values for its allocated n θ values in its registers resulting in saving memory access delays which would occur if all quantized sinθ and cosθ values are stored with each processor in its local memory. A brief explanation of the algorithm is as follows. Each processor begins working on a small part of the image. It computes the required r values for each of the θ values stored in its registers. It then increments the appropriate count in the Acc_array. If the count increases beyond a certain threshold value, there exists a possibility of this being a local maxima. Therefore, another array called Link_array is updated marking this fact. This step reduces the search space tremendously when looking for local maxima since normally a very small fraction of the image contributes to lines and entire accumulator array need not be searched when looking for local maxima. Figure 5 shows the Link_array. Once the above computation is finished for the entire image, the local maxima are computed in the Acc_array using the information available in Link_array as follows. Only those locations in the Acc_array need to be searched which are marked in Link_array because it contains only those locations which are candidates for local maxima. Therefore, the search space is reduced.

```
ALGORITHM FEATURE_MATCH;
 Each Processor $P_{i,j}$, $0 \le i \le p-1$, $0 \le j \le p-1$ (in parallel) do
 $Row\_count_{i,j} := \left| \frac{n}{p^2} \right|$
repeat
 If $Row\_count_{i,j} > 0$ (more rows left in $P_{i,j}$'s local memory)
        Select a row($Row\_count_{i,j}$) from its local memory.
        $Row\_count_{i,j} := Row\_count_{i,j} - 1$.
         Mark row($Row\_count_{i,j}$) as selected in the
         common data memory of cluster.
         Update load information in the common data memory.
         For each edge (zero crossing) in the selected row Do
                Look for match in the corresponding
                 row in the other image
                If match found Then
                  Compute depth point $z_i$ for point $(x_{i,}y_i)$
                  Store in Depth_Array($x_i\, y_i, z_i$)
                 Else
                  Store $(x_i\, y_j)$ in the list of ambiguous edges
         End_For
 Else
  If $Row\_count_{j,k} \neq 0$ for some processors $P_{j,k}$ Then
   Select a row from the processors with maximum Row_count.
   Mark the selected row.
   Update load information in the common data memory
  End_If
Until finished (i.e., no more rows left to be selected)
End FEATURE_MATCH
```

The second subtask in surface fitting is that of fitting quadratic patches. This algorithm falls in the category of MIMD algorithm which works on a database of information produced by the previous steps. The output of this algorithm is dependent on the input data globally. Two adjacent patches are compatible if there depth orientation do not differ by more than a certain threshold value. Compatibility criteria is an input parameter to the algorithm. The planar patches at each grid point are placed into sets of planes such that they are mutually compatible. Now a variation of relaxation algorithm is applied to the planes at each grid point to test for mutual compatibility. This task can be done in parallel using MIMD mode shared memory model for synchronization and load balancing of the subtasks. Shared memory model is preferred instead of message passing because the input data size is large. Therefore, there may be large communication requirements at each stage of the algorithm because compatibility label may propagate a large distance. Further, task scheduling and load balancing is easier because the load information is available centrally. Therefore, scheduling of tasks can be done easily without having to transfer huge amount of data from one processor to another. Let's assume that there are a total of n grid points in the image on which quadratic patches are to be fitted. The following steps in the stereo vision algorithms also involve similar algorithms. Due to limitations on space we are omitting the descriptions of the algorithms. However, a brief description of what is involved in rest of the steps is provided.

(4) Locating Contours : Locating contours involves checking for discontinuities in the surfaces. This also can be done in parallel using the common data memory MIMD algorithm. A similar algorithm to the one



Accum_array(i,j) for one processor

Fig. 4 : Accumulator array mapping for Hough Transform on each Processor

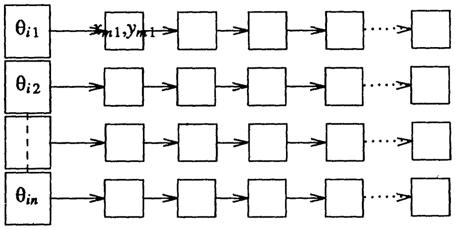
```
ALGORITHM ACCUMULATE_COUNT
 Each processor $P_i$, $1 \le i \le p^2$ does the following (in parallel)
 For j = 1 to n do
  For each (x,y) in the array such that (x,y) is significant do
   compute $r(\theta_{ij}) := x\cos\theta_{ij} + y\sin\theta_{ij}$
   Acc_array($\theta_{ij}$,r($\theta_{ij}$ )/$r_{res}$) :=Acc_array
($\theta_{ij}$,r($\theta_{ij}$ )/$r_{res}$)+1
   If Acc_array($\theta_{ij}$,r($\theta_{ij}$ )/$r_{res}$) > threshold value then
        Link_array($\theta_{ij}$,r($\theta_{ij}$ )/$r_{res}$) := true
   End_if
  End_For
  Transfer (x,y) value to next processor in the circular pipeline
 End_For
END_ACCUMULATE_COUNT
```

above can be used easily with the main body being different. Instead of checking for the compatibility at each grid point, a bipartite circular patch is fit (in four directions) in order to detect edges due to discontinuity. If enough processing power is available then check for edges can be done in more than four directions.

(5) Surface Interpolation : Surface interpolation can be done in parallel for each point P on the surface by taking the weighted height of the point (where weight depends on the distance of the point from each surface within a distance of 2w from the point). Again the algorithm can be easily implemented in MIMD mode using the shared memory model. Essentially, the algorithm reduces to computing a weighted average within a three dimensional neighborhood window for each point. Familiar algorithms for weighted average can be used. However, the algorithms needs to be MIMD because unlike the standard SIMD neighborhood algorithms, the computation is data dependent and neighborhood size itself depends on the data. Therefore, neighborhood for each point may be of different size depending on the number of different surfaces within a distance 2w from the point.

(6) Computing Parameters for Next Level : This step requires computation opposite to that needed in data compression. In this step, data is expanded for the next finer level. Essentially, the grid is doubled in each direction but preserving the surfaces in the present level. Then the quadratic patches are interpolated using the compatibility of the neighboring existing quadratic patches. This can be accomplished by relaxation algorithm used previously.

```
ALGORITHM LOCAL_MAXIMA
 Each processor does the following in MIMD mode
  For each entry Link_array(i,j) do
   If (Acc_array(i,j) > Acc_array(i-k,j)) AND
          (Acc_array(i,j) > Acc_array(i+k,j))
          for all k s.t. $1 \le k \le w$ for a certain
          neighborhood of size w Then
             declare Acc_array(i,j) as local maxima
             This gives a line with the $(r, \theta)$ parameters
   End_If
END LOCAL_MAXIMA
```



Lists of nodes indicating possible local maxima

Fig. 5: Link_Array data structure to reduce search space for computing maxima

```
ALGORITHM QUAD_PATCH;
Each Processor $P_{i,j}$, $0 \le i \le p-1$, $0 \le j \le p-1$ in parallel do
    $Grid\_point\_count_{i,j} := \left\lceil \frac{n}{p^2} \right\rceil$
repeat
 If $Grid\_point\_count_{i,j} > 0$ (Local Grid_points with $P_{i,j}$)
  Select a row($Grid\_point\_count_{i,j}$) from its local memory.
  Mark Grid_points as selected in the common data memory.
  Update load information in the common data memory
  For each selected Grid_point Do
      Check the compatibility of the each plane in
      the neighborhood with the averaged parameters
      of each set.
      Choose the two most compatible sets make the
      plane member of these two sets.
  End_For
 Else
  Check load information in the common data memory.
      If $Grid\_point\_count_{j,k} \ne 0$ for some $P_{j,k}$ Then
       Select a Grid_point from the processors with
    maximum Grid_point_count.
       Mark the selected Grid_point.
    Update load information in the common data memory
      End_If
Until finished (i.e., no more Grid points left to be selected)
End QUAD_PATCH
```

Once the parameters for the next finer level are computed, the entire algorithm (involving all the steps) is executed on a finer image and more accurate description of the 3-D surface is obtained. This iterative process is continued until the finest level of the image is reached which provides the most accurate description of the object.

IV. Summary

In this paper we considered a multiprocessor architecture for integrated vision. Its processing power is concentrated in clusters of powerful processors connected through flexible and programmable crossbar. Its system control functions are distributed over a hierarchy of controllers. We considered a parallel image processing model and discussed the architecture as well as the 3-D stereo vision algorithm in the context of the model. We illustrated how various steps of the integrated system can be mapped in parallel, pipelined and what are the computation, communication, data flow, scheduling and load balancing requirements. Furthermore, we argued why the proposed mapping and integration of tasks is efficient.

We are in the process of simulating the architecture and algorithms both in an independent environment as well as in an integrated environment to investigate its performance in the light of various issues discussed earlier. Furthermore, we propose to compare performances of various low level, high level and hybrid algorithms mapped on architectures such as hypercube with the mapping on the proposed architecture. This performance study is also aimed at identifying other issues to be considered for an integrated vision system architecture which may have been overlooked and also suggest refinements in the architecture.

REFERENCES

[1] J. L. Bentley, "Multidimensional Divide-and-Conquer," *Comm. of the ACM*, vol. 23, No. 4, pp. 214-229, April, 1980.

[2] N. Ahuja and S. Swamy, "Multiprocessor Pyramid Architectures for Bottom-Up Image Analysis," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, pp. 463-475, July, 1984.

[3] M. Sharma, J. H. Patel, and N. Ahuja, "NETRA: An Architecture for a Large Scale Multiprocessor Vision System," in *Workshop on Computer Architecture for Pattern Analysis ans Image Database Management*, Miami Beach, Florida, pp. 92-98, Nov. 1985.

[4] F. A. Briggs, K. S. Fu, J. H. Patel, and K. H. Huang, "PM4 - A Reconfigurable Multiprocessor System for Pattern Recognition and Image Processing," *1979 National Computer Conference*, pp. 255-266.

[5] H. J. Siegel et al., "PASM - a Partitionable SIMD/MIMD system for Image Processing and Pattern Recognition," *IEEE Trans. on Comput.*, vol. C-30, pp. 934-947, Dec. 1981.

[6] Y. W. Ma and R. Krishnamurti, "The Architecture of REPLICA - A Special-Purpose Computer System for Active Multi-Sensory Perception of 3_Dimensional Objects," *Proc. International Conference on Parallel Processing*, pp. 30-37, 1984.

[7] W. A. Perkins, "INSPECTOR - A Computer Vision System that Learns to Inspect Parts," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-5 No. 6, pp. 584-593, Nov., 1983.

[8] M. J. B. Duff, *Computing Structures for Image Processing*. New York, N. Y.: Academic Press, 1983.

[9] H. T. Kung and J. A. Webb, "Global Operations on the CMU WARP Machine," *Proceedings of 1985 AIAA Computers in Aerospace V Conference*, Oct. 1985.

[10] T. Gross, H. T. Kung, M. Lam, and J. Webb, "WARP as a Machine for Low-Level Vision," in *IEEE International Conference on Robotics and Automation*, ST. Louis, Missouri, pp. 790-800, March, 1985.

[11] H. T. Kung, "Systolic Algorithms for the CMU Warp Processor," in *Tech. Rep. CMU-CS-84-158, Dept. of Comp. Sci., CMU*, Pittsburgh, PA, Sept., 1984.

[12] F. H. Hsu, H. T. Kung, T. Nishizawa, and A. Sussman, "LINC: The Link and Interconnection Chip," in *Tech. Rep., Dept. of Comp. Sci., CMU, CMU-CS-84-159*, Pittsburgh, May, 1984.

[13] Marr D., *Vision*. San Fransisco: Freeman, 1982.

[14] Hoff W. and Ahuja N., "Extracting Surfaces from Stereo Images : An Integrated Approach," *CSL Tech. Rep. UILU-ENG-87-2204*, January, 1987.

[15] Marr D. and Poggio T., "A Theory of Human Stereo Vision," *Proc. fo R. Soc. Lond.*, vol. B 207, pp. 187-217, 1980.

[16] Z. Fang, X. Li, and L. Ni, "Parallel Algorithms for 2-D Convolution," *IEEE*, pp. 262-269, 1986.

AN OPTIMAL SOLUTION FOR CONSENSUS PROBLEM IN AN UNRELIABLE COMMUNICATION SYSTEM

K.Q. Yan and Y.H. Chin
Institute of Computer & Decision Sciences
National Tsing–Hua University
Hsinchu, Taiwan 30043, R.O.C.

Abstract Traditionally, consensus problem is solved in a fully connected network with node failure assumption. This paper discusses the consensus problem with the assumption of link failure. A simple and efficient protocol FLINK is proposed. The complexity of information exchange required by the protocol is $O(n^2)$. The protocol uses minimum number of rounds to achieve a consensus and can tolerate maximum number of allowable faulty components.

1. Introduction

To achieve an agreement on a predefined value in a distributed system, protocols are required so that the system will run even if certain components in the distributed system were failed. Such a unanimity problem was studied by Lamport [7,8], and it is called a *Byzantine Agreement* (BA)[2,3,4,6,7,8,9,10,11]. A closely related sub–problem, *Consensus* problem, has been studied extensively in the literature [1,7]. In the paper, we concern the solution of consensus problem. The definition of such a problem is to make the correct nodes in an n nodes fully–connected network to reach a common agreement. Each node chooses an initial value to start with, and communicates each other by means of message. The desired protocol is to solve the consensus problem if it satisfies the following constraints:

(*Agreement*): All correct nodes agree on the same value.

(*Validity*): If the initial value of all nodes is v_i, then all correct nodes shall agree on v_i.

Many results in a Byzantine Agreement or consensus problem are based on the assumption of node failure in a fail–safe network [1–11]. Base on this assumption, a communication link fault is treated as a node fault, regardless the correctness of an innocent node; hence an innocent node does not involve an agreement. This is contradiction with the definition of a BA (or consensus problem) which requires all correct nodes to achieve an agreement.

In this paper, we consider a distributed system whose nodes are reliable during the consensus execution; while message links may be disturbed by some noise or an intruder and results in the exchanged message maliciously. A new efficient and reliable protocol to achieve consensus in an unreliable communication environment is proposed first; then its efficiency and reliability are proved later. The common term *round* [1,6] is used to denote the interval of message exchange. The proposed protocol can tolerate $\lfloor n/2 \rfloor -1$ faulty links, and requires only two rounds of message exchange. The amount of necessary information exchange is only $O(n^2)$ [4]. If a link fault is treated as a node fault, the number of rounds required by the protocol is better than the previous results [1,7].

In the subsequent sections, Section 2 defines the model and the concepts. Section 3 presents the proposed protocol and proves its correctness. Section 4 discusses the impossible cases of an unreliable distributed system and the optimization of the protocol. Section 5 gives the conclusion and the future work.

2. Model

In a fully connected n–node network, if each node has at least n*n bytes memory; then a sender's message is always identifiable by a receiver; and the protocol's processing time can be negligible. If each node always works well during the execution of consensus protocol, but links may be damaged due to some noise or intruder, thus a link may be in *faulty* when its transferred message is changed or delayed. Conversely, a link is *perfect* when the transferred message is always received correctly and on time.

Usually a node's computation time is faster than the message communication time through a link; hence a node's computation time for protocol is ignored. Under such an assumption, the protocol can make the correct node in a fully–connected network to reach a predefined common value with the least number of rounds.

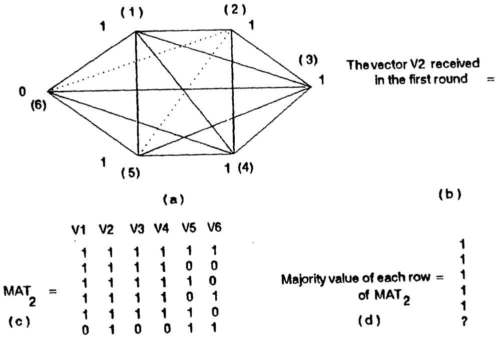
Let MAT_i be the matrix set up at node i by the procedure MATRIX shown in Figure 1 for i=1,...,n. In the first round, node i receives the preset initial value from each other nodes, and V_i be the vector $[v_1,v_2,...,v_k,...,v_n]$, where v_k represents the initial value received from node k or the initial value of node i itself, $1 \leq k \leq n$. For simplicity, any value not identified or not received within the predefined time limit will be set to the complementary value $\neg v_k$ by the receiver.

```
Procedure MATRIX ( for node i with initial value $v_i$ )
Step 1:   Receive the initial value $v_j$ from node j,
          for $1 \leq j \leq n$ and $j \neq i$.
Step 2:   Construct the vector $V_i$ $[v_1, v_2,..., v_j,..., v_n]$, $1 \leq j \leq n$.
Step 3:   Receive the vector $V_j$ from node j, $1 \leq j \leq n$, $j \neq i$.
Step 4:   Construct a $MAT_i$. (Setting the vector $V_j$
          in column j, for $1 \leq j \leq n$.)
```

Figure 1. The procedure for setting MAT_i on node i.

In the second round, each node broadcasts its V vector to other nodes and receives n–1 vectors from the other nodes. MAT_i is established by using vector V_j as the j–th column in the matrix. Note that the i–th column in MAT_i is the vector constructed by node i in the first round. In the second round, node i can receive n–1 vectors from the rest n–1 nodes; therefore MAT_i is an n*n matrix. Each element v_{jk} in MAT_i represents the value of the node k received from node j in the first round. Let the majority value of $[v_1, v_2,..., v_n]$ in the k–th row to be MAJ_k=v_i if the number of v_i's is greater than n/2; otherwise MAJ_k is set to ?. Figure 2 shows an example of six nodes and v_i = 1, for i=1, 2, 3, 4 and 5;

and $v_6 = 0$. The vector V_2 received by node 2 in the first round is shown in Figure 2(b), and the corresponding MAT_2 is shown in Figure 2(c). The second column in MAT_2 is the vector V_2 shown in Figure 2(b), and the rest 5 columns are the vectors received by node 2 in the second round. If link 25 and link 26 are faults, the values v_{52} and v_{62}, and the vectors V_5 and V_6 may have changed maliciously. The majority value of each row is shown in Figure 2(d).



The vector V2 received in the first round $= \begin{array}{c}1\\1\\1\\1\\1\\1\end{array}$

$$MAT_2 = \begin{array}{cccccc} V1 & V2 & V3 & V4 & V5 & V6 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{array}$$

Majority value of each row of MAT_2 $= \begin{array}{c}1\\1\\1\\1\\1\\?\end{array}$

Figure 2. A network with six nodes shows the way to construct MAT_2 and get the majority value of each row. The two dash lines represent two faulty links.

Based on the properties of the consensus problem, the initial value of node i (denoted as v_i) should be known by itself prior to the execution of consensus protocol. If node i makes a decision after executing the consensus protocol, it must determine whether or not a disagreement exists with the initial value, and it has to decide the initial value or a "default" (ϕ). In any case, a node with initial value v_i should not decide on the complementary value $\neg v_i$.

As for multivalue consensus problem, Turpin and Coan [11] have already shown that the protocol of a binary value consensus problem can be extended to a multivalue consensus problem, therefore only binary initial value is discussed.

3. Protocol

In the section the proposed protocol based on the model developed in Section 2 is formally presented. The following definitions make the protocol different from the previous results.

Definition 1: Every correct node should always know the initial value of itself; and

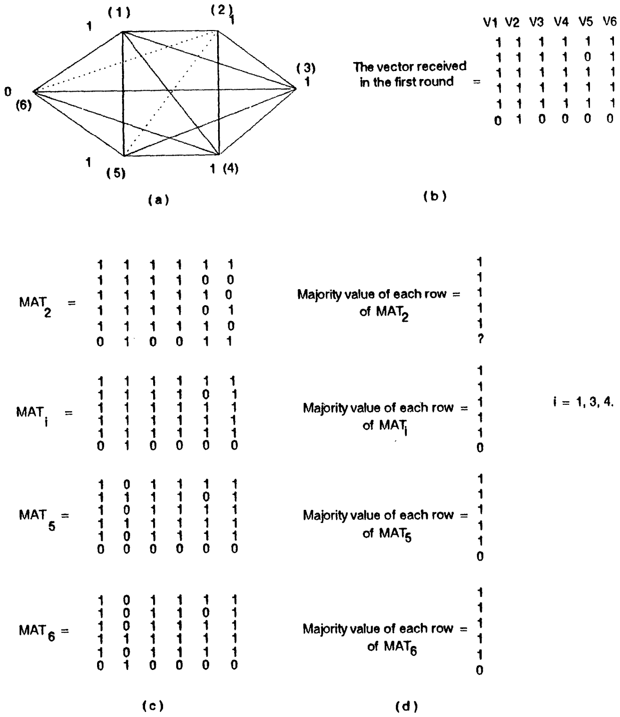
Definition 2: If the initial value is v_i, then the decision made must be either v_i or ϕ, but not $\neg v_i$.

Figure 3 shows the protocol FLINK which can tolerate $\lfloor n/2 \rfloor - 1$ fault links; and it achieves consensus by only two rounds of message exchange. Later, we will prove 1) the efficiency of the method, and 2) the necessary and sufficient conditions for the number of rounds and faults required by FLINK protocol. Let DEC_i be the value chosen by node i to agree on with others.

Figure 4 shows the complete procedure and the result of the protocol FLINK for the six node example mentioned in Figure 2. Since node 2 has MAJ_6=? in MAT_2, and $v_{62} = 1 = v_2$. By step 3 in FLINK, $DEC_2 = \phi$. Nodes 1, 3, 4 and 5 find that there is a $MAJ_6 = 0$ ($= \neg v_i$) in MAT_i, so $DEC_i = \phi$ for i = 1, 3, 4 and 5 by step 2 in FLINK. For the same reason, node 6 has a $MAJ_1 = 1$ ($= \neg v_6$), so $DEC_6 = \phi$. Therefore, all nodes agree on the same value ϕ. Consensus achievement is done.

```
Protocol FLINK ( For node i with initial value $v_i$ )
Message Exchange Phase:
    Round 1:  Broadcast ($v_i$), then receive the
              initial value from the other nodes,
              and construct vector $V_i$.
    Round 2:  Broadcast ($V_i$), then receive the
              vectors broadcasted by other nodes
              and construct $MAT_i$.
Decision Making Phase:
    Step 1:   Take the majority value of each
              row k of $MAT_i$ to be $MAJ_k$.
    Step 2:   Search for any $MAJ_k$. If ($\exists$ $MAJ_k$
              $= \neg v_i$), then $DEC_i := \phi$;
    Step 3:   else if ($\exists$ $MAJ_k$ = ?) AND ( $v_{ki}$ =
              $v_i$), then $DEC_i := \phi$; else $DEC_i :=$
              $v_i$, and halt.
```

Figure 3. The FLINK protocol to achieve consensus.



The vector received in the first round $= \begin{array}{cccccc} V1 & V2 & V3 & V4 & V5 & V6 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{array}$

$$MAT_2 = \begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{array}$$

Majority value of each row of MAT_2 $= \begin{array}{c}1\\1\\1\\1\\1\\?\end{array}$

$$MAT_i = \begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Majority value of each row of MAT_i $= \begin{array}{c}1\\1\\1\\1\\1\\0\end{array}$ i = 1, 3, 4.

$$MAT_5 = \begin{array}{cccccc} 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Majority value of each row of MAT_5 $= \begin{array}{c}1\\1\\1\\1\\1\\0\end{array}$

$$MAT_6 = \begin{array}{cccccc} 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Majority value of each row of MAT_6 $= \begin{array}{c}1\\1\\1\\1\\1\\0\end{array}$

$DEC_i = \phi$, (for $MAJ_6 = 0 = \neg v_i$ & $v_i = 1$); i= 1,3,4,5;
$DEC_2 = \phi$, (for MAJ_6 = ? AND $v_{62} = 1 = v_2$)
$DEC_6 = \phi$, (for $MAJ_j = 1 = \neg v_6$ and $v_6 = 0$); $1 \leq j \leq 5$;

Figure 4. The result of FLINK for the six node example in Figure 2.

The following lemmas and theorems are used to prove the correctness and complexity of FLINK.

Lemma 1: If there is a $MAJ_k = \neg v_i$ in MAT_i, then at least there is one node with an initial value which disagrees with v_i in the network.

Proof: The majority value in the k–th row $=\neg v_i$ means that there are at least $\lceil n+1/2 \rceil$ $\neg v_i$'s in the k–th row. Since the number of faulty links is at most $\lfloor n/2 \rfloor -1$, there exists at least one value $\neg v_i$ received from a perfect link. In other words, there is a node which has an disagreeable initial value.■

Lemma 2: Let the initial value of node i be v_i and the link ij is in perfect, then the majority value at the i–th row in MAT_j should be v_i.

Proof: Since link ij is perfect, the node j will receive v_i from node i in the first round and $v_{ij} = v_i$ in MAT_j. Mean while, the value v_i of node i will be broadcasted to the other nodes. There are at most $\lfloor n/2 \rfloor -1$ faulty links in the system. In the second round, node j receives at least $(n-1)-(\lfloor n/2 \rfloor -1) = \lceil n/2 \rceil$ v_i's in the i–th row of MAT_j. Hence, there are at least $\lceil n/2 \rceil +1$ v_i's in the i–th row, and the majority values in the i–th row should be equal to v_i.■

Lemma 3: If the initial value of node i is v_i, whether or not link ij is in perfect, the majority value at the i–th row of MAT_j, $1 \le j \le n$, should be either v_i or be ? with $v_{ij}=\neg v_i$.

Proof: By Lemma 2, when link ij is perfect, the majority value of the i–th row in node j is v_i, for $1 \le j \le n$. When link ij is faulty, we consider the following two cases in the first round.

Case 1: $v_{ij} = v_i$

Since there are at most $\lfloor n/2 \rfloor -1$ faulty links connected with node j, there are at most $\lfloor n/2 \rfloor -1$ values that may be $\neg v_i$'s in the second round. The number v_i's is $[(n-1)-(\lfloor n/2 \rfloor -1)]+1 = \lceil n/2 \rceil +1$ in the i–th row; therefore, the majority of the i–th row in MAT_j is v_i.

Case 2: $v_{ij} = \neg v_i$

There are at most $\lfloor n/2 \rfloor -1$ faulty links. In the second round, the number of $\neg v_i$'s is no more than $(\lfloor n/2 \rfloor -1)+1 = \lfloor n/2 \rfloor$ and the number of v_i's is at least $[(n-1)-(\lfloor n/2 \rfloor -1)]=\lceil n/2 \rceil$. If n is an even number, then $\lfloor n/2 \rfloor = \lceil n/2 \rceil$, the majority of the i–th row in MAT_j is ?. If n is an odd number, then $\lfloor n/2 \rfloor < \lceil n/2 \rceil$, hence the majority of i–th row in MAT_j is v_i.■

Lemma 4: If $(\neg\exists\ MAJ_k=\neg v_i)$ AND$\{(\exists\ MAJ_k = ?)$ AND $(v_{ki}=v_i)\}$ in MAT_i, then $DEC_i := \phi$ is correct.

Proof: If there has a $MAJ_k = ?$, there are exactly n/2 v_i's and n/2 $\neg v_i$'s in the k–th row. If $v_{ki} = v_i$ in MAT_i, then all n/2 $\neg v_i$'s should be received in the second round. There are $\lfloor n/2 \rfloor -1$ faulty links in the system. Therefore, in the second round, node i at least receives a value from node k without disturbance. The initial value of node k should disagree with the initial value of node i; hence it is correct to choose $DEC_i = \phi$.

If $v_{ki} = \neg v_i$, we claim that $\neg v_i$ ought to be passed by a faulty link from node k, and the initial value of node k should be $\neg v_{ki} = v_i$.

To prove, if link ki is perfect, then the initial value of node k should be $\neg v_i$. By Lemma 2, the majority value of the k–th row in MAT_i is $\neg v_i$. This is contradiction with the condition of $(\neg\exists\ MAJ_k = \neg v_i)$.

If the initial value of node k was $\neg v_i$, then, by Lemma 3, MAJ_k should be either $\neg v_i$ or ? for $v_{ki} = v_i$. It is a contradiction.■

Theorem 1: FLINK protocol is correct.

Proof: By Lemma 1, 2, 3 and 4, the theorem is proved.■

Theorem 2: FLINK protocol can achieve consensus.

Proof: (1) *Agreement*:

Part 1: If a correct node agrees on ϕ, all correct nodes should agree on ϕ.

If the correct node m with initial value v_i agrees with ϕ, by Theorem 1, at least there is a correct node k with initial value $\neg v_i$ in the network. By Lemma 4, the majority value in the k–th row of MAT_j, $1 \le j \le n$, should be either $\neg v_i$ or ? for $v_{kj}=v_i$. All correct nodes with initial value v_i agree on ϕ. Similarly, for the correct nodes with initial value $\neg v_i$, the majority value of the m–th row in MAT_j, $1 \le j \le n$, should be either v_i or ? with $v_{ij} = \neg v_i$. All correct nodes with initial value $\neg v_i$ agree on ϕ, too.

Part 2: If a correct node agrees on v_i, all correct nodes should agree on v_i.

If the correct node i with initial value v_i and $DEC_i = v_i$, but there exists some correct node j, $j \ne i$, has $DEC_j \ne v_i$, then that is impossible. To show this, if $DEC_j = \phi$, by Part 1, then $DEC_i = \phi$. This is a contradiction with the assumption as above.

If $DEC_j = \neg v_i$, unless the initial value of node j is $\neg v_i$, otherwise it is impossible according to the Definition in Section 3. But if the initial value of node j is $\neg v_i$, by Lemma 4, MAJ_i is equal to $\neg v_i$ or ? with $v_{ji} = v_i$ in MAT_i; then, $DEC_j = \phi$. It is a contradiction; hence, all correct nodes should agree on the same value.

(2) *Validity*:

The initial value of all correct nodes should be the same. If there is a value $\neg v_i$ in MAT_j, $1 \le j \le n$, then the value must be caused by a faulty link. There are at most $\lfloor n/2 \rfloor -1$ faulty links, hence there are at most $\lfloor n/2 \rfloor -1$ faulty $\neg v_i$'s in each row. Since the value received in the first round may be $\neg v_i$, the majority of each row for all MAT_j, should be

$$MAJ_j = \begin{cases} ? & \text{if the value received in the first round is } \neg v_i, \quad 1 \le j \le n. \\ v_i & \text{otherwise.} \end{cases}$$

So, by step 3 of the FLINK protocol, all correct nodes should agree on v_i.■

Theorem 3: The amount of information exchange by FLINK is $O(n^2)$.

Proof: In the first round, each node sends out (n–1) copies of its initial value to other nodes. In the second round, an n–element vector is sent to the other n–1 nodes in the network; therefore, the total number of message exchange is $(n-1) + (n^*(n-1))$. This result implies that the complexity of information exchange is $O(n^2)$.■

4. Impossibility

In this section, some impossibility of the consensus problem is presented for the case of all perfect nodes on an unreliable message communication system. First we show that the completeness of a consensus by using less than two message exchanges is impossible. Next, when the number of the faulty links is greater than $\lfloor n/2 \rfloor -1$, it is impossible to obtain a consensus. Based on these results, we can show that the FLINK protocol is optimal in the sense that it uses the minimum number of rounds and can tolerate the maximum number of faulty components by the following theorems.

Theorem 4: One round of message exchange to achieve consensus is impossible.

Proof: Part 1: Message exchange is necessary.

Without message exchange, a node can't know whether or not a disagreeable value exists in ·other nodes; hence consensus achievement is impossible.

Part 2: One round message exchange is not enough to achieve consensus.

If node i is connected with node j by faulty link ij. Node i may not know the initial value in node j by using only one round of message exchange.

Therefore it is impossible to achieve consensus by using only one round message exchange.■

Theorem 5: If the number of the faulty links t > $\lfloor n/2 \rfloor -1$, achieving consensus is impossible.

Proof: When t > $\lfloor n/2 \rfloor -1$ and n is an even number, then each node has n–1 links in the system, it is possible that there is a node which has more faulty links than a perfect link. Regardless of the number of rounds of message exchange, this node will always be confused by the message transferred through those faulty links. The decision made by the node may conflict with other nodes. In this case, consensus achievement is impossible.■

Theorem 6: Using the minimum number of rounds, FLINK can tolerate the maximum number of faulty links in a perfect node, fully–connected network.

Proof: From Theorem 2, Theorem 4 and Theorem 5, the theorem is proved.■

5. Conclusion

Previous works about consensus problem are based on the assumption that nodes are the only fallible components in the network [1,7]; however in a generalized case, both the nodes and links of a fully–connected network could be in faulty. The behavior of a faulty node can effect the other nodes in a fully–connected network; while the behavior of a faulty link will only effect the two adjacent nodes. If the number of allowable faulty components in the system is given; then in a generalized case, the correct nodes connected by faulty components is less than the correct nodes connected in a conventional case; therefore the consensus obtained in a generalized case will be reached earlier than that of a conventional case. For the similar reason, if the number of required rounds is fixed, the fault tolerant capability in a generalized case could be stronger than that in a conventional case. For a faulty link case, FLINK protocol solves the consensus problem by using $\lfloor n/2 \rfloor -1$ faulty links and two rounds of message exchange. The amount of message exchange is $O(n^2)$.

In short, the faulty link case or the conventional faulty node case can be viewed as a special case for the generalized consensus problem in which both node and link can be in faulty. In a generalized case, FLINK protocol can still be used to solve consensus problem by using less rounds than that of a conventional faulty node case. In a generalized case, the number of faulty nodes is less than that of a conventional faulty node case, if the number of faulty components is the same.

REFERENCES

[1] G. Bracha, and S. Toueg, "Asynchnorous Consensus and Broadcast Protocols," JACM, vol. 32, no. 4, pp. 824–840, Oct. 1985.

[2] D. Dolev, "Unanimity in an Unknown and Unreliable Environment," IEEE FOCS, 1981.

[3] D. Dolev, "The Byzantine Generals Strike Again," J. of Algorithm, vol. 3, pp. 14–30, 1982.

[4] D. Dolev, and R. Reischuk, "Bounds on Information Exchange for Byzantine Agreement," JACM, vol. 32, no. 1, pp. 191–204, Jan. 1985.

[5] M. Fischer, and N. Lynch, "A Lower Bound for the Assure Interactive Consistency," Information Processing Letters, vol. 14, no. 4, pp. 183–186, June 1982.

[6] M. Fischer, "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)," Lecture Notes in Computer Science, Proceeding of the 1983 International FCT–Conference, Borgholm, Sweden, pp. 127–140, Aug. 1983.

[7] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," ACM Transactions on Programing Languages and Systems, vol. 4, no. 3, pp. 382–401, July 1982.

[8] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in Presence of Faults," JACM, vol. 27, no. 2, pp. 228–234, April 1980.

[9] R. Reischuk, "A New Solution for the Byzantine Generals Problem," IBM Research Report, RJ–3673, Computer Science, 1982.

[10] H. Strong, and D. Dolev, "Byzantine Agreement," IBM Research Report, RJ–3714, Computer Science, 1982.

[11] R. Turpin, and B. Coan, "Extending Binary Byzantine Agreement to Multivalued Byzantine Agreement," Information Processing Letters, vol. 18, no. 2, pp. 73–76, Feb. 1984.

A RELIABILITY PREDICTOR FOR MIN-CONNECTED MULTIPROCESSOR SYSTEMS

John J. Macaluso, Chita R. Das, and Woei Lin

Computer Engineering Program
Department of Electrical Engineering
The Pennsylvania State University
University Park, PA 16802

ABSTRACT

In the world of cost-effective supercomputing, the use of a multistage interconnection network (MIN) as a means of connecting many processing elements to many memory modules is widespread. Whenever such a system is used in a critical environment, reliability becomes an important issue. Up to this point reliability evaluation methods for multiprocessor systems have been *ad hoc*, that is, designed for, and applicable to, only one or a few types of topology.

This paper presents the first automated simulation package with the ability to perform the reliability simulation of MIN-connected systems. The program is automated in that the required MIN topology is built by the program. The user need only specify the type of MIN and other system characteristics. The underlying strategy of the program is to find the system reliability from the system reachability matrix, which is built by a search procedure requiring $O(NS(N))$ time, where $S(N)$ is the number of switches in an $(N \times N)$ MIN. The package was used to simulate the reliabilities of many topologies proposed in the literature. Some results are presented and used for a comparison of the systems.

1. INTRODUCTION

Multiprocessor systems using multistage interconnection networks (MINs) have been an active area of research for more than a decade. A plethora of different MIN topologies have been proposed to provide communications among N processors (PEs) and N memory modules (MMs). These MINs are generally designed with stages of $(n \times m)$ switching elements (SEs), where n and m are small integers such as 2, 3, or 4. A good body of literature on MINs can be found in [1], and a survey of some fault-tolerant multipath MINs is reported in [2].

The novelty of a multiprocessor lies in its ability to provide high computing power with assured reliability. Reliability becomes important especially when the system is used in a critical application. While performance analyses of the MIN-based systems have been carried out extensively along with their design, relatively little attention has been paid to the reliability issues. Work on fault-tolerant MINs has been mostly confined to finding alternate paths between source and destination sets.

In the past, research pertaining to the reliability evaluation of MINs has addressed either full connectivity without degradation [3] or terminal reliability [4]. A couple of papers have addressed the complete reliability of MIN-based systems considering the failure of PEs, MMs, and SEs [5], [6]. However these works are restricted in a sense that either the system size is limited or the evaluation technique is not applicable to all types of MINs. Recently a combinatorial approach for reliability evaluation of multiprocessors using (4×4) SEs is given in [7]. This analysis is applicable to only unique-path MINs with (4×4) SEs.

As more and more fault-tolerant MINs are proposed, it is essential to develop a methodology for characterizing and comparing one system with another from the reliability standpoint. This type of unified evaluation technique will solve two purposes. First, the reliability of any existing or new MIN-based system can be predicted. Second, depending on the implementation requirements, a cost-effective MIN can be selected. The survey work in [2] has compared the fault-tolerant property of various multipath MINs. However, the work is not complete in a sense that the usual evaluation criterion such as the reliability/availability issue is not addressed. In this paper we are concerned with developing a unified reliability evaluation technique for various types of MIN-based systems.

Analytical evaluation of system reliability considering the degradation of PEs, MMs, and SEs is very difficult due to the NP-hardness of the problem [8]. Therefore, all the analytical evaluation techniques have been restricted to mostly unique-path MINs. As we are interested in analyzing and comparing different multipath strategies, an analytical approach seems almost impossible. Hence, simulation is used as the evaluation tool. This paper presents the first proposed automated simulation package with the ability to perform the reliability evaluation of MIN-connected multiprocessor systems.

The package takes from the user an input file containing the specifications of the topology and the details of the type of analysis desired. The user is freed from the interconnection details because the program has the ability to automatically build the proper topology. A search algorithm is employed during the course of the simulation to find the connectivity between PEs and MMs in the presence of component failures. While the size of the system is not limited by the program, the host machine environment and simulation time may be limiting factors. The reliability model used in this paper is known as task-based reliability [9], where a system remains operational as long as a task can be executed on it. Results of (16×16) and (64×64) systems using the following topologies are analyzed in the paper with and without system cost factor involved.

The topologies considered are (2×2) baseline [10], an extra-stage baseline, the (4×4) butterfly [11], the chained MIN [12], [13], the F network [14], the merged delta network (MDN) [15], the inverse augmented data manipulator (IADM) [16], and the interconnection network designed for reliable architectures (INDRA network) [4]. Although this selection covers almost the whole spectrum of MINs proposed in the literature, the program is not limited to only these topologies. It also has the ability to include virtually any MIN-based system. We do not fully describe each of these systems; more complete system descriptions can be found in the literature cited. Since the topologies chosen are representative of those surveyed in [2], this work could be considered a follow-up or extension of the work presented there.

In Section 2 we present an overview of the topologies considered. The simulation techniques are explained in detail in Section 3, including algorithm time complexities. Section 4 gives the results of the system simulations and offers a comparison between them. Concluding remarks are given in Section 5.

2. SYSTEMS SURVEY

This section briefly surveys the different MIN-connected multiprocessor systems listed in the introduction to this paper. We consider a tightly coupled multiprocessor environment where the PEs and MMs are connected through the MIN. The difference between the systems lies solely in the type of interconnection network used for communications among the processors and memories, and therefore each system will be described by its MIN.

2.1 Unique-Path MINs

A unique-path MIN provides only one path between each processor and each of the memory units. The advantage of unique-path networks lies in the simplicity of their implementation. The network uses uncomplicated selector or crossbar switches that require no look-ahead capability (i.e., they need not be independently cognizant of the conditions of the other components in the system). Each switch, if operational, merely routes an input to the proper output link depending upon the value of the request tag bit corresponding to the switch's stage. This simplicity results in fewer internal components and consequently a high average switch reliability relative to more complicated switches, such as those used in the multiple-path systems described later.

The disadvantage of unique-path networks lies in the fact that if any of the switches along a desired path fails, the entire path is eliminated, and the requesting module is unable to access the requested module. However, if a strategy exists whereby another path can be found to act as a detour around the failed element, this fault may be tolerated. These extra paths can be provided at the expense either of redundant passes through the network [17], [18] or of additional hardware; we will consider the latter.

The baseline MIN is one example of an unique-path network. The topology of an (8×8) baseline MIN is shown in Fig. 1. The baseline network consists of $n = \log_2 N$ switch stages, each containing $N/2$ (2×2) crossbar switches. The baseline MIN was chosen to represent other unique-path MINs proven to be topologically equivalent to the baseline by Wu and Feng [10]. We use the baseline in our explanations because of its simplicity of representation.

Another unique-path network, comprised of (4×4) switches, is the butterfly MIN. The BBN Butterfly Parallel Processor^{TM} is a commercially available system with up to 256 processors [11]. Because of the additional links in each switch, the butterfly MIN has only $\log_4 N$ stages each consisting of only $N/4$ switches. Thus, the communications delay of a butterfly MIN is only $O(\log_4 N)$, whereas the baseline delay is $O(\log_2 N)$.

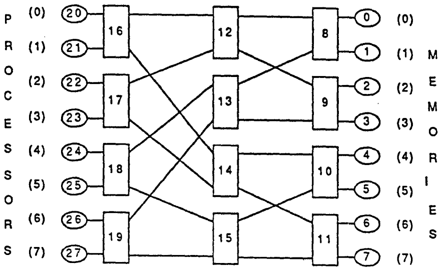


Fig. 1. A (8×8) baseline system.

2.2 Path Redundancy for Unique-path MINs

It may be possible to provide redundant paths to an unique-path MIN by at least four methods. The first involves the addition of an extra stage of switches to the input of the MIN. This is done by duplicating the MIN input stage along with its output link interconnection pattern; this extra stage is inserted between the processors and the previous input stage. This method is examined in this research by the addition of an extra stage to both the baseline and the butterfly MINs. A second redundancy method adds chaining links to each switch, partitions all the switches, and then chains together the switches in the same partition. The chained baseline MIN was examined as an example of this method. A third redundancy strategy consists of replicating r times a network consisting of $(r \times r)$ switches (e.g., the baseline MIN consists of (2×2) crossbar switches, therefore 2 network copies would be provided). This INDRA technique is examined for the baseline network. Finally, the fourth technique crosslinks c copies of an $(\frac{N}{c} \times \frac{N}{c})$ unique path network. This technique can be applied to any of the topologically equivalent unique-path MINs, but traditionally, it is employed using the delta topology to form the MDN.

2.3 Inherently Path-redundant MINs

Anticipating the need for redundant paths in a MIN, some topologies have been designed to provide these multiple paths without a need for further modification. One example, the IADM, uses $(\log_2 N) + 1$ switch stages, each consisting of N (3×3) switches. Another example, the F network, uses $\log_2 N$ switch stages, each with N (4×4) switches. These two networks show most clearly the truism that statically redundant paths require redundant hardware.

3. SIMULATION TECHNIQUES

In this research, it is assumed that the MIN of each of the systems considered has an input side and an output side, and all communications between modules are carried out in one pass from an input position to an output position. Using the system of Fig. 1 as an example, communications between processor 0 (node 20) and processor 3 (node 23) would follow the node path 20 → 16 → 12 → 9 → 3 and not 20 → 16 → 12 → 17 → 23. This example also illustrates the equivalence of input and output positions.

Depending upon the implementation, each of the topologies can be operated under either a circuit-switched or a packet-switched communications protocol. Under circuit switching, a physical link is established between two modules, and is used for transmissions in both directions. In contrast, packet switching is an asynchronous simplex protocol where information packets are exchanged via the network. The program has the ability to simulate either of these protocols.

To use the program, the user need merely edit an already-existing input file. The information in the input file includes the following.

a. The system size.
b. The system type (from a menu list).
c. The communications protocol.
d. The failure rates of PEs, MMs, and SEs.
e. Whether each PE is assigned a local MM.
f. The number of copies of the MIN (for INDRA case).
g. The output file name.

At the beginning of the simulation, this information is read into the program. The program then calls the appropriate procedure to build the desired topology. The ability to build the topology automatically (described later) is especially important in the case of large systems, when hand entry of the interconnection pattern becomes very difficult.

The simulator determines the system condition, whether operational (up) or failed (down), from **R**, the $(N \times N)$ system reachability matrix. **R** describes the connectivity between modules in the following way: if at least one path exists between processor p and memory m, then the matrix element $R[p, m] = 1$, otherwise $R[p, m] = 0$. In an unfailed system (at system startup), $\mathbf{R}=\mathbf{1}$. As components fail, the degree of system degradation can be determined from **R**. If a system is defined as being up if it has at least i processors and j memories all being both operational and completely connected to each other, then the system condition can be determined by examining **R** to ascertain whether a submatrix of order at least $(i \times j)$ and with elements all of value 1 can be found in **R**.

The simulator is based upon the following concept. The reliability evaluation of any system is dependent upon its reachability matrix. Therefore, if a program can be developed to find system reliability from **R**, and if any MIN-based system can be reduced to its reachability matrix, then the reliability of any MIN-based system can be simulated.

The algorithm for simulating system reliability from **R** is as described in [5] and will not be detailed here. The remainder of this section will explain the program with respect to internal system representation and the characterization of a search-traversal algorithm capable of finding **R**. The serial version of the search algorithm is given, and possibilities for a parallel implementation are explained.

3.1 System Representation

The topology of each network is represented by certain constants and arrays as follows (all parenthesized examples correspond to the system of Fig. 1). The system constant a is the total number of PEs, MMs, and SEs present in the system (e.g., $a = 2N + \frac{N}{2}\log_2 N = 28$); the system constant $offset$ $(= a - N)$ is the vertex number of processor 0 (e.g., $offset = 20$); finally, the system constant b is the maximum number of output links per node present in the system (e.g., $b = 2$). The vertices are numbered from 0 to $a-1$ in column-major order beginning with the vertex corresponding to memory 0 and ending with that of processor $N-1$. The output links (if any) are numbered from 0 to $b-1$ from top to bottom. The $(a \times b)$ matrix, **T**, describes the interconnection pattern of each topology as follows: the element $T[i, j]$ is the component connected to output link j of component i, where $i \in \{0..a-1\}$ and $j \in \{0..b-1\}$ (e.g., $T[13, 0] = 8$). By convention, if $T[i, j] = -1$, then output link j does not exist for component i (e.g., $T[3, 1] = -1$). The one-dimensional boolean array **living** represents the system component failure condition as follows: for all $i = 1 \cdots a-1$, **if** $living[i]$ **then** component i is operational **else** i is failed.

Since all the systems can be represented by this scheme, the user need only indicate the network topology and size. The program calculates the system constants and calls the appropriate procedure to build **T**.

These representations are needed because the procedure which finds the reachability matrix of any system is a search-traversal algorithm. A search-traversal strategy is necessary since the conventional methods of finding **R** reported in [5] do not work in the case of multipath systems such as those surveyed in [2].

3.2 A Serial Algorithm for Finding R

As described in [19], a MIN-based multiprocessor system can be represented by a directed graph. Since this is true, it follows that an $(N \times N)$ system can also be conceptualized as a grove of N search trees, where the root of each tree corresponds to an individual processor vertex, its leaves represent the memory units, and its shape depends upon the network topology. The search tree for the unfailed system of Fig. 1 can be seen in Fig. 2(a). The effect of a component failure will be to prune the tree at the appropriate position(s) held by that component in the tree. For example, if the switch cořresponding to node 12 of Fig. 1 fails, the resulting search tree is as seen in Fig. 2(b).

The problem, then, of finding the connectivity between any processor p and any memory m at a given time reduces to: is m present in the search tree of p? This can be accomplished by initializing **R** to **0**, and then performing a reverse preorder traversal of the search tree of p, setting to 1 the appropriate elements

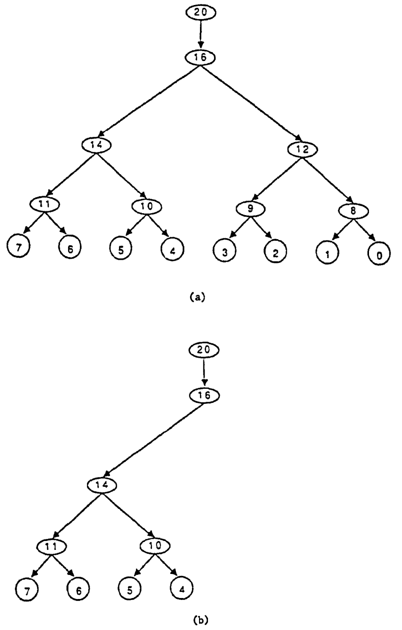


Fig. 2: Search tree for processor 0 of Fig. 1. (a) For an unfailed system. (b) Pruned after failure of node 12.

of **R** each time a leaf vertex (memory) is reached. The termination of the traversal search of the tree of processor p results in the completion of row p of **R**. Therefore, N such searches will complete the entire reachability matrix. The *Search* algorithm of Fig. 3 performs the search of one tree, and the *Reach* algorithm of Fig. 4 uses *Search* to build the reachability matrix, **R**.

The stack of *Search* is initially empty. At the beginning of the search, the *source* vertex is pushed onto the stack. The following steps are repeated while the stack is not empty.

1. A *node* is popped from the stack and checked if it is a leaf; if so, then the proper element of **R** is set to 1.
2. The processed *node* is tagged "visited."
3. The unvisited neighbors of *node* are each pushed onto the stack in birthright order.

Birthright order is from leftmost child to rightmost child in the tree representation. For example, birthright order for vertex 16 of Fig. 1 is vertex 14 and then vertex 12. This order ensures the depth-first traversal desired. The marking of the processed vertices as "visited" serves two purposes.

1. The algorithm is kept from infinitely traveling around any closed loops possibly inherent in a topology (such as the very visible loops of the chained baseline MIN structure).
2. A mechanism is provided by which future consideration of a vertex once processed is denied. This helps to reduce search time by ensuring that each vertex is processed only once, thereby providing an extra stage of pruning in systems whose search trees have vertices holding multiple positions. It is important to note that this pruning mechanism makes this algorithm suitable only for finding whether at least one path exists to each memory—not for finding the number of paths to each memory.

```
procedure  Search (source : integer)
begin
   Mark each vertex "not visited";
   Zero row $source - offset$ of R;
   Reset the local stack;
   Push source onto the stack;
    while stack is not empty  do  begin
      node := stack pop;
       if $node < N$  then  begin
         (* node is a memory *)
         $R[source - offset, node] := 1$
       end;  (* if *)
      Mark node "visited";
       for each of node's children $i$  do
       begin
          if ($i$ is  not visited)
          and (($i < N$)
          or ($i$ is living))  then  begin
            Push $i$ onto stack;
          end;  (* if *)
       end;  (* for *)
    end;  (* while *)
end; (* procedure *)
```

Fig. 3. Tree-search algorithm.

```
procedure Reach;
    procedure  Search;
begin
L1: for $p := 0$  to $N - 1$  do begin
       $source := p + offset$;
       Search (source);
     end;  (* for *)
     if packet-switched protocol  then
     begin (* packet adjust *)
L2:     for $p := 0$  to $N - 1$  do begin
          for $m := p + 1$  to $N - 1$  do
          begin
             if $R[p, m] = 0$
             or $R[m, p] = 0$  then
             begin
                $R[p, m] := 0$;
                $R[m, p] := 0$;
             end;  (* if *)
          end;  (* for *)
        end;  (* for *)
     end;  (* if *)
 L3: for each processor $p$  do begin
        if $p$ is failed  then
          zero row $p$ of  R;
     end;  (* for *)
 L4: for each memory $m$  do begin
        if $m$ is failed  then
          zero column $m$ of  R;
     end;  (* for *)
 end; (* procedure *)
```

Fig. 4. Algorithm for finding **R**.

3.3 Algorithm Time Complexities

It is important that the time complexities of *Search* and *Reach* be calculated since they are used frequently in the simulation program. In the calculations that follow, the function $S(N)$ gives the number of switches in the MIN as a function of N (e.g., $S(N) = \frac{N}{2}\log_2 N$) for a $(N \times N)$ baseline MIN). The relative growths of N and $S(N)$ are topology dependent. In the case of the F network, for example, $S(N) = N\log_2 N > N$ always. However, in the separate example of the butterfly network, $S(N) = \frac{N}{4}\log_4 N$, which is greater than N only for large N (i.e., $N > 256$). Since the asymptotic time complexities concern systems with large N, it is assumed in the derivations below that in all cases $S(N) > N$.

3.3.1 Time Complexity of the *Search* Procedure

The *Search* procedure of Fig. 3 consists of two parts: the nonsearch statements (those before the **while** loop) and the statements of the search loop (those making up the **while** loop). The time complexity of the nonsearch statements is $O(S(N))$ because the dominating term is the Mark statement since it initializes all $2N + S(N)$ vertices, and $S(N)$ grows faster than N. The search statements consist of: three assignment statements, each with constant time complexity, and a **for** loop of $O(b) = O(1)$ time complexity since the maximum number of links per node, b, remains constant as the size of the system grows. Thus the statements within the **while** loop are all of constant time complexity, and the time complexity of the entire loop is governed by the maximum number of iterations of the loop as follows. The marking as visited of each processed node ensures that each vertex is considered only once during each tree traversal. Since the maximum number of vertices that a single search can consider is the $S(N)$ switches plus the N memories plus the processor root, the time complexity of a search is $O(S(N))$. Therefore, the time complexity of procedure *Search* is

$$T_S(N) = O(S(N)). \qquad (1)$$

3.3.2 Time Complexity of the *Reach* Procedure

Procedure *Reach* can be seen to consist of four loops labeled $L1$ through $L4$ in Fig. 4. Loop $L1$ performs N successive calls on *Search*, and therefore has an $O(NS(N))$ time complexity. Loops $L3$ and $L4$ each perform N iterations of constant-time conditional assignment statements, so each has an $O(N)$ time complexity. Loop $L2$ performs constant-time conditional assignments as many times as there are elements of $\mathbf{R}$ above the main diagonal, or $\frac{1}{2}(N^2 - N)$. Since L1 is the dominant loop, the time complexity of *Reach* is

$$T_R(N) = O(NS(N)). \qquad (2)$$

3.4 A Parallel Algorithm for Finding R

Time requirements for the searches can be lessened even further by performing them in parallel. The parallelism of the *Search* algorithm is apparent; each search is completely independent of every other search (i.e., there are no data dependencies between the searches). This parallelism can be easily exploited on an array of P processors, where $P = 2^n$ for positive integer n. If $N > P$, the technique of loop concurrentization [20] could be used without much difficulty by partitioning the rows of $\mathbf{R}$ (searches) and assigning a different partition to each processor. For ease of explanation, however, we will consider the case where $P = N$, where each of the processors would be assigned a different row of $\mathbf{R}$.

When a packet-switched protocol is being simulated, the execution of the packet-adjustment statements in loop L2 of Fig. 4 introduces data dependencies, and communications between processors becomes necessary. Specifically, each processor adjusts its row of $\mathbf{R}$, but only the elements to the right of the main diagonal. Each of these elements $R[i,j]$, where $i > j$, is compared to the element $R[j,i]$ symmetrical to it with respect to the main diagonal. But before a processor i can properly examine an $R[j,i]$ value, it must receive a signal from processor j that the search of row j is complete.

The blocks labeled L1 through L4 of Fig. 4 can each be reduced by a factor of N if the procedure is implemented in parallel. In this case, the dominant execution sequence would be the *Search* procedure of L1. Therefore, if procedure *Reach* is performed in parallel with $P = N$, the time complexity is $O(S(N))$ by Eq. 1.

4. RESULTS AND DISCUSSION

This section compares the selected topologies with respect to their simulated reliabilities. In addition, since the design emphasis on MINs is inspired by a need for cost-effective communication networks, the systems were compared with respect to the ratio of system reliability to system cost (or reliability-to-cost ratio, RCR). The program also has the ability to simulate a system with any given coverage factor, C [21]. Results were obtained for systems of different sizes under both circuit- and packet-switched protocols and with different values of C. However only selected outputs are presented here.

4.1 Elements of the Comparison

For the comparisons to be valid, the processors as well as the memories were assumed to be homogeneous within each system, and the same processor and memory failure statistics were used in each type of system. In this way, each network differed from the others only in its type of MIN.

The reliability of each system is directly related to the mean failure rates of the individual elements making up the interconnection network. The mean failure rates for processors, λ_p, and memories, λ_m, were each taken to be one per 10^4 hours. Since the systems differ only in the type of MIN used, any change in λ_p or λ_m will affect all of the systems equally. Therefore, the comparison depends upon the failure rate of the MIN, which depends upon the failure rates of the individual switching elements. In the absence of any practical failure data, switch failure rates were calculated using the MIL-HDBK-217B reliability model for metal-oxide-semiconductor integrated circuits [22]. Details of the assumed switch design and failure-statistic calculations can be found in [23]. The program considers the MIN to consist of three sets of switches: the input bank, the output bank, and the banks between the input and the output banks. Then the characteristic switch failure rate is assigned to the switches of each set.

The network cost factor is the sum of the costs of all the individual switches comprising the network. The cost of each switch is calculated as a function of the number of its input links, n, and the number of its output links, m, using the equation of the cost function, $C(n,m)$, of Eq. 3.

$$C(n,m) = \begin{cases} nm & \text{for a crossbar switch;} \\ n+m & \text{for a selector switch.} \end{cases} \quad (3)$$

Eq. 4 calculates C_N, the cost of a MIN consisting of x different types of switch, each type i having a cost C_i and a population N_i. The network cost factors are then calculated by dividing each network cost by the minimum cost of all the networks of the same size (in this case the baseline).

$$C_N = \sum_{i=1}^{x} N_i C_i \quad (4)$$

4.2 System Reliability Comparison

The reliability curve for a task requiring 50% of the total number of PEs and 50% of the total number of MMs was obtained for each of the systems.

Figs. 5 and 6 contain the (16 × 16) system curves for a circuit-switched protocol and a coverage factor of $C = 1.0$ and $C = 0.8$ respectively. The difference between the curves of Fig. 5 is noticeable, however when the system's ability to reconfigure itself is relatively weak as in Fig. 6, the fault-tolerant scheme has less of an effect on reliability. In fact, with a coverage factor of 0.8, the topology of the MIN seems to have almost no effect at all on reliability; the curves are almost indistinguishable from each other. This observation follows intuition: a topologically inherent fault-tolerance scheme is of benefit only if the maintenance processor is able to utilize it.

The reliability curves for (64 × 64) systems under a circuit-switched protocol with $C = 1.0$ are shown in Fig. 7. As expected, the INDRA, F network, chained MIN, MDN, and extra-stage MINs give high reliability compared to the unique-path MINs. Also, as the system size increased, the reliability gain of multipath networks becomes more pronounced.

Probably the most surprising curve, however, is that of the IADM system. It does not seem to agree with intuition that it would have a reliability consistently below all the others, including the unique-path systems. However, upon further inspection, the reasons become clear. The IADM has multiple paths, but they are not evenly distributed between all processor-memory pairs. For example, there is only one path between processor i and memory j when $i = j$. However, probably the most significant reason for the low reliability is that the IADM contains $N(\log_2(N)+1)$ switches—many more than the $\frac{N}{2}\log_2 N$ switches of the baseline or the $\frac{N}{4}\log_4 N$ switches in the butterfly. If the failure rate of an individual switch is λ_s, then the failure rate of the switches in the system is given by $\sigma\lambda_s$, where σ is the number of active switches at any time. Clearly, if σ is very large (as in the IADM), then the switches will fail much more frequently than if σ is small (as in the baseline, or especially the butterfly). Therefore, the combination of unevenly distributed paths and quick-failing switches makes the IADM a less reliable system compared to the others.

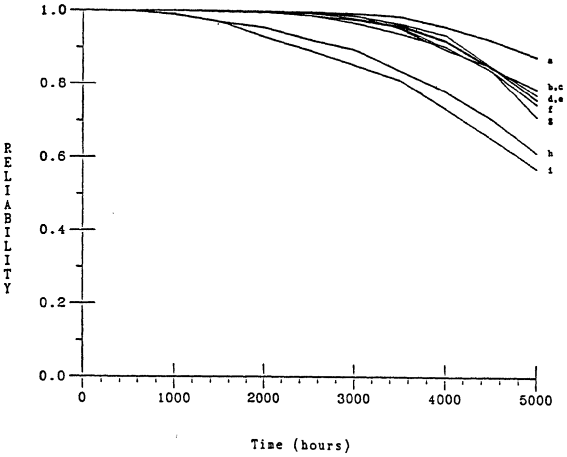


Fig. 5.

$R(t)$ for (16 × 16) circuit-switched systems with $C = 1.0$.
(a) INDRA.
(b) F network.
(c) MDN.
(d) Baseline w/ extra stage.
(e) Chained baseline.
(f) Butterfly.
(g) Butterfly w/ extra stage.
(h) Baseline.
(i) IADM.

4.3 System RCR Comparison

When the cost of a system is to be considered along with its reliability, a useful measure is the reliability-to-cost ratio (RCR), i.e., the reliability is divided by the system cost factor. This ratio serves as a comparison between the networks surveyed in Section 2. We observed that for smaller (16 × 16) systems, the RCR puts the unique-path butterfly and baseline MIN systems at the top of the ranking. The increased reliability of the more complicated multiple-path MIN systems does not compensate for the extra system cost. However, as the size of the system grows, the curves for the unique-path systems fall below some of those with multiple-paths, as seen for a (64 × 64) system in Fig. 8. In these larger systems, the extra cost begins to be of some benefit, especially the addition of an extra stage to the baseline which takes the number one spot. When cost is considered, the F network falls from the upper positions of the $R(t)$ curves to occupy the lower two positions along with the IADM system in the RCR curves.

4.4 Summary of System Comparisons

From the examination of the curves of Figs. 5 through 8, the following system evaluation is offered.

These observations are based upon the particular switch failure calculated as described above.

The best overall reliability is offered by the extra-stage baseline MIN. The reliability of this topology ranked in the top four. Its value is most clearly seen, however, in the RCR comparison of (64 × 64) systems, where it ranks in the number one spot. This indicates that for large systems where cost is a consideration, the best fault tolerance technique is the addition of an extra stage on a baseline (or topologically equivalent) system. The IADM system was the least reliable of the systems compared due to the reasons mentioned earlier.

Although the simulation of large systems takes a lot of computer time, a comparison of Figs. 5 and 7 shows that the effect of the MIN topology on reliability has a greater effect as the size of the system grows. This indicates rather strongly that the algorithms described in this report should be implemented on a large-grain parallel processor.

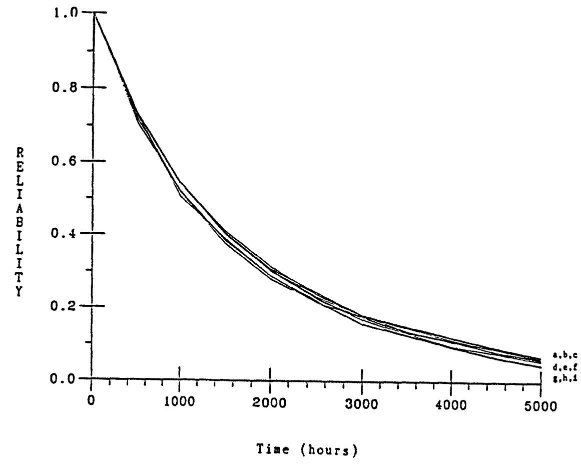


Fig. 6.

$R(t)$ for (16 × 16) circuit-switched systems with $C = 0.8$.
(a) Chained baseline.
(b) Baseline w/ extra stage.
(c) INDRA.
(d) Butterfly w/ extra stage.
(e) Butterfly.
(f) MDN.
(g) Baseline.
(h) F network.
(i) IADM.

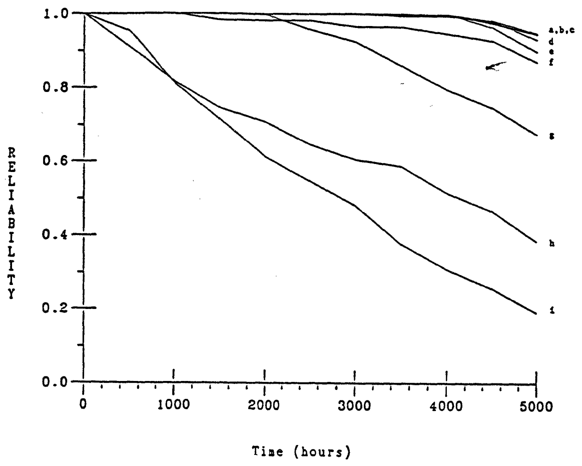


Fig. 7.

$R(t)$ for (64 × 64) circuit-switched systems with $C = 1.0$.
(a) Chained baseline.
(b) INDRA.
(c) Baseline w/ extra stage.
(d) MDN.
(e) Butterfly w/ extra stage.
(f) F network.
(g) Butterfly.
(h) Baseline.
(i) IADM.

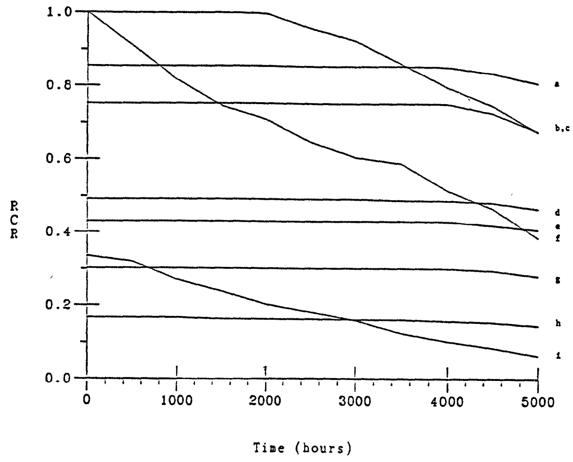


Fig. 8.

RCR for (64 × 64) circuit-switched systems with $C = 1.0$.
(a) Baseline w/ extra stage.
(b) Butterfly w/ extra stage.
(c) Butterfly.
(d) Chained baseline.
(e) INDRA.
(f) Baseline.
(g) MDN.
(h) F network.
(i) IADM.

5. CONCLUSIONS

This paper reports the first automated package with the ability to simulate the system reliability of virtually any MIN-based multiprocessor system. The program accepts the type of MIN and various failure rates from the user. It builds the MIN automatically and stores the interconnection pattern in a matrix. A traversal-search algorithm is used to find the reachability matrix of the system with random faults. The reachability matrix in turn is used in the calculation of the system reliability. The unified approach of this package makes possible system reliability predictions as well as system comparisons with respect to reliability issues.

The package in its present form provides the framework around which many other features can be built. For example, one important extension could be the ability to measure system performance in the presence of faults. Addition of this performance predictor to the reliability model will give the program the capability to predict performance-related reliability measures. Another addition could be the capability to predict the coverage factor of the system from a model of the individual processors and the maintenance processor. In this way, the coverage factor, shown in this research to be so important to system reliability, could be calculated rather than estimated and provided by the user.

REFERENCES

[1] C. -L. Wu and T. -Y. Feng, "A tutorial on interconnection networks for parallel and distributed processing," IEEE, 1984.

[2] G. B. Adams III, D. P. Agrawal, and H. J. Siegel, "A survey and comparison of fault-tolerant multistage interconnection networks," *IEEE Comput.*, vol. 20, pp. 14–27, June 1987.

[3] J. T. Blake and K. S. Trivedi, "Multistage interconnection network reliability," submitted to *IEEE Trans. Comput.*, 1988.

[4] C. S. Raghavendra and A. Varma, "INDRA: a class of interconnection networks with redundant paths," in *1984 Real-Time Systems Symp.*, Computer Society Press, Silver Spring, MD, 1984, pp. 153–164.

[5] C. R. Das and L. N. Bhuyan, "Reliability simulation of multiprocessor systems," in *Proc. 1985 Int. Conf. Parallel Processing*, pp. 591–598.

[6] J. T. Blake and K. S. Trivedi, "Comparing three interconnection networks embedded in a multiprocessor system," Tech Report, Duke Univ. Oct. 1987.

[7] L. Tien and C. R. Das, "Reliability evaluation of butterfly network based multiprocessor systems," submitted to *8th Int. Conf. on Distributed Comput. Systems*, June 1988.

[8] M. O. Ball, "Complexity of network reliability computation," *Networks*, vol. 10, pp. 153–165, 1980.

[9] A. D. Ingle and D. P. Siewiorek, "Reliability models for multiprocessor systems with and without periodic maintenance," in *Proc. 7th Annu. Int. Conf. FTC*, Los Angeles, CA, June 1977, pp. 3–9.

[10] C. -L. Wu and T. -Y. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-29, pp. 694–702, Aug. 1980.

[11] B. Thomas, "Overview of the Butterfly parallel processor," BBN Laboratories Incorporated, Aug 1985.

[12] V. Kumar, and S. M. Reddy, "Augmented shuffle-exchange multistage interconnection networks," *IEEE Comput.*, vol. 20, pp. 30–40, June 1987.

[13] N. -F. Tzeng, P. -C. Yew, and C. -Q. Zhu, "The performance of a fault-tolerant multistage interconnection network," in *Proc. 1985 Int. Conf. Parallel Processing*, pp. 458–465.

[14] L. Ciminiera and A. Serra, "A connecting network with fault tolerance capabilities," *IEEE Trans. Comput.*, vol. C-35, pp. 578–580, June 1986.

[15] S. M. Reddy and V. Kumar, "On fault-tolerant multistage interconnection networks," in *1984 Int. Conf. Parallel Processing*, pp. 637–648.

[16] R. J. McMillen, Jr., and H. J. Siegel, "Performance and fault tolerance improvements in the inverse augmented data manipulator network," in *9th Symp. Comp. Arch.*, Apr. 1982, pp. 63–72.

[17] J. P. Shen and J. P. Hayes, "Fault-tolerance of a class of connecting networks," in *7th Int. Symp. Comput. Architecture*, May 1980, pp. 61–71.

[18] J. P. Shen and J. P. Hayes, "Fault-tolerance of dynamic full-access interconnection networks," *IEEE Trans. Comput.*, vol. C-33, pp. 241–248, Mar. 1984.

[19] D. P. Agrawal, "Graph theoretical analysis and design of multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-32, pp. 637–648, July 1983.

[20] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, vol. 29, pp. 1184–1201, 1984.

[21] T. F. Arnold, "The concept of coverage and its effect on the reliability model of a repairable system," *IEEE Trans. Comput.*, vol. C-22, pp. 251–254, Mar. 1973.

[22] D.P. Siewiorek and R. Swarz, *The Theory and Practice of Reliable System Design*, Bedford, MA: Digital Press, 1982.

[23] J. J. Macaluso, "On the reliability evaluation of multistage interconnection network based multiprocessor systems," M.S. Thesis, The Pennsylvania State University, 1988.

ADAPTIVE CHECKPOINTING AND ROLLBACK IN MULTIPROCESSOR SYSTEMS

Chung-Yang Chiang and Chuan-lin Wu
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712

Abstract -- An adaptive checkpointing method and a companion rollback method which are mostly suitable for tightly-coupled multiprocessor systems, are proposed in this paper. An interprocess communication protocol is employed to synchronize checkpointing. Based on the checkpointing method, the companion rollback method restores the system to a recoverable state. Comparison of performance, in terms of checkpointing and recovery overheads, among our method, an existing unplanned method and an optimistic unplanned method which is mainly used as a performance index is provided to contrast these rollback recovery methods. Two comparison results are presented. The first result shows the performance level of the three methods at the same parameter values. The second result illustrates the optimum performance level of these methods. The performance evaluation reveals that our method is better than the existing unplanned method in most cases and sometimes better than the optimistic method. Performance breakpoints of these three methods are also depicted to investigate the constraints on individual methods.

1. Introduction

Rollback recovery methods [2]-[12] have been proposed to cope with reliability, availability and performance issues of computing systems. In a system with rollback recovery mechanism, a user or system program can be decomposed into recovery blocks. Recovery block [5] is a program structure consisting of checkpoint, primary process, alternate processes and acceptance tests for both primary and alternate processes. Rollback recovery methods can be categorized into two groups based on checkpointing policy. The first is the unplanned method which doesn't impose any constraints on processes regarding scheduling and communication among processes to establish checkpoints. The second method, termed planned or global method, on the contrary, does impose some constraints [6]. Unplanned recovery has the advantage of freeing processes for useful computation during checkpointing period, yet it suffers the domino effect [5] due to its lack of coordination among processes. Planned recovery observes opposite effects.

An adaptive method is proposed here to improve rollback recovery performance. Our method, a mixture of planned and unplanned methods, is mostly suitable for tightly-coupled systems with centralized control due to low time overhead on checkpointing and recovery in such systems. The forthcoming performance comparison result shows our method is usually better than the FTMR^{2}M method [2]-[3] and sometimes better than the optimistic method [3].

Section 2 describes the adaptive checkpointing and rollback methods along with their contrast to the two counterparts of the unplanned methods to be compared. Section 3 and 4 convey the analyses and comparisons on shared-memory and message-passing systems respectively. Last section envisions future research areas and concludes our observations.

2. Adaptive Rollback Recovery Method

The concept of rollback recovery is illustrated in Fig.1. Tp is the intercheckpointing interval. Trb is the time taken to establish a checkpoint. A set of checkpoints is consistent and thus constitutes a recoverable line if each process Pi after having established its checkpoint only communicates with other processes in the same subset that have also established their checkpoints [4].

2.1. Adaptive Checkpointing Method

The interprocess communication protocol we propose to enforce synchronization among processes is based on the following discussion. Fig. 2 illustrates a few cases regarding interprocess communication during checkpointing period to be resolved to avoid domino effect. 1j and 2j are the instants that process 1 and 2 respectively recognize the checkpointing signal for recovery block j. They establish checkpoints RP1(j) and RP2(j) at a later time. m(n,j,d) is the nth request message to the requested process d during recovery block j of the requesting process, and a(n,j,s) is the acknowledgement of the requested process to the nth request message from requesting process s in s's recovery block j. For all messages initiated before RP2(j) and after RP1(j), they must be rejected or delayed to avoid domino effect. For example, m(n+1,i,1) is issued by process 2 in its recovery block i and recognized by process 1 in its recovery block j, this request must be rejected by process 1 and reissued by process 2 after checkpoint RP2(j). m(n+1,i,1) in recovery block i thus becomes m(0,j,1) in recovery block j. For m(0,j,2), it will be received as a tentative message and processed after RP2(j). A tentative message in current recovery block will be committed as a permanent message in next recovery block. In a system where acknowledgement is supported the acknowledgement can be issued as a(0,i,1) in the old recovery block i or delayed as a(0,j,1) in the new recovery block j.

In contrast to our checkpointing method, the global checkpointing method disallows initiation and recognition of such messages as m(n+1,i,1) and m(0,j,2) whereas the unplanned method allows all messages to be initiated and recognized at any time.

Processes which have already established the checkpoint and involved in interprocess communication with processes yet to establish checkpoint form a global group, and those processes which have established the checkpoint and/or are not involved in interprocess communication form an unplanned group.

Synchronization among processes can be implemented implicitly by interprocess communication protocol. No additional phases, such as the phases in the two-phase commit protocol [6], are needed to synchronize the processes. Messages can be sent with a sequence number specifying in which recovery block they are generated. In shared-memory system, the sequence number can be implemented by "checkpoint bits" in the address bus.

2.2. Adaptive Rollback Method

Our rollback method is based on the criteria founded by the checkpointing method. Rollback might seem deceptively simple if cares are not taken. Lack of synchronization regarding rollback results in a situation similar to the domino effect caused by lack of synchronization regarding checkpointing. 'Livelock' described in [12] is one example.

We will analyze cases in Fig. 3 in which rollback is performed between two processes. Extension to more than two processes can be achieved through rollback propagation and is not addressed here.

Case 1. P2 fails at t1 or P2 fails at t4 --- P2 rolls back to RP2(i-1) and P1 rolls back to either RP1(i-1) or RP1(i) depending on if it has communicated with P2 in RB2(i-1), even though P1 is already in RB1(i). It is the logical recovery block RB2(i-1) that we just referred to since it is the same as logical RB1(i-1). The physical boundaries of RB1(i-1) and RB2(i-1) are not the same.

Case 2. P2 fails at t5 or P2 fails while establishing RP2(i) --- The message m(0,i,2) is recorded in P1's message log and received by P2 in RB2(i-1). Yet m(0,i,2) will not be committed until P2 has established RP2(i). P2 will then roll back to RP2(i-1) and P1 to either RP1(i) or RP1(i-1).

Case 3. P2 fails at t7 --- P2 rolls back to RP2(i) and P1 rolls back to RP1(i) since message m(0,i,2) is treated as the message after RP2(i) and recorded in both message logs in RB1(i) and RB2(i).

Case 4. P1 fails at t2 or P1 fails at t3 --- P1 rolls back to RP1(i) and P2 goes on as usual.

Case 5. P1 fails at t6 --- P1 rolls back to RP1(i) and P2 goes on as usual.

Case 6. P1 fails while establishing RP1(i) --- P1 rolls back to RP1(i-1) and P2 rolls back to RP1(i-1) if it has communicated with P2 in RB1(i-1). Otherwise P2 goes on as usual.

2.3. Unplanned Recovery Methods

Two unplanned recovery methods will be briefed to contrast the differences among our method and these methods to be compared. We first brief the optimistic method. This method rolls back only the necessary number of steps as determined by interprocess communication pattern. It is concluded in [3] that only a few checkpoints are needed depending on various system parameters. We present this method only to demonstrate its performance as an index in the performance comparison of the other two methods. This method isn't necessarily a better performer in all cases than the adaptive method. Obviously, it betters the FTMR^{2}M method in all cases.

FTMR^{2}M method is another unplanned recovery method. It heavily relies on the assumption that probability of single-step rollback overwhelmingly dominates probabilities of multiple-step rollbacks. The system thus records only two checkpoints to rollback one step if single-step rollback is determined. Otherwise, the system simply rolls back to the origin of the task as if the failure is fatal.

3. Shared Memory System

Based on the following assumptions, a mean value analysis will be given to compare the performances of these rollback recovery methods.

(1) mean time to rollback to last checkpoint is Tr.
(2) mean checkpointing time is Trb.
(3) interprocess communications are uniformly distributed.
(4) independent exponential failure distribution is assumed for all processing modules.
(5) probabilities of fatal and nonfatal failures are constants P_f and P_{nf} respectively, and $P_f + P_{nf} = 1$. They are independent of the underlying Poisson failure distribution
(6) probability of i-step rollback is $P_{rb}(i)$, i = 1,2,...,M-1, and $\sum P_{rb}(i) =1$ for all i's. They are also independent of the underlying Poisson failure distribution.

3.1. Derivation of Mean Execution Time

(A) Optimistic Unplanned Method --- The mean task execution time is :

$$T_A=Tp*M+\mu_A*\{P_{nf}*[Tr*P_{rb}(1)+\sum_{i=2}^{M-1}[Tr+Tp*(([(i-1)*(i-2)]/2 +(M-i+1)*(i-1))/M)*P_{rb}(i)]] + P_f*[Tr + Tp*(M-1)/2]\}$$

$$=Tef+\mu_A*\{Tr + P_{nf} *Tp* \sum_{i=2}^{M-1} P_{rb}(i)*[([(i-1)*(i-2)]/2+ (M-i+1)*(i-1))/M] + P_f*(Tef-Tp)/2\} \qquad (1)$$

μ_A is the mean number of failures during actual task execution time. Tef is Tp*M, the failure-free execution time including checkpointing overhead.

(B) FTMR^{2}M Method --- The mean task execution time is :

$$T_B = Tef+ \mu_B*\{Tr+ [(Tef-Tp)/2]*[1 - (P_{rb}(1)*P_{nf})]\} \qquad (2)$$

(C) Adaptive Method --- The mean task execution time is :

$$T_C = Tp*M+Trb*(Trb/Tp)*(T_C/Tp)+\mu_C*\{P_{nf}*[Tr* \sum_{i=1}^{M-1} Prb(i)]+P_f*[Tr+Tp*(M-1)/2]\}$$

$$=\{Tef + \mu_C*[Tr + P_f*(Tef-Tp)/2]\}/\{1- (Trb/Tp)**^2\} \qquad (3)$$

3.2. Comparison of Mean Execution Time

For the simplicity of comparison, we assume the mean number of failures during task execution time is μ' and remains the same for all three models. μ' is μ_A, μ_B and μ_C for optimistic, FTMR^{2}M and adaptive models respectively.

3.2.1. Comparison of FMTRRM and Adaptive Methods. Two performance comparisons will be studied, optimum and nonoptimum comparisons. The speedup of the FTMR^{2}M method over the adaptive method is derived from Eqns. (2) and (3) :

$$dT=T_B-T_C=[Tef(B)-Tef(C)*F]+\mu'*[(Tef(B)-Tp(B))*(1- Prb(1)*P_{nf})/2-(Tef(B)-Tp(B))*P_f*F/2]+\mu'* [Tr(B)-Tr(C)*F] \qquad (4)$$

where F is 1/(1-(Trb/Tp)**2).

3.2.1.1 Nonoptimum Performance Comparison. We assume all parameters in Eqns. (2) and (3) are the same for both methods. The speedup can be approximated as :

$$dT=Tef*[\mu*((1-1/M)/2)*(1-P_{rb}(1)) - ((Trb/Tp)**2)] \qquad (5)$$

$\mu = \mu'*P_{nf}$, i.e., number of nonfatal failures.

Assume μ is 1 for simplicity. For the FTMR^{2}M method to better the adaptive method, $P_{rb}(1)$ must be at least 0.98 if checkpointing overhead (Trb/Tp) is 0.1. If the overhead is only 0.01, $P_{rb}(1)$ will be at least 0.9999. Fig. 4.b depicts the effect of μ and Trb/Tp on min$\{P_{rb}(1)\}$. Fig. 4.a illustrates the difference in task execution time between these two methods.

3.2.1.2 Optimum Performance Comparison. Some parameters aforementioned have to be adjusted so that optimum can be realized. This optimization has been studied in [9]-[11]. Based on this concern, Tp, Tr and M vary with recovery method since we assume Trb is the same for both systems. We use triplet {Tp(i), Tr(i), M(i)} to represent the triple elements {Tp, Tr, M} for different methods.

To optimize system performance, the following equates must hold :

$$Tp(C) - Trb =[(2Trb * MTBF)^{1/2}]/P_{nf} \qquad (6)$$

$$Tp(B) - Trb =[(2Trb * MTBF)^{1/2}]/[P_{nf}*P_{rb}(1)] \qquad (7)$$

where MTBF is the mean time between failures. The above equations are derived in the same way as in [9]. In FTMR^{2}M method, every failure requiring more than one rollback will retstart the system from the origin of the task. Thus, they are essentially the same as fatal failures. P_{nf} in Eqn. (6) should then be replaced by $P_{nf}*P_{rb}(1)$ to acquire Eqn. (7).

Since (Tp-Trb)*M remains constant for both methods, we have $M(B) \leq M(C)$. Tr's are :

$$Tr(B) = MTBF - [Tp(B)*exp(-Tp(B)/MTBF)]/[1-exp(-Tp(B)/MTBF)] \qquad (8a)$$

$$Tr(C) = MTBF - [Tp(C)*exp(-Tp(C)/MTBF)]/[1-exp(-Tp(C)/MTBF)] \qquad (8b)$$

It is apparent that $Tr(B) \geq Tr(C)$.

Since Tp(B) varies as $P_{rb}(1)$ changes, we acquire the following :

$$dT_B=dTef+\mu_B*\{dTr+[(dTef-dTp)/2]*[1-(P_{rb}(1)*P_{nf})]+[(Tef-Tp)/2]*P_{nf}*[-dP_{rb}(1)]\} \qquad (9)$$

$dTef < 0$ and $dTp > 0$ improve, whereas $dTr > 0$ and $[-dP_{rb}(1)] > 0$ degrade performance of the FTMR^{2}M method. The last item in the above equation dominates others, making the optimized performance of the FTMR^{2}M method even worse than that of the adaptive method. As $P_{rb}(1)$ decreases, the probability of restarts is a lot higher even though Tef is slightly shorter. That is why the performance of the FTMR^{2}M method is worse.

3.2.2. Comparison of Optimistic Unplanned and Adaptive Methods. The execution time speedup is $T_C - T_A$, i.e., dT:

$$dT=Tef*[\mu* \sum_{i=2}^{M-1} (([(i-1)*(i-2)]/2+(M-i+1)*(i-1))/M)*P_{rb}(i) - (1+P_f*(Tef-Tp)/2)*((Trb/Tp)**2)] \qquad (10)$$

We assume the distribution of Prb(i) is geometric. In the geometric distribution, $P_{rb}(1)$ dominates other $P_{rb}(i)$'s and $Prb(i) \geq Prb(j)$ if $i \leq j$, which makes the optimistic unplanned method a superb performer. The total rollback overhead during task execution time is depicted in Fig. 5 for the three recovery models.

4. Message Passing System

The difference among the three rollback recovery methods, in terms of rollback recovery overhead, between shared-memory and message-passing systems is none when there is no error. The difference surfaces when rollback recovery is needed. Due to this difference, the adaptive method is even better than the unplanned methods.

The two unplanned methods require logging of interprocess communications, which in nature is the same as checkpointing synchronization in the adaptive method. Logging of interprocess communications requires that, upon checkpointing signal, outstanding interprocess communications be finished before other phases of checkpointing can proceed. Otherwise it can't guarantee the correctness of received messages. For instance, if a process saves its internal states and executes the validation test while some interprocess communications are still outstanding, even if this process passes its validation test, that implies only that part of the interprocess communication occurred before the end of the validation test is valid. There is no guarantee on the later part of the interprocess communication. Hence, from the point that checkpointing is first recognized by one of the processes to the point that the checkpointing ends, all three methods behave identically. Only after this checkpointing period can we see the difference incurred by different recovery methods.

4.1. Derivation of Mean Execution Time

The equations remain much the same as those of shared-memory system except some minor modifications incurred by transmitting interprocess communication logs to all processes.

(A) Optimistic Unplanned Method ---

$$T_A = Tef+\mu_A*\{Tr + P_{nf}*Tcl+ P_{nf} *Tp* \sum_{i=2}^{M-1} P_{rb}(i) *[((i-1)*(i-2)+(M-i+1)*(i-1))/M] + P_f*(Tef-Tp)/2\} \qquad (11)$$

Tcl is the time taken to form the global interprocess communication log from partial local logs recorded by each process.

(B) FTMR^{2}M Method ---

$$T_B = Tef + \mu_B*\{Tr+ P_{nf}*Tcl + [(Tef-Tp)/2]*[1 - P_{rb}(1)*P_{nf}]\} \qquad (12)$$

(C) Adaptive Method ---

$$T_C=\{Tef+\mu_C*[Tr+P_f*(Tef-Tp)/2]\}/\{1- (Trb/Tp)**^2\} \qquad (13)$$

This is exactly the same as that of the shared-memory system since constructing global communication log is not required.

5. Conclusion

An adaptive rollback recovery system is proposed and compared to two other methods. Two essential components of this system, adaptive checkpointing and rollback methods are introduced and analyzed. Performances of these three rollback recovery methods have been analyzed and compared in terms of task execution time for both shared-memory and message-passing systems. The adaptive recovery method outperforms the other two methods whenever single-step rollback probability is low, checkpointing overhead is low and number of failures is high. Besides the comparison on the task execution time, we should also consider the fact that the adaptive recovery method indeed needs less hardware which implies less number of failures during task execution. We thus conclude that the above comparison is pessimistic, the performance advantage of the adaptive method is actually better than what is revealed in the above comparison. The adaptive rollback recovery method performs even better in a message-passing system.

This adaptive method is mostly suitable for both shared-memory and message-passing systems with a centralized control mechanism. For sparsely distributed systems, the unplanned recovery method, such as the one in [7], is prone to domino effect, it has yet to be assessed which of the three methods is more efficient. Further research is necessary to compare these three different recovery methods on this type of system.

6. References

[1] R.J. Swan, S.H. Fuller, D.P. Siewiorek, "Cm* : a Modular Multi-Microprocessor", AFIPS Conf. Proc., (1977), pp. 637-644.

[2] Y. H. Lee, K. G. Shin, "Rollback Propagation Detection and Performance Evaluation of FTMR^2 -- A Fault-Tolerant Multiprocessor", Symp. on Computer Architecture, (1982), pp. 171-180.

[3] ---, "Design and Evaluation of a Fault-Tolerant Multiprocessor Using Hardware Recovery Blocks", IEEE Trans. on Computers, (Feb. 1984), pp. 113-124.

[4] T. Anderson, P.A. Lee, Fault Tolerance - Principles and Practices, MacGraw-Hill, (1981), 369 pp.

[5] B. Randell, "System Structure for Software Fault Tolerance", IEEE Trans. on Software Engineering, (June 1975), pp. 220-232.

[6] J.N. Gray, "Notes on Database Operating Systems", pp. 393-481 in "Lectures Notes in Computer Science 60", ed. R. Bayer, R.M. Graham and G. Seegmuller, Springer-Verlag, Berlin, (1978).

[7] P.M. Merlin, B. Randell, "State Restoration in Distributed Systems", Conference on Fault Tolerant Computing, (1978), pp. 129-134.

[8] D. L. Russell, "State Restoration in Systems of Communicating Processes", IEEE Trans. on Software Engineering, (Mar. 1980), pp. 183-194.

[9] J.W. Young, "A First Order Approximation to the Optimum Checkpoint Interval", Communications of the A.C.M., (Sep. 1974), pp. 530-531.

[10] E. Gelenbe, D. Derochette, "Performance of Rollback Recovery Systems under Intermittent Failures", Communications of the A.C.M., (June 1978), pp. 493-499.

[11] E. Gelenbe, "On the Optimum Checkpoint Interval", Communications of the A.C.M., (Apr. 1979), pp. 259-270.

[12] R. Koo, S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", IEEE Trans. on Software Engineering, (Jan. 1987), pp. 23-31.

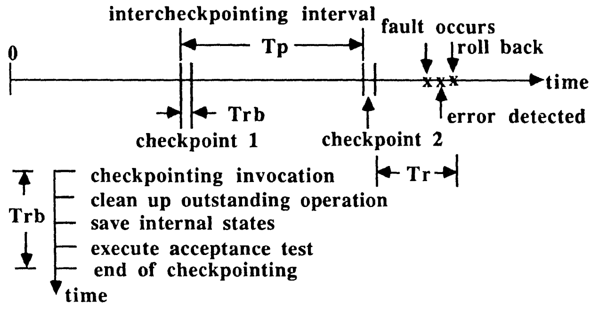


Fig. 1 Concept of checkpointing and rollback recovery

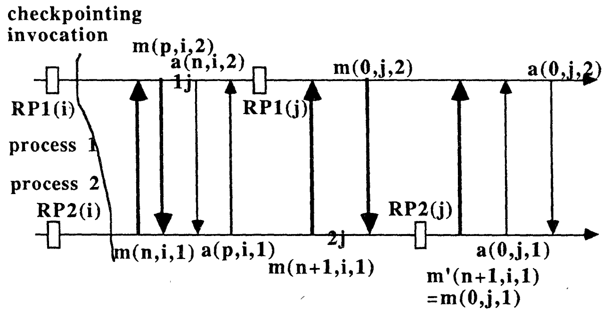


Fig. 2 Cases of interprocess communication which should be resolved to avoid domino effect

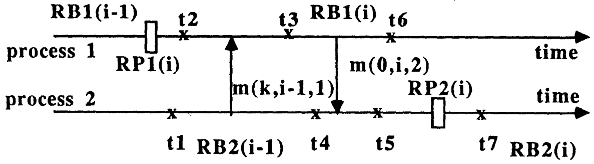


Fig. 3 Cases to consider regarding rollback involving two processes

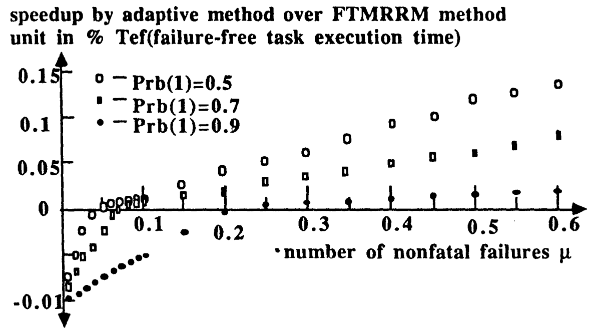


Fig. 4.a Speedup of task execution time, assuming Trb/Tp is 10%, as a function of μ and Prb(1)

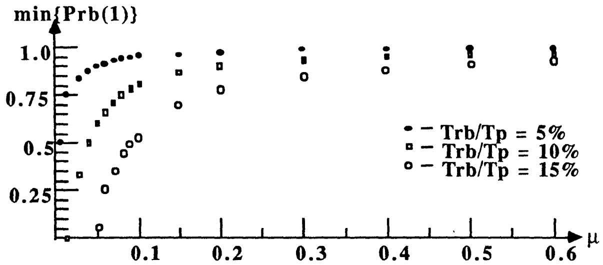


Fig. 4.b min{Prb(1)}, minimum probability of single-step rollback, for FTMRRM method to outperform adaptive method-as a function of checkpointing overhead and μ

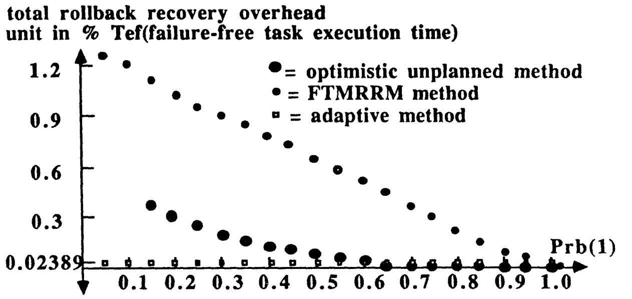


Fig. 5.a Comparison of rollback recovery overhead assuming 1 % checkpointing overhead, 0.02778 system failures, no fatal failure and geometric distribution of Prb(i)'s

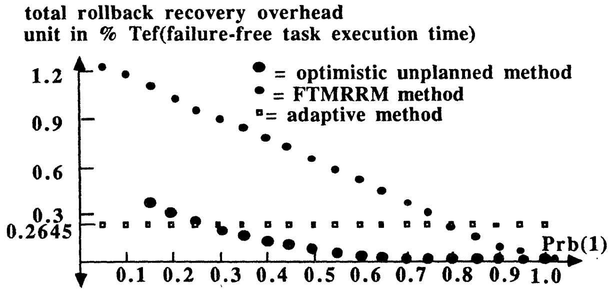


Fig. 5.b Comparison of rollback recovery overhead assuming 5 % checkpointing overhead, 0.02778 system failures, Pf=0.0005 and geometric distribution of Prb(i)'s

MEASUREMENT-BASED ANALYSIS OF MULTIPLE LATENT ERRORS AND NEAR-COINCIDENT FAULT DISCOVERY IN A SHARED MEMORY MULTIPROCESSOR

S.G. Mitra* and R.K. Iyer

Coordinated Science Laboratory and
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1101 W. Springfield Ave.
Urbana, Illinois 61801

ABSTRACT

This paper presents a methodology to study multiple latent errors and near-coincident fault discovery in the memory of a shared memory multiprocessor. The delay between the generation of an error due to a fault and its detection (error latency) can cause multiple latent errors and near-coincident fault discovery in a system. The latter effect is widely known to be catastrophic to the continued operation of a system even in highly fault tolerant systems. The methodology is illustrated on the Alliant FX/8 under real concurrent workload conditions over a five-day period. This study finds that for a conservative error rate of one error per day, one out of four errors may manifest itself as a multiple latent error. At the same error rate, 8% of the error discoveries are near-coincident in nature for a time-window size of 50 microseconds (approximately 250 instruction cycles). A strong correlation between existences of multiple latent errors and their near-coincident discovery is quantified.

1. INTRODUCTION

A prerequisite for designing high reliability systems is to understand the effect of faults and their manifestations. Behavior of faults in a computer system is not easy to comprehend. This is even more so in a multiprocessing environment, where the mannner in which faults manifest themselves is usually complex. Analytical models suffer from constraining assumptions and developmental complexity. An alternative are measurements and experiments on production multiprocessor systems. These aid the model building process and provide valuable insight for designing new systems.

This paper studies the fault discovery process in a shared memory multiprocessor system. There is usually a delay between the generation of an error (caused by a fault) and its discovery by a detection mechanism. This time is commonly referred to as error latency. Long error latencies can potentially lead to accumulation of undiscovered errors (called latent or "lurking" errors) in the system. We define multiple latent errors as a condition within a system where two or more errors are yet undiscovered by the system. Latent errors can be major threats to the reliability of the system. This is because there exists a possibility that they can be discovered simultaneously, behaving as though multiple faults have occurred. Most recovery mechanisms however are not designed to handle multiple faults. There is also a possibility for multiple latent errors to be discovered close in time, thus stressing the error recovery mechanism. Such situations are referred to as near-coincident fault discovery and are known to be catastrophic in real systems [1,2].

The purpose of this experimental study is to quantify the characteristics of multiple latent errors and near-coincident fault discovery in a shared memory multiprocessor system under a real concurrent workload[1]. A multiprocessing system presents a new dimension from the workload point of view, since a number of processes can be active at the same time. This casts a new perspective on the study of latent error behavior since the probability of error discovery is potentially higher.

The experiment employs actual hardware measurements from an Alliant FX/8 system to simulate error occurrence in the system and to investigate multiple latent error occurrence and near-coincident fault discoveries. The Alliant FX/8 is a key component in the "Cedar" parallel supercomputer project at the Center for Supercomputing Research and Development in the University of Illinois at Urbana-Champaign [3]. The measured Alliant FX/8 runs the current version of "Xylem," the Cedar operating system. Specifically, the methodology is applied to the Alliant memory subsystem. The fault model used in this study assumes that a permanent error has occurred[2]. The physical mechanism causing the faults can be varied and do not affect the results.

The results are unique in that they provide new insight into the behavior of multiple latent errors and near-coincident fault discovery in a complex parallel processing environment. At a conservative error occurrence rate of approximately one error per day, there is a 25% chance that errors cause multiple latent errors in the system. Thus one out of four errors may manifest itself as a multiple latent error. Further it was found that 8% of the error discoveries are near-coincident in nature with a time window size of 50 microseconds. It was also found that the probability of multiple latent errors tends to saturate after a threshold error occurrence rate of approximately one error per day. The probability of near-coincident fault discovery was found to saturate also, but at a slower rate.

1.1 Related Research

There is little or no research cited in the literature which experimentally investigates the occurrence of multiple latent errors or near-coincident fault discovery. Fault injection studies in the FTMP (Fault Tolerant Multiprocessor) showed that the most likely threat to system failure in the short run was arrival of two failures so close to each other that system reconfiguration was not possible [1,2]. These experiments used pin-level fault injection while running specific programs. An analytical model for near-coincident faults in NMR systems with different voting schemes is presented in [4]. The general validity of such a model however is not established.

Other related research consists of experiments conducted to measure fault/error latency. Experiments to measure fault latency via pin-level fault injections in FTMP are discussed in [5]. In this study, the researchers measured latency times for faults in

This work was supported by NASA under Grant No. NAG-1-613.

* Author is presently with SUN Microsystems Inc., Mt View, CA

[1] When two or more processors are active the system is said to be in concurrent operation.

[2] An error is that part of the system state which is liable to lead to failure. The cause - in its phenomenological sense - of an error is a fault.

different system components and obtained a standard distribution fit to their measured fault latency distribution. CPU fault latency for the digital microprocessor in FTMP was studied in [6,7] via gate-level simulation. A set of specific programs was used to exercise the CPU to reveal faults injected into the simulation.

The above approaches and results are, however, not applicable in general to multiuser systems. More recently, latent fault behavior in the memory of a VAX 11/780 was studied in [8]. The memory system was instrumented for measurements, and fault/error latencies were calculated by simulated fault injection in the memory. Also the effect of workload on fault/error latencies was investigated in [10].

Although the above studies investigate the subject of latency quite systematically, the question of multiple latent errors or near-coincident fault discovery is not addressed. As mentioned earlier past measurements indicate these problems as usually catastrophic to the system.

The next section describes the experimental methodology used to calculate multiple latent errors and near-coincident fault discovery probabilities. Section 3 presents the results and discusses the multiple latent error and near-coincident fault discovery behavior seen. Section 4 summarizes the important results of the paper.

2. EXPERIMENTAL METHODOLOGY

Figure 1 shows the Alliant FX/8 components related to our study. Detailed information on the Alliant system is given in [9]. The system runs the current version of "Xylem," the Cedar operating system. Thus, from the software point of view, many features of the Cedar supercomputer are running on the Alliant FX/8. The workload on the Alliant FX/8 consisted mostly of scientific applications such as circuit simulation, weather modeling, digital animation and fluid dynamics.

This study concentrates on the error characteristics within the main memory of a system. An important reason for this is, measured field results show the largest number of failures occur in the memory [10]. A large number of CPU errors can also be traced to the memory [11]. Further, since shared memory is a common resource, the possibility of it being the source of failures can be significant.

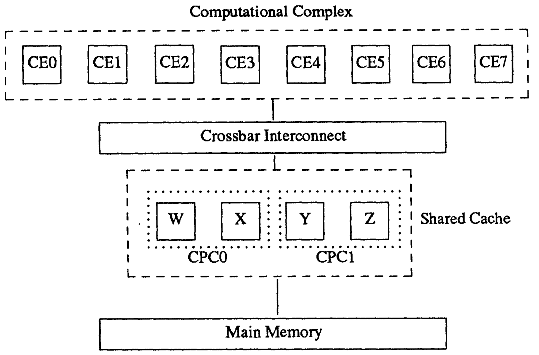


Figure 1. Configuration of Measured Alliant FX/8

2.1 Hardware Measurements

The Alliant FX/8 backplane was sampled to collect data on memory access operations from the shared cache. A Tektronix DAS 9200 with a 32K trace buffer was used for this purpose [12]. Hardware probes were attached primarily to the main memory address bus on the Alliant backplane. Other probes were used to monitor signals so that appropriate triggering could be performed.

As mentioned in the Introduction, the measurements were performed while the system was executing concurrent workload. The measurements were conducted over a five-day period, 8am to 5:30pm daily, Monday to Thursday and 8am to 3:45pm on Friday (primarily due to drop in concurrent operations). Samples were taken approximately every 4 minutes[3], with each sample containing 8K address references (representing 8K machine cycles). The total measurement period was approximately 46 hours.

Table 1 shows the filtered version of the raw data output. The addresses represent memory block start addresses. The memory is accessed in blocks of 32 bytes (transfer size between the shared cache and memory). The fields cntl0 and cntl1 provide additional status information about the state of the memory bus.

Table 1. Concurrent Workload Memory Address Trace

Line no.	time stamp	address	cntl0	cntl1
1	00033316579	0D3FF8	F	0
2	0003331666B	000230	F	4
3	000333166BC	000232	F	4
4	0003331670D	000234	F	4
5	000333167BB	0D3FF7	F	0
6	000333167CC	1AE0F7	F	8
7	00033316869	0C2FF5	F	0

2.2 Simulation

The memory address trace obtained above was then used as input to a simulation system, which essentially reconstructed the address space into which simulated error injections were performed over the entire measurement period (the simulator was driven by the address trace). An error was discovered when the time of error injection at an address location was less than or equal to the time of arrival of that address in the concurrent workload address trace. The simulation environment consisted of three simulators, ELS (Error Latency Simulator), MLEI (Multiple Latent Error Identifier) and NCFI (Near-Coincident Fault discovery Identifier). Detailed information on the simulation environment is given in [13].

For error injection purposes no distinction was made between specific locations within a block. Since the transfers from main memory occur in blocks of 32 bytes, an error in one location within the block is equivalent to an error in any other location in the block from a discovery point of view. This simplified the simulation somewhat and more importantly, smoothed out discontinuities arising out of the fact that the data were sampled.

Simulated error injections were performed assuming an exponential distribution for error occurrence over the entire measurement period. The error injection rates (λ) were varied from 0.009 to 0.058 (times 6 error injections per hour or "x6 ei/hr"). Address locations for the error injection were chosen randomly. The exponentially distributed intervals between error injections were also chosen randomly.

In order to obtain statistically consistent results, approximately 600 faults were injected at each error injection

[3] The sampling rate chosen reflects a compromise between an adequate sample size and delay in transferring data to a data logger.

time. This is equivalent to the simulation being run 600 times for each error injection rate. In each run, a randomly chosen location is injected with an error.

2.3 Measurement of Multiple Latent Errors

Multiple latent errors occur when two or more errors are yet undiscovered in the system. In order to determine the probability of multiple latent errors at a given error injection rate, we first construct a latency profile for each injection. The latency profile for an injection is the profile of discovery times for all errors injected at that injection time. Once the time to discovery for each error injected is available, a latency profile can be plotted as in Figure 2.

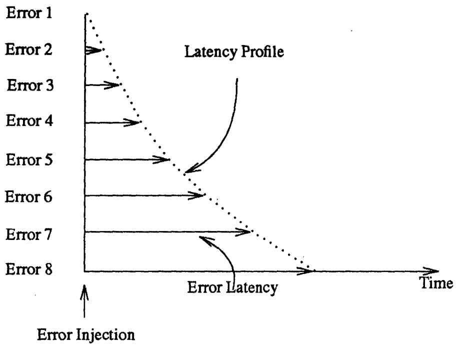


Figure 2. Example of a Latency Profile

Consider for simplicity a case in which two error injections are made in a measurement period. Figure 3 shows the three possible overlap scenarios for the latency profile. The multiple latent error regions between the two error injections is shown. The multiple latent error region area versus the total latency profile area of both injections give a rough view of the probability of multiple latent errors in the system. The probability of multiple latent errors is defined as the ratio of number of errors in the multiple latent error region to the total number of injected errors.

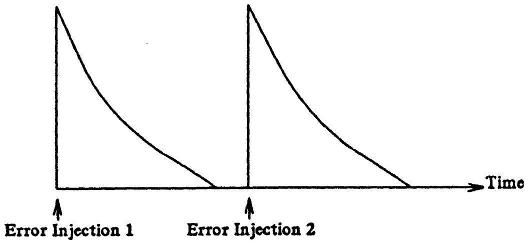


Figure 3(a). No Overlap

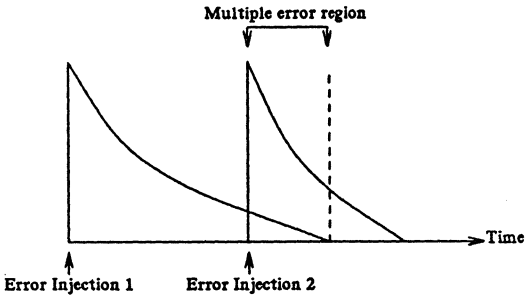


Figure 3(b). Type 1 Overlap

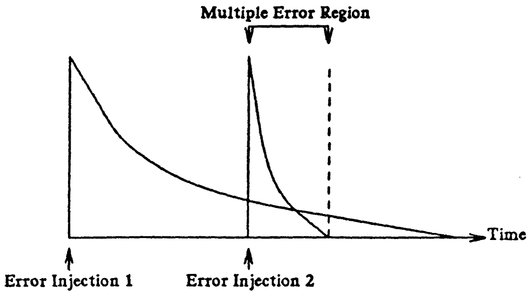


Figure 3(c). Type 2 Overlap

Let E_i represent the number of errors injected at error injection number i. Also let $Me_{ij,\lambda}$ represent the number of errors of error injection i that exist as multiple latent errors with error injection j at the error occurrence rate λ (e.g., $Me_{12,\lambda}$ represents number of errors in injection 1 that exist as multiple latent errors with injection 2 at the error occurrence rate λ and $Me_{21,\lambda}$ is the number of errors in injection 2 that exist as multiple latent errors with injection 1 at the error occurrence rate λ). Then between two error injections n and m where $n<m$,the probability of multiple latent errors $Mp_{nm,\lambda}$ for an error occurrence rate of λ is

$$Mp_{nm,\lambda}=\frac{Me_{nm,\lambda}+Me_{mn,\lambda}}{E_n+E_m}$$

In the complex case of more than two error injections within the measurement period, the multiple latent error probabilities can be individually calculated with respect to one particular error injection for the given error occurrence rate λ (i.e., $Mp_{12,\lambda}$ is a multiple latent error probability between fault injections 1 and 2, $Mp_{13,\lambda}$ is multiple latent error probability between 1 and 3 etc.). For each $Mp_{nm,\lambda}$, multiple latent errors may exist in either of the two forms shown in Figures 3(b) and 3(c). But given the definition of multiple latent errors ,where **at least** two errors must exist undiscovered in the system, only adjacent error injection probabilities need be considered. Thus only $Mp_{12,\lambda}$, $Mp_{23,\lambda}$, $Mp_{34,\lambda}$ etc. values are used to give an overall multiple latent error probability (Mp_{λ}) for the error occurrence rate chosen. Thus, if n_{ei} represents the number of error injections achieved at the error occurrence rate λ, the overall probability of multiple latent errors at error occurrence rate λ is

$$Mp_{\lambda}=\frac{\sum_{i=1}^{i=n_{ei}-1} Mp_{(i)(i+1),\lambda}}{n_{ei}-1}$$

2.4 Measurement of Near-Coincident Fault Discovery

In order to measure the probability of near-coincident fault discoveries, we first choose an appropriate time window of size T. Next we move this window over the total measurement time in increments equal to T, each time observing the number of errors discovered within the time window. The ratio of total number of errors found in that time window to the total number of errors injected gives the probability of near-coincident faults in the system. Note that, if errors from the same error injection are discovered within the time window, they do not qualify as a near-coincident fault discovery.

The total measurement period is divided into n time slices t_1 ... t_n, each T long except one (if true integer division is not possible). The number of errors discovered **(from different error injections)** in each t_k were $N_{k,\lambda}$ at an error occurrence rate λ where $1 \le k \le n$. Again n_{ei} represents the number of error injections achieved at error occurrence rate λ. If the total number of errors injected into the system is E, then, the probability of near-coincident faults (NC_λ) for an error occurrence rate of λ is

$$NC_\lambda = \frac{\sum_{i=1}^{i=n} N_{i,\lambda}}{E}$$

$$\text{where } E = \sum_{i=1}^{i=n_{ei}} E_i$$

3. RESULTS

This section presents the experimental results on multiple latent errors and near-coincident fault discoveries for the Alliant memory subsystem. Recall that errors were injected at exponentially distributed intervals (with an error injection rate λ). The memory address trace was then used for determining multiple latent errors and near-coincident fault discovery probabilities. For purposes of this study, errors were injected in the high usage regions of the memory. The region of injection represented 93% of the address references in the real concurrent workload trace but occupied only an eighth of the memory address space available. Clearly, the behavior of faults in this region is critical for continued system operation.

On the average, 14% of all errors injected remained undetected during the measurement period (approximately 5 days). The choice of the error injection rate λ for the experiment was chosen to reflect realistic error occurrence rates (see [10]). The range was chosen to be $0.009 \le \lambda \le 0.058$ (x6 error occurrences per hour - approximately 2 to 16 error injections over 5 days). The time-window sizes chosen for analysis in the near-coincident fault discovery calculations represent reasonable error recovery times for a high performance system. The time-window range was varied from 1 microsecond to 250 microseconds (approximately 6 to 1500 instructions on the Alliant FX/8).

3.1 Multiple Latent Errors

Figure 4 shows the variation in the probability of multiple latent errors $Mp_{(i)(i+1),0.043}$ during the measurement period for an error occurrence rate of approximately two errors per day (0.043 x6 ei/hr). Figure 5 shows the probability of multiple latent errors being present in the system at different error occurrence rates. We find that the probability of multiple latent errors increases from a low of 0.04 at an error occurrence rate of approximately one error every two days to a high of 0.50 which is more or less a saturation probability. The oscillatory behavior of the graph is primarily due to statistical variations.

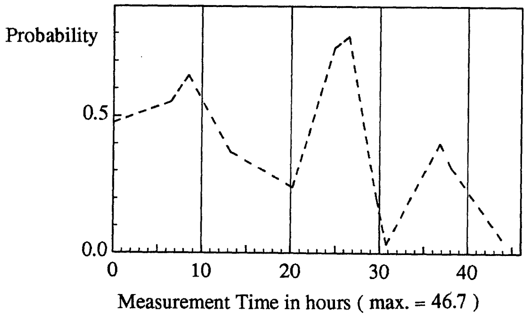


Figure 4. An Example of Multiple Latent Error Presence

Figure 5 shows that, at a conservative error occurrence rate of approximately one error per day (λ=.022), there exists a 25% chance ($Mp_\lambda \ge 0.25$) of multiple latent errors. This suggests that one out of four errors has the potential of manifesting itself as a multiple fault. On further examining of the plot in Figure 5 , we find at low error injection rates the plot has a higher slope than at high error injection rates. As expected the error occurrence rate (or the number of error injections) does have an impact on the multiple latent error probability, but this effect subsides as the error rate increases. The reason for this is that at higher error occurrence rates seemingly more latent errors tend to be discovered or "swept away", thereby resulting in a tapering effect on the plot.

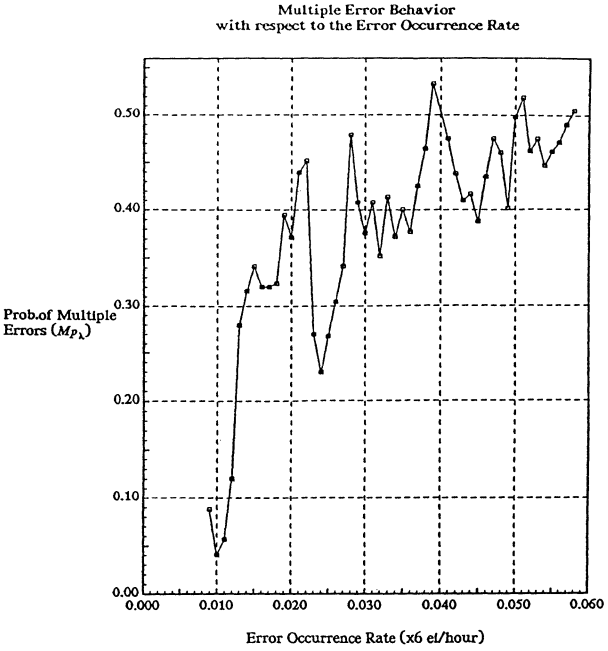


Figure 5. Probability of Mulitple Latent Errors

To show in detail how the multiple latent error probability changes during the course of error injections, a plot of variation in probability of multiple latent errors for an error occurrence rate of 0.031 (x6 ei/hour) is shown in Figure 6. There were nine error injections in the measurement period for this error occurrence rate. Each dotted line represents a multiple latent error probability plot with respect to a specific error injection number. L_1 represents multiple latent error probabilities for error injection 1 (E_1) with error injections E_2, E_3, E_4 and E_5. The $Mp_{12,0.031}$, $Mp_{13,0.031}$, $Mp_{14,0.031}$ and $Mp_{15,0.031}$ values are represented on this line. Similarly L_2 represents multiple latent error probabilities of error injection two ($Mp_{23,0.031}$, $Mp_{24,0.031}$ and $Mp_{25,0.031}$) and so on. A downward behavior is seen for all the lines. This seems intuitive; say for L_1, the errors of E_1 will tend to be discovered as time progresses, thereby reducing the probability of multiple latent errors being present in the system when E_5 is introduced.

To highlight one of the error discovery patterns in the system, the high multiple latent error probability of 0.9 for L_2 will be explained. We find that most of the errors injected in error injection 2 are discovered during the interval between error injections 3 and 4. This is because $Mp_{23,0.031}$=0.9 implies that both

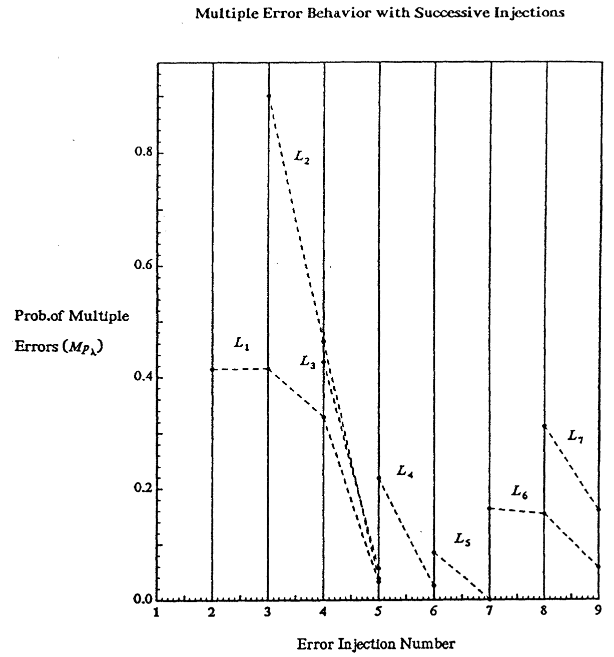


Figure 6. Probability of Multiple Latent Errors at Diff. Injections

$Me_{23,0.031}$ and $Me_{32,0.031}$ have high values. The multiple latent errors probability plot of error injection 2 continues till error injection 5. This shows that all errors discovered during the measurement time period are discovered before error injection 6. Thus the remaining errors injected at error injection 2 are discovered between error injections 2 and 3, error injections 4 and 5, and error injections 5 and 6.

3.2 Near-Coincident Fault Discovery

Figure 7 shows the variation of probability of near-coincident fault discovery with time-window sizes from 10 to 250 microseconds for three different error rates. As expected, the near-coincident fault probability increases monotonically with time window size. But however, the rate of increase in probability of near-coincident faults slowly decreases for larger time-window sizes. Figure 8 shows a microscopic view (1 to 10 microseconds) of the behavioral change in the near-coincident fault probabilities. The step function behavior is easily understood by the fact that if we have near-coincident faults in time-window size T, then those same near-coincident faults must exist in time-window size T+1.

The variation of probability of near-coincident faults ,for three time-window sizes (10us, 100us and 200us), is shown in Figure 9. The range of error rates used is 0.009≤λ≤0.049 (x6 error injections per hour), approximately 2 to 14 error injections over

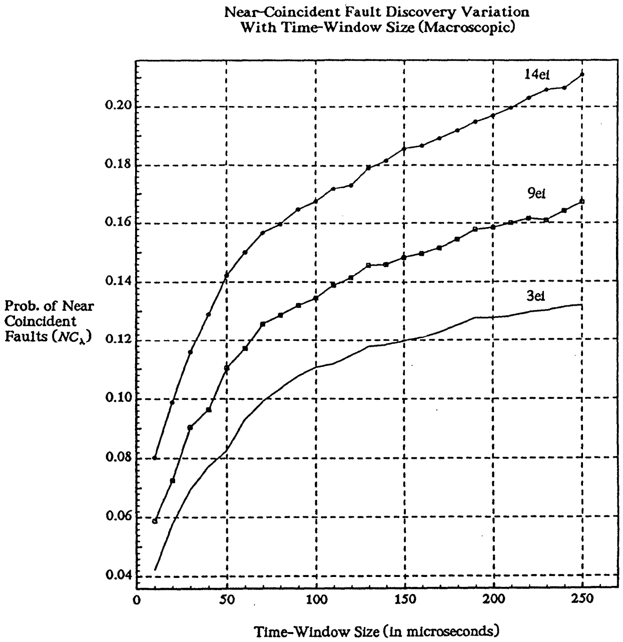


Figure 7. Macroscopic Time Window Size Variation of Probability

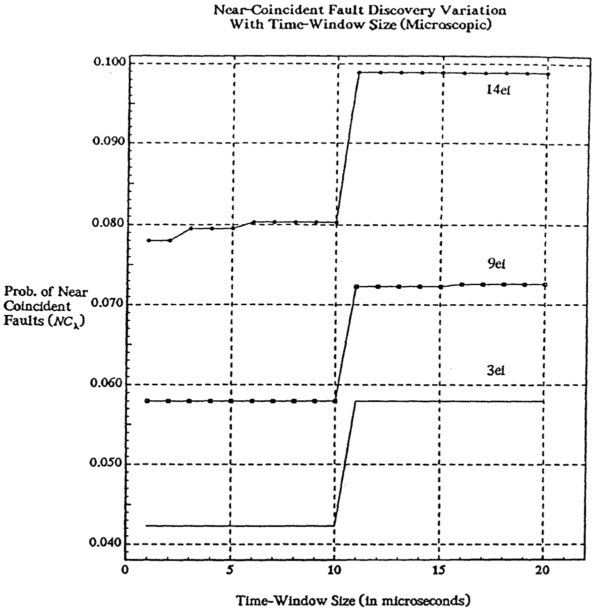


Figure 8. Microscopic Time Window Size Variation of Probability

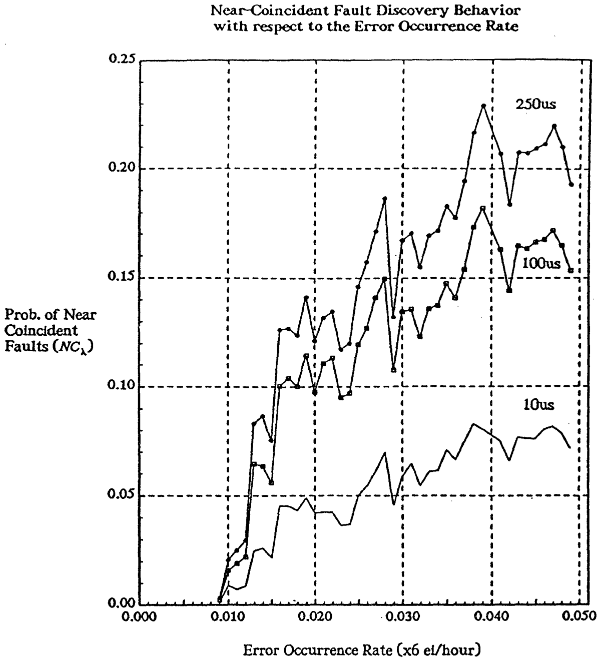


Figure 9. Probability of Near-Coincident Fault Discovery

the measurement period. From Figure 9, the near-coincident fault probability values range from 0.003 to approximately 0.21, over the 10 to 250 microsecond time-window size.

In comparing Figure 5 and Figure 9, we can see there exists a high correlation between the existence of multiple latent errors and their discoveries in near-coincidence. From Figure 9, the plot after an initial steep rise starts to taper as in the multiple latent error probability case. The saturation effect comes about more slowly in Figure 9 though, becoming more apparent at higher error occurrence rates than Figure 5. The reason for this is, as the rate of number of latent errors being "swept out" increases, the probability of near-coincident fault discovery also increases as a side effect. However after a certain error occurrence rate, the rate of removal of latent errors from the system has a more pronounced effect on the probability of near-coincident fault discovery. Thus the probability plot saturates slower as a result.

4. CONCLUSIONS

This paper has described a methodology to study the behavior of multiple latent errors and near-coincident fault discovery in the memory subsystem of a shared memory multiprocessor. Past studies have shown that these effects can seriously degrade the reliability of a system. The methodology was illustrated on a production multiprocessor system, the Alliant FX/8, running the operating system environment of the "Cedar" supercomputer.

The results show that even with a conservative error occurrence rate of one error per day, there is a 25% chance that errors result in multiple latent errors. It was also found that 8% of the error discoveries are near-coincident in nature with a time window size of 50 microseconds. It was also seen that the probability of multiple latent errors tends to saturate after a threshold error occurrence rate of approximately one error per day. The near-coincident fault discovery probability increases monotonically with larger time-window sizes. A strong correlation was found between the existence of multiple latent errors and their near-coincident discovery. The saturation effect on probability of near-coincident fault discovery was seen to take effect slower than that for the probability of multiple latent errors.

Future work is expected to involve investigation of methods to use such experimental results to make reliability and availability predictions for measured systems. It is suggested that other parallel systems be similarily studied so that more information on error characterization is available.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Ed Davidson for providing access to the Alliant FX/8 for experiment purposes. Many thanks to Mark Sloan, who wrote the core of the error latency simulator. Acknowledgments to Tracy Tilton, Mark Washburn, Madhu Sharma, Al Malony, Mike Haney and the other wonderful staff members at Center of Supercomputing Research and Development for their invaluable assistance.

REFERENCES

[1] A.L. Hopkins, T.B. Smith, J.H. Lala, "FTMP- A highly reliable fault-tolerant multiprocessor for aircraft," *Proceedings of the IEEE*, vol.66, No. 10, pp. 1221-1239, October 1978

[2] J.H. Lala, "Fault detection, isolation and reconfiguration in FTMP: methods and experimental results," *Proc. IEEE National Aerospace Electronics*, vol.1, pp. 21.3.1-21.3.9, 1984

[3] D. Kuck, D. Lawrie, A. Sameh and D. Gajski, "Cedar - A large scale multiprocessor," *Proc. 1983 International Conference on Parallel Processing*, pp.524-529, August 1983

[4] J. McGough, "Effects of near-coincident faults in multiprocessor systems," *Proc. IEEE/AIAA Fifth Digital Avionics Systems Conf.*, pp. 16.6.1-16.6.7, 1983

[5] K.G. Shin and Y.H. Lee, "Measurement and application of fault latency", *IEEE Transactions on Computers*, vol. C-35, No. 4, pp. 370-375, April 1986

[6] F.L. Swern, S.J. Bavuso, A.L. Martensen and P.S. Miner, "The effects of latent faults on highly reliable computer systems", *IEEE Transactions on Computers*, vol. C-36, No. 8, pp. 1000-1005, August 1987

[7] J. McGough and F.L. Swern, *Measurement of Fault Latency in a Digital Avionic Miniprocessor part-II*, NASA Contractor Report 3651, 1983

[8] R. Chillarege and R.K. Iyer, "Measurement-Based analysis of error latency," *IEEE Tranactions on Computers*, vol. C-36, No. 5, pp. 529-537, May 1987

[9] *Alliant FX/Series Product Summary*, Alliant Computer Systems Corp., Littleton, MA., October 1986

[10] R.K. Iyer, D.J. Rossetti and M.C. Hsueh, "Measurement and modeling of computer reliability as affected by system activity," *ACM Transactions on Computer Systems*, vol. 4, No. 3, pp. 214-237, August 1986

[11] R.K. Iyer and D.J. Rossetti, "A statistical load dependency of CPU errors at SLAC," *Proc. 12th International Sym. on Fault Tolerant Computing*, pp. 363-372, 1982

[12] *DAS 9200 System and Module A60 User's Guide*, Tektronix, Beaverton OR., 1987

[13] S.G. Mitra, Masters Thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana IL., 1988

A Distributed Architecture for the PEPSys Parallel Logic Programming System

Uri Baron, Bounthara Ing, Michael Ratcliffe, Philippe Robert

ECRC, Arabellastr. 17, 8000 Muenchen 81, West Germany

Abstract

The PEPSys project is concerned with the definition and evaluation of a parallel logic programming system addressing a complete spectrum of issues, from high-level language and applications to implementation on machine architectures. This paper discusses the design issues and trade-offs involved in the specification of a particular distributed architecture to support the sequential, OR , and Independent AND-parallel mechanisms of PEPSys. Of particular interest are the load balancing strategies adopted and their evaluation by simulation. The simulation of the architecture has also produced many other results which are presented along with a discussion of their implications.

1. Introduction

The general goal of the PEPSys (Parallel ECRC Prolog System) project, which started in mid - 1984, is to study and evaluate new and practicable solutions to the problems of parallel logic programming which was found to be a useful vehicle for expressing parallelism [6]. The PEPSys language, [4] was designed to exploit the OR and Independent-AND parallelism inherent in declarative logic programming languages. Together with the language, a superset of conventional PROLOG, a new computational model [7] and an abstract machine were designed. Besides the authors, all members of the PEPSys team have contributed to this work: H. Westphal, D. Peterson, J. Chassin, JC Syre.

Of particular interest, is the class of architectures most amenable to efficient execution of PEPSys. A class of architectures which supports the basic characteristics of the language, its computational model and abstract machine has been identified. This paper describes the architecture, its features, the underlying design philosophy and presents some preliminary performance results obtained from simulation of the architecture. The problems of load-balancing in the machine are discussed. A more detailed description can be found in [1].

The PEPSys computational model provides efficient solutions to problems central in any parallel Prolog implementation: the management of variable bindings in a parallel environment and the control of the search space to produce all (wanted) solutions. Its main features, detailed in [7], are retro-active parallelisation at very little cost, shallow binding with an explicit time-stamping mechanism and full combination of OR and AND parallelism with sequential backtracking execution. The implications of such a model on this architecture are discussed in the next section.

2. The PEPSys Architecture

The overall goals of PEPSys have greatly influenced the specifications of this architecture. In this section, an overview of the major decisions made in the design of the architecture is presented including justification thereof.

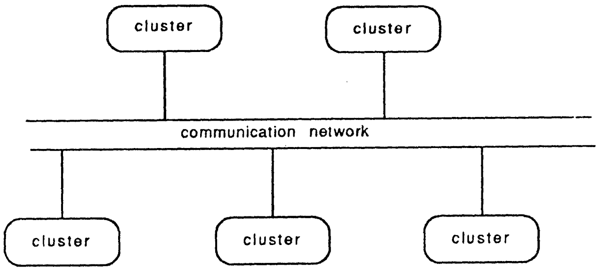


Figure 1: PEPSys Multi-Cluster Architecture.

Requirements of an Architecture

The architecture must deliver a scalable performance as the computing power of the machine is increased. Therefore it was critical to restrict parallelism to where it is really useful and to limit the increased communication overhead. A further requirement was that the architecture be 'open-ended' to allow for future modifications and extensions to be incorporated with relative ease. Finally, the architecture should be flexible enough to allow the machine to assume different roles, e.g. as a backend symbolic processor or as a front-end dedicated Prolog machine. To match the coarse-grained parallelism of PEPSys with communication costs, a *cluster* based design was chosen. Figure 1 depicts the abstract view of the cluster architecture.

2.1. Architectural Specifications

Making use of the experience gained in the implementation of PEPSys on a shared-memory Siemens MX-500 machine [3], the number of PEs has been scaled up by adding more clusters, thereby introducing the notion of distance between PEs; when PE*i* of cluster *j* wishes to access a variable on PE*k* on cluster *l*, it induces two levels of communication: intra-cluster and inter-cluster.

The PEPSys computational model solves the problem of maintaining multiple binding of variables in an OR-parallel environment through the use of *hash windows*. Combining (and nesting) AND-parallelism with OR-parallelism is done using an additional data-structure, *join-cells* together with hash-windows. From the architectural point of view - such a model does not impose the choice of memory used in the architecture (shared or unshared) as a process is an independent entity, identified by a process number, a hash-window and a root-frame [5], and accesses common variables by searching ancestor hash-windows. Thus the implementation of PEPSys using shared or private memories for each processing element is possible.

2.2. The Structure of a Cluster

A cluster is a small number of identical processing elements sharing a common memory and communicating with the shared memory through a high-speed bus (Figure 2). Several successful attempts have been made to implement OR-parallel computational models on limited-resources, shared memory machines [2] and therefore it was natural to extend this approach by connecting several such clusters to a communication network. A program's search space could then be divided between several clusters in order to improve performance and to be able to run much larger programs.

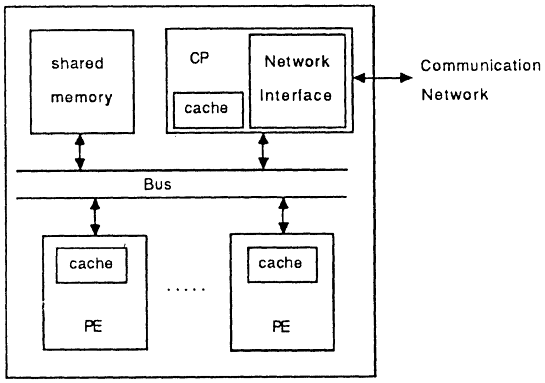


Figure 2: A cluster connected to a communication network.

It is clear that while increasing the processing power of the machine, its complexity has increased in the form of the extra level of communication between clusters. To alleviate this undesired consequence the problem was attacked on two levels: adding additional hardware and using sophisticated methods to reduce such communication. Each cluster is augmented with a Cluster Processor (CP) whose primary function is to handle *inter-cluster communication.* Other CP functions include: servicing remote dereferencing requests, managing local load-balancing (acquiring remote work when necessary), servicing PE requests, aborting processes and local bookkeeping.

The Inter-cluster Communication Network

The CPs communicate with each other over a common bus via *message passing*. The decision to use message passing between clusters was based on the following factors:

- message-passing is far more flexible and enables the implementation of different inter-cluster communication networks
- message-passing lends itself naturally to a distributed architecture in which individual components execute asynchronously.
- messages are of a higher level of abstraction - it is possible to implement a global address space with messages.

At implementation level the distribution of work amongst clusters is restricted to avoid a communication bottleneck. This was corroborated by simulation results for fairly large programs, which show an average bus utilisation of between 12%-30% of the total runtime, using a communication-inducing configuration: 1 PE per cluster for 10 clusters.

2.3. Cluster Processor (CP)

The Cluster Processor is viewed as the cluster's "work-horse" - a powerful processing unit performing a host of tasks, whose ultimate aim is to satisfy the local PEs' demands for work, values etc., while doing additional work in its idle time to speed up overall performance.

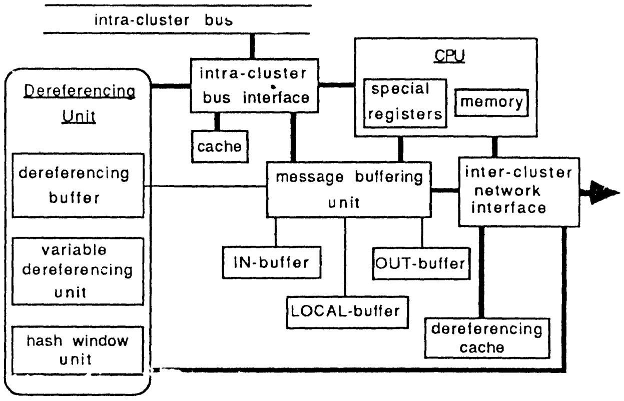


Figure 3: A Cluster Processor Block Diagram.

The block diagram of a cluster processor (CP) is shown in Figure 3. The CP has four message queues managed by hardware as FIFOs. The size of these buffers is small as only a few messages are expected to be in a queue at any given time. Buffer overflow is handled by directing excess messages to the CP's private memory. The CP has a small private memory used for cluster bookkeeping, temporary scratchpad and overflow areas. It has a fairly large set of dedicated registers in addition to a set of general purpose registers. The dedicated registers are used for fast access to CP data tables and counters. The basic execution cycle of a CP is sketched below:

```
loop:
  { process a message from each buffer
   if all buffers are empty then
          do_idle_time_work
  }
```

3. Load Balancing Strategies

Work is distributed between processing elements at two levels: intra-cluster and inter-cluster. The intra-cluster work-distribution strategy is an extension of [5] and is not discussed here; in this section the load balancing scheme implemented is described along with other possible schemes suitable for the architecture.

A workpool containing potential pieces of work for OR-branches and AND-branches is maintained in each cluster.

As these workpools are global and can be accessed by any PE in the cluster and of course by the CP; mutual exclusion must be ensured when modifying the workpool.

A PE executes a *local-search-for-work* procedure when it becomes idle, attempting to find work in one of the cluster's local workpools. If the PE finds work - it modifies the workpool and executes the work. On the other hand, if no work is available, the PE sends a message to the local CP asking for remote work. In other words, a *lazy* scheme for load-balancing, based on demand-driven activation by idle PEs is used. A backtracking PE can either wait for its children to terminate, or it can speed up their termination by taking work from their descendants (constrained remote work). The latter scheme was implemented and it was found that this operation must be severely restricted to prevent delocalizing computation (by the spreading of too small sized sub-trees) and thrashing of goals between clusters. Figure 4 depicts a simple example:

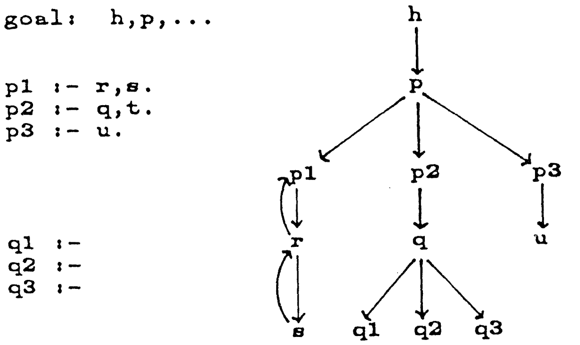


Figure 4: A Load Balancing Example.

p and *q* are or-parallel predicates and in the example their clauses are numbered *p1,q1,p2,q2*... for clarity. Assume the first clause of *p* is executed on cluster*1* by PE*1* and the second clause is executed by PE2 on cluster2. PE*1* executes *h*, followed by *p1* then *r* then *s* (Figure 4). Execution of *s* fails resulting in PE backtracking to *p1*, now PE*1* cannot backtrack beyond this point until p2 has terminated on cluster2. At the same time *p2* has spawned *q*, another parallel predicate which in turn may have alternative clauses 'stolen' by other clusters. PE*1*'s decision to ask for work from *p2* or rather to wait for *p2*'s termination is extremely difficult to make, depending heavily on a particular program's behaviour. This kind of inter-cluster work distribution can be restricted by 'capturing' entire sub-trees in one cluster, or by refusing such a work request on the remote cluster.

Only half the picture has been explained. The CPs play a crucial role in distributing the load over the entire machine. The CP maintains a queue of idle PEs which have requested 'remote' work. Periodically, this queue is inspected by the CP and when local conditions are met - the CP selects a remote cluster and sends it a 'request_for_work' message. The CP manages a list of remote CPs to query for work, based on partial information it has obtained while polling these CPs and from messages received from them. Alternative approaches would be to remove polling CPs altogether and provide some random choice function instead or to monitor the bus.

A CP receiving a request for work initiates the *search_for_work* procedure. If the cluster is too busy or alternately too idle, it can refuse the request immediately. When work is found it is sent to the requesting cluster and the local workpool structures are updated.

This load-balancing strategy places the burden of finding work locally on the PE, which is idle anyway. The CPs control the amount of external work requests outstanding per cluster, even so, requests for work must be kept to a minimum through the use of pragmas (at program level) and run-time information. On the receiving side - finding work for remote PEs is done by the CP without interrupting local PEs.

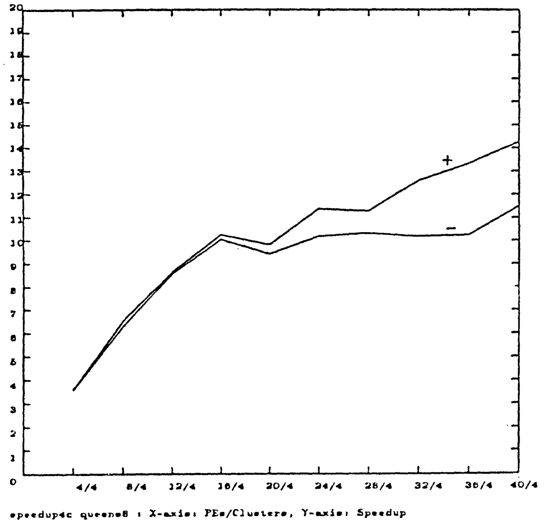


Figure 5: 8 Queens problem with work distribution optimisation.

By reserving branches high up in the search tree for remote clusters, additional performance gains averaging 20 per cent have been achieved. Figure 5 shows the performance for various configurations of the architecture running the 8 queens, all solutions program with this improved, initial work distribution. The speedups delineated in the graphs below, are obtained by comparing against the performance of a uni-processor on the same benchmarks.

4. Performance Analysis

In this section some preliminary performance results obtained from architectural simulation are presented. A multitude of configurations with the number of clusters and PEs ranging between one to ten were simulated. For a single cluster architecture up to 30 PEs were simulated. No optimizations of any kind were included in these architectures, i.e. the CP was an ordinary processor running at the same speed as a PE, it had *no* parallel sub-units and management of its message buffers was done by software. A simple load-balancing scheme was employed: no restrictions were imposed on suspended PEs' requests for work from their (remote) children and a threshold equal to half the number of PEs in a cluster was used to control external requests for unconstrained work. In the implementation of a cluster none of the important optimizations to PE dereferencing or local work management have been made. The graph in Fig. 6 shows the speedup obtained for the 8 queens program, run on a particular set of configurations: 4

PEs per cluster, 6 PEs per cluster, 8 PEs per cluster and 10 PEs per cluster, for 1 to 10 clusters. The three factors mentioned below account for the less-than ideal speedups achieved:

- simple load-balancing -
  - the scheme should severely restrict the taking of remote work
  - the initial 'spreading' of work has to be improved
- no optimizations were performed - the CP must execute faster than its local PEs and should have asynchronous, hardware sub-units.
- the test-programs must be large enough with respect to the amount of sustainable parallelism they exhibit

Detailed statistics were gathered from two representative configurations: a 100 PEs on 10 clusters, and 10 PEs on 10 clusters. As expected, the inter-cluster bus does not cause a communication bottleneck. This is due largely to the process-oriented nature of PEPSys' computational model which was discussed previously.

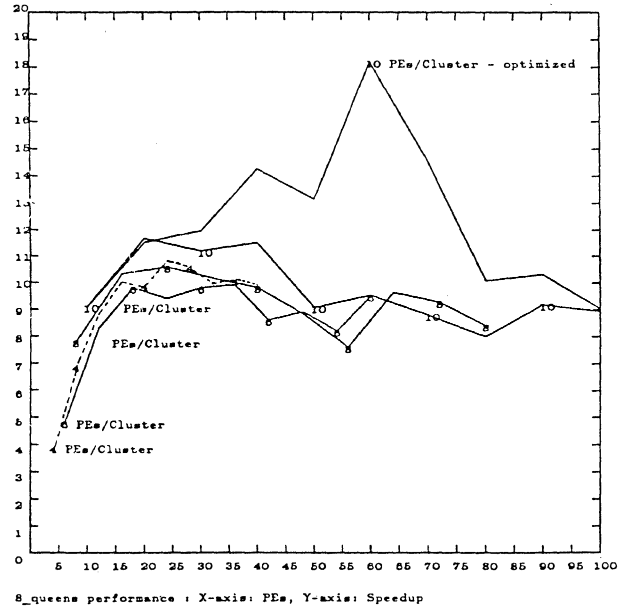


Figure 6: Overall performance of 8 Queens for different cluster configurations.

The distribution of communication (in the form of message passing) between any two clusters in a configuration is almost uniform, excluding the cluster initiating the computation which always has a heavier load. By dividing the work to be done at the highest possible level in the search tree, this additional overhead can be eliminated on the initiating cluster.

5. Conclusion

The architecture presented above fulfills the initial requirements -

- an increase in processing power is viable through parallelism and is its implementation in hardware is feasible with state-of-the-art technology
- flexibility - communication is limited and evenly distributed between clusters making replacement of the communication network easy, once the target environment of the machine is known. Adding complexity to the CP can be done in a straightforward and efficient manner.

Partial answers to questions posed by the computational model such as the frequency of dereferencing, the availability of long sequential branches in PEPSys programs and hash-window chain lengths have been obtained. Many important questions regarding the implementation of PEPSys were investigated, in particular, work-distribution strategies which were found to influence performance immensely. The concept of distance between processing elements was introduced, allowing greater processing power while not vitiating performance too severely. More "real" programs need to be measured to provide empirical validation of the design and performance must be boosted significantly. The existence of a large class of applications, generating sufficient amounts of parallelism to sustain the machine, must be ascertained.

References

[1] U. Baron, B. Ing, M. Ratcliffe, P. Robert.
A Distributed Architecture for the PEPSys Parallel Logic Programming System.
Technical Report 25, ECRC, Nov, 1987.

[2] R. Butler, E.L. Lusk, R. Olson, R.A. Overbeek.
ANLWAM - A Parallel Implementation of the Warren Abstract Machine.
Internal Report, Argonne National Laboratory, 1986.

[3] Chassin, J., Westphal, H. and Peterson, D.
The Implementation of PEPSys on a MX-500 MultiProcessor..
Internal Report, ECRC, December, 1987.

[4] M. Ratcliffe, J.C. Syre.
The PEPSys Parallel Logic Programming Language.
In *IJCAI.* ECRC, August, 1987.

[5] Philippe Robert.
An Emulator for the PEPSys Abstract Machine.
Internal Report PEPSy 17, ECRC, April, 1987.

[6] Akikazu Takeuchi and Koichi Furukawa.
Parallel Logic Programming Languages.
In *Third International Conference on Logic Programming*, pages 242-254. July, 1986.

[7] H. Westphal, P. Robert, J. Chassin, J.-C. Syre.
The PEPSys Model: Combining Backtracking, AND- and OR-parallelism.
In *Proceedings - 1987 Symposium on Logic Programming*, pages 436-448. IEEE, September, 1987.

A DATAFLOW ARCHITECTURE FOR OR-PARALLEL EXECUTION OF LOGIC PROGRAMS

A. V. S. Sastry and L. M. Patnaik
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560012
INDIA

ABSTRACT Logic programming languages have gained wide acceptance because of two reasons. First is their clear declarative semantics and the second is the wide scope for parallelism they provide which can be exploited in parallel implementation. In this paper, a dataflow architecture (based on Manchester ring) to support **OR-Parallelism and Argument Parallelism** is proposed. A new scheme for handling deferred read mechanism using the matching unit of the machine is suggested. The required data structures and the built-in dataflow procedures for OR-parallel execution are discussed. Multiple binding environments are handled by a modified form of directory tree method that is suitable for dataflow implementation. This method is illustrated by an example. The dataflow graphs of the program clauses are calls to the built-in procedures, therefore they are modular and independent of argument complexity. This feature makes the compilation of the clauses very easy.

1. INTRODUCTION

Logic programming is a novel programming style with clear declarative semantics. It means that the user program is more like a specification of the problem than the specification of the algorithm - as is the case with conventional von Neumann languages. Therefore writing programs in this paradigm is very simple and elegant. Comparing logic languages with functional languages, we find that a logic program can be thought of as an equivalent of a set of functional programs, one functional program corresponding to one instance of the input-output mode of the arguments of the clause in the logic program. Hence logic programs are more compact as compared to functional programs.

One important application of logic programming is in its use in knowledge representation and reasoning. Both declarative and procedural knowledge can be represented quite succinctly in the form of clauses. Facts and rules can be represented with the same ease. Reasoning with knowledge can be done using the inference rule of first order logic called **resolution.** This ability makes this language paradigm quite suitable for artificial intelligence applications.

A promising feature of logic languages from implementation point of view is that they do not obscure any parallelism present in the program. Parallel architectures based on control flow, dataflow and reduction,[3,10..15] have been proposed. The motivation for parallel architectures is to speed up symbolic computation - where problems are quite unstructured and weak methods of problem solving are applied.

In this paper, we propose an extension of Manchester dataflow machine that can support **OR-Parallelism** of logic programs. In addition to that, our machine also supports **Argument Parallelism.** A new scheme for handling the **deferred read mechanism** , which permits a read request at an empty memory location, using the matching unit of the machine is discussed. This simplifies the design of the memory modules. We propose suitable data structures and explain how the **OR-Parallel execution** is possible on our machine. The rest of the paper is organized as follows. Section 2 contains a brief description of the definitions and the basic computational model of logic programs. Section 3 gives a description of our architecture and discusses the deferred read mechanism. Section 4 deals with the data structures, dataflow procedures and the scheme for handling the binding environments. Section 5 contains the preliminary simulation results of execution of a simple logic program on this machine.

2. PRELIMINARIES

2.1 Basic Definitions

A logic program is a set of **clauses** expressed in first order logic. We restrict ourselves to a specific subset of first order logic called **Horn clause logic.** The syntax of a clause is given below:

A:-B1,B2,...,Bn

where A,B1,B2,...,Bn are called Predicates which are relations over the given domain, A is called the **head** of the clause and B1,B2,...,Bn together constitute the **body** of the clause. Each predicate has a fixed arity. A **predicate** is represented as

P(t1,t2,t3,...,tn)

where P is the predicate name and t1,t2,...,tn are the arguments which are the terms of the first order logic language. An example of a predicate is Father(john,mary) which asserts the relationship between two terms john and mary of the domain.

A **term** is recursively defined as
1. A variable is a term, e.g. x,y and z
2. A functor f(t1,t2,...,tn) is a term where t1,t2,...,tn are terms and f is the functor name. The arity of the functor is n. The constants of the domain are the functors with 0-arity.

There are three kinds of clauses. A **unit clause** does not have a body. A **definite clause** has both head and body. A **goal clause** has body but no head.

2.2 Interpretation of Logic Programs

The declarative meaning of a clause A:-B1,B2,...,Bn is that the predicate A is true if B1,B2,...,Bn are simultaneously true. A unit clause is unconditionally true as it does not have a body. Unit clauses form the facts of the program. Definite clauses are rules and goal clauses are the intended queries made on the logic program. In logic parlance, the set of clauses form the set of axioms. A goal clause is a theorem to be proved. The meaning of execution of a logic program is to find the instances of the goal clause implied by the given set of clauses.

From the point of view of execution of logic programs, Kowalski has given a nice **procedural interpretation** to logic programs. In this view, each clause can be considered as a procedure. The body of the clause is nothing but a set of procedure calls. A goal clause can be considered as the initial set of calls to the various procedures in the programs. The passing of parameters from goal to body is done by a bidirectional syntactic pattern matching procedure called **unification.**

2.3 Basic Computational Model

The underlying model of computation is **unification.** Comparing it with **reduction** which is the computational model of functional languages, we find that in case of unification there is no commitment of the variables as input or output. Specification of a subset of variables of the goal clause and execution of the program result in the solution which specifies the values of the unspecified variables. In reduction, a set of variables is designated as input variables. Input variables have to be specified to get the output implying that reduction is unidirectional. The unification of two predicates results in a minimal set of variable bindings called the **Binding Environment(BE).** If the two predicates are unifiable, the BE created is unique and known as the most general unifier of the two predicates. If the two predicates are not unifiable then the result of unification is a 'fail' message.

In order to solve a goal in a given logic program, an inference rule called **resolution** is applied. The basic algorithm for solving the goal clause of a logic program is outlined below:

```
Initialize goalset to the given goal
While goalset not empty do
Begin
  step1 : select a goal from goalset
  step2 : find the matching clause/clauses
  step3 : unify the head of the clause and the
          goal to generate the BE
  step4 : pass the bindings to the body of
          the selected clause and include the
          body in the goalset
End
```

2.4 Parallelism in Logic Programs

In the above mentioned computational model, many steps can be performed in parallel. They are classified[2] as follows:

Search Parallelism: Searching for the candidate clauses for unification and resolution can be done in parallel by associative search.

OR-Parallelism: When more than one candidate clause are present in the program, all of them can be attempted simultaneously in order to obtain alternative solutions.

AND Parallelism: Solving a goal clause reduces to solving the subgoals in that clause. These subgoals can be solved simultaneously the only constraint being the consistency of the bindings generated by the subgoals.

Argument Parallelism: The arguments of the two predicates can be unified in parallel. The parallel unification requires consistency check on the BE. The reason is that shared variables taking part in unification should get bound to consistent values.

3. THE DATAFLOW ARCHITECTURE

The motivation to choose the dataflow architecture for executing logic programs is the inherent computational structure of the problem. A goal in a logic program initiates the computation, therefore the program is goal-driven. Thus if the goal is considered as a data item, there is a direct correspondence between execution of logic programs and execution of dataflow graphs on a dataflow machine.

Our architecture is based on the Manchester ring[4,5,6] which is shown in figure 1. The original architecture does not provide any special hardware for structure handling. Array data structures are supported by the matching unit using specialized matching functions[7]. Recently Manchester machine has been augmented with structure store[9] similar to Arvind's I-Structure store[1]. As the execution of logic programs requires efficient handling of structures, we also provide the structure memory (SM) modules in our machine. The structure memory is functionally same as Arvind's I-structure memory. The basic architecture of the proposed machine is shown in figure 2(a). We have added one more unit to the machine which we call the **Definition Search Unit.**

A logic program is assumed to be compiled into a set of **definitions.** A definition is the set of clauses having the same head. The goal predicate identifies its definition and attempts to unify its arguments with all the clauses in its definition. This function of selecting the candidate clauses for unification is accomplished by the definition search unit which is shown in figure 2(b). It has two memory units **Definition Search Memory** and **Clause Address Memory.** The clause address memory stores the starting address of the dataflow graphs of each of the program clauses such that addresses of all the clauses in a definition are stored contiguously. Starting address of each definition in the clause address memory is stored in **Definition Search Memory.** When a goal arrives at the definition search unit, it searches for the address of its definition in the definition search memory. After getting the starting address of the definition, a copy of the goal is sent to all the clauses of that definition using clause address memory. Thus the purpose of the definition search unit is to initiate computation in all the solution paths of the goal simultaneously.

3.1 Deferred Read Mechanism

In our architecture, we provide a new way of handling deferred read requests using the matching unit. The reason to do so is two-fold. First, it simplifies the design of the memory units. Second, communication between the processor and the memory is through a bus, minimizing control in the memory units would reduce the processing time of the memory unit thereby reducing the latency between the processor and the memory module. Another advantage is that the hashing mechanism of the matching unit[4,5,6] can be used to support the deferred read mechanism without any extra hardware.

A token in the machine can be represented as a tuple

<data,c,i,destination,operand type,token type>

The first three fields following the data field namely c, i and destination fields are required to identify an instruction of a particular invocation[5] where c and i called the **color** and the **iteration count** constitute the tag. The 'operand type' decides whether the token requires matching. We define another field called **token type.** This is necessary for supporting deferred read mechanisms. The various token types are 'ordinary' 'deferred ' and 'signal release'. The **'ordinary'** tokens are the ones which are generated by the processor during the normal course of execution of the dataflow graph. The other two types of tokens namely **'Deferred '** and **'Signal Release'** are generated when a memory read operation at a particular location occurs before the memory write operation at that location. Their generation and use are described below:

A memory location can be in one of the three states, **present, absent,** or **waiting** [1]. When a memory read is requested at a particular location 'l', there are three possibilities. If the state of the memory location is **present,** the read is said to be successful and the result is routed to the destinations of the read instruction. If the state is **absent or waiting,** the read instruction cannot be satisfied immediately because the location does not contain any data value. Such a read request is deferred[1]. The processor changes the state of the memory word to **waiting** and generates a 'deferred' token of the format

<dest-i, c, iter count, l, op-i, "deferred">

where 'dest-i' and 'op-i' of the deferred token are the destination and operand type of the ith result token of the read instruction. The destination field is 'l' which is the address of the memory location where the read was attempted, 'c' and 'iter count' are obtained from the tag of the read instruction. The number of 'deferred' tokens generated is equal to the number of destinations of the read instruction. These 'deferred' tokens wait in the matching unit for the 'signal release' token which is generated by some write instruction at the memory location 'l'.

When a write instruction into the memory location 'l' is executed, there are three possibilities. If the state is **present,** the write instruction is invalid[1]. If the state is **absent,** the data is written into memory location 'l' and its state is changed to **present.** If the state is **waiting,** the data is written into the memory location 'l', its state is changed to **present** and the a 'signal release' token of the format

<dl, 0, 0, l, -, "signal release">

is generated by the processor. The data field contains 'dl' which is the value written in the memory location 'l'and the destination field contains 'l' which is the address of the memory location where write operation is performed. This 'signal release' token searches for a partner token present in the matching store. A token 'i' is its partner if the destination field of 'i' matches with the destination field of the incoming 'signal release' token and the token 'i' is 'deferred'. The 'signal release' token extracts all the 'deferred' tokens from the matching unit which match successfully with it. Corresponding to each matched token 'i' of the matching store a new token 'k' of the form

<dl, c, iter count, dest-i, op-i, "ordinary">

is generated. This token is nothing but the result of the read instruction at the memory location 'l'. Its data field contains dl, the content of the memory location 'l' and is obtained from the data field of the 'signal release' token. The destination field of token 'k' contains the destination to which the this 'ordinary' token 'k' should be routed. The destination address of token 'k' is obtained from the data field of the matching deferred token 'i'. The color, iteration count and operand type fields of token 'k' are copied from the corresponding fields of the deferred token 'i'. Depending on the operand type, token 'k' is either put back in the matching unit distributor or is sent forward to the node store. Thus the 'signal release' token releases all the deferred read requests generated for the memory location 'l'. There are two ways of implementing this scheme. One is to provide a separate store in the matching unit where only deferred tokens are stored. This scheme does not affect the search of ordinary tokens but effective utilization of the matching unit is reduced. The other scheme does not allocate any separate memory and is based on the assumption that the number of deferred tokens is a minor fraction of the total number of tokens generated by the program, therefore the effect of the deferred tokens residing, in the matching store, on the ordinary tokens is insignificant.

4. DATA STRUCTURES AND DATAFLOW PROCEDURES

To execute logic programs, the machine has to support all the basic data types that are used in logic programs. These are constants, variables, lists and structures. In fact, list and constant types are special cases of structures but are treated separately in PROLOG[3]. We also follow the same convention. We represent the BE as a list of <variable, binding value> pair. Apart from these data types, some structures are necessary for representing a goal, a context and a binding environment. These are goal node, context and binding node respectively. They are described as follows

Goal Node: It is a 2-tuple <<predicate name, argument address>, context pointer>. The predicate name is used to identify the corresponding definition of the predicate in the definition search unit. The argument pointer points to the array containing the arguments of the goal. The context pointer points to the last context created.

Binding Environment: It is a list of binding nodes where each binding node is represented as a 3-tuple <variable,binding value,next node> where variable and binding value fields are used to represent the bindings created during unification. Next node is used to form the list of bindings.

Context: It is a record like data structure used to hold the BE along with other control information. The following are the fields in a context.

Context number: Each context is identified by a unique number. This number is used in renaming the variables of a clause in order to distinguish the variables of the clause under different invocations.

Tag: It is a <color, iteration count> pair which is used in restoring the tag of a data token when it returns from a clause. At the time of the creation of a context, the return tag of the token is stored in the 'tag' field.

Destination: It gives the address in the dataflow graph to which the result token should return after exiting from the clause.

Prev context: It is a pointer to the previous context.

First: It is a pointer to the first element in the BE .

Last: It points to the last element in the BE.

Unify fail: A boolean variable that indicates the status of the environment which can be valid or invalid.

4.1 Handling Multiple Binding Environments

Jim Crammond[8] has discussed three methods for handling multiple BEs for OR-parallel execution of logic programs. These are Directory tree, Hash windows and Variable importation. A modified form of directory tree method which is suitable for dataflow implementation is employed in our architecture. The advantage of our scheme is that no environment copying is done. Since we represent the environment as a list of bindings, the time required to search for the value of a variable is O(n) where n is the number of elements in the BE.

When a clause is invoked, a new context which is referred to as **present context** is allocated in the structure memory. The return address and the return tag are stored in the present context. The arguments of the clause head are unified with those of the goal and the BE is created. The prev_context field of the present context is made to point to the previous context. The pointer to the previous context is obtained from the context pointer field of the goal node. When the clause does not have any more goals to be solved, the BE of the previous context is appended to the BE created in the present context and the new BE so created is stored in a new context called **return context** whose other fields are copied from the corresponding fields of the previous context. The return context is returned from the clause to the destination indicated in the destination field of the present context. The two distinct advantages of doing so are,

1. Search for a variable need be done only in one context. On the contrary, in case of the directory tree technique, search has to be done in a list of contexts.
2. No copying of environment is required at any stage, but in the case of directory tree method, the uncommitted contexts which contain at least on unbound variable are to be copied and passed on to the subsequent contexts.

We illustrate by a hypothetical example how environments are managed. Consider a program having six clauses given below.

(1) P:-Q,R. (2) P:-A. (3) Q. (4) Q. (5) A.(6) R. The goal is P. We assume that each predicate has some arguments. The goal P unifies with the clauses 1 and 2 to produce two contexts c1 and c2 as shown in figure 3(a). The BE created context i is represented as '(bei)'. These two clauses 1 and 2 form the two alternative solution paths for the goal P. The subsequent goals in the two solution paths are Q and A. The goal Q unifies with clauses 3 and 4 to generate two more contexts c3 and c4. Both these contexts have c1 as their predecessor. The goal A unifies with clause 5 and creates context c5. Its predecessor is c2. The configuration of the contexts at this stage is shown in figure 3(b). The clauses 3, 4 and 5 ,being unit clauses, do not have any subgoals in their body, therefore the 'return' contexts r1, r2 and r3, shown in figure 3(c), are returned to the destinations specified in contexts c3, c4 and c5 respectively. The contexts r1 and r2 from clauses 3 and 4 return to clause 1 and form two new goals with the predicate R. The context r3 from clause 5 returns to clause 2. As clause 2 does not have any more goal, the BE in that context (r3) is returned as one of the three solutions as indicated in figure 3(c). The two goals R corresponding to the two 'return' contexts r1 and r2 unify with clause 6 and create contexts c6 and c7 as shown in figure 3(d). Subsequently 'return' contexts r4 and r5

are returned from clause 6 to give the other two solutions as shown in figure 3(e). Looking it another way, we find that during the process of execution of a goal, a tree of BE is created with each path corresponding to one alternative solution. Whenever a goal enters a clause, it increases the level of the tree grown so far by one. Whenever an exit is made from a clause, the level of the tree is reduced by by one. Ultimately when all the goals are solved, the tree is converted into a set of BEs where each is an alternative solution to the goal.

4.2 Dataflow Procedures

As the user is relieved of the burden of specifying the algorithm, the machine must have some built-in mechanisms for controlling the execution of programs. In the case of sequential PROLOG unification and backtracking are built into the machine[3]. Analogously, we provide the following built-in dataflow procedures for execution of logic programs. (i) **UNIFY** (ii) **WRITE IN BE** (iii)**SEARCH** (iv)**COPY ARGUMENTS** (v) **EXIT** (vi)**CALL COPY AND UNIFY** (vii)**PASS ARGUMENTS.** We describe below these procedures in a Pascal-like syntax and give explanations wherever necessary.

```
procedure UNIFY(tl,t2,ct);/* tl and t2 are the
two terms to be unified and ct is the context */
begin
 if not ct.Unify fail then
 begin
  if tl.dtype=t2.dtype then
  begin
   if tl.dtype='atom' then
   begin
    if tl.val<>t2.val then
    ct.unify fail:=true;
   end;
   if tl.dtype='var' then
   WRITE IN BE(tl, t2, ct);
   if tl.dtype='list' then
   begin
    UNIFY(tl.head,t2.head, ct);
    UNIFY(tl.tail,t2.tail, ct);
   end;
   if tl.dtype='struct' then
   begin
    if different struct names then
    ct.unify fail:=true else
    begin
     for i:= 1 to n do
     /* n:no. of arguments*/
     UNIFY(tl.arg(i),t2.arg(i),ct);
     synchronize;
    end;
   end;
  end
  else
  begin
   if tl='var' then
   WRITE IN BE(tl,t2, ct);
   else if t2='var' then
   WRITE IN BE(t2, tl, ct);
   else ct.unify fail:=true
  end;
 end;
 return ct;
end.
```

The procedure **UNIFY** unifies the two terms and calls the procedure **WRITE IN BE** to write the <variable, value> pair in the BE.

```
procedure WRITE IN BE(vl,dl,ct); /* vl:variable,
dl: binding value, ct : context*/
begin
 if not ct.unify fail then
 begin
  bn:- request new binding node;
  bn.variable:-vl;
  bn.binding value:-dl;
  search for vl in BE,
  if vl is present  then
  begin
   dl':-value of vl already present in BE;
   UNIFY(dl',dl,ct);
  end
  else  put bn in the BE;
 end;
 return ct;
end.
```

The procedure WRITE IN BE checks for the presence of a variable in the BE. If the variable is already present, it performs the consistency check on the BE by unifying the two values of the variable, one already present and other one the new binding value dl discussed above. If the variable is not present, the <variable, value> pair is written in BE.

```
procedure SEARCH(al, dl, ct);/* al :the address
of an empty memory location, dl : term and ct :
the context */
begin
 stepl :if dl is ground or env=nil then  write
 dl in address al;
 if dl is a variable then
 begin
  search for dl in BE;
  if not found then write dl in al;
  else
  begin
   new dl:-binding value of dl;
   new  ct:-ct;
   new al:-al;
   goto  stepl
  end;
 end;
 if  dl is a list then
 begin
  al':-request new list node;
  write al' in al;
  SEARCH(al'.head,al.head,ct);
  SEARCH(al'tail, al.tail, ct);
 end;
 if dl is a structure  then
 begin
  fl':-request new structure;
  store functor name and number of arguments;
  for i:=1 to number of arguments do
  SEARCH(fl'.arg(i), dl.arg(i),ct);
 end;
end;
```

The procedure searches the environment for the given term and writes the result of the search process in address a1.

```
procedure COPY ARGUMENTS(a1, d1, i);/* a1 :
address of a memory location, d1 : term and i :
integer */
begin
 if d1 is ground then write d1 in a1;
 if d1 is variable then
 begin
  d1':=rename(d1,i);
  write d1' in a1;
 end;
 if d1 is a list then
 begin
  d1':-request new list node;
  COPY ARGUMENTS(d1'.head, d1.head, i);
  COPY ARGUMENTS(d1'.tail, d1.tail, i);
 end;
 if d1 is a structure then
 begin
  f1:- request new structure;
  store functor name and arguments
  for j:=1 to number of arguments do
  COPY ARGUMENTS(f1.arg(i),d1.arg(i), i);
 end;
end.
```

The procedure COPY ARGUMENTS writes the arguments into the address location specified. This procedure is required because each unification is performed on a new copy of the arguments. The instruction 'rename', renames a variable by tagging it with a number. The renaming is required to distinguish the variables of a clause under different invocations.

```
procedure exit(ct);
/* ct is a context pointer */
begin
 if ct.prev_context=nil then
 begin
  return ct to the destination specified in its
  destination and set the tag of the return
  token with ct.tag;
 end
 else
 begin
  ct':-request new context;
  copy the four fields context name, tag,
  destination and prev_context of ct' from the
  corresponding four fields of ct.prev_context;
  return ct' to the address specified in the
  destination field of ct and set its tag (color
  and iteration count) using tag field of ct;
  if ct.first=nil then
  begin
   ct'.first:-ct.prev_context.first;
   ct'.last:-ct.prev_context.last;
  end
  else
  begin
   ct'.first:-ct.first;
   if ct.prev_context.first=nil then
   ct'.last:-ct.last
   else
   begin
    ct'.last:-ct.prev_context.last;
    ct.last.next_node:-ct.prev-context.first;
   end;
  end;
 end;
end.
```

The procedure EXIT appends the BE of the previous context (ct.prev_context) to the BE of the present context and creates a return context ct' if the previous context is not nil , otherwise it returns ct. The return address and return tag are obtained from the present context(ct).

```
procedure CALL COPY AND UNIFY(g1,ct,n,addr,r);
/* g1 : goal node, ct the context, n : number of
arguments in the head of a clause, addr : the
address of array of arguments of the head, and
r : return address */
begin
 a1:-request new array(n);
 a2:-request new array(n);
 k:=ct.context number;
 for i:=1 to n do
 begin
  COPY ARGUMENTS(a1(i),addr.arg(i), k);
  COPY ARGUMENTS(a2(i),g1.arg addr.arg(i),0);
  /* 0 indicates that renaming is not done*/
  UNIFY(a1(i), a2(i), ct)
 end;
 synchronize;
 return ct;
end;
```

The above procedure invokes unification of all the arguments of a clause head with those of the goal. The instruction 'synchronize' is used to wait till all the unifications of the arguments are complete which ensures the complete formation of the BE.

```
procedure PASS ARGUMENTS(a1,d1, n, ct); /* a1,
d1 : pointers to arrays, n : integer and ct :
context */
begin
 for i:=1 to n do
 SEARCH(a1(i), d1(i), ct);
end;
```

The above procedure carries out the search for each argument of d1 in the BE and writes the result of the search process in the corresponding element of array a1.

A clause is represented as a dataflow graph consisting of calls to procedures CALL COPY AND UNIFY, PASS ARGUMENTS and EXIT. The dataflow graphs for a definite clause and unit clause are shown in figure 4(a) and 4(b) respectively. The operators 'split goal', 'form goal' are used for manipulation and creating of the goal node. The main feature of the dataflow graphs of the clauses is their modularity. The complexity of the graphs is independent of the argument complexity which enables easy compilation of the program clauses.

5. SIMULATION OF THE ARCHITECTURE

A simulator for this architecture is developed in SIMULA-67 on a DEC-1090 system with a view to studying the performance of the machine in terms of the speedup of the machine and utilization of various hardware units namely processor, matching unit, node store unit and memory modules. The variables are, number of processors, number of matching units and the number of memory modules. We have simulated the Grandparent relationship problem on our machine. The problem,though simple, captures all the features of a logic program. The relation between execution time and number of processors for this problem is shown in figure 5(a). We find that for this particular problem, the maximum speedup achievable is 3 with eight processors. The variation of utilization of the matching unit and the processing element with the variation in processing elements is shown in figure 5(b). The utilization of the matching unit is low (26%) for a single processor and reaches a maximum (67%) with eight processors. Further simulations of more complex problems are under progress.

6. CONCLUSIONS

A dataflow machine for executing logic programs is proposed. The machine supports OR-parallelism and argument parallelism. A new scheme for handling deferred read mechanism is suggested. The dataflow graphs for the program clauses are quite modular and are independent of the complexity of the arguments, hence the compilation of the clauses is easy. Work is in progress to devise better schemes for representing the BE with a view to minimizing search process.

ACKNOWLEDGEMENTS

The authors thank Mr S. Sundaram, Centre for Computer Aided Design, Indian Institute of Science, Bangalore for helping them in the preparation of the manuscript.

REFERENCES

[1] Arvind and R.E.Thomas, I-Structures:An Efficient Datatype for Functional Languages, Technical Memo TM-CSG-174, Laboratory for Computer Science, MIT, September 1980.

[2] J.S.Conrey and D.Kibler, "Parallel Interpretation of Logic Programming," Proceedings of the International Conference on Functional Programming and Computer Architecture, 1981.

[3] D.H.D.Warren, Implementing Prolog-Compiling Predicate Logic Programs, D.A.I., Research Report 39, University of Edinburgh, 1977.

[4] I.Watson and J.R.Gurd, "A Practical Dataflow Computer", IEEE Computer, February 1982.

[5] J.R.Gurd, I.Watson and J.R.W.Glauert, A Multilayered Dataflow Architecture, Internal Report, Department of Computer Science, University of Manchester, 1980.

[6] J.R.Gurd, C.C.Kirkham and I.Watson, " The Manchester Prototype Dataflow Computer", CACM, Vol. 28, No. 1, January 1985.

[7] J.Sargeant and C.C.Kirkham, "Stored Data Structures on the Manchester Dataflow Machine", Proceedings of the 13th Annual Symposium on Computer Architecture, 1986.

[8] Jim Crammond, "A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages", IEEE Transactions on Computers, Vol. C 34, No. 10, 1985.

[9] K.Kawakami and J.R.Gurd, "A Scalable Dataflow Structure Store", Proceedings of the 13th Annual Symp. on Computer Architecture, 1986.

[10] N.Ito et al., "The Architecture and Preliminary Evaluation Results of The Experimental Parallel Inference Machine PIM-D", Proceedings of the 13th Annual Symp. on Computer Architecture, 1986.

[11] N.Ito et al., "Dataflow Based Execution Mechanisms of Parallel and Concurrent Prolog", New Generation Computing 3(1985).

[12] R.Hasegawa and Makato Amamiya, "Parallel Execution of Logic Programs Based on Dataflow Concept", Proceedings of the International Conference on Fifth Generation Computer Systems, 1984.

[13] R.Hasegawa et al, "An Architecture for List-Processing-Oriented Dataflow Machine", REVIEW of the Electrical Communication Laboratories, Vol. 32, No. 5, 1984.

[14] S.Umeyama and K.Tamura, "A Parallel Execution Model of Logic Programs", Proceedings of the 10th Annual Symposium on Computer Architecture,1983.

[15] Zahrin Halim, "A Data-Driven Machine for OR-Parallel Evaluation of Logic Programs", New Generation Computing 4(1986).

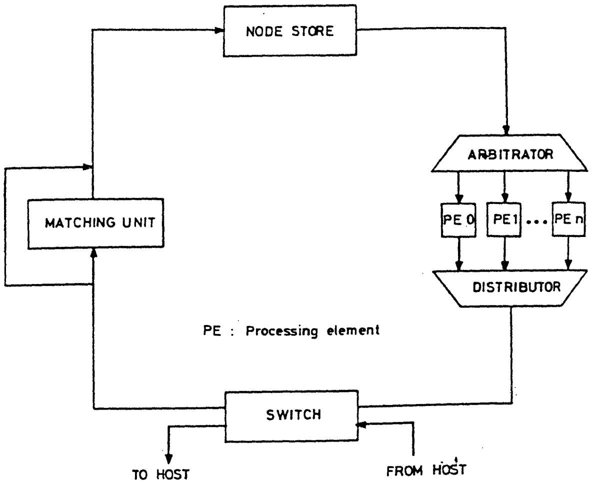


FIG.1 THE MANCHESTER RING

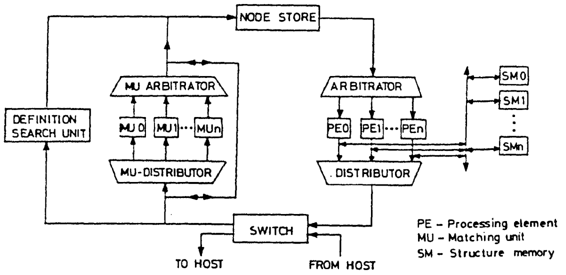


FIG. 2(a) DATAFLOW ARCHITECTURE TO EXECUTE LOGIC PROGRAMS

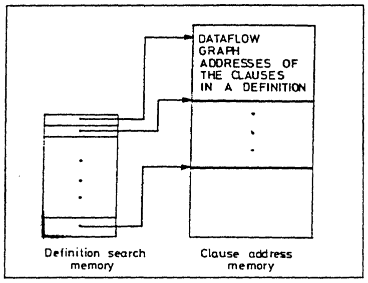
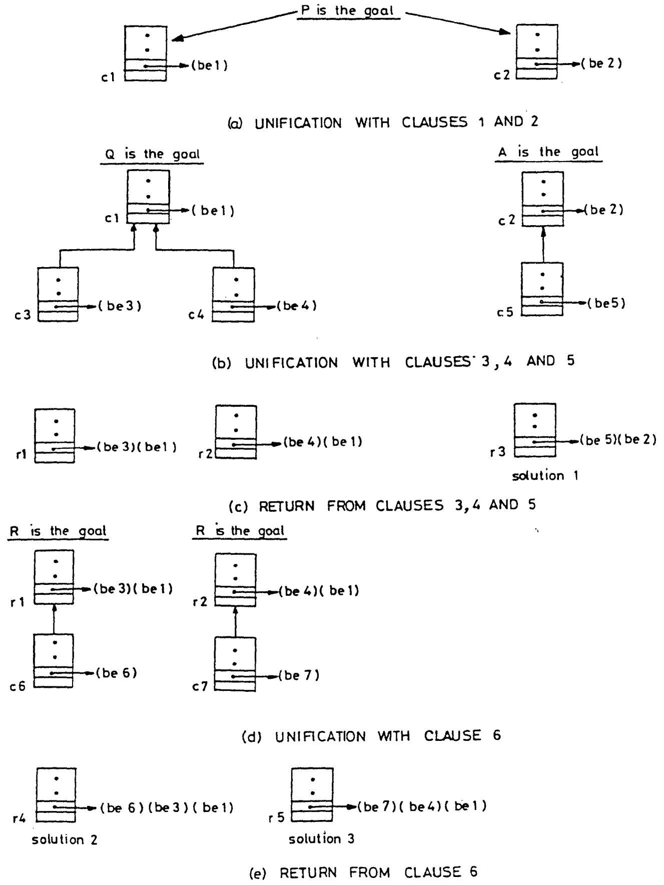


FIG. 2(b) DEFINITION SEARCH UNIT



(a) UNIFICATION WITH CLAUSES 1 AND 2

(b) UNIFICATION WITH CLAUSES 3, 4 AND 5

(c) RETURN FROM CLAUSES 3,4 AND 5

(d) UNIFICATION WITH CLAUSE 6

(e) RETURN FROM CLAUSE 6

FIG.3 MULTIPLE BINDING ENVIRONMENTS IN OR-PARALLEL EXECUTION OF LOGIC PROGRAMS

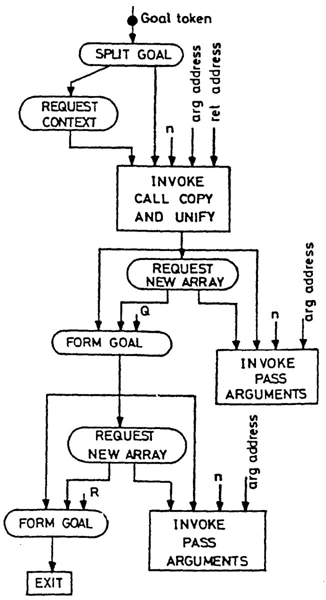


FIG.4(a) DATAFLOW GRAPH FOR A CLAUSE P(X,Y) :— Q(X, Z), R(Z,Y)

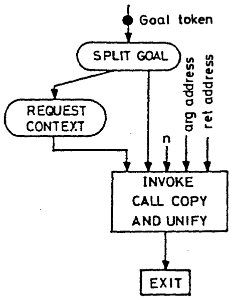


FIG.4(b) DATAFLOW GRAPH FOR A CLAUSE P(X,Y)

n : Number of arguments

arg address : pointer to argument array

ret address : return address

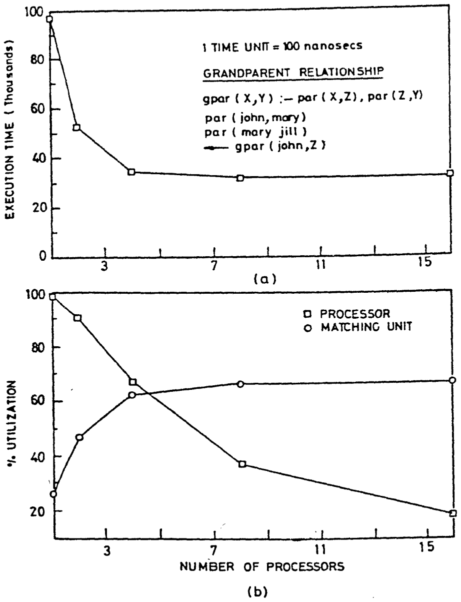


FIG 5 (a) EXECUTION TIME vs. NUMBER OF PROCESSORS
(b) UTILIZATION vs. NUMBER OF PROCESSORS

Storage Schemes for Efficient Computation of a Radix 2 FFT in a Machine with Parallel Memories

D. T. Harper III and D. A. Linebarger

Erik Jonsson School of Engineering and Computer Science
The University of Texas at Dallas
Richardson, Texas 75083-0688
(214) 690-2974

Abstract

Efficient bit-reversed access of vectors is an important consideration in designing architectures for use in signal processing applications. In particular, this type of access occurs in radix 2 FFT algorithms.

In this paper two skewing schemes which permit efficient bit-reversed access are discussed and compared in the context of a simple computer architecture such as might be designed around a low-cost, high performance specialized DSP chip. Performance measurements are shown for each scheme and simple address generation hardware is presented.

Introduction

Efficient algorithms for computation of the discrete Fourier transform have been studied intensely over the past 25 years. Most of these investigations centered around attempts to minimize the arithmetic complexity of such algorithms. The motivation for this direction was the fact that the time consuming operations in the algorithms were the arithmetic ones. However, improvements in VLSI processing and architectural optimization now permit extremely fast arithmetic operations to be performed. Current commercial multipliers are capable of sub 50ns operation on 32-bit data and all indications are that that figure will continue to fall. One effect of these advances is that the processing bottleneck has moved out of the arithmetic unit and into the memory. The limiting factor is now the rate at which data can be transferred to the arithmetic unit.

There are several approaches to alleviating this bottleneck. The first approach is to use faster memory. The same technological advances that achieved fast arithmetic circuits also achieved fast memory circuits. The disadvantage of this approach is that for large amounts of memory the cost becomes prohibitive. Other concerns are the problems of dissipating the large quantities of heat generated by the fast memory devices and the additional space required by the lower bit densities of the fast memories.

The second approach is to use a cache to achieve an apparent memory cycle time which is lower than the cycle time of the main memory. Disadvantages of this approach are the added complexity of hardware to implement the cache and the added complexity of the software or hardware to manage the cache.

A third approach, and the one pursued here, is to use parallel banks of slower memory, each of which can operate independently, so that overlap between multiple banks, or modules, creates an effective memory cycle time which is low enough to support the data rates required by the arithmetic unit. This technique is particularly applicable to systems which perform vector operations. The disadvantage of using parallel memory architectures is that severe performance degradation results if references are directed to modules which are busy processing prior references. This event is known as a *memory collision.* The rate at which collisions occur is determined by three factors: which data items are being referenced, the temporal order in which these items are accessed, and how the items are distributed over the parallel modules. Since the first and second factors are usually determined by the particular problem being solved it is useful to focus on the third factor.

The issues of determining a storage policy to allow efficient access to vectors using strides common to matrix operations and of how much performance degradation occurs when collisions do occur have been considered by several authors [1,2,3,4,5,6] with most conflict-free systems being based on a prime number of modules. Harper and Jump [7] have also considered the performance implications of using a composite number of modules. Melton and Norton [8] have proposed a storage scheme to solve the problem of vector accesses with strides equal to powers of 2 ($S = 2^s$) for memory systems with a power of 2 number of modules ($N = 2^n$).

In this paper accessing patterns common to a typical Cooley-Tukey radix 2 FFT algorithm are considered. The Cooley-Tukey radix 2 FFT operates on a vector with a length equal to a power of 2 and accesses the input data with strides equal to powers of 2. The performance of the proposed storage schemes with the FFT access patterns is discussed in the context of a simple architecture which does not have an expensive parallel interconnection network such as a multistage network or a crossbar.

Architecture

In the architectural model considered in this paper, computation is performed by a processor which is assumed to be capable of processing data at the bus rate. Equivalently, the system bottleneck is assumed to be caused by access conflicts in the memory. This is not an unreasonable assumption given the speed of current hardware and the prevalence of pipelining in execution units. This architecture differs sharply from the system considered in [8] which employed a highly parallel interconnection network.

One important feature required of the processor is a decoupling of the data fetch and execute cycles. This allows the fetch hardware to generate a stream of addresses to the memory independently of the operation of the execution unit. After some delay, the data referenced by the address stream is returned to the execute unit of the processor. These two tasks operate asynchronously with each other. The data-independent nature of vector accesses in the FFT algorithm makes this decoupling advantageous. Each memory module is assumed to have a bus interface register so that a reference transmitted to a memory does

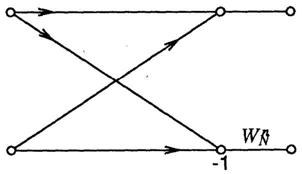


Figure 1: Butterfly

not require the use of the bus for the entire memory cycle (subsequent transmissions can be overlapped with the memory cycle).

FFT Algorithms

The fast Fourier transform (FFT) is a computationally efficient way of computing the discrete Fourier transform - an operation that is often performed in signal-processing applications. The focus of this paper is on implementation of the standard radix 2 Cooley-Tukey FFT [9, 10] on a machine with parallel memories. Although FFT's have been developed that are faster than the radix 2 FFT, the radix 2 FFT is still widely used. It is assumed that the input sequence is of length $L = 2^l$.

The fundamental operation in any FFT is known as a *butterfly*. A radix 2 butterfly has two inputs and two outputs. The butterfly computation consists of one complex multiplication and two complex additions as shown in Figure 1. Each node represents a complex addition with any indicated negations. W_L^n is notation often used with FFT's and is a symbol for $exp(-j2\pi n/L)$. W_L^n is a multiplicative constant for the branch it appears by. Each stage of a radix 2 FFT consists of $L/2$ butterfly computations with the entire computation requiring $\log_2 L$ stages. The entire FFT is a sequence of butterfly computations. Each butterfly has two inputs and two outputs, but for the standard implementation of the Cooley-Tukey radix 2 FFT, the separation between the two inputs (and the two outputs) varies from one stage to the next. Hence, the accessing pattern is variable.

The most commonly used algorithms for the radix 2 FFT are in-place. The in-place algorithms have their input vectors in order and their output vector is produced in a scrambled order, or vice-versa. Figure 2a shows an algorithm where the input is in order. There are two important observations to be made concerning Figure 2a:

(1) The output points for each butterfly are adjacent to its input points. This implies that the implementation can be calculated in-place.

(2) The separation between the input (and output) points for each butterfly decreases in each successive stage of the FFT. This implies that the accessing pattern is different from one stage of the FFT to the next.

The second characteristic of the in-place implementation of the radix 2 FFT makes it difficult to develop an efficient algorithm for storing the input data across multiple memories.

The nodes at a given stage of the FFT can be reordered without changing the function of the FFT as long as none of the connections are changed and the multiplicative constants move with the original connection with which they are associated. Thus a rearrangement of the node ordering for the flow graph in Figure 2a can be considered which might be more efficient for an architecture with parallel memories. The implementation illustrated in Figure 2b is known as a *constant geometry* algorithm since all stages of the FFT have the same connection pattern. The same computations are performed by the FFT's illustrated in Figures 2a and 2b; only the order of computation has changed. The input and output vectors of the constant geometry FFT are in the same order as those for the in-place FFT only the intermediate nodes have been reordered. However, it should be noted that the constant geometry implementation of the FFT cannot be calculated in-place.

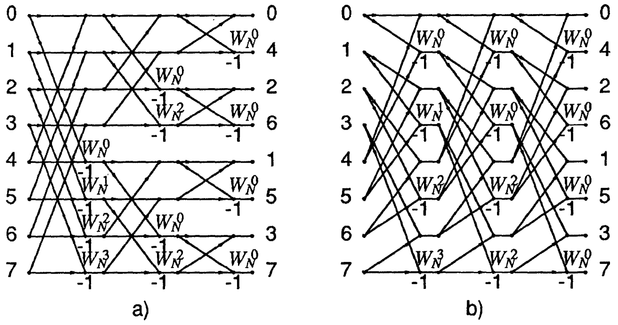


Figure 2: FFT flow graphs

Since each stage of the constant geometry FFT is identical the number of accessing patterns to be considered is reduced. The butterfly inputs and outputs are accessed in pairs; on the output the pairs consist of consecutive, adjacent pairs (stride 1). The input pairs consist of elements separated by half the sequence length, L. At the last stage, the output appears in scrambled order.

Storage Schemes

Vector accesses in the constant geometry FFT algorithm [9] consist of three different patterns. One is the stride 1 access which is easily handled by interleaving the addresses across the modules. The second is consecutive pairs separated by half the sequence length. The third access pattern is the pattern referred to in the previous section as "scrambled". The pattern is not truly random; it is a "bit-reversed" pattern. In this pattern the sequence of elements required is equivalent to the binary numbers formed by reversing the order of the bits of a binary counter. The most significant bit of the counter becomes the least significant bit of the address, the least significant bit of the counter becomes the most significant bit of the address, *etc.* Figure 3 shows the order of elements required by a bit-reversed access of a length 16 vector. Unlike constant stride accessing patterns, bit-reversed accesses are dependent on the length of the vector being accessed. In-place FFT algorithms require bit-reverse access for vector lengths of all powers of 2 less than or equal to L. The constant geometry implementation only requires bit-reverse access for length L and only performs this type of access after the final stage of computation is completed.

Stride 1	0000	0001	0010	0011	0100	...
	(0)	(1)	(2)	(3)	(4)	
Bit-Reversed	0000	1000	0100	1100	0010	...
	(0)	(8)	(4)	(12)	(2)	

Figure 3: Stride 1 and Bit-Reversed Access Patterns

It has been recognized by several authors [11] that generating bit-reversed sequences of addresses under software control is prohibitively expensive. To reduce the penalty for these accesses it has been proposed that hardware support for bit-reversed address generation be added to the memory system [12] . While this improves performance by removing the address generating task from the software, only part of the problem is solved. The problem of memory contention in bit-reversed accesses has not been considered.

To address the problem of memory collisions in bit-reversed accesses the vector storage scheme of the system must be considered. It has been noted that interleaved schemes based on low-order interleaving perform well only when the access stride is relatively prime to the number of memory modules. Referring to the bit-reversed sequence in figure 3, it seems as though there is no fixed stride involved. However, the sequence can be viewed as the concatenation of $L/2$ length 2 vector accesses each of which has a stride of $L/2$ but differs in their starting address. Thus, there are two accessing patterns involving pairs separated by $L/2$, the difference being the order that the pairs are accessed. Since the number of modules, $N = 2^n$, is a power of 2, and since the length of vectors in FFT algorithms is often also a power of 2, $L = 2^l$, system performance can be degraded due to the effects of memory collisions when low-order interleaving is used.

A more desirable storage scheme therefore must provide for both stride 1 and stride $L/2$ accesses. If L is restricted to be a power of 2 then Norton and Melton have proposed such a scheme based on a set of boolean transformations. For a system with $N = 4$ and $L = 16$ then their storage scheme maps the elements of the vector into the modules as shown in Figure 4.

M_0	M_1	M_2	M_3
0	3	2	1
5	6	7	4
10	9	8	11
15	12	13	14

Figure 4: Norton/Melton Storage Scheme

Under this scheme all power of 2 stride accesses can be made in a conflict-free manner. Although they do not explicitly state the ability of their scheme to provide conflict-free access to bit-reversed patterns it is clear that the capability is present. These statements are true as long as the architecture has a parallel interconnection network between the processor and the memory so that an address can be delivered to each module simultaneously. In the architecture considered here that is not the situation. Since only a single address is delivered to the memory in each time period contention can occur due to references to the same module in a sliding window of N references rather than in a fixed window of N references. Figure 5 uses the sequence of module addresses generated for a stride 1 access in a 4 memory system with the Norton/Melton storage policy to show collisions due to the lack of a parallel interconnection network. Unfortunately, the scheme proposed by Norton and Melton produces multiple conflicts when used with a sequential network. An alternative scheme for storing vectors accessed in bit-reversed order can be constructed as follows. Note that this scheme does not eliminate conflicts but serves to reduce the frequency of their occurrence compared to the scheme of Norton and Melton.

Begin by dividing the vector into N parts each of length L/N. Elements $0, 1, ..., L/N - 1$ are placed in module 0. Elements $L/N, L/N + 1, ..., 2L/N - 1$ are placed in module 1, *etc.* This storage scheme, shown in Figure 6a, permits conflict-free access for consecutive $L/2$ pairs and reduced conflict access for a bit-reversed pattern but does not permit efficient stride 1 access. To obtain the stride 1 access it is sufficient to skew each row by 1 module from the preceding row. The resulting patterns are shown in Figure 6b.

Performance Comparison

To evaluate the relative performance of the two schemes several simulation experiments were performed. Using a discrete-event simulation package a model of the architecture was developed. In the performance measurements it was assumed that the bus cycle time, t_b, was matched to the memory cycle time, t_b: $t_m = N \cdot t_b$. Also, timing was normalized to the bus cycle time ($t_b = 1$). The input to the simulations was the sequence of module addresses required to fetch the particular vector elements. The simulation measures memory system throughput, TP, as a function of the sequence of module addresses. Figure 7 compares the performance of the scheme proposed by Norton and Melton with the scheme proposed here. The solid lines indicate performance for stride 1 accesses. The dotted line measures throughput for a bit-reversed access under the proposed scheme. Bit-reversed access performance under the RP3 scheme perform identically to stride 1 accesses. The graphs indicated by the dashed lines represent performance on stride $L/2$ accesses. It is clear that for the three access patterns and the architecture considered here that the proposed scheme leads to better memory performance. It should also be noted that the proposed scheme requires $L \geq N^2$ to distribute vector elements correctly. This is not viewed as a strong constraint since in architectures similar to the one discussed here N is typically on the order of 4 to 16. Larger values of N are not required since the bus quickly becomes the bottleneck of the system. These values of N do not require a particularly large value of L.

Parallel Network: conflict occurs if a single module is referenced
more than once in each group of 4 references.

0 3 2 1 | 3 0 1 2 | 2 1 0 3 | 1 2 3 0 : no conflicts

Serial Network: conflict occurs if successive references to a
module are separated by fewer than 3 references.

0 3 2 1 3 0 1 2 2 1 0 3 1 2 3 0 : several conflicts
↑ ↑

Figure 5: Sequence of Module References To Demonstrate the Effects of the Interconnection Network - $N = 4$

M_0	M_1	M_2	M_3
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Figure 6a

M_0	M_1	M_2	M_3
0	4	8	12
13	1	5	9
10	14	2	6
7	11	15	3

Figure 6b

Figure 6: Proposed Storage Scheme

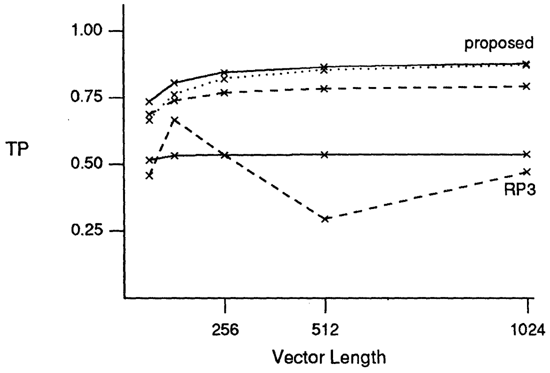


Figure 7: Performance Comparison

Other simulations have been performed by varying the value of N used. Results of these simulations are similar to those presented in Figure 7.

Address Generation

For any storage scheme to be practical the question of address generation must also be considered. Norton and Melton devised an elegant method for implementing their address mapping. In this section a simple method is demonstrated which generates the address of the k^{th} element of a vector access (either stride 1 or bit-reversed). The hardware required to perform the mapping of k into the address of the appropriate vector element is shown to be inexpensive. In the following discussion the length of the vector is given by $L = 2^l$ and the number of memory modules is given by $N = 2^n$.

Equations (1.r) and (1.m) specify the row and module address of the k^{th} element referenced during a stride 1 access. The module address is the number of the module being referenced; the row address is the location of the reference within the module.

$$r_1(k) = k \ mod \ \frac{L}{N} , \tag{1.r}$$

$$m_1(k) = \left[k + \left\lfloor k \frac{N}{L} \right\rfloor \right] \ mod \ N \tag{1.m}$$

Let bits of a value be numbered from 0 at the least significant position and let $x_{i:j}$ represent bits i through j of x. By using the fact that L and N are both powers of 2 equation (1.m) can be rewritten as:

$$m_1(k) = \left[k_{0:n-1} + k_{l-n:l-1} \right] \ mod \ N$$

Equations (2.r) and (2.m) specify the addresses for the bit-reversed access pattern. $BR(x)$ indicates the value of x after reversing its bits.

$$r_{br}(k) = BR(k_{n:l-1}) , \tag{2.r}$$

$$m_{br}(k) = \left[BR(k_{0:n}) + BR(k_{n:l-1}) \right] \ mod \ N \tag{2.m}$$

Circuits which compute the values of $m_1(k)$, $r_1(k)$, $m_{br}(k)$, and $r_{br}(k)$, are shown in Figure 8. Blocks labeled reverse perform bit-reversal on their inputs. This is achieved simply by mapping a permutation of the input bits to the output bits. The additional hardware required to implement the storage scheme consists of two n bit adders.

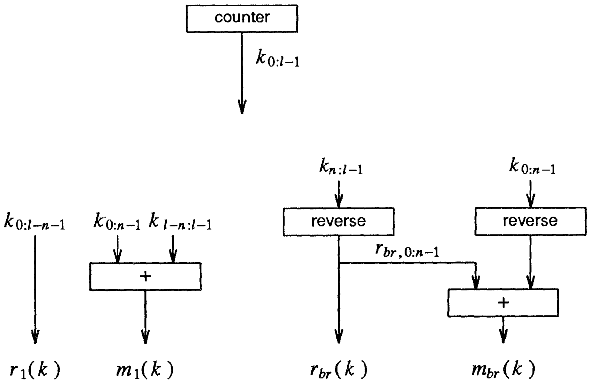


Figure 8: Address Generation Hardware

Conclusions

To summarize, advances in technology have permitted the fabrication of extremely fast arithmetic units. This has served to move the bottleneck of computationally intensive problems, such as the FFT, from the ALU to the memory. One solution to this problem is to provide parallel memories so that overlap can occur between successive references if the references are directed to different modules.

For FFT algorithms designed for an architecture with a parallel memory system the constant geometry version of the radix 2 FFT will better utilize memory bandwidth. The proposed storage scheme allows for fast memory access in the patterns required for the constant geometry radix 2 FFT. This may be of particular importance in real-time applications.

The analysis was based on an architecture with a low-cost, serial interconnection network. Under these constraints it was shown that the proposed storage scheme provided better memory performance than the scheme proposed by Norton and Melton. The problem of generating addresses was also considered and it was shown that a simple circuit based on two adders is capable of generating the proper address sequence.

References

1. P. Budnik and D.J. Kuck, "The Organization and Use of Parallel Memories," *IEEE Trans. Comp.* **C-20**(12) pp. 1566-1569 (1971).
2. D.H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Comp.* **C-24**(12) pp. 1145-1155 (1975).
3. D.H. Lawrie and C.R. Vora, "The Prime Memory System for Array Access," *IEEE Tran. Comp.* **C-31**(5) pp. 435-442 (1982).
4. H.D. Shapiro, "Theoretical Limitations on the Efficient Use of Parallel Memories," *IEEE Trans. Comp.* **C-27**(5) pp. 421-428 (1978).
5. H.A.G. Wijshoff and J. van Leeuwen, "The Structure of Periodic Storage Schemes for Parallel Memories," *IEEE Trans. Comp.* **C-34**(6) pp. 501-505 (June 1985).
6. Wilfried Oed and Otto Lange, "On the Effective Bandwidth of Interleaved Memories in Vector Processing Systems," *IEEE Trans. Comp.* **C-34**(10) pp. 949-957 (Oct. 1985).
7. D. T. Harper III and J. R. Jump, "Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme," *IEEE Trans. Comp.* **C-36**(12) pp. 1440-1449 (Dec. 1987).
8. A. Norton and E. Melton, "A Class of Boolean Linear Transformations for Conflict-Free Power-of-Two Stride Access," *Proc. 1987 Int. Conf. on Parallel Processing*, pp. 247-254 (1987).
9. Alan V. Oppenheim and Ronald W. Schafer, *Digital Signal Processing*, Prentice-Hall (1975).
10. C.S. Burrus and T.W. Parks, *DFT/FFT and Convolution Algorithms*, Wiley-Interscience (1985).
11. E. O. Nwachukwu, "Address Generation in an Array Processor," *IEEE Trans. Comp.* **C-34** pp. 170-173 (Feb. 1985).
12. P. T. Hulina and L. D. Coraor, "A Hardware Memory Mapping Unit for Efficient Address Computation," *Proc. 1987 Int. Conf. on Parallel Processing*, pp. 340-343 (1987).

Distributed Instruction Set Computer

Lingtao Wang and Chuan-lin Wu
Department of Electrical and Computer Engineering
The University of Texas at Austin,
Austin, Texas 78712.

Abstract -- Distributed Instruction Set Computer (**DISA**) is a multiple functional-unit computer system. It employs a new architecture, Distributed Instruction Set Architecture (**DISA**) to explore the execution parallelism at the instruction level. DISA expands data flow concept to combine the operation and execution information together. The execution information, such as data dependence, is detected with a post-compiler and attached to the opcode. DISA instructions are self-contained execution units which can be executed independently with one another in multiple functional units. This alleviates the performance bottleneck in a conventional multiple functional-unit system which uses a large associate memory table to decode instructions. DISC and DISA together have demonstrated a new and efficient way to incorporate data flow concept into von Neumann computer architecture.

Introduction

The current thinking in speeding up the instruction execution in a multiple functional-unit system is to apply concurrent instruction issuing" [1], "out-of-order execution" [2] or "branch prediction" techniques[3]. Each techniques has run into some difficulties when it is implemented in a von Neumann type instruction set. For instance, we need a large table to support a high degree of instruction issuing. The data tag search in a large table has contributed to a lengthy clock cycle. For the out-of-order execution, the system needs to repair the side effect of the execution whenever it encounters an exception[4]. This repair work becomes an overhead which slows down the processing speed. Our investigation into these techniques has shown the problem is not caused by the techniques but by the von Neumann type instruction set architecture. A von Neumann type instruction set architecture is designed for a sequential execution in a uni-processor system. It separates the operation information from the execution information. The operation information, such as opcode and operand, is given in the source code. The machine decodes the instruction to determine the execution information, such as data dependencies in the run time to execute the instruction. It is this run-time execution information detection that puts a tremendous burden on hardware design. It becomes a performance bottleneck when we intend to execute multiple instructions at the same time. A direct solution to this problem is to give up the von Neumann type instruction set architecture for a multiple functional-unit machine.

In this paper, we introduce the concept of Distributed Instruction Set Architecture(DISA) to solve this problem. We first describe the concept of DISA. We then develop a generic hardware system based on DISA. The system which we call Distributed Instruction Set Computer(DISC) is modeled with software to study its performance. We report our initial evaluation of DISC by running several benchmark programs on the model.

Distributed Instruction Set Architecture

There are three steps to issue an instruction : 1) fetch the instruction; 2) detect status of the execution unit; 3)decode and issue the instruction for execution. The concept of DISA is to speed up this procedure by eliminating the first two steps in a multiple functional-unit environment. We use software techniques to detect data dependence among instructions during compiling time. A post-compiler is used to detect the data dependencies among the instructions. It converts data dependence in the instruction into a data tag which shows the number of cycles the data needs to become "mature" in this instruction.The data tag is then attached to the instruction. By doing this, an instruction which is fetched from the memory can be immediately issued to a functional unit for execution. It is the responsibility of each functional unit to check the data tag before it pursues execution. This is the same concept as a tagged-token data flow architecture[5]. However, we detect the data dependence with a software which eliminates a lengthy table search in the conventional data flow approach.

An example is given in Figure 1 to explain the idea. In Figure 1, R3 has a dependence between the first and the second instruction. In DISA, we assign R3 in the first instruction with a 0 tag, and R3 in the second instruction with a 1 tag. When the first two instructions are sent to two functional units for execution in the same cycle, the second functional unit finds R3 with a non-zero tag. It decreases the tag by one and waits for one cycle. In the next execution cycle, it re-checks tag and finds a zero tag, it then proceeds to complete the execution. For an instruction that has unknown status during compiling time, such as memory instruction and conditional dependable instruction, we assign a flag bit to the instruction. The flag indicates that a conditional bit is required to be checked before the execution. When a functional unit receives the instruction, it checks the conditional bit in addition to the data tag checking. It only executes the instructions when both are satisfied. An example is R7 in instruction 4 and b. Both depend on the outcome of instruction 3.

```
1.Add       R1,R2,R3      /*R1+R2->R3*/
2.Or        R3,R4,R5      /*R3 OR R4 ->R5*/
3.CBra,=    R5,0,#b       /* jump to #b if R5=0*/
4.Sub       R4,R6,R7      /* R4-R6->R7*/
.
b.Sub       R1,R2,R7      /*R1-R2->R7*/
```

Figure 1: Data dependence among instructions.

The control flow of DISA is shown in Figure 2. A DISA processing cycle consists of a transmission sub-cycle and an execution sub-cycle as shown in Figure2a. The instruction dispatcher pre-fetches instructions. It then sends n instructions to n functional units in the forward routing phase of the transmission sub-cycle. A free functional unit accepts a new instruction. It checks the data tag and conditional bit in the instruction during the checking phase of the execution cycle. If tag shows an executable status, the functional unit pursues the execution in the second phase. Otherwise, it decreases the tag by 1 and idles for one cycle. In the next cycle, this functional unit refuses any new instruction and repeats the checking on the same instruction until it finishes the execution. Meanwhile, the unaccepted instructions are automatically routed back to the instruction dispatcher during the second backward sorting phase of the transmission cycle. They will be re-tried in the next transmission.

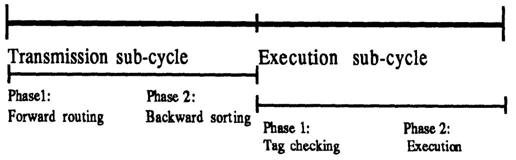
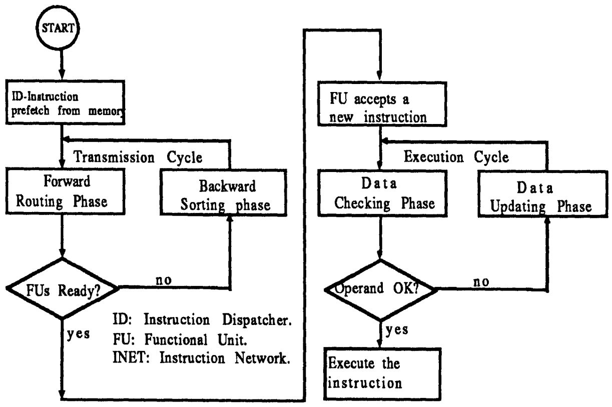


Figure 2a: DISA Processing Cycle



Fiugre 2b: DISA System Control Flow Diagram

Instruction Set Format

DISA is a register intensive instruction set architecture with load and store as the only memory access instructions. Each register is associated with a data tag which shows the dependencies with other instructions. The instruction format is affected by the structure of its targeted machine. In our research, we characterize a multiple functional-unit DISC system with four factors, [c,n,m,b]. c is the number of execution cycle per instruction, n is the number of functional units in the system, m indicates the number of memory ports and b is the level of branch prediction.

DISA has three instruction formats: one operand with long immediate, two operands with short immediate and three operands. Each operand in the instruction is assigned a tag, as shown in Figure 3. The tag(TAG) is an i-bit field. In a [1,n,m,b] system, the relation is i = log2(n). A dynamic tag(DTAG) field is defined to handle the dynamic flow information. Each bit in the DTAG corresponds to a conditional flag. It requires the functional unit to check the flag in addition to all the TAG fields. A good post-compiling detection algorithm is the key in this idea which is described in the following section.

OPCODE	OP1	TAG1	OP2	TAG2	OP3	TAG3	DTAG

Figure 3a: 3-operand tagged instruction format

OPCODE	OP1	TAG1	OP2	TAG2	IMMEDIATE	DTAG

Figure 3b: 2-operand tagged instruction format

OPCODE	OP1	TAG1	IMMEDIATE	DTAG

Figure 3c: 1-operand tagged instruction format

Post-compiling Algorithm

The post-compiler is a piece of software. It reads the assembly source code, decodes the instruction and generates the data-tag DISA instructions for a specific DISC [c,n,m,b] system. An "active window" algorithm is used in calculating the data tag. The size of an active window, w, is defined as the number of instructions which can be active at one instance in the system. The post-compiler checks one window at a time to calculate tags. In the worst case, it needs to check p-n+1 windows for a program with p instructions in a [1,n,m,b] DISC system. A general post-compiling algorithm is shown in the Algorithm 1. A detailed description is in another report[6].

Algorithm 1: Post-compiling scheme

1) Scan one instruction. If it is a load instruction, we adjust its memory ports. If it is a branch-target instruction and not the first instruction in a new window, we adjust the instruction sequence to make it as the first instruction and jump to step (4).
2) Calculate TAG and DTAG.
3) Repeat steps 1 to 2 to scan a new instruction until we fill the current window.
4) Calculate the instructions for next window.
5) Repeat steps 1 to 3 to finish a new window.
6) repeat step 1 to 5 to finish the program.

Distributed Instruction Set Computer

Based on DISA developed above, we propose a DISC system with 8 functional units. A [1,8,4,1] DISC system is shown in Figure 4. A processing cycle starts with the Instruction Dispatcher(ID) to fetch multiple instructions from the system instruction cache. ID cooperates the instruction issuing with the instruction network(INET) to send n instructions to n functional units(FU) every cycle. A free FU accepts a new instruction and checks the data tag. If the tag shows an operand is not ready at that moment, the FU updates the tag and holds the instruction for one cycle. Meanwhile, INET routes rejected instructions back to ID. In the next transmission cycle, the busied FU rejects a new coming instruction and checks the tag again. It repeats the same sequence until the operand is ready. It then fetches data from Register File(RF) through data network(DNET) and executes the instruction. If it is an ALU instruction, FU stores the result back into RF and becomes available to accept a new instruction in the next transmission cycle.

Instruction Dispatcher

ID is an interface logic between CPU and the instruction cache. It fetches multiple instructions from instruction cache and issues them to FUs through INET.

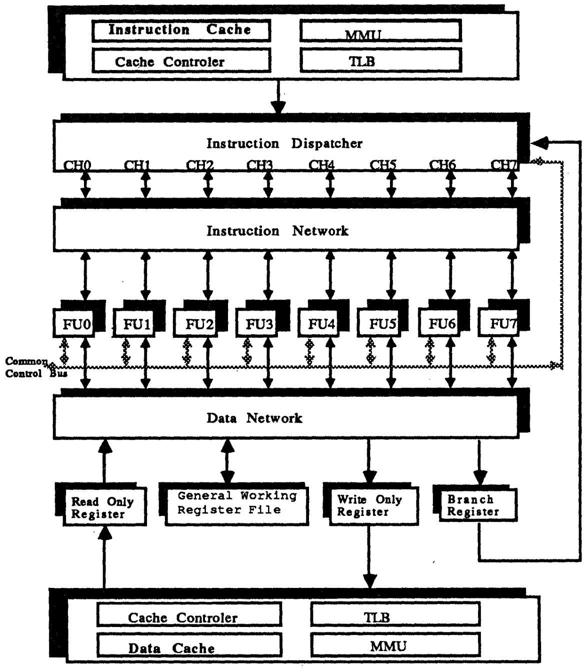


Figure 4: DISC[1,8,4,1] System Organization

Instruction Network

INET is a multistage n by n, circuit switching, synchronous control network. The INET with n=8 is shown in Figure 5. Each 2*2 switch element has a third control port to the elements above and below it. A built-in logic enables the upper output to have a higher priority than the lower one. A switch element always tries to send an input instruction to the upper output port if it is available. The element in the last row has an internal buffer to hold blocked instructions.

The INET provokes an automatic routing scheme that the network routes the instructions by itself to minimize the transmission overhead. It employs a three-step routing scheme. The first step is to detect the status of FU and set up the forward routing paths for each channel. A FU raises its input port low or high to indicate that it is able or unable to accept a new instruction in this cycle. The switch element next to it raises both input ports and control output high if it senses a "disable" FU. Then, each switch element detects the status of its previous elements and the element above it to set up its switch pattern. If both its outputs are high, it raises both input ports and control output high. It raises the lower input port and control output high, if only its control port is high. A high port indicates the path is blocked. An element with both its outputs high is unable to receive and transmit any instruction. In the second step, INET sends the instructions from ID to FUs by following the set-up physical paths. In the third step, INET routes un-accepted instructions which are blocked in INET back to channels of ID.

Functional Unit

A FU has a 32-bit ALU to provoke a three-stage pipelined execution. It fetches two data, executes the instruction and stores the data within one cycle.

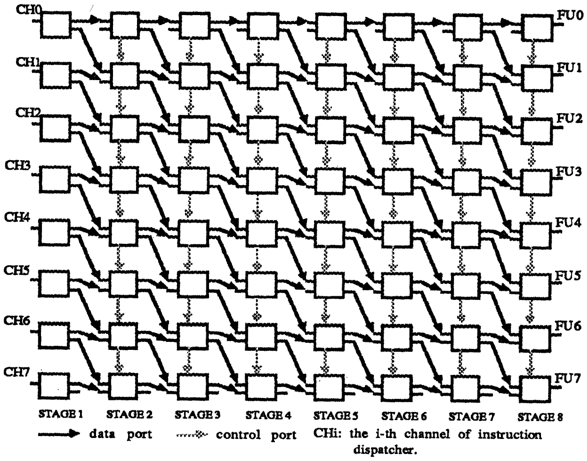


Figure 5: INET with n=8.

Data Network

DNET is a 32 by 8 cross-bar network. It transmits the data between RF and FUs. It supports one-to-one and many-to-one transaction for every execution cycle.

Common Control Bus

The common control bus(CCB) provides the inter-unit communication among FUs. It is used to support any running time information needed to execute the instructions, such as an exception or interrupt. An interrupt or exception is detected and reported to ID through CCB by a FU. When ID receives an exception report, it holds the instruction issuing and sends status report instruction(SRI) to each FU to ask a status report. A FU executes SRI to report its status, either complete or incomplete. When ID sees an "incomplete" status report, it needs to repair the system to a re-startable point before it issues the exception handler instructions. The overall flow chart is shown in Figure 6.

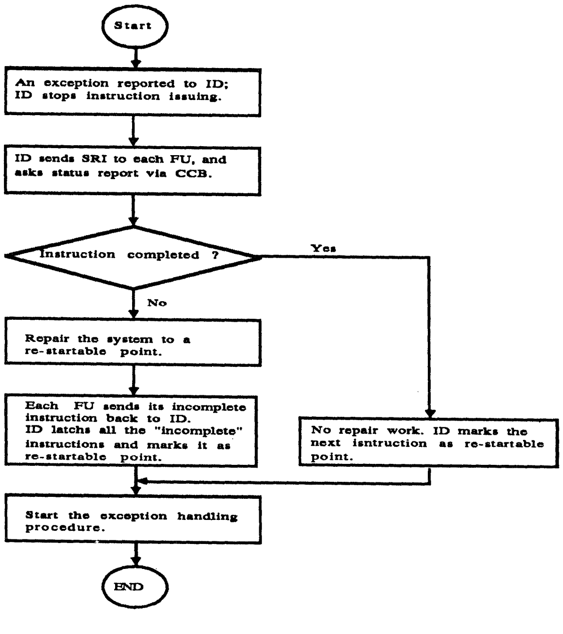

Start
An exception reported to ID; ID stops instruction issuing.
ID sends SRI to each FU, and asks status report via CCB.
Instruction completed ?
Yes
No
Repair the system to a re-startable point.
Each FU sends its incomplete instruction back to ID. ID latchs all the "incomplete" instructions and marks it as re-startable point.
No repair work. ID marks the next isntruction as re-startable point.
Start the exception handling procedure.
END

Figure 6. DISC Exception Processing Flow Chart.

Evaluation

Four programs are written in a generic DISC assembly language. The first program, Matrix, calculates matrix addition and subtraction on two linear arrays. Each array has 100 elements. The second program, Matrix-I, is the same as Matrix but it is optimized for DISC system. The optimization techniques are: 1) unfold the loop, 2) register renaming and 3) instruction re-ordering. The third program, Bubble, is a non-numerical one which uses bubble sort algorithm to sort 10 elements of a linear array in an ascending order. We only sort 10 elements for an easy tabulation with other programs. The fourth program, Salesman, is a traveling salesman problem. It finds the minimal distance for a salesman who visits 6 cities once and returns to the starting city.

Each program is traced in a DISC system with four configurations;[1,1,1,1], [1,2,1,1], [1,4,1,1] and [1,8,1,1]. Since we insert NOOP(no-operation) instructions into the program during the compiling time, Table 1a shows the size of program change in different system configuration. A [1,1,1,1] DISC system is equivalent to a conventional von Neumann type machine. Table 1b shows a gradually performance improvement when the number of FU increases. During the trace analysis, a non-memory instruction takes one cycle and a memory instruction takes two cycles. The impact of memory latency is neglected by assuming no cache miss or TLB miss. It will be studied in the future.

Table 1a: DISC programs size in number of instructions

no. of FUs / programs	1	2	4	8
Matrix	19	19	19	19
Matrix-I	32	32	32	32
Bubble	20	20	22	27
Salesman	55	60	78	123

Table 1b: DISC program execution time(cycles) and,

$$\text{Performance ratio} = \frac{\text{execution time of } n=1}{\text{execution time of } n=2,4,8}$$

no. of FUs / programs	1	2	4	8
Matrix	1905 1.00	1203 1.58	1102 1.73	1102 1.73
Matrix-I	1855 1.00	1153 1.61	1052 1.76	1052 1.76
Bubble	714 1.00	532 1.34	469 1.52	424 1.68
Salesman	2674 1.00	2073 1.29	1909 1.40	1845 1.45

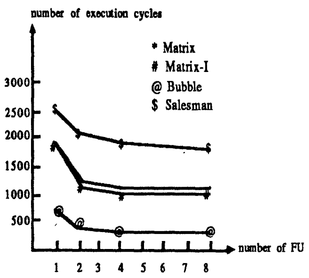


Figure 7a: DISC programs excution time

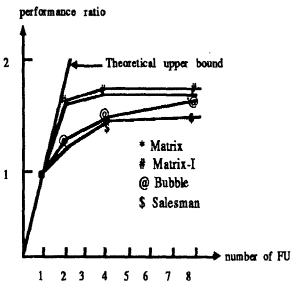


Figure 7b: DISC programs performance ratio

Figure 7a shows the total execution cycles for each four programs in a DISC system. Figure 7b shows the equivalent performance gain. The performance gain is not linearly proportional to the number of functional units in the system. An 80% improvement is gained for numerical programs and 60% for non-numerical programs. The numerical programs contain few control instructions and many computation instructions. They tend to have a better performance improvement than non-numerical programs. However, non-numerical programs show better performance improvement than numerical programs in a DISC[1,8,1,1] system. This is because non-numerical programs tend to have multiple branch instructions which can only be explored with a large number of functional units.

Conclusion

The concept of DISA is to apply the concept of data flow to a multiple functional-unit machine. It modifies von Neumann instruction set architecture to combine the operation and execution information together. Software techniques are applied to coordinate the data dependencies among the instructions. It minimizes the hardware complexity and system overhead in a distributed execution environment. These all together make DISA an ideal candidate for a distributed, parallel processing, multiple functional-unit machine. A generic system DISC has been proposed to study various aspects of DISA. The instruction streams and data streams are shown to be the most critical components in DISC. We invent INET which maintains a constant instruction stream into the multiple-FU engine. The regular cell and simple routing scheme have made INET very attractive in the real world.

Two software simulation models are under construction. One is used to study INET issuing mechanism, the other is used to investigate DISC system. Our future effort is to set up a [1,8,m,1] DISC system model. Then, benchmark programs will be written or collected to study the system performance on the model. When DISA concept is proven, we will concentrate on writing a DISC compiler and post-compiler to translate a program from C-language into DISC assembly language. At this early stage of research, we feel that we have invented a simple way to incorporate the data flow concept into von Neumann computer architecture. DISA enables us to speed up the program execution at the instruction set level in a very nice and efficient way.

Reference

[1] R.D.Acosta, J.Kjelstrup and H.C.Torng,"An instruction issuing approach to enhancing performance in multiple functional unit processors", IEEE Tran. Com., vol.c-35, no.9, pp.815-828, Sep. 1986.

[2] Y.N.Patt, W.M.Hwu and M.Shebanow,"HPS, A new microarchitecture: rationale and introduction", the Proc. of the 18th Microprogramming Workshop, pp.103-108, Dec. 1985.

[3] J.E.Smith,"A study of branch prediction strategies", 8th Int. Symp. on Com. Arch., pp.135-148, May 1981.

[4] W.W.Hwu and Y.N.Patt, " Checkpoint repair for high-performance out-of-order execution machines", IEEE Tran. on Com. v.C-36, n.12, pp.1496-1514, Dec. 1987.

[5] Arvind and R.S.Nikhil," Executing a program on the MIT tagged-token dataflow architecture ", the Proc. PARLE Conf., Eindhoven, The Netherlands, Jun. 1987.

[6] L. Wang, D.C.Zu and J.M.Chai," DISC Post-compiler structure and algorithm", University of Texas, ECE Department, Internal Tech. Rep. No. DISC-H-88-01.

An Improved Approximation Algorithm for Scheduling Pipelined Machines

David Bernstein

IBM Research
T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Consider a pipelined machine which can issue one instruction every machine cycle, but can use its result only $d+1$ machine cycles after it has been issued. Instruction scheduling is an important phase of the compilation process whose goal is to generate optimized code for such machines. Since the problem of producing optimal instruction schedules for pipelines for arbitrary expressions, with possibly common subexpressions, is NP-complete (except of a few restricted cases), we concentrate on approximation solutions. A class of scheduling algorithms, called *leveling* algorithms, is defined and analyzed. The basic leveling algorithm sometimes yields bad schedules such that the ratio of their length over the length of an optimal schedule can be made arbitrarily close to $2 - 1/(d+1)$ which is the upper bound of *list* schedules. We refine this algorithm to improve the worst case ratio to $2 - 2/(d+1)$. The time complexity of the refined leveling algorithm is $O(n\alpha(n) + e \log n)$ where n is the number of instructions, e is the number of dependences among the instructions, and $\alpha(n)$ is a very slow-growing function.

1. Introduction

Pipelining is a common technique for building fast processors. In contrast to parallel processing, in which computational jobs can be initiated simultaneously, only one instruction can be issued every machine cycle in a pipelined machine; several instructions may be executed concurrently, one in every stage of the pipe. In general, recently designed computer architectures ([HB84], [K81]) include both pipelining and parallelism. In this paper we concentrate on the effect of pipelining which mostly characterizes recently proliferating RISC machines [R83], [K84], [P85].

Pipelining may cause the insertion of NOPs (No OPerations) into the sequence of machine instructions either by hardware or software. In both cases a certain penalty is paid in increased execution time. Minimizing the number of NOPs increases the effective speed of the machine. It is a task of the compiler that produces code for a pipelined machine to schedule the instructions as to get rid of maximum number of NOPs. Previously, this problem was tackled both in production compilers by implementing different (heuristic) algorithms and in theoretical scheduling papers.

Li assumed identical delays of all the instructions in the pipeline [L77]. In this case, assuming that the input is limited to tree expressions, an optimal computation can be constructed by executing first the instructions furthest from the root of the tree. Directed acyclic graphs (dags) were considered by Bruno et. al. [BJS80]. They showed that if the delays of all the instructions are equal to one time unit, then Coffman-Graham's algorithm ([CG72]) can be used to produce an optimal solution. This result was generalized in [BG86] and [BRG87] where an optimal solution was given for dags for the case when the delays are either 1 or 0. However, if no bound is put on the maximal delay d then the problem of finding an optimal computation turns out to be NP-complete as was proved in [HG83] for dags. A recent survey on the complexity of scheduling for pipelined machines can be found in [LLM87]. Since it is unlikely to find a polynomial solution to the pipeline scheduling problem for arbitrary d, we turn to approximation algorithms and study their worst case behavior.

In [AH82], [HG83], and [GM86] different heuristic algorithms were implemented in production compilers, but for none of them worst case bounds are known, even though satisfactory results were reported on average. In [BRG87] it was proved that an upper bound for *list* schedules ([C76]) on pipelined machines is $2 - 1/(d+1)$. So, the question is how better than this upper bound we can do.

We propose an algorithm that follows the critical path approach by assigning a *level* to each instruction. Then, a computation is constructed in such a way that instructions at higher levels are computed first. This has been a natural heuristic approach to multiprocessor scheduling ([CL75], [LS77], [S76b]). Unfortunately, there are examples in which this basic leveling algorithm can perform on dags as badly as the upper bound of list schedules.

Then, we define a *refined leveling* algorithm that improves the worst case ratio of the length of the schedule produced by the algorithm over the length of an optimal schedule to $2 - 2/(d+1)$. Also, we mention a family of examples that approaches this worst case ratio arbitrarily closely. In [BRG87] the same worst case ratio of $2 - 2/(d+1)$ was proved for Coffman-Graham's algorithm, but on average the refined leveling algorithm is advantageous.

The time complexity of the refined leveling algorithm is $O(n\alpha(n) + e \log n)$ where n is the number of instructions, e is the number of dependences among instructions and $\alpha(n)$ is a very slow-growing function. However, the refined leveling algorithm (similarly to Coffman-Graham's algorithm) requires the given dag of dependences among instructions to be free of transitive edges. Usually, it can be assumed that the dag is given in that form, but if it is not true, the removal of transitive edges can dominate the time complexity of all the process since it takes time $O(\min(en, n^{2.61}))$ to do that [G82].

In the next section we start with some preliminary definitions. Then, in Section 3 the leveling algorithm is described, and we conclude with directions for future research.

2. Background

The scheduling model we consider consists of a single *processor* P and a *job system* $T = (J, D, G)$. T comprises a set of unit execution times *jobs* $J = \{J_1, \dots, J_n\}$, a set of *delays* $D = \{D_1, \dots, D_n\}$ where $D_i \in \{0, \dots, d\}$ for some fixed integer d, and a directed graph $G = (J,E)$ of *precedence constraints*. (The delays model the pipelined structure of P.) In this paper we limit ourselves to consider the case where for all i, $D_i = d$.

A *legal* schedule is defined as a one-to-one mapping S from the elements of J into the set N of positive integers (interpreted as *time slots*) such that for all $(J_i,J_j) \in E$, $S(J_j) - S(J_i) > D_i$. A time slot of S, in which no job can be executed because of delay limitations, is called a NOP.

We assume that G has no transitive edges since they do not impose additional restrictions on a schedule S of T. Also, the leveling algorithm which will be presented in Section 3 requires to distinct transitive and non-transitive edges of G.

For example, consider the job system of Figure 1(a). The jobs are represented by circles, and their indices appear inside the circles. Also, it is assumed that $d = 2$. Two legal schedules for the job system are shown in Figure 1(b), where i in column j means that J_i is executed in time slot j. Notice that time slots 7 and 11 of S^1 are NOPs since $D_5 = 2$ and $D_8 = 2$.

The *completion (maximum finishing) time* $c(S)$ of a schedule S is defined by $\max S(J_i)$. For example, in Figure 1(b), $c(S^1) = 14$ and $c(S^2)^i = 13$. In this work we will be interested in minimizing the completion time, which is equivalent to minimizing the number of NOPs. An *optimal schedule* S is a legal schedule for which $c(S)$ is smallest. It turns out that S^2 of Figure 1(b) is an optimal schedule for the job system of Figure 1(a).

3. Scheduling algorithms

3.1. List schedules

Let $T = (J, D, G)$ be a job system with n jobs. If $(J_i,J_j) \in E$ we say that J_j is an *immediate successor* of J_i, and J_i is an *immediate predecessor* of J_j. Given a schedule S for T, J_j is *ready in time slot k*, if each of its immediate predecessors J_i has been scheduled not later than time slot $k - 1 - D_i$.

Now we consider an important class of schedules, called *list schedules* ([C76]). Informally, given a *priority list* L of the jobs of J, the list schedule S that *corresponds* to L can be constructed by the following procedure:

1. Iteratively schedule the elements of S starting in time slot 1 such that during the i-th step, L is scanned from left to right, and the first ready job not yet scheduled is chosen to be executed in time slot i.
2. If no such job is found, a NOP is inserted into S in time slot i.

Consider a class of optimal schedules for T. Since all the jobs in T have unit execution times, there is no reason in optimal schedules to leave the processor P idle whenever a ready job exists. Therefore, for our problem, an optimal schedule can always be found among list schedules. The obvious question is how to obtain the right priority list L.

Analyzing a class of list schedules, we would like to know how far from the optimum an arbitrary list schedule can be. Let us denote optimal schedules by S_{opt} and arbitrary list schedules by S_{list}. The upper bound for list schedules was proved in [BRG87] to be as follows: $R = c(S_{list})/c(S_{opt}) \leq 2 - 1/(d + 1)$. In the next section, an algorithm that improves on this upper bound is presented.

3.2. Leveled schedules

In this section, a subclass of list schedules, called *leveled schedules* is considered. If $J_i \in J$ has no immediate successors, we say that J_i is a *sink* of G. Also, let $IS(J_i)$ (or IS_i for short) be the set of immediate successors of J_i.

First, the original leveling algorithm that was introduced at first in [BRG87] is described. The *level* $l(J_i)$ of a job J_i is defined as follows:

$$l(J_i) = \begin{cases} 0 & J_i \text{ is a sink of } G \\ D_i + \max\limits_{X \in IS_i} l(X) & \text{otherwise} \end{cases}$$

Notice that the total execution time of the jobs does not affect the levels as defined above.

Let L be a priority list of the jobs in J constructed in a non-increasing order of their levels (the order among the jobs of the same level is arbitrary). A schedule S corresponding to such an L is called a *leveled schedule.* Intuitively, in leveled schedules we first schedule jobs whose delays are maximal, hoping that the NOPs induced by these jobs will be replaced by other jobs.

However, it turns out that the leveling algorithm defined above is not successful enough. For example, in Figure 1(a), for $9 \leq i \leq 12$, $l(J_i) = 0$, for $6 \leq i \leq 8$, $l(J_i) = 2$ and for $1 \leq i \leq 5$, $l(J_i) = 4$. This may lead to a priority list 1,2, ...,12 that results in a non-optimal schedule S^1 of Figure 1(b). In general, it was shown in [BRG87] that in the worst case the leveling algorithm described above does not improve on the upper bound of list schedules.

In the sequel, a *refined leveling* algorithm (or RL for short) that improves on the upper bound of list schedules is defined. Let the *refined level* of J_i be denoted by $rl(J_i)$ and let $M_i = rl(J_{i_1}), ..., rl(J_{i_{|IS_i|}})$ be a sequence on non-negative integers constructed from the refined levels of the immediate successors of J_i ordered in a way that $rl(J_{i_1}) \geq ... \geq rl(J_{i_{|IS_i|}})$. Then, $rl(J_i)$ is defined recursively as follows:

1. If J_i is a sink of G then $rl(J_i) = 0$.
2. Otherwise, $rl(J_i) = D_i + \max(rl(J_{i_1}), rl(J_{i_2}) + 1, \ldots, rl(J_{i_{|IS_i|}}) + |IS_i| - 1)$.

Apparently, the number of the immediate successors of a job and their refined levels are taken into consideration while computing the refined level of a job. A priority list L produced by RL is computed as follows:

1. Compute the levels $l(J_i)$ for all i.
2. Compute the refined levels $rl(J_i)$ for all i.
3. Create a priority list L by first ordering J_i in a non-increasing order of l, and then ordering the jobs with the same value of l in a non-increasing order of rl. The order among the jobs with the same values of l and rl is arbitrary.

Using the refined levels of the jobs to create a priority list as described above, results in somewhat less arbitrary decisions which are made for the jobs of the same level as compared to the original leveling criterion.

For example, consider the job system of Figure 1(a), and let us demonstrate how the refined levels are computed. First, for $9 \leq i \leq 12$, $rl(J_i) = 0$. Then, $M(J_6) = M(J_7) = M(J_8) = \{0,0,0\}$. Therefore, $rl(J_6) = rl(J_7) = rl(J_8) = 4$. Then we get $M(J_1) = M(J_2) = M(J_3) = \{4\}$. Therefore, $rl(J_1) = rl(J_2) = rl(J_3) = 6$. On the other hand, $M(J_4) = M(J_5) = \{4,4\}$. Therefore, $rl(J_4) = rl(J_5) = 7$. This leads to a priority list 4,5,1,2,3,6, ...,12 that results in an optimal schedule S^2 of Figure 1(b).

Because of lack of space we are not able to present the analysis of RL and only mention the two main results:

1. The worst case ratio of the length of the schedule S_{rl} produced by RL over the length of an optimal schedule S_{opt} is as follows: $R = c(S_{rl})/c(S_{opt}) \leq 2 - 2/(d + 1)$.
2. The time complexity of RL is $O(\alpha(n)n + e \log n)$ where n is the number of instructions, e is the number of dependences among the instructions, and $\alpha(n)$ is a functional inverse of Ackermann's function.

4. Conclusions

In this paper we presented a refined leveling algorithm to schedule instructions under pipelined constraints whose worst case ratio is $2 - 2/(d + 1)$. This result is proved for the case when all the delays are exactly d machine cycles. One of the relaxations of our pipelined model is to allow the delays to be any integer between 0 and d. The question is how to extend RL to this case in order to achieve the worst case ratio of $2 - 2/(d + 1)$.

Another direction for further research is to search for a scheduling algorithm which improves on RL. RL asymptotically achieves its worst case bound of $2 - 2/(d + 1)$ on a complex job system presented by Lam and Sethi in [LS77], Fig 10. One of the alternatives to improve RL is to construct a priority list L of jobs in the non-increasing order of their refined levels (without taking into consideration the basic levels l at all). This extended leveling algorithm can be shown to do better than $2 - 2/(d + 1)$ on all known families of worst case examples including that of [LS77]. Our conjecture is that the worst case bound of this algorithm is not 2 anymore when d increases, however, we are not able to proof this claim at a moment. We conclude by demonstrating in Figure 2 a job system that, for the best of our knowledge, is worst for the extended leveling algorithm we propose.

The job system T = (J, D, G) of Figure 2 consists of $k + 1$ groups of jobs. The group t, $0 \leq t \leq k$, consists of d type-A jobs $(A_{t1}, \ldots, A_{td})$ and d type-B jobs $(B_{t1}, \ldots, B_{td})$. The precedence constraints described in Figure 2 are such that for all t, $1 \leq t \leq k$, every type-A (type-B) job of group $t - 1$ is an immediate successor of every type-A (type-B) job of group t. Executing the jobs of group t in order $A_{t1}, \ldots, A_{td}, B_{t1}, \ldots, B_{td}$ results in an optimal schedule with no NOPs. Thus, $c(S_{opt}) = n = 2d(k + 1)$.

It turns out that by applying the extended leveling algorithm to T, we get that all the jobs of the same group have the same refined level. Thus, we might get a priority list L in which the jobs of group t, $0 \leq t \leq k$, appear in order $A_{t1}, \ldots, A_{td-1,}\ B_{t1}, \ldots, B_{td}, A_{td}$. By applying a list scheduling process to L, we get a schedule S that has $d - 1$ NOPs after each A_{td} job except of A_{0d}. Thus, $c(S) = n + k(d - 1)$ and

$R = c(S)/c(S_{opt}) = 1 + k(d-1)/2d(k+1)$. By increasing k, R can be made arbitrarily close to $3/2 - 1/2d$.

Acknowledgment. The author would like to thank M. Rodeh for many valuable discussions and observations.

References

[AH82] Auslander, M., and Hopkins, M., "An overview of the PL.8 compiler", *Proceedings of the ACM Symposium on Compiler Construction* (June 1982), 22-31.

[BG86] Bernstein, D., and Gertner, I., "Computing expressions on a pipelined processor with a small number of stages", EE PUB No. 594, Dept. of Elec. Eng., Technion, Haifa, Israel, (June 1986).

[BRG87] Bernstein, D., Rodeh, M., and Gertner, I., "Approximation algorithms for scheduling arithmetic expressions on pipelined machines", *TR-88.227*, IBM Haifa Scientific Center, (July 1987).

[BJS80] Bruno, J., Jones, J.W., and So, K., "Deterministic scheduling with pipelined processors", *IEEE Transactions on Computers*, C-29, 4 (Apr. 1980), 308-316.

[C76] Coffman, E.G., *Computer and job-shop scheduling theory*, John Wiley and Sons, New York, 1976.

[CG72] Coffman, E.G., and Graham, R.L., "Optimal scheduling for two-processor systems", *Acta Informatica*, 1 (1972), 200-213.

[CL75] Chen, N.F., and Liu, C.L., "On a class of scheduling algorithms for multiprocessors computing systems", *Lecture notes in computer science*, Vol. 24, Springer-Verlag, New York, 1975, 1-16.

[G82] Gabow, H.N., "An almost-linear algorithm for two-processor scheduling", *JACM* 29, 3 (July 1982), 766-780.

[GM86] Gibbons, P., and Muchnick, S., "Efficient instruction scheduling for a pipelined architecture", *Proceedings of the ACM Symposium on Compiler Construction* (June 1986), 11-16.

[HB84] Hwang, K., and Briggs, F.A., *Computer architecture and parallel processing*, McGraw-Hill, New York, 1984.

[HG83] Hennessy, J.L., and Gross, T.R., "Postpass code optimization of pipeline constraints", *ACM Transactions on Programming Languages and Systems*, 5, 3 (July 1983), 442-448.

[K84] Katevenis, G.H., *Reduced instruction set computer architecture for VLSI*, MIT Press, Cambridge, 1984.

[K81] Kogge, P.M., *The architecture of pipelined computers*, McGraw-Hill, New York, 1981.

[L77] Li, H.F., "Scheduling trees in parallel/pipelined processing environments", *IEEE Transactions on Computers*, C-26, 11 (Nov. 1977), 1101-1112.

[LS77] Lam, S., and Sethi, R., "Worst case analysis of two scheduling algorithms", *SIAM J. Computing*, Vol. 6, No. 3, (Sep. 1977), 518-536.

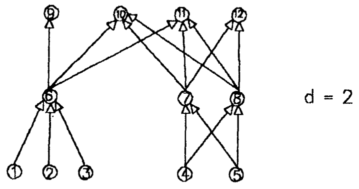
[LLM87] Lawler, E., Lenstra, J.K., Martel, C., Simons, B., and Stockmeyer, L., "The complexity of scheduling pipelined machines", *RJ-5738*, IBM Almaden Research Center, (July 1987).

[P85] Paterson, D.A., "Reduced instruction set computers", *Communication of the ACM* 28, 1 (Jan. 1985), 8-21.

[R83] Radin., G., "The 801 minicomputer", *IBM J. Res. Dev.* 27, 3 (May 1983), 237-246.

[S76a] Sethi, R., "Scheduling graphs on two processors", *SIAM J. Computing*, Vol. 5, No. 1, (Mar. 1976), 73-82.

[S76b] Sethi, R., "Algorithms for minimal length schedules", in Coffman, E.G., ed., *Computer and job-shop scheduling theory*, John Wiley and Sons, New York, 1976, 51-99.



(a)

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S^1	1	2	3	4	5	6		7	8	9		10	11	12
S^2	4	5	1	2	3	7	8	6		12	9	10	11	

(b)

Figure 1. A job system and two legal schedules

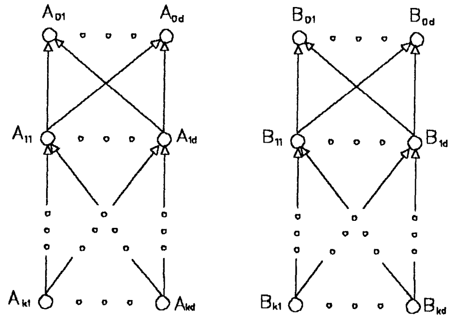


Figure 2. A worst case example for extended RL

The Processor Partitioning Problem In Special-Purpose Partitionable Systems

Ramesh Krishnamurti
School of Computing Science
Simon Fraser University
Burnaby, B.C. V5A 1S6
Canada
(604) 291-4116

Eva Ma
Department of Computer Science
University of Pennsylvania
Philadelphia, PA 19104
(215) 898-8549

Abstract:

We address the problem of processor partitioning in partitionable systems used for special-purpose applications. We demonstrate that the partition size for a task should depend on the task characteristics, the workload, and the availability of resources. Thus, to maximize throughput, the partition sizes for a set of tasks should be determined at run time. Such an approach could be supported in special-purpose applications since the set of tasks the system needs to support are usually known in advance. We first show that given a set of tasks and their characteristics, the problem of determining the optimal partition sizes for the set of tasks is NP-Complete. We then present a polynomial time approximation algorithm for this problem. We also derive a worst-case bound on the solution obtained by the algorithm as compared to the optimal solution.

Section 1: Introduction

Partitionable architectures, also called Multiple SIMD-/MIMD architectures or MSIMD/MIMD architectures, consist of a set of processors and controllers [NUT77, PRE80, SIE81]. Such architectures can be partitioned into independent subsystems, each comprising of a variable number of processors and a controller assigned to the execution of a task. Each of these subsystems may either be in the SIMD or MIMD mode of computation. In addition to the flexibility of supporting both SIMD and MIMD modes, the ability to form multiple independent subsystems to execute several tasks in parallel provides such a system with the potential of achieving better utilization of processing resource.

An important problem that needs to be addressed in the partitioning of these systems is one of determining the number of processors allocated to each subsystem, that is, *the partition size* for each task. One possible approach to determining this size is to first derive the maximum degree of parallelism available in the program, and then choose the partition size to be either this maximum degree of parallelism or the maximum number of processors in the system that may be allocated to the task, whichever is smaller. Such an approach to determine the number of processors allocated to a task has been used widely in conventional SIMD and MIMD systems, and much work in the areas of programming languages and compiler design has been done to support this approach [KUC77].

Such an approach, however, may not be optimal from the standpoint of either minimizing the execution time for a task or the completion time for a set of tasks (we define completion time to be the least time by which all tasks in the set have completed execution). Since most parallel programs require communication among processors during their execution, for many parallel programs, the communication between processors may play a dominant role in the overall execution time with increasing partition size. As a result, the improvement in execution time may level off as the partition size increases. In other words, there is an effect of diminishing return in performance with larger partition sizes. Furthermore, beyond a certain partition size, the execution time may actually increase. The optimal partition size for a task depends on the computation and communication structure of the program, the size and values of the input data, and the computation and communication support of the system [LIN81, NIC87, MA87, MA88]. This size could be smaller than both the maximum degree of parallelism in the program and the maximum number of processors that may be allocated to the task.

In a partitionable system, due to the effect of diminishing return in performance with larger partition sizes, when there is a multiple number of tasks ready for execution, using a smaller partition size for each task and executing as many tasks in parallel as possible could lead to a shorter completion time than using the partition size that gives the minimum execution time for each individual task. For example, in a simulation study of a histogramming algorithm [KUE84], Kuehn and Siegel have shown that given a set of four histogramming tasks of the same size ready for exe-

This work has been supported by National Science Foundation Grant DCR84-51408, National Science Foundation CER Grant MCS82-19196, AT&T Information System Research Grant, Army Research Office Grant DAAG-29-84-K-0061, and the President's Research Grant at Simon Fraser University.

cution and a partitionable system of 256 processors, using a subsystem of 64 processors for each task and executing the four tasks in parallel gives a shorter completion time than using a partition of 256 processors for each task and executing the four tasks sequentially.

Given a set of tasks which are ready for execution, the optimal partition sizes for these tasks depend on the number of tasks in the set, their characteristics, and the amount of available resource. Since the information on what tasks are ready for execution, which we refer to as *workload*, and what resources are available for allocation cannot be determined until run time, the optimal partition sizes can be determined only at run time, and not at program design time or at compile time. Furthermore, in order to determine at run time the optimal partition sizes for a set of tasks, it is necessary for the system to know the characteristics of each individual task in the workload. Such characteristics, however, could be difficult to obtain for a system designed for general purpose application because the tasks the system needs to support may vary widely. But for a system designed for a special-purpose application, the tasks the system needs to support are relatively fixed and known in advance. For example, for the application of image processing, such tasks include FFT, Histogramming, Convolution and Image Smoothing. It is thus possible to pre-analyze the characteristics of the tasks and make them available to the system. As a result, a partitionable system for special-purpose application can be designed with the ability to determine optimal partition sizes at run time. The feasibility and advantage of this approach is naturally determined by the overhead involved, which include the effort to pre-analyze the task characteristics, the storage required to record these characteristics in the system, and most importantly, the time it takes for the system to determine these partition sizes. In this paper, we focus on analyzing the time complexity of using such an approach to determining optimal partition sizes, assuming that the required information on task characteristics can be made available to the system. We show the problem of determining such optimal partition sizes to be NP-Complete; we also propose a polynomial time approximation algorithm for this problem and derive the performance bound for the algorithm.

In Section 2, we illustrate through a sequence of examples the impact of task characteristics and workload on optimal partition sizes. In Section 3, we formulate the processor partitioning problem, review the multiprocessor scheduling problems in the literature related to this problem, and establish the NP-completeness of this problem. In Section 4, we propose a polynomial time approximation algorithm for the partitioning problem and derive its performance bound. In Section 5, we apply the approximation algorithm on some examples to illustrate the possible reduction in completion time by using the partition sizes determined by the algorithm as opposed to using the partition sizes that minimize the execution time for each individual task.

Section 2: Impact of Task Characteristics and Workload on Optimal Partition Size

We illustrate the impact of task characteristics and workload on optimal partition sizes with some examples on the following model of a partitionable system. The system consists of 512 processors interconnected as a linear array. The links in the array are bidirectional. The system can be partitioned into several subsystems, each of which consists of a subset of consecutive processors in the linear array, operating in the SIMD mode. Furthermore, the time to communicate a data item between two adjacent processors equals the time to perform an arithmetic or logical operation over two data items. Note that the characteristics of the architecture, particularly those of the supporting interconnection network, have an important impact on task characteristics, and thus also affect the optimal partition sizes. More detailed analyses of the impact of the characteristics of tasks, workload, and system on optimal partition sizes are given in [MA88].

In the following examples, let N denote the number of data items for a task, and K denote the partition size allocated for the task. To simplify our presentation, we restrict K to be those integers such that N is divisible by K. Let T denote the completion time for a set of tasks executed by the system, which is the least time by which all tasks in the set have completed execution. If the set consists of only one task, then T is simply the execution time of the task, which is the sum of the computation and communication time for the execution of the task.

2.1: Impact of Task Characteristics on Optimal Partition Size

Example 1. Summing N Numbers

We use a recursive doubling algorithm to sum the N numbers. Initially all the processors are active, and each processor is assigned $\frac{N}{K}$ numbers. Each processor first forms the partial sum of $\frac{N}{K}$ numbers. These partial sums are then accumulated to form the final sum in $\log K$ iterations. In each iteration, starting from the leftmost active processor, every alternate processor sends its partial sum to the active processor immediately to its right. An active processor that receives a data forms a new partial sum by adding the received data to its own partial sum. At the end of an iteration, all the sending processors are disabled. This parallel algorithm takes $(\frac{N}{K}-1)+\log K$ additions and $K-1$ communication operations on the linear array. The completion time T is given by

$$T=\frac{N}{K}+K+\log K-2.$$

For $1 \le K \le N$, the computation time, which is $\frac{N}{K} - 1 + \log K$, is a decreasing function of K, and the communication time, which is $K - 1$, is an increasing function of K; combining the effects of both, the execution time is concave upward with respect to K, with the minimum occuring at some K' between 1 and N. The variation between T and K for $N = 512$ is shown in Figure 1. For $N = 512$, the execution time has minimum at $K = 16$, and this is the optimal partition size for one summing task of size 512 on the given partitionable system. ▮

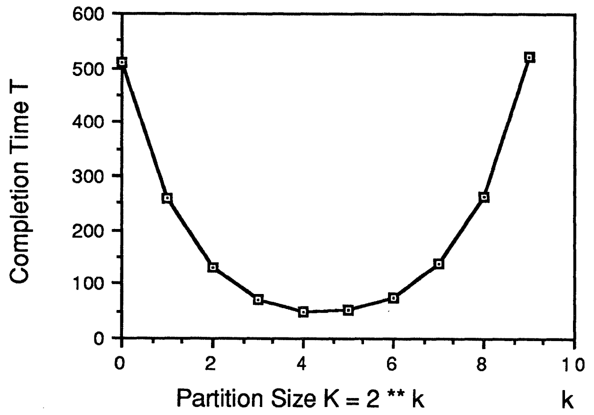


Figure 1 Summing 512 Numbers

Example 2. Sorting N Elements

We use the parallel algorithm given in [BAU78] to sort the N elements. The algorithm is a generalization of the odd-even transposition sort. Each processor is assigned $\frac{N}{K}$ elements. These elements are first sorted in each processor. The resulting subsequences are then merged and redistributed for K iterations to form the final sequence. For all odd iterations, processor $i+1$, where $i = 1, 3, \ldots, 2\lfloor\frac{K}{2}\rfloor - 1$, first sends its subsequence to processor i, processor i then merges the two subsequences it has, retains the first half of the resulting subsequence (the $\frac{N}{K}$ smallest elements), and sends the second half of the subsequence (the $\frac{N}{K}$ largest elements) back to processor $i + 1$. For all even iterations, the same steps as the odd iterations are executed but for $i = 2, 4, \ldots, 2\lfloor\frac{K-1}{2}\rfloor$. After K such iterations, the final sorted sequence will be partitioned among the N processors, with each processor holding a subsequence of $\frac{N}{K}$ elements. These subsequences are in increasing order from processor 1 to processor N. Since the initial sorting takes $\frac{N}{K}\log\frac{N}{K}$ comparisons and each iteration takes $2\frac{N}{K}$ comparisons and $2\frac{N}{K}$ communication operations on the linear array, the overall completion time is given by $T = \frac{N}{K}\log\frac{N}{K} + 4N$. For $1 \le K \le N$, the computation time, which is $\frac{N}{K}\log\frac{N}{K} + 2N$, is a decreasing function of K, and the communication time, which is $2N$, is a constant with respect to K; as a result, the execution time is a decreasing function of K, with the minimum occuring at $K = N$. The variation between T and K for $N = 512$ is shown in Figure 2. For $N = 512$, the execution time has minimum at $K = 512$, and this is the optimal partition size for one sorting task of size 512 on the given partitionable system.

As illustrated in the above two examples, the optimal partition size for a task depends on the computation and communication requirements of a task, which in turn are determined by the characteristics of the corresponding program, input data, and supporting architecture. For the task of summing, the optimal partition size is smaller than the maximum degree of parallelism in the task, but for sorting, these two quantities are equal.

2.2: Impact of Workload on Optimal Partition Size

Due to the need for communication in most parallel programs, as we increase the number of processors allocated to a task by a factor of k, the execution time of the task is usually reduced by a factor less than k. For instance, for the summing task with $N = 512$ in Example 1, as the partition size increases from 1 to 16 by a factor of two each time (that is, from 1 to 2, 2 to 4, 4 to 8, and 8 to 16), the execution time decreases from 511 to 50 by factors of 1.988 (511 to 257), 1.947 (257 to 132), 1.808 (132 to 73), and 1.460 (73 to 50); when the partition size increases beyond 16, the execution time increases. For the sorting task with $N = 512$ in Example 2, as the partition size increases from 1 to 512 by a factor of two each time, the execution time decreases from 6656 to 2048 by factors of 1.625, 1.391, 1.210, 1.101, 1.046, 1.019, 1.008, 1.003, and 1.001. Due to such diminishing return in performance with larger partition sizes, when there are a multiple number of tasks ready for execution in a partitionable system, using a smaller partition size for each task and executing as many tasks in parallel as possible could lead to a shorter completion time than using the partition size that gives the minimum execution time for each task. We illustrate this impact of workload on optimal partition size in the next example.

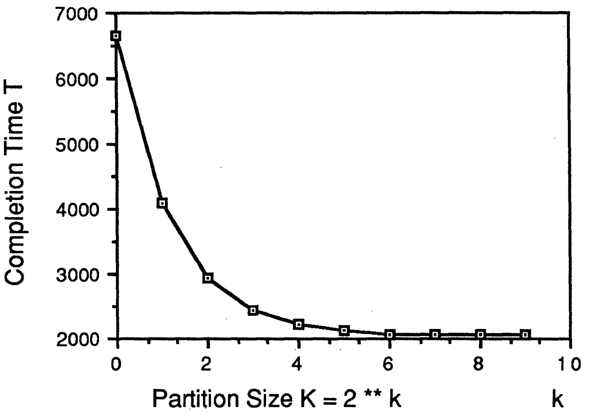


Figure 2 Sorting 512 Elements

Example 3. Multiple Sorting Tasks of N Elements

Suppose we have eight sorting tasks to be executed, with each task having to sort N elements. Assume that each task is allocated a partition of size K. Since there are 512 processors in the system, for $K \leq 64$ the eight sorting tasks can be executed in parallel, and the completion time T for these eight sorting tasks is the same as the execution time of one sorting task of size N using a partition of size K. However, for $K > 64$, the eight sorting tasks have to be executed in multiple batches, with each batch, except possibly the last, having $\lfloor \frac{512}{K} \rfloor$ tasks. The completion time T in this case is the product of $\lceil \frac{8K}{512} \rceil$ (the number of batches) and the execution time of one sorting task on a partition of size K. Figure 3 shows the variation between the completion time T and partition size K for $N = 512$. The least completion time for a set of eight tasks is obtained when $K = 64$, and this is the optimal partition size for each such task. The corresponding completion time T for the eight tasks is 2072. On the other hand, if we use a partition size of 512 for each of the eight tasks, which is the optimal partition size for the execution of one task, the completion time T is 16384 instead.

The least completion time for a set of sixteen tasks is obtained when $K = 32$, and this is the optimal partition size for each such task. The corresponding completion time T for the 16 tasks is 2112. If we use a partition size of 512 for each of the sixteen tasks, the completion time T is 32768 instead.

As illustrated in the above examples, determining the partition sizes at run time based on task characteristics, workload, and amount of resource available could provide higher throughput than determining such partition sizes at either the program design time or compile time. For a partitionable system designed for special purpose applications, since the set of tasks the system needs to support is usually known in advance, by pre-analyzing the characteristics of the tasks and making them available to the system, it is possible for it to determine the partition sizes at run time. The feasibility and advantage of such an approach is determined by the overhead involved. In the remainder of this paper, we focus on analyzing the time complexity of the problem of determining optimal partition sizes at run time.

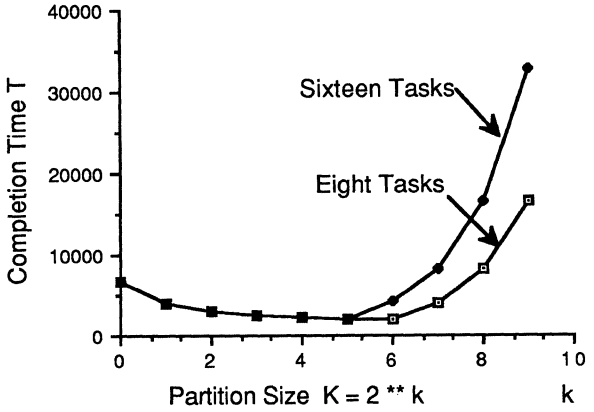


Figure 3 Multiple Sorting Tasks of Size 512

Section 3: The Processor Partitioning Problem

In multiprocessor scheduling problems, we are given a number of processors and a set of tasks, and the goal is to schedule the tasks on these processors such that some objective on the execution times of the tasks is optimized. These are variants of the general multiprocessor scheduling problem, and most of these problems are NP-Complete [GAR79]. In the following, we first review the general multiprocessor scheduling problem and two of its variants most related to our problem.

Let Z^+ denote the set of positive integers. Let n denote the number of processors and m the number of tasks, where n, $m \in Z^+$. For any positive integer k, we use $[k]$ to denote the set $\{1, \cdots, k\}$.

The general multiprocessor scheduling problem can be stated as follows [GAR79]: Given n processors and m tasks, where each task requires one processor for execution with a specific execution time and there is no precedence constraint among the tasks, the objective is to find a nonpreemptive schedule with the least completion time for all the tasks. This problem is NP-Complete in the strong sense for arbitrary n, but can be solved in pseudo-polynomial time for any fixed n. The problem remains NP-Complete for $n = 2$.

A variant of the general multiprocessor scheduling problem is the multiprocessor scheduling problem with nonfragmentable resource constraint [GAR75]. In a special-case of this problem, we are given n processors and m tasks, each task requiring one processor for execution with execution time equal to unity and no precedence constraints among the tasks. Further, we are given a resource R with a total amount B available, and a nonnegative resource requirement $R(i)$ for each task $i \in [m]$. The objective is to find a nonpreemptive schedule with the least completion time for all the tasks such that the sum of the resource requirements of all the tasks scheduled simultaneously does not exceed the total amount of the resource available. An important characteristic of the problem is that the resource does not have to be allocated in contiguous blocks since the resource does not suffer from fragmentation. This problem is shown to be NP-Complete by transforming the Three-Dimensional Matching problem to this problem [GAR75].

Yet another version of the scheduling problem is the multiprocessor scheduling problem with fragmentable resource constraint [BAK83]. In this, tasks share a resource such as memory, where such a resource may only be allocated in contiguous blocks. In this problem, we are given n processors and m tasks. We are also given a resource R with a total amount B available, and a nonnegative resource requirement $R(i)$ for each task $i \in [m]$. Once again, the objective is to find a nonpreemptive schedule with the least completion time for all the tasks such that the sum of the resource requirements of all the tasks scheduled simultaneously does not exceed the total amount of the resource available. The distinguishing characteristic between this problem and the earlier problem is that the resource may only be allocated in contiguous blocks since the resource is fragmentable. This problem is NP-Complete since it is equivalent to the 2-D bin packing problem.

In the processor partitioning problem, we are given n processors, r controllers, and m tasks among which there are no precedence constraints. Each task can be executed by a number of different partition sizes. The partitions may comprise of processors which need not be contiguous in any address space. The objective is to choose partition sizes for the tasks and to find a nonpreemptive schedule with the least completion time for all the tasks such that the maximum number of tasks scheduled simultaneously does not exceed the number of controllers r and the sum of the chosen partition sizes of all the tasks scheduled simultaneously does not exceed the total number of processors n. The above problem can be shown to be NP-Complete by transforming the multiprocessor scheduling problem with fragmentable resource constraint to a restricted version of the above problem. Details of the proof are omitted in this paper.

The processor partitioning problem that we study in the remainder of this paper is a special version of the problem stated above. In this version, we are given n processors, r controllers, and m tasks among which there are no precedence constraints, and each of which can be executed by a number of different partition sizes. The partitions may comprise of processors which need not be contiguous in any address space. For all $i \in [m]$, let q_i denote the total number of such partition sizes for task i, and let the functions $p_i : [q_i] \rightarrow [n]$ and $t_i : [q_i] \rightarrow Z^+$, define respectively the partition sizes and the corresponding execution times for task i. The functions p_i and t_i have properties such that for all $k, l \in [q_i]$ where $k < l$, we have the following:

(a) $p_i(k) < p_i(l)$ (the partition sizes defined by p_i are in increasing order)

(b) $t_i(k) > t_i(l)$ (the execution time of a task decreases as the partition size increases)

(c) $p_i(k)\, t_i(k) < p_i(l)\, t_i(l)$ (the execution time of a task using a larger partition size decreases by a factor which is less than the increase in partition size since $t_i(l)/t_i(k) > p_i(k)/p_i(l)$)

In addition, we have the assumption:

(∗) $m \le r$ and $\sum_{i=1}^{m} p_i(1) \le n$ (all the tasks available for execution can be executed in parallel when using the smallest partition sizes for these tasks).

The objective is to choose partition sizes (and their execution times, which obey the above three properties) for the tasks and to find a nonpreemptive schedule with the least completion time for all the tasks such that the maximum number of tasks scheduled simultaneously does not exceed the number of controllers r and the sum of the chosen partition sizes of all the tasks scheduled simultaneously does not exceed the total number of processors n. This problem can also be shown to be NP-Complete by transforming the multiprocessor scheduling problem with fragmentable resource constraint to a restricted version of the above problem. Details of the proof are omitted in this paper.

In a partitionable system, properties (a) and (b) imply ordering the partition sizes in increasing order of index and selecting only that part of the task characteristics where the execution time continues to decrease with increasing partition size. For example, for the tasks of summing and sorting discussed in Section 2.1, we only include five partition sizes 1, 2, 4, 8, 16 for summing, while for sorting, we include all the ten partition sizes. Property (c) implies that speed-up in a parallel system is less than linear due to communication and control overhead. We make assumption (∗) to simplify the presentation of our solution to the problem. Our solution may be extended to the case where this assumption does not hold.

Section 4: Solution Techniques for the Processor Partitioning Problem

In this section, we first derive some lower bounds on the completion time for the processor partitioning problem. Using these lower bounds, we derive a condition on the processor partitioning problem under which optimal solutions can be determined easily. We then present a polynomial time approximation algorithm for the processor partitioning problem. We also derive a worst-case bound on the solution obtained by the algorithm and the conditions under which it gives optimal solutions.

Let S_o be an optimal schedule, and for each $i \in [m]$, let u_i denote the index for the partition size for task i in schedule S_o. Let T_o be the completion time for schedule S_o, which is the optimal completion time.

4.1: Lower Bounds on Completion Time

Lemma 1 provides us with a relation between the optimal completion time T_o and the partition sizes and respective execution times for the tasks in an optimal schedule S_o.

Lemma 1. $T_o \ge \sum_{i=1}^{m} p_i(u_i) t_i(u_i)/n$.

Proof. The term $\sum_{i=1}^{m} p_i(u_i) t_i(u_i)$ represents the total time units that the allocated processors are busy in schedule S_o. Since S_o is a feasible schedule, at any time instant, each of the n processors is allocated to at most one task. Thus each processor is busy for at most T_o time units. Since the total number of processors allocated at any instance is bounded by n, we have $\sum_{i=1}^{m} p_i(u_i) t_i(u_i) \le nT_o$. The Lemma follows consequently. ∎

As per Property (c) stated in Section 3, in a partitionable system, as we transit from a given partition size to a higher partition size, the execution time decreases by a factor which is less than the increase in partition size. This property asserts that as we increase the partition size for a task, the time to execute on the new partition cannot decrease below a certain limit. Based on this relationship between partition sizes and execution times, Lemma 2 derives, under certain conditions, a lower bound on completion time for the special case where all the tasks to be executed are of the same type.

Lemma 2. *Assume that there are m tasks of the same type to be executed on an n processor system. Let q denote the number of possible partition sizes for this type of task. In addition, let the functions $p : [q] \rightarrow [m]$ and $t : [q] \rightarrow Z^+$ denote the partition sizes and the corresponding execution times respectively. Assume further that there exists some $l \in [q]$ such that $p(l) \geq \lceil \frac{n}{m} \rceil$. Let l^* be the smallest index in $[q]$ such that $p(l^*) \geq \lceil \frac{n}{m} \rceil$. Then $T_o \geq t(l^*)$.*

Proof. For each $i \in [m]$, let u_i be the index in $[q]$ such that $p(u_i)$ is the partition size for task i in some optimal schedule S_o. For each $i \in [m]$, the completion time for task i is $t(u_i)$. If for some $i \in [m]$, $u_i \leq l^*$ then $T_o \geq t(u_i) \geq t(l^*)$ from Property (b) and the lemma is trivially true. Thus assume that for every $i \in [m], u_i > l^*$. It follows from Property (c) that

$$\text{for every } i \in [m],\ p(l^*)t(l^*) < p(u_i)t(u_i).$$

The above implies that

$$m\ p(l^*)t(l^*) < \sum_{i=1}^{m} p(u_i)t(u_i).$$

Since $p(l^*) \geq \lceil \frac{n}{m} \rceil$, it follows that

$$m\lceil \frac{n}{m} \rceil t(l^*) < \sum_{i=1}^{m} p(u_i)t(u_i).$$

Since $\lceil \frac{n}{m} \rceil \geq \frac{n}{m}$, we have

$$t(l^*) < \sum_{i=1}^{m} p(u_i)t(u_i)/n.$$

Since the optimal schedule S_o has completion time T_o, from Lemma 1, we have $T_o > t(l^*)$. ▮

For the special case where all the tasks to be executed are of the same type, Theorem 3 states a condition under which a parallel schedule with $\frac{n}{m}$ processors in each partition has the least completion time.

Theorem 3. *Assume there are m tasks of the same type to be executed on an n processor system. Let q denote the number of possible partition sizes for this type of task. In addition, let the functions $p : [q] \rightarrow [m]$ and $t : [q] \rightarrow Z^+$ denote the partition sizes and the corresponding execution times respectively. Assume further that n is divisible by m and there exists some $l^* \in [q]$ such that $p(l^*) = \frac{n}{m}$. Then a parallel schedule with $\frac{n}{m}$ processors in each partition has the least completion time.*

Proof. Since the allocation is feasible, the theorem follows from Lemma 2. ▮

4.2: An Approximation Algorithm for Partitioning

We now present an approximation algorithm which runs in $O(\min\{n, \sum_{i=1}^{m} q_i\} \log m)$ time. This algorithm explores only parallel schedules and does not explore any serial-parallel schedules. By assumption (*), there always exists a feasible, parallel schedule for the given set of tasks. We first give an informal description of the algorithm below.

Initially, each task is allocated a number of processors equal to the smallest partition size for this task. By assumption (*), such an allocation is always possible. Then, the task with the longest execution time is selected. As many processors are allocated to this task as is neccessary to transit to the next larger partition size. This process is repeated until we run out of free processors.

Intuitively, the algorithm allocates processors to tasks in an efficient manner. To account for the effect of diminishing return with larger partition size, the algorithm starts with the smallest partition size for each task, and increases a partition size only if the execution time corresponding to such a partition size determines the completion time. Since the criterion to be minimized is completion time, the algorithm isolates the task with the longest execution time at every iteration, since this is what determines the completion time in a parallel schedule. It then allocates as many processors as is neccessary to reduce the execution time of this task so as to reduce the overall completion time. If this additional allocation results in a different task having the longest execution time, the algorithm allocates additional processors to this task. Thus, it provides processors to tasks that need them the most, in some sense. Given below is a more formal statement of the algorithm.

Approximation Algorithm: Partitioning

Input: n, m, for every $i \in [m]$, q_i, p_i, t_i.
Output: a set of indexes $\{l_i | l_i \in [q_i] \text{ for } i \in [m]\}$.

```
begin
    $remain := n$;
    for $i := 1$ to $m$ do
    begin
        $l_i := 1$;
        $remain := remain - p_i(1)$
    end;
    $done := false$;
    while ($remain > 0$) and (not $done$) do
    begin
        find $j$ such that $t_j(l_j) = max_{i \in \{1,\cdots,m\}} t_i(l_i)$;
        if ($l_j < q_j$) and ($remain \geq p_j(l_j + 1)$
                                  $- p_j(l_j)$)
          then
```

```
            begin
                $l_j := l_j + 1$;
                $remain := remain - (p_j(l_j+1) - p_j(l_j))$
            end
            else
              $done := true$
        end
    end.
```

In the above algorithm, the *while loop* will be executed no more than min $\{n, \sum_{i=1}^{m} q_i\}$. Inside the loop, we have to find the maximum of the t_i's, which can be done in time $O(\log m)$ if we use a priority queue to store the t_i's. The rest of the algorithm can be done in $O(m)$ time. Therefore, the approximation algorithm has complexity $O($min $\{n, \sum_{i=1}^{m} q_i\} \log m)$. Since $\sum_{i=1}^{m} p_i(l_i) \le n$, the partition sizes chosen from the approximation algorithm allow a parallel schedule. In the next section, we derive some bounds on the performance of the algorithm.

4.3: Performance Analysis of the Approximation Algorithm

Let S_a be the schedule determined by the approximation algorithm. For each $i \in [m]$, let l_i denote the index for the partition size for task i in schedule S_a. Let T_a be the completion time for schedule S_a. By definition, we have the following set of inequalities: $T_o \le T_a$, and for all $i \in [m]$, $t_i(u_i) \le T_o$, $t_i(l_i) \le T_a$.

Lemma 4 derives a relationship between partition sizes in the schedule S_a and partition sizes in a feasible schedule with completion time no greater than the completion time of S_a.

Lemma 4. *Let S_f be a feasible schedule. For all $i \in [m]$, let r_i denote the index for the partition size for task i in S_f. Let T_f denote the completion time due to schedule S_f. Assume that $T_f \le T_a$. Then, for all $i \in [m], r_i \ge l_i$.*

Proof. Let i be an arbitrary element in $[m]$. We consider two cases.

a) $l_i = 1$. Since in any feasible schedule, every task needs at least as many processors as in the least-sized partition, we have $r_i \ge l_i$.

b) $l_i > 1$. In this case, we have $t_i(l_i - 1) > T_a$, otherwise the algorithm will not augment $l_i - 1$ to l_i. We thus have the following relationship:

$$t_i(r_i) \le T_f \le T_a < t_i(l_i - 1).$$

This implies that $r_i > l_i - 1$ from Property (b). Thus, $r_i \ge l_i$. ▮

Lemma 5 derives a lower bound on the optimal completion time T_o in terms of the partition sizes and the execution times of the tasks in the schedule S_a.

Lemma 5. $T_o \ge \sum_{i=1}^{m} p_i(l_i)t_i(l_i)/n$.

Proof. Since S_o is a feasible schedule and $T_o \le T_a$, from Lemma 4, we have, for all $i \in [m], u_i \ge l_i$. Hence from Property (c), we have, for all $i \in [m]$,

$$p_i(u_i)t_i(u_i) \ge p_i(l_i)t_i(l_i).$$

Therefore,

$$\sum_{i=1}^{m} p_i(u_i)t_i(u_i) \ge \sum_{i=1}^{m} p_i(l_i)t_i(l_i).$$

From Lemma 1, we also have

$$nT_o \ge \sum_{i=1}^{m} p_i(u_i)t_i(u_i).$$

Thus, $T_o \ge \sum_{i=1}^{m} p_i(l_i)t_i(l_i)/n$. ▮

Theorem 6 derives a worst-case bound on the completion time T_a.

Theorem 6. *Let k be the index in $[m]$ such that task k is the one that determines the completion of T_a. Then $T_a \le (n/p_k(l_k))T_o - (\sum_{\substack{1\le i\le m \\ i\ne k}} p_i(l_i)t_i(l_i))/p_k(l_k)$.*

Proof. From Lemma 5, we have

$$\sum_{i=1}^{m} p_i(l_i)t_i(l_i) \le nT_o.$$

Since $t_k(l_k) = T_a$, we can rewrite the above expression as

$$\sum_{\substack{1\le i\le m \\ i\ne k}} p_i(l_i)t_i(l_i) + p_k(l_k)T_a \le nT_o.$$

Thus, we have

$$T_a \le nT_o/p_k(l_k) - \sum_{\substack{1\le i\le m \\ i\ne k}} p_i(l_i)t_i(l_i)/p_k(l_k),$$

and the theorem follows. ▮

Theorem 7 proves that the schedule due to the approximation algorithm is an optimal schedule among all parallel schedules.

Theorem 7. *The completion time T_a due to the approximation algorithm is the optimal completion time among all parallel schedules.*

Proof. Assume to the contrary that there exists a parallel schedule S_p with completion time T_p such that $T_p < T_a$. For each $i \in [m]$, let v_i denote the index for the partition size for task i in S_p. Let k be the index in $[m]$ such that task k is the one that the approximation algorithm tries to increase the partition size of before it terminates (in case two or more tasks are tied in determining the task with the longest execution time). We have $t_k(l_k) = T_a$. From

Lemma 4, we have,

$$\text{for all } i \in [m], v_i \geq l_i.$$

Consequently, for each $i \in [m], p_i(v_i) \geq p_i(l_i)$.

Further, from our assumption that $T_p < T_a$, we get $t_k(v_k) < t_k(l_k)$. This implies that $v_k > l_k$, and hence $v_k \geq l_k + 1$. Therefore, $p_k(v_k) \geq p_k(l_k + 1)$. It follows that

$$\sum_{i=1}^{m} p_i(v_i) \geq \sum_{\substack{1 \leq i \leq m \\ i \neq k}} p_i(l_i) + p_k(l_k + 1).$$

Thus,

$$\sum_{i=1}^{m} p_i(v_i) \geq \sum_{i=1}^{m} p_i(l_i) + p_k(l_k + 1) - p_k(l_k). \quad (1)$$

When the algorithm terminates, the number of remaining processors is strictly less than $p_k(l_k + 1) - p_k(l_k)$ (task k determines the completion time). Otherwise, the algorithm would have reduced the completion time $T_a = t_k(l_k)$ by augmenting the partition size of task k from $p_k(l_k)$ to $p_k(l_k + 1)$. Thus, we have,

$$n - \sum_{i=1}^{m} p_i(l_i) < p_k(l_k + 1) - p_k(l_k).$$

This implies that

$$\sum_{i=1}^{m} p_i(l_i) + p_k(l_k + 1) - p_k(l_k) > n. \quad (2)$$

Combining inequalities (1) and (2), we have $\sum_{i=1}^{m} p_i(v_i) > n$, which is a contradiction since S_p is a parallel schedule and there are at most n processors. Thus, no parallel schedule S_p exists with completion time T_p such that $T_p < T_a$. The completion time due to the approximation algorithm is therefore optimal among all parallel schedules. ▮

Theorem 8 proves that the schedule due to the approximation algorithm is optimal under the condition that no processors remain to be allocated when the algorithm terminates, and in addition, the execution times of the tasks for the partition sizes allocated by the approximation algorithm are equal.

Theorem 8. *Assume that $\sum_{i=1}^{m} p_i(l_i) = n$ and for all $i \in [m], t_i(l_i) = T_a$. Then $T_a = T_o$.*

Proof. Assume to the contrary that $T_a > T_o$. From Lemma 5 we have

$$T_o \geq \sum_{i=1}^{m} p_i(l_i) t_i(l_i)/n.$$

By the assumption of the theorem, we have

$$T_o \geq \sum_{i=1}^{m} p_i(l_i) T_a/n.$$

Since $\sum_{i=1}^{m} p_i(l_i) = n$, the above implies that

$$T_o \geq T_a,$$

which is a contradiction. Thus, $T_a = T_o$. ▮

Corollary 9. *Assume there are m tasks of the same type to be executed on an n processor system, and n is divisible by m. Assume further that there exists some $l^* \in [q]$ such that $p(l^*) = \frac{n}{m}$. Then the approximation algorithm obtains an allocation with the optimal completion time, allocating partitions consisting of $\frac{n}{m}$ processors to each of the m tasks.*

Proof. When the approximation algorithm terminates, it allocates a partition of size $p(l_i) = \frac{n}{m}$, for each $i \in [m]$. Since every task is of the same type, $t(l_i) = T_a$, for each $i \in [m]$. Since $\sum_{i=1}^{m} p(l_i) = n$, the corollary follows from Theorem 8. ▮

The analyses in this subsection shows that under certain conditions (those stated in Theorem 8 and Corollary 9) the approximation algorithm produces an optimal schedule. Further, if we constrain ourselves to strictly parallel schedules, then the schedule due to the approximation algorithm is always optimal among all such schedules. In general, the performance of the schedule due to the approximation algorithm is always within the bound given in Theroem 6.

Section 5: Applications of the Approximation Algorithm

We now give an example of using the approximation algorithm in a typical application on the model of a partitionable system described in Section 2. The application we choose to illustrate the approximation algorithm is in the area of image processing. In image processing applications using stereo images, there is a need to compute the Histogram and perform Image Smoothing for a pair of images. Since both these computations may be carried out in parallel, the workload may then comprise of the following tasks: two Histogramming tasks and two Smoothing tasks (one for the right image, and one for the left image).

Assume that the image is a square of $\sqrt{N} \times \sqrt{N}$ pixels, where $\sqrt{N}$ is a positive integer. Let K denote the size of a partition, where $\sqrt{K}$ is a positive integer. Assume also that N is divisible by K.

We assume that the image is divided evenly over the K processors so that each processor has a square subimage of $\frac{N}{K}$ pixels. In computing the histogram of an image, the frequency count of each grey level is computed over the entire image. The final histogram is represented as an array of b elements, each element being a count of the number of pixels in the image with that grey level. Each processor first computes the histogram of the subimage local to it in time $\frac{N}{K}$. These partial histograms are then accumulated to form the total histogram in $\log K$ iterations using a recursive doubling algorithm similar to the one for the summing task given in Section 2, Example 1. Computing the new partial histogram in each iteration takes b time units since it amounts to a vector addition of b elements. Communication in the first iteration takes b time units since an array of b elements is sent to an adjacent processor. In general, for all $i = 1, \dots, \log K$, communication in the i^{th} iteration takes $b + 2^{i-1} - 1$ time units since an array of b elements is sent to a processor which is 2^{i-1} away, and the array of b elements can be sent in a pipelined fashion. The total number of computation operations is $\frac{N}{K} + b \log K$, and the total number of communication operations is $(b-1)\log K + K - 1$. The execution time for Histogram computation is given by

$$T = \frac{N}{K} + K - 1 + (2b-1)\log_2 K.$$

For $N = 512 \times 512$, $b = 256$, and K varying from 1 to 4096 as squares of powers of two, the variation of the execution time T with partition size K is shown in Figure 4.

K	1	4	16	64	256	1024	4096
T	262144	66561	18443	7225	5367	6389	10291

Figure 4 Histogramming

In the problem of Image Smoothing, the grey value of each pixel in an image is averaged with the surrounding eight neighbouring points for a given number of iterations. In each iteration of the algorithm, each processor needs to perform $8\frac{N}{K}$ additions and $\frac{N}{K}$ divisions, and send a message to each of the eight processors operating on its surrounding subimages; four of these messages are of size $\sqrt{\frac{N}{K}}$ elements, and four of size one element. If the subimages are mapped row by row into the linear array, each processor has to communicate with two processors at a distance of one, two at a distance of $\sqrt{K}$, two at a distance of $\sqrt{K}+1$, and two at a distance of $\sqrt{K}-1$. The net communication time on the linear array is thus:

$$2\sqrt{\frac{N}{K}} + 2\sqrt{\frac{N}{K}}K + 2(\sqrt{K}-1) + 2(\sqrt{K}+1).$$

The total execution time for Smoothing is given by

$$T = 9\frac{N}{K} + 2\sqrt{\frac{N}{K}} + 2\sqrt{N} + 4\sqrt{K}.$$

For $N = 512 \times 512$, the variation of the execution time T with partition size K is shown in Figure 5.

K	1	4	16	64	256	1024	4096
T	2361348	591368	148752	38048	10368	3488	1872

Figure 5 Image Smoothing

Next we apply the approximation algorithm for the job mix of two Histogramming tasks and two Smoothing tasks, all of which are available for execution. The approximation algorithm stops after 18 iterations with a schedule whose overall completion time is 5367 time units. Table 6 shows the partition sizes for the tasks and the completion time. The table has one entry for every two iterations since the partition size has to increase for both Smoothing tasks or both Histogramming tasks to reduce completion time at each iteration.

Smooth Size	Hist Size	Comp Time
1	1	2361348
4	1	591368
16	1	262144
16	4	148752
64	4	66561
64	16	38048
256	16	18443
256	64	10368
1024	64	7225
1024	256	5367

Figure 6 Iterations in Approximation Algorithm

If we use the maximum partition size consisting of all 4096 processors for each task, and a strictly sequential schedule to execute the tasks, then the overall completion time is 24326 time units, implying a factor of 4.5 improvement in the overall completion time with the parallel schedule obtained by the approximation algorithm. If, on the other hand, we use the partition size with the least execution time for each task (the optimal partition size for the execution of a single task) and the best schedule possible to execute the tasks, then the overall completion time is 9111 time units, implying a factor of 1.6 improvement in the overall completion time with the parallel schedule obtained by the approximation algorithm.

Applying Theorem 6 to obtain a worst-case performance bound for the approximation algorithm in the above example, we get

$$T_a \leq \frac{4096}{256}T_o - \frac{2 \times 1024 \times 3488 + 256 \times 5367}{256},$$

which implies that

$$T_a \leq 16T_o - 33271,$$

from which we can infer that $T_o \geq 2415$. Using this bound, we can deduce that the approximation algorithm obtains a completion time which is less than 2.23 times the optimal completion time.

For this particular example, the schedule determined by the approximation algorithm is actually optimal among all possible schedules, that is, $T_a = T_o = 5367$.

Section 6: Conclusion

In this paper, we address the problem of processor partitioning in partitionable architectures. We demonstrate the importance of determining the partition sizes based on task characteristics, workload, and availability of resources. An underlying assumption is that the task characteristics are available. For a system designed for a special-purpose application, it is possible to pre-analyze the characteristics of the tasks and make them available to the system since the set of tasks the system needs to support is usually known in advance. For such systems, we advocate determining the partition sizes at run time. To support such an approach, we investigate the design of an efficient approximation algorithm to determine the partition sizes based on task characteristics and workload. We derive the worst-case performance bound for the approximation algorithm, and conditions under which the algorithm is optimal.

Other important issues that may affect the feasibility of such an approach such as the overhead involved and the fragmentation of the processing resources in such systems will be studied in the future. The fragmentation of processors in such systems is influenced by the network interconnecting the processors in the system. We have preliminary analysis [KRI87] on the physical subset of processors that may comprise a partition in a partitionable system. We plan to investigate further these and related issues in the future.

Acknowledgements: The authors would like to thank the referees for carefully reading the paper and giving detailed comments that have helped improve the paper.

Bibliography

[BAK83] B. S. Baker, and J. S. Schwarz, "Shelf Algorithms for Two-dimensional Packing Problems," SIAM J. Comput. Vol. 12, No. 3, August 1983, pp. 508 - 525.

[BAU78] G. Baudet, and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," IEEE Transactions on Computers, Vol. C-27, January 1978, pp. 84 - 87.

[GAR75] M. R. Garey, and D. S. Johnson, "Complexity Results for Multiprocessor Scheduling under Resource Constraints," SIAM J. Comput. Vol. 4, No. 4, December 1975, pp. 397 - 411.

[GAR79] M. R. Garey, and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Company, 1979.

[KRI87] R. Krishnamurti, "Reconfigurable Parallel Architectures for Special Purpose Computing," PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1987 (also available as Technical Report MS-CIS-87-81).

[KUC77] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," ACM Computing Surveys, Vol. 9, No. 1, March 1977, pp. 29 - 59.

[KUE84] J. T. Kuehn, and H. J. Siegel, "Simulation Studies of a Parallel Histogramming Algorithm for PASM," 7th International Conference on Pattern Recognition, 1984, pp. 646 - 649.

[LIN81] B. Lint, and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," IEEE Transactions on Software Engineering, Vol. SE-7, March 1981, pp. 174 - 188.

[MA87] Y. W. Ma, R. Krishnamurti, L. Tao, D. G. Shea, B. Narahari, R. Varadarajan, "Reconfigurable Special-Purpose Computers," Second International Conference on Supercomputing, 1987, pp. 343 - 351.

[MA88] Y. W. Ma, and D. G. Shea, "Downward Scalability of Parallel Architectures," to appear in Third International Conference on Supercomputing, 1988.

[NIC87] D. M. Nicol, and F. H. Willard, "Problem Size, Parallel Architecture, and Optimal Speedup," Proceedings of 14th Annual International Symposium on Computer Architecture, June 1987, pp. 347 - 354.

[NUT77] G. J. Nutt, "Multiprocessor Implementation of a Parallel Processor", Proceedings of the Fourth Annual Symposium on Computer Architecture, 1977.

[PRE80] U. V. Premkumar, R. Kapur, M. Malek, G.J.Lipovski, and P.Horne, "Design and implementation of the Banyan Interconnection Network in TRAC," Proceedings of the National Computer Conference, 1980.

[SIE81] H. J. Siegel L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley Jr., and S. D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," IEEE Transactions on Computers, Vol. C-30, December 1981, pp. 934 - 947.

Non-Deterministic Instruction Time Experiments on the PASM System Prototype

Samuel A. Fineberg, Thomas L. Casavant, Thomas Schwederski*

Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

*H.J. Siegel***

Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Abstract

Experimentation aimed at determining the minimum-granularity at which variable-length SIMD operations may be decoupled into identical asynchronous MIMD streams for a performance benefit is reported. The experimentation is based on timing measurements made on the PASM system prototype at Purdue. The application used to measure and evaluate this phenomenon was matrix multiplication, which has feasible solutions in both SIMD and MIMD modes of computation, as well as in a hybrid of SIMD and MIMD modes. Matrix multiplication was coded in these three ways and experiments were performed which examine the tradeoffs among all of these modes.

1. Introduction

While extensive past efforts have dealt with analytical and simulated performance analysis of SIMD and MIMD algorithms, computations, and machines, this work describes empirically-based research generated from experiments on a parallel machine. This research was performed in an attempt to gain insight into the effect of certain aspects of novel architectures on applications programs. Specifically, the performance of the PASM prototype, a machine capable of both SIMD and MIMD modes of computation, is evaluated from the perspective of matrix multiplication. This application was chosen because it has obvious optimal solutions and a simple enough structure to permit analysis of architecture features through controlled measurements of program execution time. The experiments described are based on SIMD, MIMD, and hybrid S/MIMD algorithms for multiplying n × n matrices for values of n ranging from 4 to 256. Operations were performed on 16-bit integers utilizing 16 processors in several 4, 8, and 16 processor configurations.

The primary architecture feature being evaluated in this work is the ability to decouple small grains of variable execution-time operations from SIMD sections of code into multiple asynchronous MIMD threads of control. This unique feature derives from the ability to dynamically reconfigure the parallelism mode of PASM.

Results indicate that when mode-changing operations induce a minimal overhead, benefits of such decoupling may be found even for relatively small amounts of variation in the execution-time of individual operations. This same low-overhead mode-changing feature was also used to greatly improve the performance of the inter-process communication components of parallel programs by using the implicit hardware synchronization of SIMD mode to reduce the complexity of message passing protocols through the PASM interconnection network. Finally, experiments indicate that due to the existence of finite queues for issuing instructions from the control units to the processing elements in SIMD mode, superlinear speed-up is achievable. (We define superlinear speed-up as the condition in which the speed-up to number of PEs (processing elements) ratio is greater than 1.)

Section 2 briefly describes generally related work, and Section 3 overviews PASM and its prototype. Section 4 describes the basic algorithm that was used while Section 5 describes the programmed variations of this algorithm as implemented on PASM for use in the experiments presented in Section 6. In Sections 7 through 11, the empirical results are discussed under special consideration of the PASM architecture as well as the central issue of decoupling variable-length SIMD operations into multiple asynchronous MIMD streams.

2. Background and Related Work

Related experimental research has been carried out on several machines through the use of both simulation and experimental techniques. Simulation-based analysis was performed by Su and Thakore for the SM3 system and a hypercube architecture [SuT87]. Experimental work involving measurements on working machines has also been performed. Examples include work involving several machines: the BBN Butterfly [CrG85], Cm* [GeS87], the Encore Multimax [Hud88], the Intel Hypercube [Hud88], PASM [FiC87], and the Warp system [AnA87]. In these efforts, matrix multiplication was normally employed as an example algorithm. Other reported work involving efficiency measurements and algorithm optimization on parallel machines includes work done on an Alliant FX/8 [JaM86, Han88], a CRAY XMP [Cal84], and a combination of Apollo work-stations and an Alliant FX/8 [KuN88].

3. Overview of PASM and the PASM Prototype

The PASM (partitionable SIMD/MIMD) system is a dynamically reconfigurable architecture in which the processors may be partitioned to form independent virtual SIMD and/or MIMD machines of various sizes [SiS81]. A 30-processor prototype has been constructed and was used in the experiments described in Section 6. This section discusses the PASM architecture characteristics which are most relevant to the reported experimentation. For a more general description of the architecture, see [SiS87].

*Supported by the Supercomputing Research Center under contract number 6925.
**Currently on leave from Purdue.

The *Parallel Computation Unit* of PASM contains N PEs where N is a power of 2 (numbered from 0 to N–1), and an interconnection network. Each *PE* (processing element) is a processor/memory pair. The *PE processors* are sophisticated microprocessors that perform the actual SIMD and MIMD operations. The *PE memory modules* are used by the processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The *Micro Controllers (MCs)* are a set of $Q=2^q$ processors, numbered from 0 to Q–1, which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. Each MC controls N/Q PEs. PASM has been designed for N=1024 and Q=32 (N=16 and Q=4 in the prototype). A set of MCs and their associated PEs form a virtual machine. In SIMD mode, each MC fetches instructions and common data from its associated memory module, executes the control flow instructions (e.g., branches), and broadcasts the data processing instructions to its PEs. In MIMD mode, each MC gets instructions and common data for coordinating its PEs from its memory.

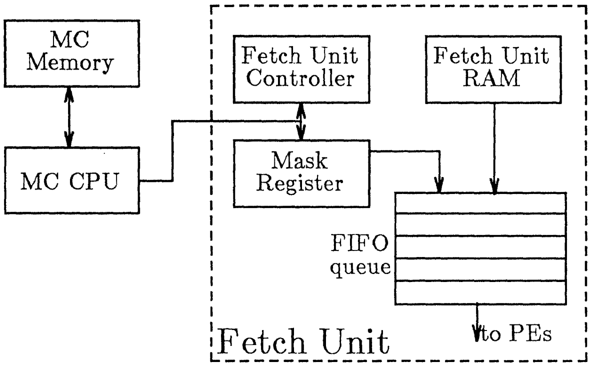


Figure 1: Simplified MC structure.

The PASM prototype system was built for N=16 and Q=4. This system employs Motorola MC68000 processors as PE and MC CPUs, with a clock speed of 8 MHz. The interconnection network is a circuit-switched Extra-Stage Cube network, which is a fault-tolerant variation of the multistage cube network. Because knowledge about the MC and the way in which SIMD instructions are implemented with standard MC68000 microprocessors is essential to the understanding of the behavior that was observed in the experiments, the SIMD instruction broadcast mechanism is overviewed below.

Consider the simplified MC structure shown in Figure 1. The MC contains a memory module from which the MC CPU reads instructions and data. Whenever the MC needs to broadcast SIMD instructions to its associated PEs, it first sets the Mask Register in the Fetch Unit, thereby determining which PEs will participate in the following instructions. It then writes a control word to the Fetch Unit Controller which specifies the location and size of a block of SIMD instructions in the Fetch Unit RAM. The Fetch Unit Controller automatically moves this block word by word into the Fetch Unit Queue. Whenever an instruction word is enqueued, the current value of the Mask Register is enqueued as well. Because the Fetch Unit enqueues blocks of SIMD instructions automatically, the MC CPU can proceed with other operations without waiting for all instructions to be enqueued.

PEs execute SIMD instructions by performing an instruction fetch from a reserved memory area called the *SIMD instruction space.* Whenever logic in the PEs detects an access to this area, a request for an SIMD instruction is sent to the Fetch Unit. Only after all PEs that are enabled for the current instruction have issued a request is the instruction released by the Fetch Unit queue, and the enabled PEs receive and execute the instruction. Disabled PEs do not participate in the instruction and wait until an instruction is broadcast for which they are enabled. This way, switching from MIMD to SIMD mode is reduced to executing a jump instruction to the reserved memory space, and a switch from SIMD to MIMD mode is performed by sending a jump to the appropriate PE MIMD instruction address located in the PE main memory space.

The SIMD instruction broadcast mechanism can also be utilized for *barrier synchronization* [LuB80] of MIMD programs. Assume a program uses a single MC group, and requires the PEs to synchronize R times. First, the MC enables all its PEs by writing an appropriate mask to the Fetch Unit Mask Register. Then it instructs the Fetch Unit Controller to enqueue R arbitrary data words, and starts its PEs which begin to execute their MIMD program. If the PEs need to synchronize (e.g., before a network transfer), they issue a read instruction to access a location in the SIMD instruction space. Because the hardware in the PEs treats SIMD instruction fetches and data reads the same way, the PEs will be allowed to proceed only after all PEs have read from SIMD space. Thus, the PEs are synchronized. The R synchronizations require R data fetches from the SIMD space. Thus, the Fetch Unit Queue is empty when the MIMD program completes, and subsequent SIMD programs are not affected by this use of the SIMD instruction broadcast mechanism.

In order to make comparisons of the speed of the PASM prototype relative to other machines and to compare the relative speeds of SIMD and MIMD instruction fetches, the actual raw performance of PASM in SIMD and MIMD mode was measured on the prototype and is illustrated in Table 1 in MIPS (millions of integer instructions per second) for two different types of instructions. The difference in speed between SIMD and MIMD modes can be attributed to two factors. SIMD instructions are fetched from the Fetch Unit Queue in the MC, and the queue can deliver data with one less wait state than can the PEs' main memories. In addition, PEs' main memories are implemented with dynamic memories. While care was taken in the hardware design that all refresh operations occur simultaneously in all PEs, and are performed invisible to the PE CPU, some delay is still possible. No such delay occurs during SIMD instruction fetches because the Fetch Unit queue is implemented with static RAM components. Measurements were made with repeated blocks of straight line code which were large enough to make the loop control overlap insignificant.

4. Matrix Multiplication Algorithms Used

The parallel matrix multiplication algorithm used here had $O(n^3/p)$ time and space complexity for multiplying two n×n matrices employing p PEs. Figure 2 shows an $O(n^3)$ time and space complexity serial algorithm. This particular algorithm is provided to illustrate the ordering of multiplications as they are done in the parallel version of Figure 3. Figure 4 demonstrates the progress of the serial algorithm for n=4. The two data-flow graphs illustrate what occurs during the first two iterations of the second *j* loop of Figure 3. The *i* loop of the

Table 1: Prototype raw performance.

Mode	Operation	Processing Rate
SIMD	16-bit Reg.-to-Reg. add	22 MIPS
MIMD	16-bit Reg.-to-Reg. add	18 MIPS
SIMD	16-bit Reg.-to-Mem. add	6.4 MIPS
MIMD	16-bit Reg.-to-Mem. add	6.0 MIPS

serial algorithm simulates the PE number in the parallel algorithm. The calculation of ((i+j) mod n) in the serial version allows the rows of the B matrix to be stepped through as the *j* loop proceeds with the initial B matrix row number being *i*. The serial algorithm used in the measurements on PASM, however, was optimized in order to permit accurate evaluation of speed-up, and therefore did not perform multiplies in this columnar manner. Rather, it followed a more straightforward row-column order.

```
for i=0 to n−1 do
   for j=0 to n−1 do
      $c_{i,j}$=0;
for i=0 to n−1 do
   for j=0 to n−1 do
      for k=0 to n−1 do
         $c_{k,i} = c_{k,i} + a_{k,((i+j) \bmod n)}\ b_{((i+j) \bmod n),i}$;
```

Figure 2: Serial matrix multiplication algorithm.

```
for all i, 0≦i≦n−1, do
   for v=0 to (n/p)−1 do
      π(v)=v;
      for j=0 to n−1 do
         $c_{j,v} = 0$;
   for j=0 to n−1 do
      for v=0 to (n/p)−1 do
         for k=0 to n−1 do begin
            $c_{k,v} = c_{k,v} + a_{k,\pi(v)}\ b_{((i(n/p)+v+j) \bmod n),v}$;
      for v=1 to (n/p)−1 do
         [change the pointer to column v−1 of the A
         matrix to point to column v*]
         π(v−1)=π(v);
      for k=0 to n−1 do
         send $a_{k,\pi(0)}$ to PE (i−1) mod p;
         receive a value and move it into $a_{k,\pi((n/p)-1)}$;
```

Figure 3: Parallel matrix multiplication algorithm.

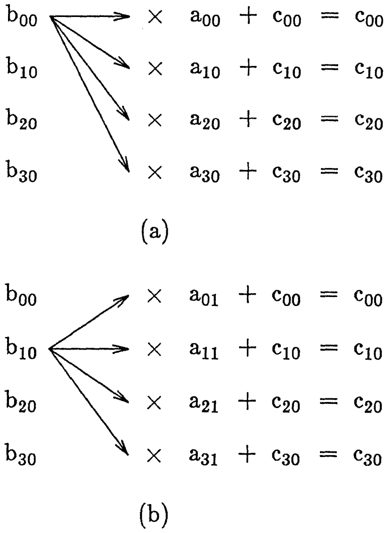

b_{00} × a_{00} + c_{00} = c_{00}
b_{10} × a_{10} + c_{10} = c_{10}
b_{20} × a_{20} + c_{20} = c_{20}
b_{30} × a_{30} + c_{30} = c_{30}
(a)
b_{00} × a_{01} + c_{00} = c_{00}
b_{10} × a_{11} + c_{10} = c_{10}
b_{20} × a_{21} + c_{20} = c_{20}
b_{30} × a_{31} + c_{30} = c_{30}
(b)

Figure 4: Two iterations of the serial algorithm for n=4. (a) i=0, j=0, 0≦k≦3. (b) i=0, j=1, 0≦k≦3.

PE 0			•••	PE 3		
a_{00} a_{01}	b_{00} b_{01}	c_{00} c_{01}		a_{06} a_{07}	b_{06} b_{07}	c_{06} c_{07}
a_{10} a_{11}	b_{10} b_{11}	c_{10} c_{11}		a_{16} a_{17}	b_{16} b_{17}	c_{16} c_{17}
a_{20} a_{21}	b_{20} b_{21}	c_{20} c_{21}		a_{26} a_{27}	b_{26} b_{27}	c_{26} c_{27}
a_{30} a_{31}	b_{30} b_{31}	c_{30} c_{31}	•••	a_{36} a_{37}	b_{36} b_{37}	c_{36} c_{37}
a_{40} a_{41}	b_{40} b_{41}	c_{40} c_{41}		a_{46} a_{47}	b_{46} b_{47}	c_{46} c_{47}
a_{50} a_{51}	b_{50} b_{51}	c_{50} c_{51}		a_{56} a_{57}	b_{56} b_{57}	c_{56} c_{57}
a_{60} a_{61}	b_{60} b_{61}	c_{60} c_{61}		a_{66} a_{67}	b_{66} b_{67}	c_{66} c_{67}
a_{70} a_{71}	b_{70} b_{71}	c_{70} c_{71}		a_{76} a_{77}	b_{76} b_{77}	c_{76} c_{77}

Figure 5: Data Layout for n=8, p=4.

In the parallel algorithm, the outer *for all* loop represents iteration across space rather than time. Each PE contains n/p adjacent columns of each matrix as shown in Figure 5. Within each PE these columns are numbered from 0 to (n/p)−1 as shown in the algorithm of Figure 3. π is a vector of indices to the n/p columns of A (in the actual implementation address pointers are used for efficiency). This layout is similar to that used by Su and Thakore in their experiments for the SM3 system [SuT87]. Using the *for v* loop, each of these adjacent columns is stepped through by each PE in sequence, and each PE appears as if it has n/p virtual PEs within it. The virtual PE number is then defined as (n/p)i+v. Thus, the row subscript of B is calculated by replacing *i* in Figure 2 with this virtual PE number. Data movement internal to each PE involves only a pointer adjustment. Only on the boundaries of the A matrix (i.e. the highest and lowest numbered columns of each PE) is the inter-PE network employed.

This particular algorithm was chosen over a more standard parallel matrix multiplication algorithm (e.g., see Stone [Sto80]) for several reasons. First, if a broadcast approach is used to distribute the "a" coefficients to the PEs, p network set-up cycles are incurred in addition to n^2 network transfer cycles (in the course of the algorithm each PE will have to broadcast its A matrix values (hence p settings) and the whole A matrix will have to be broadcast (hence n^2 transfers)). In the chosen algorithm, the network remains in one configuration (i.e., PE i connected to PE (i−1) mod p), thus eliminating any recurring network set-up costs, while not incurring any additional network transfer costs. Also, this algorithm facilitates a columnar data format which was preferable for several reasons. First, because all matrices are stored in columnar format, B×A may be calculated as well as A×B without rearrangement of the data. Second, each matrix

* This effectively rotates all internal columns of the A matrix to the left without destroying the data in column 0 of the PE, or actually moving the data.

may be used in subsequent multiplications without reformatting. Data uniformity is also desirable to facilitate parallel I/O transfers of large data sets from secondary memory.

What follows is a semantic description of the progress of the algorithm. During each of the n^2/p iterations of the innermost loop of the algorithm shown in Figure 3, each of the elements of the columns of the A matrix is multiplied by an element of the B matrix. (Note that due to the columnar storage, the column of the B matrix matches the *internal* column number of the A matrix. The absolute row number of B will match the absolute column number of the A matrix.) This value is then added to an element of the C matrix. Therefore, there is a total of n multiplications and additions in the inner loop with this loop being executed n/p times. In the final k loop, the columns of the A matrix are shifted one column to the left. Within each PE, this transfer involves a single memory move, because a pointer to the entire column is changed rather than moving its elements. However, for the lowest numbered column of each PE, the transfer employs the interconnection network. This column is transferred through the network and stored in the highest numbered column of PE ((i−1) mod p). The data received through the network is placed in the PEs memory as its highest numbered column. This transfer requires n network word operations (one for each element of the column). This procedure is repeated until all of the columns of the A matrix have been through each of the (n/p) positions of each PE for a total of n^2 network word transfer operation times incurred. During each of these elemental time periods, p values are exchanged.

Consider the time required for index calculation. The constant ix(n/p) was pre-calculated and placed in the programs data segment because it was constant in each PE for a given value of n and p. Also, the j+v operation involved in the B matrix row calculation was done outside the k loop and therefore only contributes O(n) time complexity per PE. The calculation of the A and C matrix row indices was done with the MC68000's auto-increment mode. Due to the pipelined structure of the MC68000 this does not add any extra execution time to the non-autoincrement mode. Therefore, the index calculation, as a separate component of the execution, time is not significant.

The current implementation of the network in PASM supports 8-bit data transfers. Because these experiments involved 16-bit data, each element transfer required two shift operations (one for transmitting and one for receiving), an OR operation, and two network operations. Because no DMA block transfers were possible given the current implementation of PASM, each column transfer required n single-element transfers for a total of 2n network operations per column.

Being circuit switched, setting up a path in the PASM prototype network is a time consuming operation. However, in this algorithm only a single path set-up is required, (i.e. PE i always sends to PE (i−1) mod p). Thus the measurements made do not reflect any significant influence from network reconfiguration overhead. Hence, there were $2n^2$ network accesses, n^3/p multiplications, and n^3/p additions required. This resulted in a $O(n^3/p)$ growth in execution time.

5. Implementations of the Algorithm

Three variations of the parallel algorithm, as well as an efficient serial version, were programmed in MC68000 assembly language for execution on the PASM prototype. The parallel versions included a pure SIMD, a pure MIMD, and a hybrid S/MIMD version. These three programs may be executed on 4, 8, or 16 processors simply by changing variables embedded in their data sections.

5.1. SIMD

The SIMD version executes all looping and control flow instructions in the MCs. Arithmetic, data movement, and index calculation instructions are executed on the PEs in SIMD mode. The PE instruction stream is obtained through the MC's Fetch Unit Queue and is executed synchronously on all PEs.

In PASM, the network appears to the PEs as two memory locations (transmit and receive registers). Network transfers are made directly to the transfer registers using memory-to-memory move instructions.

For several reasons, the SIMD version appeared to be the most natural choice for implementation. First, in the matrix multiplication algorithm used all PEs are always enabled, thus eliminating the need for enabling and disabling the PEs. Second, the implicit synchronization inherent in SIMD mode allowed the network transfer operations to be carried out in a straightforward fashion requiring only two memory-to-memory move instructions. Third, the only data-dependent portion of the algorithm is the actual multiplication instruction, which has a variable execution length due to its microcoded implementation in the MC68000. A final advantage of the SIMD version is due to the use of a FIFO queue in the Fetch Unit of the MCs. Because this queue buffers instructions being sent to the PEs, the execution of SIMD instructions by the PEs can be overlapped with the execution of control flow instructions by the MCs.

In addition to these conceptual factors involved in the SIMD version, there are some factors that were present due to the implementation of the PASM prototype. First, instructions may be accessed more quickly from the Fetch Unit Queue than from the PEs main memory. This is due to the use of faster memory technology in the queue. Also, the overlap of the control flow instructions with PE instructions is only present if the queue remains non-empty. In other words, the PEs can only proceed if the MCs supply instructions faster than the PEs can remove them from the queue.

5.2. MIMD

The second version was a pure MIMD program in which the MCs were only used for initiating the PE programs. The PEs executed all instructions asynchronously including all network, control flow, and arithmetic operations. Although the network hardware prevents overwriting of old data in the transfer register, the asynchronous network operations necessitated polling of the network buffer in order to determine whether it was ready to accept new data. After transmission, the network buffer must be polled to assure that the data is valid before a receive operation can be completed.

The major advantage of the MIMD version was rooted in the variation of the execution time of the MC68000 multiply instruction. Multiply or divide instructions require an amount of time which is related to the number of 1's in the binary representation of one operand. Assume an algorithm is executed on K PEs, each PE executes J instructions, and instruction j on PE k takes time τ_{jk}. Then the total execution time in SIMD mode (τ_{SIMD}) is the sum of the worst case times for each instruction as given by:

$$\tau_{\mathrm{SIMD}} = \sum_{j=1}^{J} \max_{k=0}^{K} \tau_{jk}$$

In MIMD mode each PE proceeds independently, and therefore the execution time (τ_{MIMD}) is the worst case sum of instruction execution times as given by:

$$\tau_{\mathrm{MIMD}} = \max_{k=0}^{K} \sum_{j=1}^{J} \tau_{jk}$$

In general, $\tau_{\mathrm{MIMD}} \leqq \tau_{\mathrm{SIMD}}$.

5.3. S/MIMD

The hybrid S/MIMD algorithm was developed to take advantage of the fast barrier synchronization mechanism described in Section 3 and to exploit the execution time advantage of the MIMD program (i.e. decoupling at low cost). In this version, the main program was the same as in the MIMD case. The difference was in the method of determining whether the network was ready to accept a transfer operation. Rather than polling the network buffer, barrier synchronization was used to allow network operations to be carried out as simple memory-to-memory move operations as in the SIMD version. This lowered the amount of network overhead to a level comparable but slightly greater than the SIMD version due to the mode switching time. The other advantages of SIMD mode (i.e., faster instruction fetch and control flow instruction overlap) could not be realized in this version.

6. Experiments Performed

Experiments were performed on n×n matrices and measurements were made of the execution times for n= 4, 8, 16, 64, 128, and 256. The algorithm was implemented for SIMD, MIMD, and S/MIMD mode and was run on p = 4, 8 and 16 PEs. All operations were 16-bit unsigned integer operations and overflow was ignored. To allow for varying machine and problem size, loops were utilized wherever possible.

To measure the amount of asynchronous execution necessary to yield better performance by the hybrid version over the SIMD version, the number of multiplies in each innermost loop of the algorithm was made to be a dependent variable. These multiplies were added as straight line code in order to prevent skewing of execution time data due to control flow overlap. The multiplies were added to study the effect on the total execution time and did not affect the values in the C matrix. Let T_{SIMD} and $T_{\mathrm{S/MIMD}}$ be the total execution time for the SIMD and S/MIMD programs respectively. The performance of each of the components of the execution time was measured at points corresponding to quantities of inner loop multiplications where:

$$T_{\mathrm{SIMD}} < T_{\mathrm{S/MIMD}},$$
$$T_{\mathrm{SIMD}} = T_{\mathrm{S/MIMD}}, \text{ and}$$
$$T_{\mathrm{SIMD}} > T_{\mathrm{S/MIMD}}.$$

Measurements were made with the internal system timers (MC68230). Experiments were performed for each version with the identity matrix in A and random data in B. While the value of the multiplier used in the MC68000 multiplication instruction affects the execution time, the data value of the multiplicand has no effect. Therefore, the elements of the A matrix, which were always used as the multiplicand could be chosen as the identity matrix without affecting program performance. By using the identity matrix, matrix multiplication results could be easily verified, thereby simplifying the debugging process. Random data, produced from a uniformly distributed random number generator, was chosen for these experiments in order to represent the average case, and the same data sets were used on all versions of the algorithm with the same value of n and p.

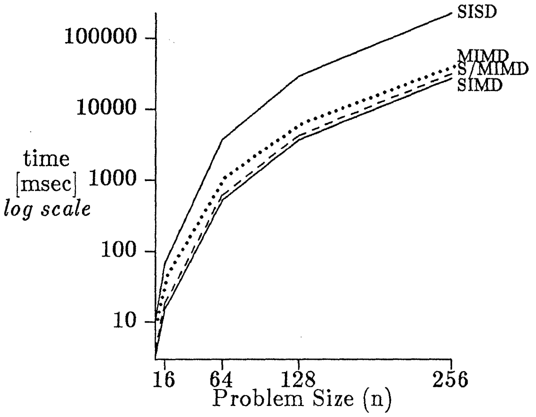


Figure 6: Execution time vs. problem size for p=8 and one multiply per inner loop.

7. Speed-up & Overall Comparison

Figure 6 illustrates execution time of matrix multiplication vs. problem size observed in the parallel versions of the algorithm for p=8. The difference between the SISD time and that of the parallel versions represents an improvement by a factor of approximately p.

Although not readily apparent in the graph, it should be noted that $T_{\mathrm{MIMD}}/T_{\mathrm{S/MIMD}}$ decreases as n increases. The only difference between these two versions is attributed to the contribution to the execution time of communication. Note that for p fixed, and small n (e.g. n=8), the time complexity of the multiplications is $\frac{n^3}{p}$ or $\frac{n^2\,(8)}{8} = n^2$. This is the same order of contribution as communication. Hence, for small n, the $O(n^2)$ communication contribution dominates the $O(n^3)$ arithmetic. However, for larger n, the $O(n^3)$ component ultimately dominates and all three curves converge.

The third aspect of this graph is the apparent advantage of the SIMD version over the S/MIMD version. The difference is caused by the ability of the MCs to execute control flow in parallel with arithmetic. However, the S/MIMD version has the potential for better performance due to the decoupling effect associated with MIMD execution of data-dependent execution time operations. In order to determine the point where these graphs cross, however, experiments were conducted which added more data-dependent instructions in a controlled way.

8. Execution Time vs. Number of Variable Length Operations

To determine the amount of asynchronous execution needed to achieve a benefit when executing a portion of a computation asynchronously in MIMD mode, additional multiplication operations were added to the innermost loop of the algorithm. Figure 7 plots total execution time for SIMD and S/MIMD programs with added multiplications vs. the number of added multiply instructions for n=64 and p=4 with random data. The lines plotted include 3 different points with the number of multiplications ranging from 13 to 15.

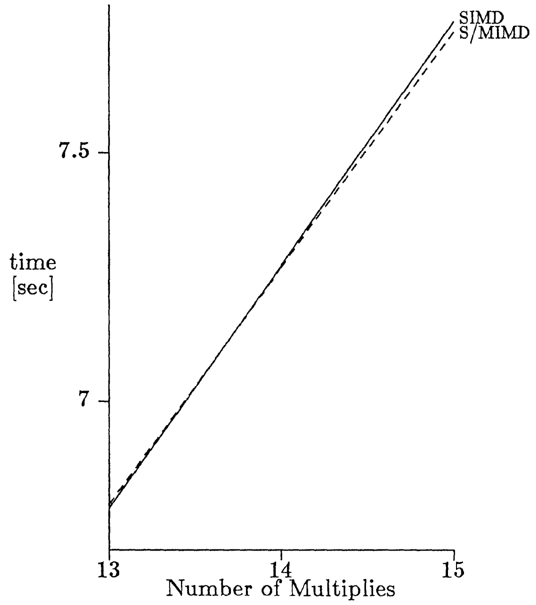


Figure 7: Execution time vs. number of inner loop multiplications for n=64 and p=4.

These lines are disjoint at the endpoints with the SIMD version being faster for small numbers of added multiplies and S/MIMD being faster as the number of added multiplies is increased. The point at which $T_{SIMD} = T_{S/MIMD}$ was with approximately fourteen added multiplications. This was due to the increase in execution efficiency when the multiplications were executed asynchronously. i.e., fewer processors were idle while waiting for all multiplications to complete.

9. Contributions to Execution Time

To further demonstrate that the execution time advantage was manifested in the multiplication instruction execution time, the contributions of the total execution time of the hybrid and SIMD programs were broken down and plotted. Figures 8, 9, and 10 contain plots of execution time vs. problem size at each of the endpoints and at the crossover point of Figure 7.

The times shown are broken down into: (i) multiplication time, (ii) communication time, and (iii) other contributions such as time for clearing the C matrix and shifting pointers for internal data movement. Multiplication and communication times include related address calculation operations. The multiplication time also includes the addition operation required to add the calculated value to the proper C matrix element. Figure 8 shows clearly that as problem size increases the time required for the multiplications increases faster than the communication time. This was mainly due to to the difference in the order of the communication time and the multiplication time (i.e. $O(n^2)$ vs. $O(n^3/p)$). Due to this difference in time complexity, the time required for the multiplication instructions becomes the largest component of execution time, even without the added multiplication instructions. The S/MIMD program, however, does not execute faster than the SIMD version due to both the control unit instruction overlap and the faster memory access time of the Fetch Unit Queue unless extra data-dependent instructions are

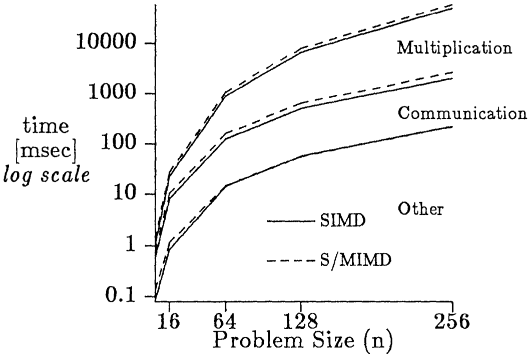


Figure 8: Contributions to execution time for matrix multiplication with one multiply per inner loop and p=4.

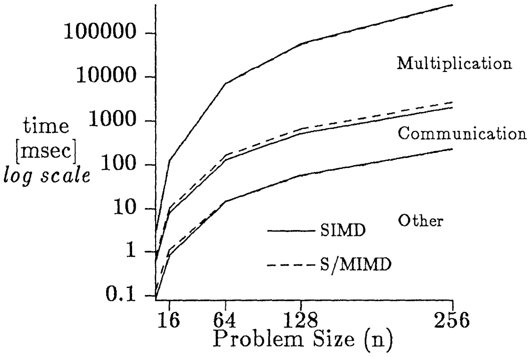


Figure 9: Contributions to execution time for matrix multiplication with 14 multiplies per inner loop and p=4.

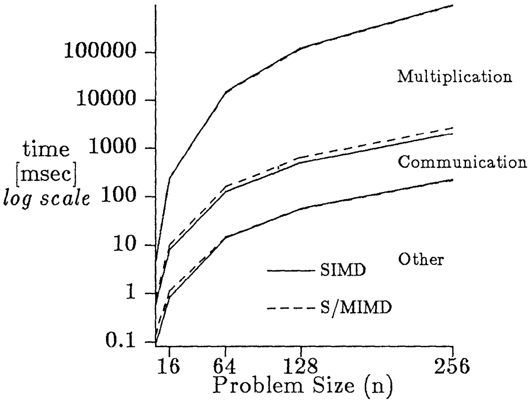


Figure 10: Contributions to execution time for matrix multiplication with 30 multiplies per inner loop and p=4.

added.

In Figure 9, the execution times are equal at n=64. With the total time broken down, it is apparent that the matrix multiplication times are close for all values of n, and when n=64 the matrix multiplication time is less in the S/MIMD program than in the SIMD program. However, the matrix multiplication time was the same because the communication time in the S/MIMD version was slightly more than in the SIMD version. Also, it should be noted that this effect would be greater if the constant value representing the instruction fetch time advantage were removed.

Figure 10 demonstrates the advantage provided by the asynchronous multiplication instructions when enough were added to make the other effects diminish in importance. In this version with 30 added multiplications per inner loop the S/MIMD version is faster for the larger values of n and this difference increases with n.

10. Efficiency vs. Problem Size

Figure 11 plots efficiency vs. problem size for the three modes of computation possible on PASM with p=4 as well as the serial case where efficiency is defined as:

$$E = \frac{T_{serial}}{T_{parallel} \times p}$$

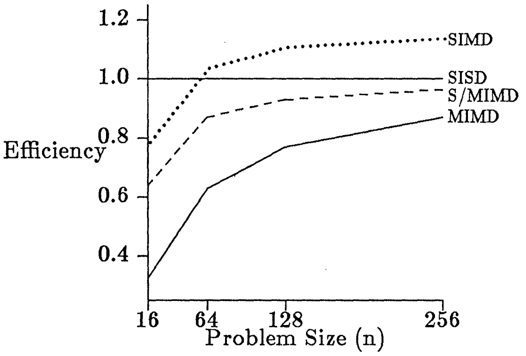


Figure 11: Efficiency vs. problem size for p=4 and one multiply per inner loop.

The efficiency of the S/MIMD and MIMD versions increased with the problem size, and never reaches or exceeds unity. The reason for the increasing efficiency can be accounted for by the fact that the quantity of communication overhead increases as $O(n^2)$, and the computation increases as $O(n^3/p)$. The best efficiency was 96% for the S/MIMD version and 87% for MIMD version (for n=256 and no added multiplies). The MIMD efficiency was lower due to the extra overhead required for the MIMD communication.

The SIMD version, however, was not only more efficient than the MIMD and S/MIMD versions, but was able to achieve an efficiency greater than unity when compared only to the number of PEs employed. This difference can be attributed to the ability of the PEs to do computation while the MCs are doing looping and other control operations. If the queue can remain non-empty and non-full at all times, it should be possible to eliminate all of the time required for the control operations. Because this amount increases with n, it is not surprising that the benefit also increases with n. This amount of benefit is related to the the ratio of control operations versus computation and communication operations. This does, however, demonstrate that the overlap of control flow and computation is possible and does provide some efficiency benefits — especially for applications that strongly exhibit a large quantity of control flow operations that can be performed on the MCs. This effect was predicted earlier by Kuehn et al in [KuS86].

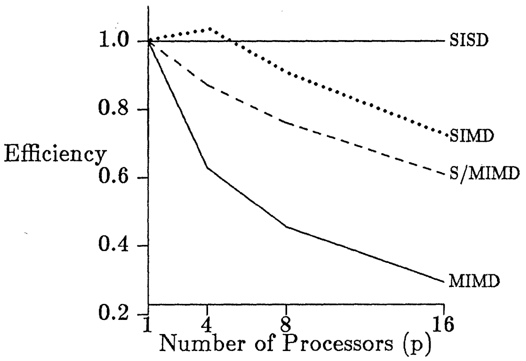


Figure 12: Efficiency vs. number of processors for n=64 and one multiply per inner loop.

11. Efficiency vs. Number of PEs

Figure 12 shows how efficiency drops as the number of processors utilized increases. This drop in efficiency is due to several factors. First, the value of n/p drops as p increases representing a decrease in the amount of computation done by each processor. While this does allow better parallelization of the algorithm, it makes the time consumed by inter-processor communication and other factors not present in the serial version become more significant compared to the time required by the computation portion of the algorithm.

12. Summary

Experiments designed to examine the tradeoffs among the SIMD, SISD, MIMD, and MIMD with barrier synchronized modes on the PASM parallel processing system prototype were described. In particular, the effects of instructions with data dependent execution times were considered. Tests were coded and executed on the prototype. Runtimes for different numbers of multiplies, numbers of processors, array sizes, and modes of parallelism were collected. This data was evaluated and discussed, analyzing the effects of the various parameters in the tests.

The experiments presented used an actual parallel system and pointed out some of the trade-offs among these modes of parallelism. Experiments such as these on working prototypes are important in order to begin to learn how to effectively harness the power of parallel processing.

Acknowledgements: The authors of this paper acknowledge many useful discussions with Pierre Pero, Tom Pusateri, Ed Bronson, Henry Dietz, Wayne Nation, and the other members of the PASM working group.

FLEXIBLY COUPLED MULTIPROCESSORS FOR IMAGE PROCESSING*

M. H. Sunwoo and J. K. Aggarwal

Computer and Vision Research Center
The University of Texas at Austin
Austin, TX 78712

ABSTRACT

There exist two main schemes for data sharing among processing elements in multiprocessors: message passing in loosely coupled multiprocessors and shared memory in tightly coupled multiprocessors. However, the former has communication overhead and the latter has shared memory contention. In this paper, two *Flexibly* (*Tightly/Loosely*) Coupled Multiprocessors (FCMs) for image processing are proposed in order to alleviate these disadvantages. A *variable space* memory scheme in which a set of adjacent memory modules can be merged by a dynamically partitionable bus, is proposed to achieve the FCMs. These architectures are quantitatively analyzed and simulated on the Intel's Personal SuperComputer (iPSC), a hypercube multiprocessor. Parallel algorithms for region labeling and median filtering are simulated on the proposed architectures. The performance of the FCMs shows remarkable improvement over the existing hypercube machine.

1. INTRODUCTION

In existing multiprocessors, there are mainly two methods for data sharing among processing elements (PEs). The first one is the message passing scheme in loosely coupled multiprocessors, and the second one is the shared memory scheme in tightly coupled multiprocessors. The Cosmic Cube [1], iPSC, and Ncube are examples of loosely coupled multiprocessors and the PUMPS [2], PASM [3], and Ultracomputer [4] are examples of tightly coupled multiprocessors.

However, both types of multiprocessors have major limiting factors towards speed-up and expansion. Loosely coupled multiprocessors have a communication overhead disadvantage due to message passing, whereas tightly coupled multiprocessors have a shared memory contention problem. Another limiting factor which is usually neglected, but is important, is the time to load input data and to unload output data. In many researches, it is often assumed that the data to be processed are already in processing structures. In other words, the time to load and unload data is ignored. However, it is not negligible because input and output may take longer than computation time in some cases on existing multiprocessors.

In order to alleviate these disadvantages (communication overhead, memory contention, and data loading and unloading overhead), and to achieve higher speed compared with that of existing multiprocessors, two *Flexibly* (*Tightly/Loosely*) Coupled Multiprocessors (FCMs) with a *variable space* memory scheme in which a set of memory modules can be merged by a dynamically partitionable bus are proposed.

Most image processing tasks require a considerable amount of computation [5] which results from the large amount of data to be processed. The throughput of the system must be very high to meet these computational demands. Specifically, the throughput should be much higher for the real-time applications where a sequence of image frames is required to be processed.

Parallel architectures for image processing can be classified into two groups in terms of functionality [6]: general purpose architectures and special purpose architectures. General purpose architectures are flexible and programmable for performing a broad range of applications. However, the desired performance often cannot be achieved due to the communication overhead, memory contention for the exchange of data and control information, and the complicated control strategies. Special purpose architectures can achieve better performance at the cost of flexibility and versatility.

The characteristics inherent to image processing which warrant the parallel processing approach are now discussed. First, a whole image processing task can be decomposed into a set of subtasks which are sequentially applied to an entire image domain. For example, the task of object recognition is composed of several subtasks which include preprocessing, boundary detection, region labeling, normalization and finally matching. Second, a sequence of images is usually processed for real-time image processing. These *temporal characteristics* can be exploited by pipelining (temporal parallelism). Third, the entire image is subjected to the same operation which is performed pixel by pixel (e.g., histogram calculation) or region by region (e.g., median filtering). This *spatial characteristic*, i.e., the *spatial locality* of the image data, suggests that a whole image may be partitioned into subimages which can be processed in parallel by a set of PEs. The spatial characteristic can be exploited by array processing or multiprocessing (spatial parallelism).

The rest of this paper is organized as follows. In the next section, two FCMs for image processing are introduced. These architectures are then quantitatively analyzed. The applications for SIMD (single instruction stream - multiple data stream) algorithms and pipelined pseudoparallel algorithms are described. The analytical model of the hypercube multiprocessor is described for comparison with the FCMs. In Section 3, the applications of the proposed architectures for image processing are described. Parallel algorithms for region labeling and median filtering are implemented and simulated on the hypercube multiprocessor. In Section 4, the experimental results and a discussion of the implementation and the simulation are presented for performance comparison. The FCMs are MIMD (multiple instruction stream - multiple data stream) machines which can exploit both types of parallelism for image processing. Some features of the FCMs proposed in this paper have several similarities to those of other architectures such as the VS [7], the SM3 [8] and the MP/C [9]. However, there exist several significant differences. Section 5 contains a discussion of the similarities and differences, and also some concluding remarks.

2. TWO FLEXIBLY COUPLED MULTIPROCESSORS (FCMs)

In this section, we describe the new architectures, and compare their features with those of the existing hypercube architecture. In general, the term *tightly coupled* is used for a multiprocessor which has shared memories, while the term *loosely coupled* is for a multiprocessor which has no shared memory. In contrast, as will be seen below, both terms may be used to describe the proposed architectures.

*This research was supported in part by IBM.

References

[AnA87] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb, "The Warp computer: architecture, implementation, and performance," *IEEE Transactions on Computers,* Vol. C-36, December 1987, pp. 1523-1538.

[Cal84] D. A. Calahan, "Influence of task granularity on vector multiprocessor performance," *1984 International Conference on Parallel Processing,* August 1984, pp. 278-284.

[CrG85] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," *1985 International Conference on Parallel Processing,* August 1985, pp. 531-540.

[FiC87] S. A. Fineberg, T. L. Casavant, and T. Schwederski, "Mixed-mode computing with the PASM prototype," *25th Allerton Conference on Control, Communications and Computing,* September 1987, pp. 258-267.

[GeS87] E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel processing: the Cm* experience,* Digital Press, Bedford, MA, 1987.

[Han88] F.B. Hanson, "Vector multiprocessor implementation for computational stochastic dynamic programming," *IEEE Technical Committee on Distributed Processing Newsletter,* Vol. 10, 1988, (to appear).

[Hud88] P. Hudak, "Exploring parafunctional programming: separating the what from the how," *IEEE Software,* Vol. 5, January 1988, pp. 54-61.

[JaM86] W. Jalby and U. Meier, "Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system," *1986 International Conference on Parallel Processing,* August 1986, pp. 429-432.

[KuN88] J. G. Kuhl, J. J. Norton, and S. R. Sataluri, "A large-scale application of coarse-grained parallel and distributed processing," *IEEE Technical Committee on Distributed Processing Newsletter,* Vol. 10, 1988, (to appear).

[KuS86] J. T. Kuehn and H. J. Siegel, "Simulation based performance measures for SIMD/MIMD processing," in *Evaluation of Multicomputers for Image Processing,* L. Uhr, K. Preston, Jr., S. Levialdi, and M. J. B. Duff, eds., Academic Press, Orlando, FL, 1986, pp. 139-158.

[LuB80] S. F. Lundstrom and G. H. Barnes, "A controllable MIMD architecture," *1980 International Conference on Parallel Processing,* August 1980, pp. 165-173.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers,* Vol. C-30, December 1981, pp. 934-947.

[SiS87] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., 1987, pp. 387-407.

[Sto80] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture (second edition),* H. S. Stone, ed., Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.

[SuT87] S.Y W. Su and A. K. Thakore, "Matrix operations on a multicomputer system with switchable main memory modules and dynamic control," *IEEE Transactions on Computers,* Vol. C-36, December 1987, pp. 1467-1484.

2.1 The FCM Model I

2.1.1 Architecture

The *Flexibly* (*Tightly*/*Loosely*) Coupled Multiprocessor Model I (FCM I) is shown in Fig. 2.1.1. The FCM I consists of N PEs (PE_i), where $N = 2^n$, N memory modules (M_i), a control unit (CU) and a programmable I/O processor (IOP), where $0 \le i \le N-1$. PE_i is connected to the CU through the communication bus and to the M_i through the dynamically partitionable address and data bus (A/D BUS) with the partitionable arbiter described below. PE_i contains its own local memory for program and intermediate results. M_i is used only for data.

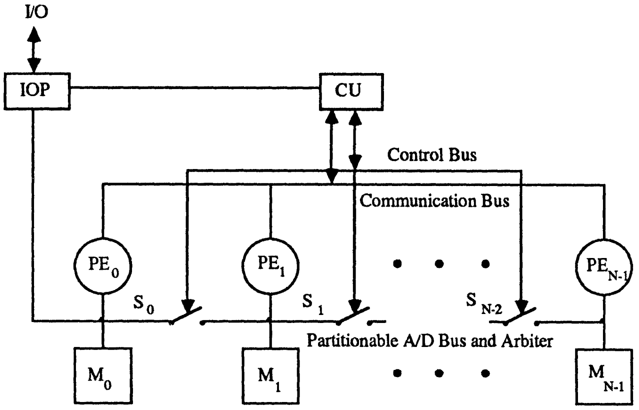


Fig. 2.1.1 The Flexibly Coupled Multiprocessor Model I (FCM I)

A set of switches, S_i, which can connect and disconnect the A/D Bus, is located between two memory modules, M_i and M_{i+1}. All switches in S_i are operated together (closed or opened together). The CU can handle each switch set S_i independently. The IOP is directly connected to this memory and the CU.

Any two successive memory modules (M_i, M_{i+1}) can be formed into one *contiguously addressable* memory module with S_i closed, in which case PE_i and PE_{i+1} can access the module through the arbiter one at a time. Moreover, a set of consecutive memory modules can become one contiguously addressable memory module with all switch sets between these modules closed. When all switch sets are closed, all memory modules become one contiguously addressable memory module. Thus, any PE, or the high speed IOP using the direct memory access (DMA) scheme, can access all memory modules. In this case, the FCM I becomes a *fully tightly* coupled multiprocessor. With all switch sets closed, if more than one PE tries to access the memory at the same time, memory contentions occur. To reduce memory contention, the CU opens some switch sets. If all switch sets are opened, then any PE (PE_i) can access its corresponding module (M_i) without memory contention.

Accordingly, when all switch sets are open, the variable space memory scheme becomes N memory modules. Therefore, each PE can access only its own memory module. Thus, the FCM I becomes a *fully loosely* coupled multiprocessor because there is no more shared memory. When some switch sets are closed and the other sets are open, the network is grouped into a set of disjoint partitions. If all sets, except S_i, are open, PE_i and PE_{i+1} are referred as *partially tightly* coupled, and other PEs are referred as *partially loosely* coupled. The variable space memory becomes N–1 disjoint memory modules. Accordingly, the network forms N–1 partitions. When all switch sets, except S_i, are closed, the variable space memory becomes two disjoint memory modules, and the network forms two partitions. Each partition, PE_j, $0 \le j \le i$, and PE_k, $i+1 \le k \le N-1$, is partially tightly coupled. The partitions are partially loosely coupled with respect to each other. Thus, the FCM I is a flexibly coupled multiprocessor.

To prevent memory contention in tightly coupled partitions, a partitionable arbiter within the A/D Bus is used between PEs and memory modules. One possible arbiter based on ripple carry logic is shown in Fig. 2.1.2 [10]. This arbiter can be modified to be a partitionable arbiter by using switches as shown. The lines G_i are for requests, P_i are for propagation of the requests from the previous stages, and C_i signal whether the request has been granted in the previous stages. The switches in the arbiter are operated in conjunction with the switch sets in the A/D Bus.

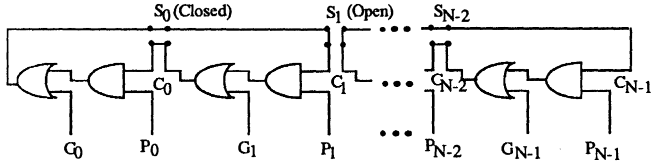


Fig. 2.1.2 The Partitionable Arbiter

2.1.2 The Variable Space Memory Scheme

For a more detailed observation, the connection scheme of PEs (PE_i, PE_{i+1}), memory modules (M_i, M_{i+1}), and a set of switches (S_i) is shown in Fig. 2.1.3. PE_i and PE_{i+1} are connected to M_i and M_{i+1}, respectively, via the partitionable A/D Bus. S_i is located between M_i and M_{i+1}.

The address space of each memory module is

$$K = 2^m \qquad (2.1.1).$$

Each module M_i, for $0 \le i \le N-1$, contains consecutive addresses from $i2^m$ to $(i+1)2^m-1$. Thus, m address lines $(a_0, \ldots, a_{m-1})$ are required for a module. Since there are N $(= 2^n)$ memory modules, the total address space is expressed by

$$N*K = 2^{(m+n)} \qquad (2.1.2).$$

The total number of address lines is $m + n$, namely, $(a_0, \ldots, a_{m+n-1})$. Thus, every item has a unique and absolute address so that it can be accessed by any PE or the IOP through the A/D Bus. The number of switches in a set is $m + n + d + a$, where d is the number of switches for the data bus, and a is the number of switches for the partitionable arbiter. The switches in a set can be operated in parallel by using the control bus, but any two sets may not be operated in parallel.

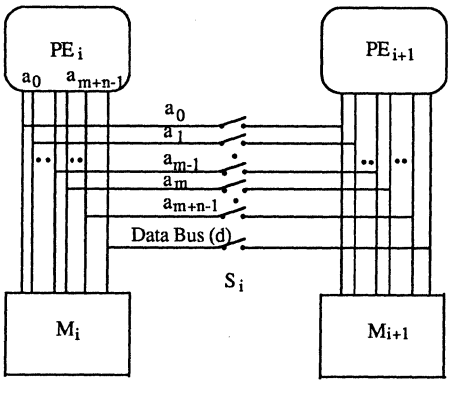


Fig. 2.1.3 The Connection Scheme of PEs and Memory Modules

In multistage interconnection networks, $\frac{N}{2}\log_2 N$ switching elements (2 X 2) are required, while only $N-1$ switching sets are required in the FCM I. The switches in the set S_i are simple switches, unlike the complicated switching elements with relatively complicated control strategy used in a multistage interconnection network [11].

2.1.3 Communications

There are three different types of communication: CU-to-PE communication (bi-directional), PE-to-PE communication and broadcast from the CU or from any PE. CU-to-PE communication can be achieved via the communication bus. PE-to-PE communication can be achieved via the communication bus or the variable space memory. The broadcast from the CU or from any PE can be realized by the communication bus in one cycle. For data sharing, the variable space memory can be used in tightly coupled partitions, and the communication bus can be used in loosely coupled partitions. Even in tightly coupled partitions, the communication bus can be used for message passing where messages are short (a few bytes).

The variable space memory is advantageous for exchange of large amounts of data, while the communication bus is advantageous for exchange of small amounts of data. In order to combine two data in M_i and M_{i+1}, only the switching time to close the switch set S_i is needed instead of message passing.

2.1.4 Control

The control scheme of the FCM I is simpler than the control of a multiprocessor which uses a multistage interconnection network. The role of the CU is to control PEs, all switch sets and the IOP.

An example of the control schemes for SIMD mode processing on the FCM I, configured as an MIMD machine, is described below.

i) The CU closes all switch sets and sends a signal to the IOP to load input data into a set of memory modules.

ii) The IOP can load input data by treating the collection of memory modules as a single module. After the IOP loads data, it sends the load completion signal to the CU.

iii) The CU opens all switches and broadcasts the task start signal to the PEs employed to execute their tasks.

iv) During execution, data sharing is achieved via the variable space memory and also via the communication bus. The CU mediates data sharing, when the variable space memory is used.

v) After each PE finishes its task, it sends the task completion signal to the CU.

vi) When the CU receives all task completion signals from the PEs employed, it closes all switch sets and sends a signal to the IOP to unload output data.

As described above, PE-to-PE communications and CU-to-PE communications are not required to load and unload data, which is in sharp contrast to the hypercube machine described in 2.3.2. Instead, only the switching time and control time are required for the FCM machines. The CU-to-PE communication is used for control, not for data movement. In other words, the communications of the CU mainly consist of the control signals for PEs, switches, and the IOP. The control and synchronization schemes are simple. The CU issues control signals infrequently. Hence, the CU and the communication bus will not cause bottlenecks. This will be discussed in greater detail in subsection 3.1.2 and Section 4. Since the size of input and output data can be predicted in image processing tasks, the IOP can load and unload data selectively. For more general tasks, the terminator which indicates the end of input data or output data can be used by the IOP and PEs. This type of procedure is well suited for SIMD algorithms, such as median filtering, convolution, edge detection, FFT (fast Fourier transform), region labeling, and so on.

2.1.5 Analytical Model for Image Processing

In the FCM I, the parallel image processing time mainly consists of six time components: input data acquisition time (t_{ac}) to digitize the image data from an input device, such as a camera and to store the digitized image into the variable space memory by the IOP, computation time (t_{cp}), merging time to take the boundary consistency problem between subimages into account (t_{mg}), switching time to close and open switches (t_{sw}), control time to exchange control signals (t_{cn}), and an output transfer time (t_{tf}) to unload data to an output device. The parallel processing overhead includes t_{mg}, t_{sw}, and t_{cn}.

t_{cp} and t_{mg} are functions of the size of the image (I) and the number of the PEs employed (N). t_{ac} and t_{tf} are functions of I and are independent of N. t_{sw} and t_{cn} are functions of N and are independent of I. The time components may be expressed as follows

$$t_{ac} = f_1(I), \qquad (2.1.3.a)$$

$$t_{cp} = f_2(I, N), \qquad (2.1.3.b)$$

$$t_{mg} = f_3(I, N), \qquad (2.1.3.c)$$

$$t_{sw} = f_4(N), \qquad (2.1.3.d)$$

$$t_{cn} = f_5(N), \qquad (2.1.3.e)$$

$$t_{tf} = f_6(I). \qquad (2.1.3.f).$$

The functions, f_2, f_3, f_4 and f_5, are very dependent on algorithms.

As discussed in the previous subsection, PE-to-PE communications are not needed for loading and unloading data. Only CU-to-PE communications are required for control.

Thus, the total time to process an image frame on the FCM I can be expressed by

$$T_I = t_{ac} + t_{cp} + t_{mg} + t_{sw} + t_{cn} + t_{tf} \qquad (2.1.4).$$

On a single PE, it may be represented by

$$T_s = t_{ac} + cNt_{cp} + t_{tf} \qquad (2.1.5)$$

where c is a constant.

Speed-up (SP_I) is defined as the ratio of the time on a single PE to that on the FCM I

$$SP_I = \frac{T_s}{T_I} = \frac{t_{ac} + cNt_{cp} + t_{tf}}{t_{ac} + t_{cp} + t_{mg} + t_{sw} + t_{cn} + t_{tf}} \qquad (2.1.6).$$

2.2 The FCM Model II with Buffering Capability

This model is useful for MSIMD (multiple single instruction stream - multiple data stream) mode processing, which is composed of a set of SIMD mode processing. It is also useful for pipelined pseudoparallel algorithms [12] discussed in 2.2.3 as well as for SIMD algorithms.

2.2.1 Architecture

The FCM with buffering capability (FCM II) is shown in Fig. 2.2.1. There are two memory sets, M_A and M_B. The set M_A consists of N memory modules $\{M_{A0}, M_{A1}, \cdots, M_{AN-1}\}$ and the set M_B consists of N memory modules $\{M_{B0}, M_{B1}, \cdots, M_{BN-1}\}$. The local memory in a PE can be used for storing program and intermediate results like in the FCM I. The IOP is connected to each memory set, with two ports for each set to load input data and unload output data from the modules, independently.

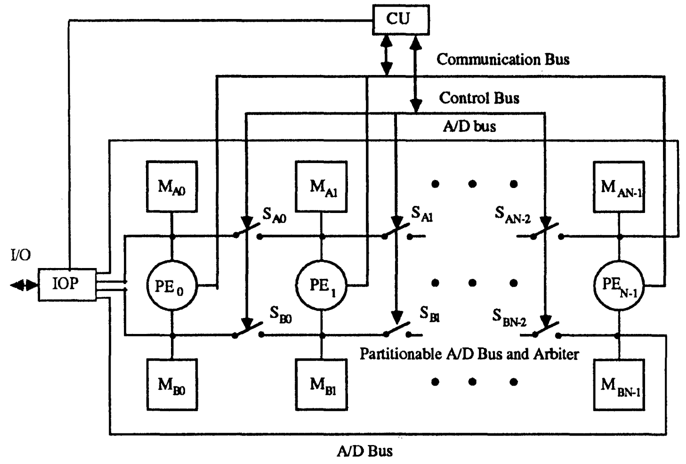


Fig. 2.2.1 The Flexibly Coupled Multiprocessor Model II (FCM II)

While PEs use M_A for processing, the IOP can unload output results from and load input data into M_B. When PEs start processing M_B, the IOP can unload output data from and load input data into M_A.

The role of the CU in the FCM II is similar to that in the FCM I. The CU has three functions: to control the upper and lower switch sets separately, to control PEs, and to control the IOP.

2.2.2 Analytical Model for Image Processing

While PEs use M_A, the CU sends a signal to the IOP to unload output data from M_B and load another input data into M_B, and vice versa. Thus t_{ac}, t_{tf}, and some portions of t_{sw} and t_{cn} for one set of data can be overlapped with t_{cp}, t_{mg}, and t_{sw} and t_{cn} for the other set of data. For example, during processing in M_A, the time for sending a signal to the IOP from the CU can be overlapped with computation time. Therefore, $t_{ac} + t_{tf} + \rho_1 t_{sw} + \rho_2 t_{cn}$ can be overlapped with $t_{cp} + t_{mg} + (1 - \rho_1)t_{sw} + (1 - \rho_2)t_{cn}$, where ρ_1 and ρ_2 are the overlapped portions of t_{sw} and t_{cn}.

The total time to process an image frame for the FCM II (T_{II}) is expressed as follows

$$T_{II} = t_{cp}+t_{mg}+(1-\rho_1)t_{sw}+(1-\rho_2)t_{cn} \,,$$
$$\text{if } t_{cp}+t_{mg}+(1-\rho_1)t_{sw}+(1-\rho_2)t_{cn} \geq t_{ac}+t_{tf}+\rho_1 t_{sw}+\rho_2 t_{cn} \qquad (2.2.1).$$

$$T_{II} = t_{ac}+t_{tf}+\rho_1 t_{sw}+\rho_2 t_{cn} \,,$$
$$\textit{otherwise}$$

Speed-up (SP_{II}) is expressed as follows

$$SP_{II} = \frac{T_s}{T_{II}} \qquad (2.2.2).$$

2.2.3 Pipelined Pseudoparallel Algorithms on the FCM II

Pipelined pseudoparallelism, where a serial algorithm is decomposed into a set of noninteractive independent subtasks so that parallelism can be used in each subtask level, is proposed in [12].

For an integrated computer vision system, pipelined pseudoparallelism is valuable, wherein temporal parallelism and spatial parallelism can be exploited. Spatial parallelism and temporal parallelism may be achieved by using multiprocessing and pipelining, respectively. Many computer vision tasks, such as pattern recognition and dynamic scene analysis, fall in the category of pipelined pseudoparallel algorithms.

Pipelined pseudoparallel algorithms can be implemented conveniently on the FCM II. The FCM II can be partioned into a set of various-size MIMD machines (or SISD, single instruction stream - single data stream, if the size is 1) by using the partitionable A/D Bus as described below.

For the sake of convenience, the FCM II with $N = 4$ is illustrated in Fig. 2.2.2. $\overline{S}_{Aj}$ and $\overline{S}_{Bj}$ denote that the switches are closed, and S_{Aj} and S_{Bj} denote the switches as being open. Suppose that a whole task g is decomposed into three subtasks (g_1, g_2, g_3), and g_1 needs two PEs (PE_0, PE_1) and each of g_2 and g_3 needs one PE (PE_2 and PE_3). For example, a pattern recognition task (g) consists of three subtasks which are preprocessing (g_1), feature extraction (g_2) and pattern classification (g_3). Assume that a sequence of images is to be processed. For simplicity, the steps for switching and control between the CU, PEs and the IOP are omitted.

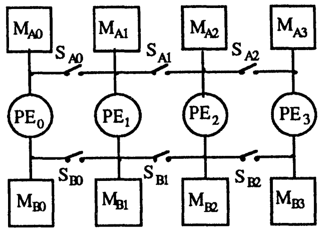


Fig. 2.2.2 The FCM II (N = 4)

In the first phase, PE_0 and PE_1 (g_1) process the image frame loaded by the IOP in M_{A0} and M_{A1} with S_{A0} and S_{A1}. After processing, PE_0 and PE_1 write output into M_{A0} and M_{A1}. In the second phase, PE_2 (g_2) processes the intermediate results in M_{A0} and M_{A1} with $\overline{S}_{A0}$ and $\overline{S}_{A1}$, and writes output into M_{A2}. In the third phase, PE_3 (g_3) processes the intermediate results in M_{A2} with $\overline{S}_{A2}$, and writes output into M_{A3}. Finally, the IOP accesses output results from M_{A3}. In other words, g_i reads input data from M_{Ai-1} (M_{Bi-1}), and writes output data into M_{Ai} (M_{Bi}). When i=0, input data is loaded from the IOP. The above steps are interleaved between two memory sets as described below.

i) While PE_0 and PE_1 (g_1) process the image frame loaded by the IOP in M_{A0} and M_{A1} with S_{A0}, PE_2 (g_2) processes the intermediate results generated by PE_0 and PE_1 in M_{B0} and M_{B1} with $\overline{S}_{B0}$ and $\overline{S}_{B1}$. Also, PE_3 (g_3) processes the intermediate results generated by PE_2 in M_{A2} with $\overline{S}_{A2}$. After processing, the processed frames of g_1, g_2 and g_3 are in (M_{A0} and M_{A1}), M_{B2}, and M_{A3}, respectively. The IOP can directly access the final results in M_{A3} with S_{A2}.

ii) Similarly, while PE_0 and PE_1 (g_1) process the image frame in M_{B0} and M_{B1}, PE_2 (g_2) accesses the previous output of g_1 in M_{A0} and M_{A1}, and write its output in M_{A2}. At the same time, PE_3 (g_3) accesses the previous output of g_2 in M_{B2}. The IOP can directly access the final results in M_{B3} with S_{B2}.

As described above, PEs can process the data in two memory sets M_A and M_B, alternately. And the PEs of the current subtask write their outputs into corresponding memory modules which are used by the PEs of the next subtask. Thus, even data transfer time between stages which causes overhead in pipelined schemes can be reduced. During processing, there are only two connection patterns between PEs and the variable space memory modules shown in Fig. 2.2.3. There is no communication overhead. For synchronization, the stage transition between subtasks must be controlled by the CU. The processing time per output is max $\{t_i\}$, where t_i is the processing time of the i-th stage. Any composition of subtasks in pipelined pseudoparallel algorithms, may be mapped into the FCM II without communication overhead.

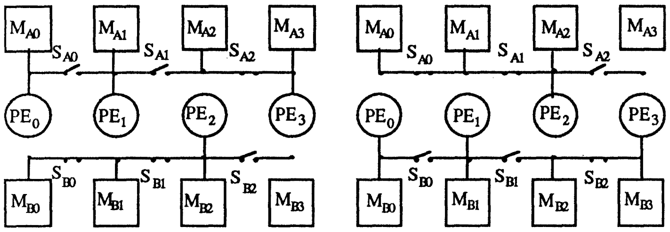


Fig. 2.2.3 Two Connection Patterns between PEs and Memory Modules

2.3 Hypercube Multiprocessor

In order to compare the performance of the FCMs with a hypercube multiprocessor, the architecture and the analytical model of a hypercube are briefly discussed.

2.3.1 Architecture

The system is composed of a controller and a hypercube structure which consists of $N = 2^n$ PEs. Each processor is connected to its n nearest neighboring PEs. The hypercube multiprocessor is a loosely coupled multiprocessor with no shared memory and no global synchronization. Thus, data sharing between PEs is achieved by message passing [1].

There are two different types of communication: controller-to-PE communication and PE-to-PE communication. The communication required to perform an image analysis task causes significant overhead which degrades performance [13].

2.3.2 Analytical Model for Image Processing

In the hypercube multiprocessor, the parallel image processing time mainly consists of six time components: input data acquisition time (t_{ac}), input data distribution time (t_{ds}) from the controller to PEs which may consist of controller-to-PE communications ($t_{ds_{cp}}$) and PE-to-PE communications ($t_{ds_{pp}}$), computation time (t_{cp}), collection time to gather local results in PEs by using PE-to-PE communications ($t_{cl_{pp}}$), merging time to take the boundary consistency between subimages into account (t_{mg}), collection time to send a whole result to the controller by using PE-to-controller communication ($t_{cl_{pc}}$), and output transfer time (t_{tf}). The parallel processing overhead includes $t_{ds_{cp}}$ $t_{ds_{pp}}$, t_{mg}, $t_{cl_{pp}}$, and $t_{cl_{pc}}$.

Each time component, except t_{ac} and t_{tf}, is a function of both the size of the image (I) and the number of the PEs employed (N). t_{ac} and t_{tf} are functions only of I. Therefore, the time components are expressed as follows

$$t_{ac} = h_1(I), \qquad (2.3.1.a)$$

$$t_{ds} = t_{ds_{cp}} + t_{ds_{pp}} = h_2(I, N), \qquad (2.3.1.b)$$

$$t_{cp} = h_3(I, N), \qquad (2.3.1.c)$$

$$t_{cl} = t_{cl_{pp}} + t_{cl_{pc}} = h_4(I, N), \qquad (2.3.1.d)$$

$$t_{mg} = h_5(I, N), \qquad (2.3.1.e)$$

$$t_{tf} = h_6(I). \qquad (2.3.1.f).$$

It was found that the functions can be represented as $h_i = C_{i0} + C_{i1}I_s + C_{i2}I_s{}^2$, where C_i's are constants, I_s is the size of a subimage $\left\lceil \frac{I}{N} \right\rceil$, and $2 \le i \le 4$ [14]. These functions are very dependent on algorithms.

The total time to process an image frame on the hypercube multiprocessor (T_h) is represented by

$$T_h = t_{ac} + t_{ds} + t_{cp} + t_{cl} + t_{mg} + t_{tf} \qquad (2.3.2).$$

Speed-up (SP_h) is defined as the ratio of the time on a single PE to that on the hypercube multiprocessor

$$SP_h = \frac{T_s}{T_h} = \frac{t_{ac} + cNt_{cp} + t_{tf}}{t_{ac} + t_{ds} + t_{cp} + t_{cl} + t_{mg} + t_{tf}} \qquad (2.3.3).$$

The time to load data into and unload data from the hypercube includes t_{ac}, t_{ds}, t_{ci} and t_{tf}. t_{ds} and t_{cl} consist of controller-to-PE communications and PE-to-PE communications which may significantly degrade performance. As N increases, communication overhead increases, and efficiency sharply decreases. In contrast, to load and unload data on the FCM I, only t_{ac}, t_{tf} and some portions of t_{sw} and t_{cn} are needed. Furthermore, to load and unload data on the FCM II, even t_{ac} and t_{tf} can be entirely overlapped, and hence only some portions of t_{sw} and t_{cn} are required.

3. APPLICATIONS TO IMAGE PROCESSING

We describe the simulations of some common image processing algorithms on the proposed architectures, and compare them with the implementation of these on the hypercube machine. We illustrate the advantages of the proposed models over the hypercube model for such algorithms. In the next section, we provide experimental results that demonstrate the advantages of the new architectures.

Region labeling and median filtering are chosen for performance evaluation. Median filtering is used for preprocessing in many image processing tasks. Region labeling is one of the basic operations in image processing. Once an image has been partitioned into regions, these regions can be studied, described and possibly identified. Applications where region labeling plays an integral part include cell classification, military target detection, parts inspection, object classification and character recognition.

3.1 Region Labeling

A parallel algorithm for region labeling has been implemented on the iPSC d5 (32 PEs) and is described elsewhere [13]. The tracking method of Agrawala and Kulkarni [15] has been modified to overcome some of the limitations of the original scheme, and the merging algorithms for parallel implementation have been developed [13].

Two major limiting factors for speed-up on the hypercube multiprocessor were discovered. One is the PE-to-PE communication overhead, the other is the controller-to-PE communication overhead. The same algorithm can be also implemented on the FCMs without communication overhead. Only switching time and control time contribute to the overhead as described below.

3.1.1 Parallel Algorithm

The parallel algorithm consists of three major operations; preprocessing, labeling, and merging. The original image is partitioned into N equal-size subimages, which are distributed to the PEs involved. Each PE is assigned a subimage. Each subimage is preprocessed in a raster scan manner, i.e., top to bottom, left to right, generating a reduced representation of the image, i.e., the transition point representation (TPR) [15]. The transition point pair (TPP) shown in Fig. 3.1.1 is composed of $(XL_i(k), XR_i(k))$, where $XL_i(k)$ and $XR_i(k)$, respectively, are the left point and the right point of the i-th region on scan line k. The TPP's are obtained by preprocessing. After preprocessing, labeling operation is performed by using a set of boundary continuation conditions [13]. The local results are combined recursively through the *pseudo* binary tree structure described in the next subsection. It is necessary to merge two adjacent subimage regions with different labels into one region with the same label. The merging is performed first across vertical subimage borders (*vertical merging*) and then across horizontal subimage borders (*horizontal merging*). Finally, one PE labels regions based on merged TPP's. Using the algorithm, any digital image represented in a 2-dimensional array can be labeled. Further details about algorithms are described in [13].

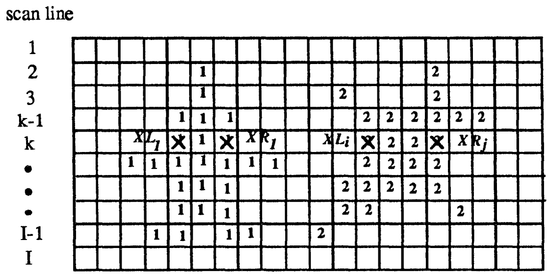


Fig. 3.1.1 Regions and TPP's for scan line *k*

3.1.2 Simulation on the FCM I

A. Pseudo Binary Tree

A pseudo binary tree (refer to Fig. 3.1.2 for a 4-level tree) is a binary tree structure which can be easily embedded into the hypercube topology such that a node in the hypercube may represent more than one node in a corresponding pseudo binary tree [14]. The pseudo binary tree is an efficient topology for distributing subimages and collecting the local results in the hypercube multiprocessors. The reason is that all PEs in the hypercube are utilized for the pseudo binary tree implementation while at most only half of PEs are utilized for a binary tree implementation. The pseudo binary tree can be embedded into not only the hypercube topology but also the FCM I and II.

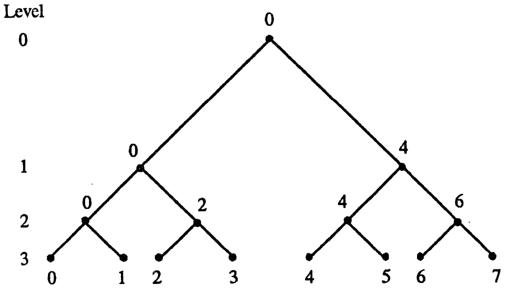


Fig. 3.1.2 The Pseudo Binary Tree

B. Embedding the Pseudo Binary Tree into the FCM I

The pseudo binary tree can be used for merging procedure on the FCMs. It is not required for distributing and collecting data. For the convenience of illustration, a 4-level pseudo binary tree shown in Fig. 3.1.2 is used. The embedding the pseudo binary tree into the FCM I for merging procedure is shown in Fig. 3.1.3. The procedure is described below.

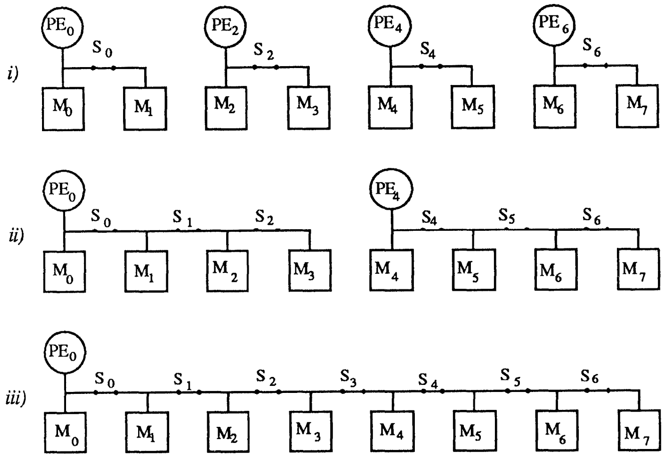


Fig. 3.1.3 Embedding the Pseudo Binary Tree into the FCM I

i) When the switch set S_0 is closed, two memory modules, (M_0, M_1), become one contiguously addressable memory module where the parenthesis represents one contiguously addressable memory module. PE_0 can access the modules like one memory module. Similarly, when S_2, S_4 and S_6 are closed, (M_2, M_3), (M_4, M_5) and (M_6, M_7) become three disjoint memory modules. PE_2, PE_4 and PE_6 are able to merge the local results. This procedure realizes the transition from level 3 to level 2 in the 4-level pseudo binary tree.

ii) In addition, when S_1 and S_5 are closed, PE_0 and PE_4 can merge the local results in (M_0, M_1, M_2, M_3) and (M_4, M_5, M_6, M_7), respectively.

iii) Finally, PE_0 can merge all local results in $(M_0, M_1, M_2, M_3, M_4, M_5, M_6, M_7)$ with S_3 closed. In this case, all switch sets are closed.

As shown above, there is no communication overhead (no message passing) that can significantly limit speed-up as on the hypercube multiprocessor. In addition, there is no memory conflict because only one PE in each partition performs the merging procedure. Instead there exist only switching and control times. The steps for the FCM II in each memory set are the same as those for the FCM I. Thus, the pseudo binary tree may also be embedded into the FCM II. In addition, general tree topologies may also be embedded into the FCMs. The embedding procedure for general tree topologies into the FCMs is similar to that for the pseudo binary tree.

C. Control Algorithm for Parallel Region Labeling on the FCM I

The control algorithm for region labeling executed in the CU is described in Fig. 3.1.4. Consider PE_p. Let the binary representation of p be $(a_{n-1} \cdots a_i \cdots a_0)$ where $a_i = 0,1$. Consider the switch set S_q. Let the q be denoted by the binary representation $(b_{n-1} \cdots b_i \cdots b_0)$ where $b_i = 0,1$. NI is the number of image frames to be processed and N is the number of PEs employed.

```
begin
  close all switch sets;
  for $i \leftarrow 0$ until $NI - 1$ do
    begin
      if ($i = 0$, i.e., the first frame) then
        send a signal to the IOP to load the first image frame;
      else
        send a signal to the IOP to unload output results and
        load another frame;
      repeat wait until the CU receives the load completion
      signal from the IOP
      open all switch sets;
      broadcast the start signals to PEs to execute tasks;
      while the number of signals received from PEs $\neq N$ do
        wait;
      /* Implementation of the Pseudo Binary Tree */
      $NS \leftarrow N$;
      for $j \leftarrow 1$ until $\log N$ do
        begin
          close the switch set $S_q$ where $q$ is all possible
          values of $(b_{n-1}, \ldots ,b_0)$ obtained with the $j$ least
          significant bits set to zero, i.e.,
          $b_0 = b_1 = \cdots = b_{j-1} = 0$;
          send the start signals to $PE_p$ to merge two local
          results, where $p$ is all possible values of
          $(a_{n-1}, \ldots ,a_0)$ obtained with the $j$ least significant
          bits set to zero, i.e., $a_0 = a_1 = \cdots = a_{j-1} = 0$;
          while the number of signals received from PEs $\neq$
          $NS$ do
            wait;
          $NS \leftarrow NS \gg 1$; /* one bit right shift */
        end
    end
end
```

Fig. 3.1.4 The Overall Control Algorithm executed in the CU

D. Time for Switching and Control on the FCM I

As described before, the switches in a set can be operated in parallel by using the control bus, but two sets can not be operated in parallel. The time for operating one set of switches is represented by t_1. The control signal can be issued by the CU, PEs and the IOP. The time for sending a control signal or for broadcast is represented by t_2. The control procedure described in the previous subsection is used for estimating the time for switching and control.

To send a signal to the IOP to unload output results and load another image frame, t_2 is required. After loading, the IOP sends the completion signal to the CU, thus t_2 is also required. When the CU opens all switches in all sets, $(N-1)t_1$ is needed. When the CU broadcasts the start signal to all PEs to execute their tasks, t_2 is needed. After processing, all PEs send the task completion signal to the CU, it takes Nt_2. The merging procedure using the pseudo binary tree was described in subsection 3.1.2.B. To combine two local results, the switches in the set between two modules should be closed. Hence, $\left[\frac{N}{2}\right]$ switching times are needed for the bottom level in the pseudo binary tree (level $\log_2N$). Then the CU sends the start signal to $\left[\frac{N}{2}\right]$ PEs to execute the merging procedure. After merging, $\left[\frac{N}{2}\right]$ PEs send the completion signal to the CU. Accordingly, $\left[\frac{N}{2}\right]$ switching times and $2\left[\frac{N}{2}\right]$ control signals are required. For the next bottom level (level $\log_2N - 1$), $\left[\frac{N}{4}\right]$ switching times and $2\left[\frac{N}{4}\right]$ control signals are needed. Since the height of the pseudo binary tree is $\log_2N$, this procedure is repeated $\log_2N$ times. Thus, the total number of switching times for merging procedure is $N \sum_{i=1}^{\log_2N} \left[\frac{1}{2}\right]^i$, and it takes $\left[N \sum_{i=1}^{\log_2N} \left[\frac{1}{2}\right]^i\right]t_1$. The total number of control signals for merging procedure is $2N \sum_{i=1}^{\log_2N} \left[\frac{1}{2}\right]^i$, and it takes $\left[2N \sum_{i=1}^{\log_2N} \left[\frac{1}{2}\right]^i\right]t_2$.

Therefore, the total time for switching in region labeling is

$$t_{sw} = (N-1)t_1 + \left[N \sum_{i=1}^{\log_2N} \left[\frac{1}{2}\right]^i\right]t_1 = 2(N-1)t_1 \qquad (3.1.1).$$

and the total time for control is

$$t_{cn} = 3t_2 + Nt_2 + \left[2N \sum_{i=1}^{\log_2N} \left[\frac{1}{2}\right]^i\right]t_2 = 3t_2 + Nt_2 + 2(N-1)t_2 \quad (3.1.2).$$

As will be seen in Section 4, these times are much smaller than computation time. Hence, bottlenecks due to the CU, the communication bus and the control bus are minimal.

3.1.3 Simulation on the FCM II

Since the FCM II has a buffering capability, another image frame can be sent to the currently unused memory set even before the previous image has been processed. In other words, we may try to overlap in time the data acquisition and the data transfer with the computation between successive images. This could improve performance especially when the data acquisition time and the data transfer time are considerably large. In contrast, there can be no overlap of the procedures in the FCM I. The former is referred to as an overlapping method while the latter as a non-overlapping method.

The pseudo binary tree can be also embedded into the FCM II. The control algorithm in the FCM II is similar to that of the FCM I described in 3.1.2.C. While PEs process the data in one memory set, the CU sends a signal to the IOP to unload output results of the previous frame from the other set and load another frame into the set (it takes t_2 for control). After the CU receives the load completion signal from the IOP (it takes also t_2), the CU opens all switches in all sets (it takes $(N-1)t_1$), and wait until PEs finish the current frame. Therefore, the time for switching, $(N-1)t_1$, and the time for control, $2t_2$ can be overlapped with computation time. Thus, the total switching time and control time are reduced as follows:

$$t_{sw} = \left[N \sum_{i=1}^{\log_2N} \left[\frac{1}{2}\right]^i\right]t_1 = (N-1)t_1 \qquad (3.1.3)$$

$$t_{cn} = t_2 + Nt_2 + \left[2N \sum_{i=1}^{\log_2N} \left[\frac{1}{2}\right]^i\right]t_2 = t_2 + Nt_2 + 2(N-1)t_2 \quad (3.1.4).$$

3.1.4 Implementation on the Hypercube Multiprocessor

The system is composed of a controller and a hypercube containing 32 PEs (the Binary 5-cube). The controller acquires an input image from an input device, and divides it into a set of equal size subimages, and sends them to the hypercube through the pseudo binary tree described in 3.1.2.A. After distribution, every PE processes a subimage concurrently. On completing, the local results are sent to the higher level PEs in the pseudo binary tree for collecting and merging. They are merged across vertical or horizontal borders through the pseudo binary tree until a PE obtains a whole result. The total number of vertical and horizontal merging steps is $\log_2N$ where N is the number of PEs employed. The labeled image is sent back to the controller. Finally, the controller sends it to an output device. As described above, there are six steps in this procedure; acquisition of an image, distribution of an image to PEs, parallel computation, collection of local results, merging for boundary consistency, and transferring output result to an output device. There exist controller-to-PE communications and PE-to-PE communications for distribution and collection. We used a scheme for distribution of data, i.e., the *modified singlecast scheme* in which the controller distributes a set of subimages to PEs on a certain level in a pseudo binary tree [14]. Every PE which receives a subimage divides it again into two subimages, sends one subimage to its child PE in the pseudo binary tree recursively until all leaf PEs receive their subimages.

3.2 Median Filtering

Median filtering is a neighborhood operation, which transforms the value of each pixel to a new value calculated from its neighboring pixels (the median of a 3 x 3 window is used). Convolution, edge detection, and smoothing are other examples of neighborhood operations. To avoid unnecessary communication due to the data partitioning, the overlapped partitioning method shown in Fig. 3.2.1 is used. In this case, merging time (t_{mg}) can be avoided. Therefore, the pseudo binary tree which is used only for the merging algorithm in region labeling, is not needed to implement median filtering on the FCMs. However, it is needed to distribute and collect data on the hypercube multiprocessor. The control algorithms of the FCMs are simpler than those of region labeling because there is no merging procedure, namely, no pseudo binary tree. Since the control steps for the pseudo binary tree are not required, the total times for switching and control on the FCM I are derived from equations (3.1.1) and (3.1.2)

$$t_{sw} = (N-1)t_1 \qquad (3.2.1)$$

$$t_{cn} = 3t_2 + Nt_2 \qquad (3.2.2).$$

As described in 3.1.3, $(N-1)t_1$ in t_{sw} and $2t_2$ in t_{cn} can be overlapped with t_{cp} on the FCM II. Therefore, on the FCM II the total time for switching can be totally overlapped, and the total time for control is

$$t_{cn} = t_2 + Nt_2 \qquad (3.2.3).$$

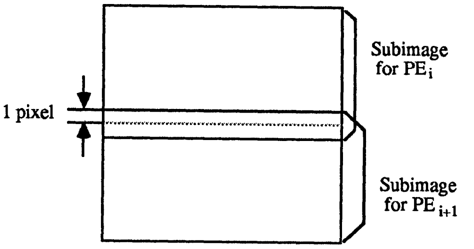


Fig. 3.2.1 The Overlapped Partitioning Method of Image

4. EXPERIMENTAL RESULTS AND DISCUSSION

Parallel algorithms for region labeling and median filtering have been implemented on the iPSC and simulated on the FCMs by using the iPSC. The simulation results are based on the following conservative assumptions. Firstly, for simplicity and sufficiency, t_1 and t_2 are assumed to be 1 msec. Secondly, the data acquisition time and the data transfer time are assumed to be 40 msec for all machines. Lastly, assume that the PEs of the FCMs have the same computing capabilities as those of the iPSC. In practice, t_1 and t_2 are usually much smaller than 1 msec (nano second order), and the data acquisition time and the data transfer time for a 256 x 256 binary image (64Kbytes), are less than 40msec [16].

4.1 Region Labeling

User-generated binary images (256 x 256) were used to test all aspects of the algorithm. The images have a variety of regions. A data set which resembles a mesh containing a variety of region types is used for performance evaluation.

To evaluate the performance of the FCMs, each time component, the total time, and the speed-up are listed in Table 4.1.1. The speed-up is plotted in Fig. 4.1.1. As shown in this figure, the speed-up of the FCMs is better than that of the hypercube multiprocessor. The reason is that communication overhead increases as N increases on the hypercube multiprocessor. On the FCMs, there is no communication overhead that increases with N, only marginal overhead in switching and control. Even with the conservative assumptions, the switching and control times on the FCMs are much smaller than computation time. Hence, bottlenecks due to the CU, the communication bus and the control bus are minimal.

The overlapping method (FCM II) achieves better performance than the non-overlapping method (FCM I) as expected. The FCM II can reduce even the data acquisition time and the data transfer time. If these times are large, the difference in performance is large. Even though the difference is small, the FCM II has more applicability as described in subsection 2.2.3.

Even with no communication overhead, the speed-up is not linearly proportional to N because there are some necessary tasks that are independent of N. For instance, the amount of time to label regions based on merged TPP's is fixed because it must be done by a single PE. This behavior can be observed in Table 4.1.1.

4.2 Median Filtering

The measured performance figures are tabulated in Table 4.2.1. The speed-up is plotted in Fig. 4.2.1. As shown in this figure, significant speed-up is achieved. The speed-up is better than that of region labeling. The reason is that there is no mering procedure, and computation time is larger than that for region labeling. In other words, the ratio of computation time to parallel processing overhead in median filtering is larger than the ratio for region labeling. The computation time (t_{cp}) decreases linearly, as N increases. The speed-up of the FCMs is approximately linearly proportional to N because there is no communication overhead, no merging procedure, and no necessary task which must be done by a single PE. Accordingly, as N increases, more speed-up may be achieved. For many neighborhood operations, such significant speed-up may be expected.

5. CONCLUSIONS

In this paper, two Flexibly (Tightly/Loosely) Coupled Multiprocessors for image processing are proposed. These multiprocessors alleviate the disadvantages of existing multiprocessors, such as communication overhead due to message passing in loosely coupled multiprocessors, memory contention due to shared memory in tightly coupled multiprocessors, and overhead due to inefficient loading and unloading data.

Some features of the FCMs have several superficial similarities to those of other architectures, such as the VS [7], the SM3 [8] and the MP/C [9]. However, there exist several significant differences. The

N	Time [ms]	t_{ac}	$t_{ds_{cp}}$	$t_{ds_{pp}}$	t_{cp}	$t_{cl_{pp}}$	t_{mg}	t_{sw}	t_{cn}	$t_{cl_{pc}}$	t_{tf}	Total	Speedup
1	Single	40	0	0	12880	0	0	0	0	0	40	12960	1.0
2	Hypercube	40	1856	128	6720	224	560	0	0	1682	40	11250	1.152
	FCM I	40	0	0	6720	0	560	2	7	0	40	7369	1.759
	FCM II	0	0	0	6720	0	560	1	5	0	0	7286	1.779
4	Hypercube	40	1872	192	3568	304	416	0	0	1664	40	8096	1.601
	FCM I	40	0	0	3568	0	416	6	13	0	40	4083	3.174
	FCM II	0	0	0	3568	0	416	3	11	0	0	3998	3.241
8	Hypercube	40	1856	208	2736	480	544	0	0	1904	40	7808	1.660
	FCM I	40	0	0	2736	0	544	14	25	0	40	3399	3.813
	FCM II	0	0	0	2736	0	544	7	23	0	0	3310	3.915
16	Hypercube	40	1872	240	2048	272	560	0	0	1832	40	6904	1.877
	FCM I	40	0	0	2048	0	560	30	49	0	40	2767	4.684
	FCM II	0	0	0	2048	0	560	15	47	0	0	2670	4.854
32	Hypercube	40	1872	208	1776	448	656	0	0	1648	40	6688	1.938
	FCM I	40	0	0	1776	0	656	62	97	0	40	2671	4.852
	FCM II	0	0	0	1776	0	656	31	95	0	0	2558	5.066

Table 4.1.1 Time Components for Region Labeling

N	Time [ms]	t_{ac}	$t_{ds_{cp}}$	$t_{ds_{pp}}$	t_{cp}	$t_{cl_{pp}}$	t_{mg}	t_{sw}	t_{cn}	$t_{cl_{pc}}$	t_{tf}	Total	Speedup
1	Single	40	0	0	70700	0	0	0	0	0	40	70780	1.0
2	Hypercube	40	1678	81	35349	18	0	0	0	1612	40	38818	1.823
	FCM I	40	0	0	35349	0	0	1	5	0	40	35435	1.997
	FCM II	0	0	0	35349	0	0	0	3	0	0	35352	2.002
4	Hypercube	40	1678	119	17771	132	0	0	0	1550	40	21330	3.318
	FCM I	40	0	0	17771	0	0	3	7	0	40	17861	3.963
	FCM II	0	0	0	17771	0	0	0	5	0	0	17776	3.982
8	Hypercube	40	1680	140	8883	150	0	0	0	1716	40	12649	5.596
	FCM I	40	0	0	8883	0	0	7	11	0	40	8981	7.881
	FCM II	0	0	0	8883	0	0	0	9	0	0	8892	7.960
16	Hypercube	40	1676	156	4440	216	0	0	0	1514	40	8082	8.758
	FCM I	40	0	0	4440	0	0	15	19	0	40	4554	15.542
	FCM II	0	0	0	4440	0	0	0	17	0	0	4457	15.881
32	Hypercube	40	1677	151	2220	206	0	0	0	1764	40	6098	11.607
	FCM I	40	0	0	2220	0	0	31	35	0	40	2366	29.915
	FCM II	0	0	0	2220	0	0	0	33	0	0	2253	31.416

Table 4.2.1 Time Components for Median Filtering

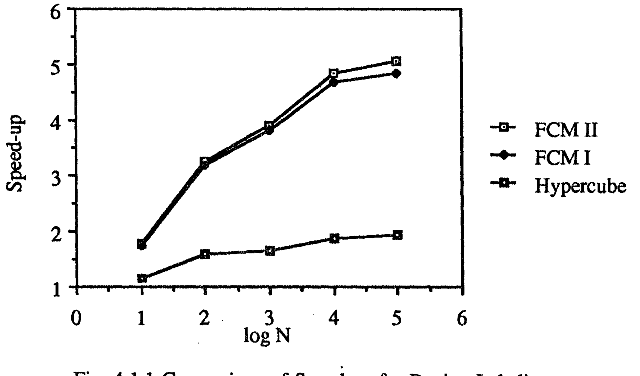


Fig. 4.1.1 Comparison of Speed-up for Region Labeling

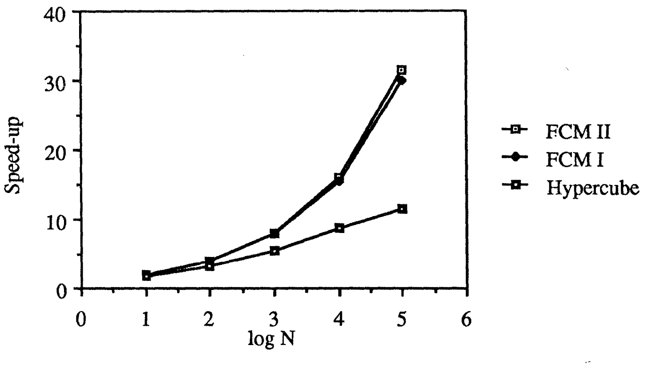


Fig. 4.2.1 Comparison of Speed-up for Median Filtering

FCMs differ considerably from the VS in that the latter is a goal-oriented architecture, being functionally dedicated, with inhomogeneous processors and memory modules. Other significant differences between the FCMs and the VS architecture are in the bus structures, switches, memory addressing scheme, and I/O scheme.

In the FCMs, a set of adjacent memory modules can be formed into one contiguously addressable module, while this feature is not found in the SM3 and the MP/C. In the worst case, if the processor in the SM3 or the MP/C tries to repeatedly access data stored in two alternate memory modules, then module switching in the SM3 or computing an effective address by the switch controller in the MP/C must precede each memory access. In image processing, this situation occurs often. One example is the merging of subimages using boundary consistency. If two adjacent subimages in two different memory modules are to be combined into one subimage using boundary consistency, the alternate memory modules need to be repeatedly accessed for every boundary pixel processed by the MP/C and the SM3. In contrast, in the FCMs only one switching time is needed to connect the memory modules.

The input/output processor is directly connected to the variable space memory in the FCMs. This scheme can minimize the overhead to load and unload data, which may be significant in the SM3 and the MP/C. In each partition of the MP/C, only one processor can be active, while in the FCMs, any PE in any partition can be active. Only one type of communication (processor-to-memory, no direct processor-to-processor communication) is supported in the MP/C, while three different types of communication are supported in the FCMs. In addition, the switch controller in the MP/C and the three different control buses with switches in the SM3 are complicated. The FCMs use simple switches in one control bus. The SM3 is a multicomputer, in which each node is an independent computer system with its own secondary storage device, but the FCMs are multiprocessors.

Parallel algorithms for region labeling and median filtering have been simulated on the proposed architectures by using the iPSC. The performance of the FCMs shows remarkable improvement over the existing hypercube multiprocessor. The FCMs can also be used for more general tasks. Image processing, MSIMD processing, SIMD processing, pipelined pseudoparallel algorithms including pipelined algorithms and tree-structured algorithms are a few examples. The FCMs are highly suited when data locality is guaranteed.

Since the CU, the communication bus, and the control bus are mainly used for control, but not for the exchange of data, bottlenecks due to these components are minimal as described in subsection 4.1. In addition, the architectures are simple and modularized, and the control strategy is straightforward. Hence, the FCMs have good scalability. The granularities of the proposed architectures may be considered in the range from coarse to fine. The applications for more general tasks will be investigated in future research.

Acknowledgment

We would like to thank Dr. N. Nandhakumar for his valuable criticism and suggestions, and P. Song for his comments on the feasibility of this work. We thank H. Asar, J. Rodriguez and V. Chaudhary for reading and commenting. We are also grateful to the referees for their helpful suggestions and comments.

REFERENCES

[1] C.L. Seitz, "The Cosmic Cube," *Commun. ACM*, vol. 28, pp. 22-33, Jan. 1985.

[2] F.A. Briggs, K.S. Fu, K. Hwang, and B.W. Wah, "PUMPS Architecture for Pattern Analysis and Image Database Management," *IEEE Trans. Comput.*, vol. C-31, pp. 969-983, Oct. 1982.

[3] H.J. Siegel, L.J. Siegel, F.C. Kemmerer, P.T. Mueller, Jr, H.E. Smalley, Jr, and S.D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Trans. Comput.*, vol. C-30, pp. 934-947, Dec. 1981.

[4] J.T. Schwartz, "Ultracomputer," *ACM TOPLAS*, vol. 2, pp. 484-521, Oct. 1980.

[5] IEEE Computer, "Special Issue on Computer Architecture for Image Processing," Jan. 1983.

[6] S. Yalamanchili, K.V. Palem, L.S. Davis, A.J. Welch, and J.K. Aggarwal, "Image Processing Architectures: A Taxonomy and Survey," *Progress in Pattern Recognition*, vol. II, pp. 1-37, North Holland, 1985.

[7] J.D. Dessimoz, J. Birk, R. Kelley, and J. Hall, "A Vision System with Splitting Bus," in *Proc 1981 IEEE Comput. Soc. Workshop Comput. Architect. for Pattern Analysis and Image Database Management*," Hot Springs, VA, pp. 62-66, Nov. 1981.

[8] C.K. Baru, and S.Y.W. Su, "The Architecture of SM3: A Dynamically Partitionable Multicomputer System," *IEEE Trans. Comput.*, vol. C-35, pp. 790-802, Sep. 1986.

[9] B.W. Arden, and R. Ginosar, "MP/C: A Multiprocessor/Computer Architecture," *IEEE Trans. Comput.* vol. C-31, pp. 455-473, May 1982.

[10] G.J. Lipovski, and M. Malek, "Parallel Computing," *Wiley*, New York, 1987.

[11] W. Lin, and C.-l. Wu, "Design of a 2 x 2 Fault-Tolerant Switching Element," in *Proc the 9th Int. Symp. Comput. Architect.*, pp. 181-189, 1982.

[12] D.P. Agrawal, and R. Jain, "A Pipelined Pseudoparallel System Architecture for Real-Time Dynamic Scene Analysis," *IEEE Trans. Comput.* vol. C-31, pp. 952-962, Oct. 1982.

[13] M.H. Sunwoo, B.S. Baroody, and J.K. Aggarwal, "A Parallel Algorithm for Region Labeling," in *Proc. 1987 IEEE Comput. Soc. Workshop Comput. Architect. for Pattern Analysis and Machine Intelligence*, Seattle, WA, pp. 27-34, Oct. 1987.

[14] S.Y. Lee, and J.K. Aggarwal, "Exploitation of Image Parallelism via the Hypercube," *the Second Conf. Hypercube Multiprocessors*, Knoxville, TN, Sep. 1986.

[15] A.K. Agrawala, and A.V. Kulkarni, "A Sequential Approach to the Extraction of Shape Features," *Computer Graphics and Image Processing*, vol. 6, pp. 538-557, 1977.

[16] G.C. Nicolae, and K.H. Hohne, "Multiprocessor System for the Real-Time Digital Processing of Video-Image Series," *Elektronische Rechenanlagen*, No. 21, pp. 171-183, 1979.