PROCEEDINGS

OF THE

1977 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

JEAN-LOUP BAER
Editor

Cosponsored by the

Wayne State University

and the

IEEE Computer Society

IEEE Catalog No. 77CH1253-4C

PROCEEDINGS
OF THE
1977 INTERNATIONAL CONFERENCE
ON
PARALLEL PROCESSING

JEAN-LOUP BAER
Editor

Papers presented on
August 23-26, 1977

Co-Sponsored by

Department of Electrical and Computer Engineering
WAYNE STATE UNIVERSITY
Detroit, Michigan

and the

IEEE Computer Society

In Cooperation with the

Association for Computing Machinery

Additional copies are available from:
IEEE Computer Society
5855 Naples Plaza, Suite 301
Long Beach, CA 90808

or

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

Manufactured in the U.S.A.

P R E F A C E

In a year filled with numerous national and international meetings, the response to the call for papers and the attendance at the 1977 International Conference on Parallel Processing have been extremely rewarding for its organizers. This conference, the sixth if one includes the Sagamore Computer Conferences to which it has succeeded, is now regarded as a regular annual event. The 1977 conference, as its predecessor, had the formal support of the IEEE Computer Society which is handling the production and distribution of these Proceedings and of the Association for Computing Machinery.

This year more than 80 papers were submitted with prospective authors coming from 9 countries. Each paper was refereed by at least two referees. The 96 individuals who made this possible are listed at the end of these proceedings and I would like to thank them personally for a job well done. Special thanks are also due to Dr. U. Herzog who served as a liaison with some European contributors, Mr. M. Kesselman who volunteered to organize a panel on multiple microprocessors systems, Mr. J. McKay who set up a session on PEPE, and Dr. M. Freeman and TCCA who helped in organizing and refereeing papers for a session on Computer Architecture.

I think that the participants at the Conference will agree with me for sending congratulations to Dr. Charles Elliott and his staff at Wayne State University for taking care of impeccable local arrangements. I would also like to thank Ms. Marcia Riedel at the University of Washington for her help.

Last, but certainly not least, we owe a great debt of gratitude to Professor Tse-yun Feng. Dr. Feng, who originated and organized the first four conferences, was General Chairman of the 1977 ICPP. With him as a constant driving force, we can look forward with great anticipation to next year's meeting.

Jean-Loup Baer

1977 ICPP Program Chairman

SPECIAL AWARDS

1977 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

Best Presentation

Mr. David Misunas
M.I.T.
Analysis of Structures for Packet Communication

Most Original Paper

Professor Jerome Rothstein
Ohio State University
Toward An Arithmetic For Parallel Computation

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT'D)

TABLE OF CONTENTS (CONT'D)

TABLE OF CONTENTS (CONT'D)

PERSPECTIVES OF PARALLEL PROCESSING

Franklin H. Westervelt

Director, Computing Services Center
Professor of Computer Science
College of Liberal Arts
Professor of Engineering
College of Engineering

Wayne State University
Detroit, Michigan 48202

Keynote Address
1977 International Conference
on
Parallel Processing

Over the past years, the keynote address for the International Conference on Parallel Processing has attempted to present one or another view of Parallel Processing and, in so doing, provide a point of beginning and a challenge for the conference and its work. Each view of parallel processing tends to see and to emphasize certain aspects of parallel processing and its state of development. In a very real sense, each view provides another perspective of parallel processing.

It has been observed that the "real world" is the union of an unbounded finite number of "unreal worlds". Each unreal world is a model or perspective of the real world, or some aspect of it, held by a particular observer. Occasionally, many observers share a common perspective, at least for a time, and cause a particular view to acquire a certain popularity and acceptance. But it is of considerable importance for each of us to develop flexibility and adaptiveness so that we may recognize and appreciate the contributions provided for us by other views or perspectives. The best perspectives recognize a great many features and fine structure and, in so doing, tend to provide unification and understanding of complex subjects, but it is also important to remember the difficulty inherent in viewing any complex, multidimensional subject from a finite number of perspectives, let alone from a single point, and thereby obtaining an adequate picture of the subject.

This address will provide yet another perspective of parallel processing. But the point of view is, hopefully, different enough that some may discern new features and new challenges.

Parallelism in computation machinery has been recognized and incorporated from the very beginning. Circuitry to provide, for example, parallel addition appeared almost concurrently with serial circuits for the same functional purpose. Designers have always recognized the improvement in speed and performance to be gained through parallelism. The ability of each member of the audience to carry on, at this very moment, extremely difficult feats of audio and visual pattern recognition and interpretation very easily while employing receptors and information processors whose performance specifications are comparatively pedestrian is possible only because of quite incredible parallel processing inherent in each one of us. The human being is, indeed, a most remarkable 80 kg non-linear parallel processing servomechanism capable of mass production by unskilled labor.

The individual logic devices which together comprise the human parallel processor can each be put to shame in many ways by components already state-of-the-art in contemporary computing systems. Yet we remain an enormous distance away from being able to assemble, package and power a parallel processor of like complexity and generality. It is interesting to note the anthropomorphic inspiration present in recent research on optical pattern recognition systems and interconnected cellular automata. Man has always derived great benefit from the study and modeling of existing systems and from using these studies and models to enhance and improve upon various aspects or features of them. Parallel Processing research should be found to be no different in this respect.

With these very general remarks as background, let me move toward the presentation of my particular personal perspective of parallel processing. Here my experience in providing systems for all aspects of computational services in

higher education at two major United States universities must necessarily bias my point of view. But I believe that my perspective may be of sufficient general interest on a larger scale to merit your consideration. At the close of this address, I hope that we share in a mutual exchange of question and comment springing from these remarks.

As we are all well aware, the world presently faces an energy crisis. But it is more nearly correct to recognize the crisis to be in the consumption of certain particular forms of fuels. In other words, the problem lies in the most appropriate use of raw materials as their finite supply decreases and the cost of acquiring them increases. The complex long chain hydrocarbons present in fossil fuels are a resource of chemical building blocks that is much too valuable to be simply oxidized by burning. I must believe that the descendants of our children's children will be most critical of our generation for having squandered and destroyed these complex molecules in such enormous quantities by burning them. Yet the world need for abundant energy in order to provide an adequate food supply and general living standards for its burgeoning population must be met.

Because the energy needs must be met if we are to survive and continue as a civilization, I will make no further comment on this situation at this time. I shall assume the solution for the energy supply problem and focus my attention on another longer range problem. In the longer run, the obviously finite size of the planet Earth and the material resources available to it within its reasonable sphere of acquisition will, in my opinion, result in the problems of most efficient and effective use of all natural resources becoming the overriding concern for all people. Materials may become too valuable and costly to permit anything but highly automated plants and machines to handle, mold, cut, shape and form them into products for our use. Reduction of the waste of materials by the progressive elimination of the human error factor in manufacture and production will come to be a dominant objective achievable through increased application of automation. Many products, particularly in the computer field, are only possible to be made at all, even today, because of complex, highly automated machines which operate with very minimal human intervention. The concurrent development and application of information processing in its most general sense and the consequent impact on further development of parallel processing methodology follows immediately in the industrial and commercial arena.

But the systematic reduction of human error in manufacture and production will carry with it another effect. It has been observed that, in any reasonably complex system, there is no such thing as a change that produces only a single effect. As the use of automation increases, the amount of conventional work performed by humans in manufacture and production will decrease. Leisure time growth today is viewed as positive by many who may have had their time occupied to too great an extent in the past by conventional work. But society does not yet compensate leisure in any general sense and for some persons bypassed by technology "leisure time" may be only another term for "unemployment".

Unemployment is a problem of concern when national levels are in the 5% to 10% range. But I can tell you from recent personal experience in the Detroit area where unemployment in some segments of the population reached 50% or more during the recent recession that "concern" is simply not an adequate term to apply to such a problem. Consider then what the situation might be if conventional work decreased such that very high levels of unemployment became common on a world scale. Each of you may construct your own image of such a world.

The world faces a dilemma: On the one hand, the Puritan work ethic will tend to decline to compensate leisure while, on the other hand, scarcity of material resources will cause conventional work to decline as well. A solution for this apparent dilemma will, in my opinion, come from a redefinition of "conventional work" and from a shift in human activities from those that are intensive in the consumption of non-renewable materials toward activities that will tend to be energy-intensive, and in many cases nearly energy-exclusive. In other words, energy will tend to become the one resource that humans will, in general, be permitted to use and consume in significant amounts because it will be the most easily replenished resource.

Consider some of the kinds of energy-intensive activity implied by such a world situation. "Work" may be structured in terms of interaction at many different levels of intellectual capability and skill using the technology of extremely advanced communication, simulation and computation to enable persons to learn complicated new skills and to be compensated for doing so. In the process of such learning and development, actual

materials would be consumed very sparingly while the process may consume significant energy in order to be carried out properly.

If this should seem too farfetched, consider only a few of the things that we are presently doing of this nature. We are all aware of the elaborate simulations used in the development of skills needed by the Astronauts and Cosmonauts. When the first Astronaut actually stepped upon the surface of the Moon, after consuming enormous quantities of real natural materials in order to get there, his pulse rate, respiration and blood pressure showed no indication of his awareness of being in surroundings that for the rest of us must still be regarded as fantastic! Of course not, this human being had already "been there before" many times through simulation that was incredibly "real" and which, by comparison, consumed almost no natural resources. Furthermore, this Astronaut was among many who experienced the same training through simulation and were paid to do it.

Airline pilots and the captains of supertankers are also examples of skill development and learning through the use of sophisticated simulations. I need not relate to this audience the critical role played by computer technology in these cases. I should only like to point out that many much less sophisticated examples exist where compensation has been given to those learning or developing new skills or capabilities. The learning of foreign language while in military service is a most familiar example. The extension and refinement of these and similar examples is, perhaps, the mechanism by which "work" in the future may come to be redefined.

If something similar to this should come about, it will require significant new developments in parallel processing for general purpose computation in addition to the special purpose forms that receive most of our attention today. It is of great importance that the necessary research and development of large general purpose parallel processors be funded now.

To develop my perspectives of parallel processing further, I should now like to turn my attention toward a much closer but highly related problem. In a very strong sense, both the foregoing problem and the one upon which I now focus are related to education and learning. Education, in general, and Higher Education, in particular, is an extremely labor intensive business today. For many, if not most, colleges and universities the fraction of General Fund Budget committed to salaries and wages is 70% to 80% or, in some cases, more.

As a result of declining numbers of students in the primary and secondary schools and general inflationary pressures on salaries and wages, colleges and universities face the very serious prospect of "pricing themselves out of business" in the next decade. Tuition is already at a level that tends to require one or more forms of student financial assistance, even for students from families nominally considered to be well-to-do. For less advantaged students, higher education in the absence of substantial student financial aid is already priced beyond reach.

In order to preserve or, better, improve the quality of higher education while holding or, better, reducing the cost, means for improving the productivity of the system must be found. Other business and industry faced with the same sort of problem turned to technology for help in solving it. Unfortunately, much of the technology relevant to industry is not relevant to higher education in trying to improve productivity.

There are, however, two general technologies with considerable relevance to this problem. One, the general "broadcast" technology, provides many ways to improve upon the dissemination of information in the "one-to-many" mode. Audio-visual techniques, including the entire scope from films and tapes to video, all extend the audience of a given educator and tend to reduce the unit or per-student cost of conveying the particular information or lesson. In general, the more effectively the broadcast technology expands the audience size, however, the less effectively does the technology accommodate to the individual needs of particular students. In other words, the "unfair advantage" that distinguishes the university or college from the correspondence school, student-teacher interaction, tends to be seriously impaired. And with this loss of interaction, the quality of the educational process is also impaired.

Again we face a dilemma: it appears essential to improve the productivity of higher education in terms of numbers of students per person engaged in the process, yet it is the interaction or feedback of the one-on-one educational experience that characterizes the finest aspects of that process.

The broad field of computer technology is the other technology

relevant and uniquely suited to assist in the resolution of this dilemma. Where the broadcast technologies tend toward simplex communications channels, computer technologies have emphasized duplex communications in many relevant forms. The essential contribution is the provision of a "many-to-one" technology to provide more efficient and economical interaction and feedback for use in higher education.

The simplest examples of currently available techniques are little more than conventional store-and-forward communications systems. Computer conferencing or asynchronous conferencing comes much closer to the level of technological assistance required for the solution of the quality/quantity versus unit cost dilemma of higher education.

A great deal can be done with existing computer systems in this area. But to reach the levels of reliability and generality required to really solve the problem, we do not yet have the computer systems available with general purpose characteristics and the configurability necessary to deliver the appropriate computational power to a very large number of dynamically created and changing tasks. Parallel processing research and development holds the promise of making the required systems available.

I should like to take a moment to describe a little of the work now being done at Wayne State University which, I hope, may be relevant to parts of the solution for the foregoing problem. Let me first give you a brief picture of the university itself.

Wayne State University is a major urban university with a number of rather unique characteristics. At any given moment, Wayne State University is an active community of about 40,000 or more students and faculty. But the momentarily active student body of 30,000 to 35,000 is drawn dynamically by personal circumstances of work and study from a student population admitted to the university numbering well in excess of 100,000 individuals.

It has been demonstrated, for example by Dartmouth, that when sufficient computer resources can be made available and accessible, nearly 70% to 80% of a university community will find the resource meaningfully relevant to their educational experience.

But it is a considerable challenge to try to provide such access using contemporary systems at an institution that is an order of magnitude larger than Dartmouth.

At Wayne State University, we began over six years ago to acquire the facilities step by step and to develop a hierarchical computing system for the university that might address the problems of higher education as rapidly and effectively as resources and technology would permit. The first decision was to purchase, in 1971, a dual processor IBM System/360 Model 67 configured as a full duplex system. Since then we have been able to retire the loan used to acquire that system and to use the system as a foundation for further system growth and development. This April, we added an Amdahl 470V/6 system as a part of the plan.

The design limitation on main memory size and the lack of Error Correcting Code capability in the standard IBM memory products for the Model 67, as well as the relatively high cost of IBM memory, resulted in a contract between Wayne State University and Fairchild Memory Systems. Under this contract, the parties combined talents to design a bipolar semiconductor memory using the same 256xl TTL 100 ns memory chips supplied by Fairchild for the Illiac IV. This memory system has several interesting features, such as a memory controller capable of executing instructions, and it has demonstrated significantly better performance and economics. Instead of being limited to a maximum of 2 Megabytes, we have 4.25 Megabytes today at an average system cost of about four cents per bit.

The Model 67 duplex architecture remains unique in the IBM family and is generally very poorly understood. This architecture provides features to enhance the parallel processing carried on by the processors and channel controllers. These features have been exploited in the MTS (Michigan Terminal System) implementation. While best known for its Address Translation hardware, the Model 67 bus structure providing for up to eight processors or channel controllers to share main memory and its extremely flexible configuration capability features are at least as important and significant. Quite unlike the more common MP systems produced by IBM. the duplex Model 67 provides system symmetry and consequent simplification of system design. In MTS, for example, the only lack of complete homogeneity among the processors is in the keeping of Time of Day. Since an independent clock for that purpose was not a part of the system hardware, this task

is uniquely assigned to whichever processor was IPLed first. Otherwise the processors are treated in a completely homogeneous fashion. The MTS software is designed to support the maximum of four processors and four channel controllers supported by the bus. IBM never built a maximum configuration to my knowledge, and only one triplex, which was never delivered to a customer. It is most unfortunate that the features of the Model 67 were never carried forward into later systems by IBM.

As but a single example of the importance and utility of these features when combined with appropriate operating system software, a soon to be released paper by Professor R. J. Srodawa of the Wayne State University Computer Science faculty reports and discusses the achievement of dual processor throughput more than double that of the single processor case. The literature commonly cites factors of 1.5 to 1.8 for such systems. While there is need for more experimentation and modeling, these results indicate that it is possible to attain significantly better systems performance than has generally been reached and reported elsewhere. It is also important to recognize the existence of practical cases in which a two processor duplex system can produce more than twice the throughput of a single processor system with the same memory size. Such a result is by no means a contradiction of the Second Law of Thermodynamics. There are many reasons why such a result is attained in this case. Clearly these results require both hardware which is designed with special attention paid to issues of symmetry, lock contention, storage contention and inter-processor communication as well as an operating system designed with special attention to these same issues and including design features that do not double or more than double system overhead in going from one to two processors. The fundamental work of Alexander et al at the University of Michigan in the design of MTS should be much better and more widely understood.

In April of this year, a 4 Megabyte Amdahl 470V/6 was added to our duplex Model 67 configuration. This well known pipelined machine was installed so quickly that our own site preparation delay in obtaining 400 Hz power held up initial operation by two days. Since power became available, we have been extremely pleased with the reliability and performance of this system. The Amdahl is being run under the VM (Virtual Machine) operating system in order to accomplish the significant system work needed to interconnect the 67s and the Amdahl using two CCAs (Channel-to-Channel Adapters) in a full-duplex communications protocol. When ready for use in this mode, the 67s will act as "frontend processors" for the Amdahl and will enable concurrent support for a large number of interactive terminals performing relatively small, quick response tasks and for a smaller number of large, more demanding tasks with slower response. The 67s are further "frontended" by intelligent terminal controllers to provide flexible and prompt communications response. One such controller, based on a PDP-11, serves as the communications controller for up to 32 terminals and the MERIT computer network linking our facility at Wayne State University to TELENET and to the CDC 6400 at Michigan State University and the Amdahl 470V/6 at the University of Michigan. The goal for this system is to form a hierarchical system capable of providing good service for 400 to 500 concurrent general purpose timesharing lines or users. These users presently connect a wide variety of remote terminals to our system ranging from "dumb" typewriter or CRT devices to quite "intelligent" micro- or mini-based graphics and laboratory systems.

The objective is to provide very economical access to a computational resource able to provide a "match" for a given problem with the computing power necessary for effective interaction. And to accomplish this in a user-transparent manner and on a scale consistent with the size of our university community of users.

We see a great many challenging problems in various aspects of parallel processing to be solved in order to achieve our goals and objectives. We believe that dealing with these problems in a real environment of demanding users will cause us to seek out and implement solutions that will contribute to the understanding necessary for improved future systems.

One characteristic of our system that should be clear to all is its combination of processors of a wide range of size, bandwidth and capability. I am frequently amused and sometimes annoyed by the various proponents and protagonists in the mini- vs midi- vs maxi-computer system arguments.

I have held that we, as computer people, have yet to build anything but minicomputer systems. Until we actually build a real maxi-computer, I believe that we have no basis for such arguments.

To illustrate my point, several years ago I served on the Board of the Argonne Universities Association, the governance body for the Argonne National Laboratory. The Laboratory had just acquired an IBM 195 system at a cost of some $10 to $12 million. Clearly a system that most might feel to be a maxi-computer.

On the return flight to Detroit, my seat companion was another member of the AUA Board who was also a vice president of the Detroit Edison Company. Thinking about the 195 acquisition, I asked him, "When was the last time that Detroit Edison acquired a major power generator for $10 million?" My companion laughed and said, "Good Heavens! The transformer substation for the Renaissance Center cost more than that!" Which is exactly my point, a single major power generating station today represents nearly $1 billion and there are many such installations over the entire United States, let alone over the world! On the other hand, while we talk of computer utilities and maxi-computers, we have yet to conceive of, let alone build, any comparable scale machine. Until we have designed and built such a scale machine, I believe that we are dealing with mini-computers and networks of them, no matter what actual mainframe we may be talking about.

When it is finally decided that a true maxi-computer should be built for the first time, it seems clear that parallel processing must infuse the entire design. Parallelism to improve speed and performance, parallelism to improve system reliability and availability, parallelism to enable dynamic configuration, partitioning and assignment of computational power appropriate to a very large number of both independent and interdependent tasks, parallelism to enable rapid and efficient processing of very large databases required to meet the needs of society. The call to this International Conference on Parallel Processing seems clear and the future exciting and challenging.

If one places today's point in the development of modern computer technology on the same time scale as the Industrial Revolution beginning with James Watt's Condensing Steam Engine, then we have just this year seen Robert Fulton's first commercial steamboat! While we recognize that the advance of technology tends to be exponential and that, as a result, progress on an absolute scale in our time is much larger than from Watt to Fulton, I believe that our relative progress in computer technology as viewed from a century or two hence will appear to have been no greater! We have no justification to feel superior or to fail to press forward with maximum effort.

Depending upon how one keeps the score, we have moved toward the serial limit of machine computation by seven to nine orders of magnitude since Eckert and Mauchly, and again depending upon who attempts to establish the serial limit, we have perhaps five or six orders of magnitude remaining. Allowing for complementary exponential effects in difficulty and technology advance, we should approach the serial limits rather closely when we have lapsed again the time interval already past in computer technology development. We must continue to press forward toward the serial limits, but it becomes quite clear that we must become increasingly aware of and sensitive to the vital role of parallelism in the future of machine computation.

These are some of my perspectives of parallel processing. I hope that you may have gained from sharing them with me even a tiny fraction of the pleasure that it has been for me to present them to such a distinguished and important audience. I want to add my welcome to that of Wayne State University and the IEEE Computer Society to the 1977 International Conference on Parallel Processing and to the important work ahead of you. Thank you for your most gracious consideration and attention.

THE IMPACT OF CLASSIFICATION SCHEMES ON COMPUTER ARCHITECTURE

Wolfgang Händler*

Universidade Estadual de Campinas
Instituto de Matemática, Estatística e Ciência da Computaçao
Campinas S.P., Brasil

1. Remarks on Classification Schemes and Formal Systems

Classification schemes, languages, and formal systems of all kinds have a considerable influence on our thinking. Structures which are inherently the subject-matter of a language as well as of classification schemes form the basic material of what can be expressed in a language or can be comprehended from its position in a classification scheme. The same statement seems to be valid for formal systems in a more specific sense. Thus the tool can be used in the application area for which it was created.

For example the Ricci-Calculus performs this role only in the area for which it was created, certain areas of physics and partial differential equations. Outside this area problems arise for which it is not suitable.

B. Whorf has said that language guides thought [11] and that therefore language sometimes prevents the appropriate solution of a problem being found. We must admit that in many cases a language (it can be referred to as a calculus or notation) can be a barrier rather than an aid in solving a problem. It is also true that a classification scheme can be a barrier, although it can provide an insight into the relationships between the elements of some group.

If such a classification scheme is to be applied to animals and plants, then the elements are existing objects and the scheme cannot completely fail, although the discovery of a new species can present difficulties in fitting it into an existing classification scheme. Such a scheme can be called a taxonomy, since all the species are considered to be descended from a single species, in accordance with the biological theory of evolution.

It seems more difficult to create a classification scheme, or even a taxonomy, for some area of contemporary technology. It is necessary to project future advances as well as placing existing examples in it.

The aim of this paper is to show that some existing schemes may fail to indicate the right direction for the development of computer architecture, as compared with a new and promising classification scheme introduced in [3], [4]. We would, however, not claim that the proposed classification scheme will cover all computer structures which will arise in the future. We do show that the proposed scheme does cover several very interesting structures which cannot be placed at an appropriate point in the scheme of Flynn [1] and Feng [2].

The justification of the proposed scheme is that it should be useful in classifying structures and concepts which will emerge in the next years, and be of use to the designers of these structures. A further justification of the scheme is that the elements of the classification scheme can be composed and decomposed by operations which are suitable for the purposes of the computer architect.

2. Contemporary Classification Schemes

Existing classification schemes differ in the information on which they are based. For instance M. Flynn [1] bases his scheme on a 'data stream' and an 'instruction stream'. By combining these simple concepts he can classify many of the new computer structures. In contrast, Feng [2] emphasises the number of bits which are processed simultaneously. These schemes are outlined in sections 2.1 and 2.2 in order to contrast them with the scheme outlined in chapter 3. In section 2.3 the definitions of multiprocessing proposed by the American National Standards Institute [5] and by Enslow [6] are discussed.

2.1 Flynn's Classification

Flynn proposed in 1966 a classification based on the instruction streams and data streams. In the conventional Princeton type computer a single data stream is processed by a single instruction stream. This is described as SISD (single instruction single data).

In an array computer such as ILLIAC IV, a single instruction stream processes many data streams. Such a computer is known as SIMD (single instruction multiple data). In ILLIAC IV 64 copies

*On sabbatical leave from: Institut für Mathematische Maschinen und Datenverarbeitung (III), University of Erlangen-Nuremberg, Martensstr. 3, D-8520 Erlangen, Fed. Republic of Germany.

of the same instruction are executed simultaneously by 64 arithmetic units. The Goodyear STARAN is also a SIMD computer. It differs from ILLIAC IV in many respects, in particular in being an associative array processor.

MISD is an abbreviation for multiple instruction single data. Some authors include various types of pipeline computers in this class though it is doubtful whether this is appropriate, and it is unsatisfactory because it does not distinguish between the three kinds of pipelining (see section 3.3 below).

MIMD is an abbreviation for multiple instruction multiple data. Here multiple processors are working on multiple data streams. The simplest case is where each processor is executing its own program on its own data. The processors can be connected via a bus system or can access multi-port memory. The classification does not contain any information about the type of connection used.

Flynn's classification is illustrated by fig. 1, where many contemporary computers can be classified by assigning them to one of the four vertices of a graph. However, the classification does not fully satisfy the needs of computer architects because it is not fine enough and because the interpretation of the class MISD is not clear (cf. [7]). In the literature many authors restrict themselves to the classes SISD, SIMD, and MIMD. A further difficulty occurs if a computer contains both parallelism and pipelining.

2.2 Feng's classification

Feng [2] classifies according to the wordlength, i.e. the number of bits which are processed in parallel in a word, and the number of words which are processed in parallel. A computer structure is represented by a point in a plane (fig.2) where the abscissa is the wordlength (normally 12, 16, 24, 32, 48, 60 or 64), and the ordinate is the number of words processed in parallel. The latter can be determined by the number of processors. For example C.mmp which contains 16 PDP-11's with wordlength 16 bits is represented by (16,16). The ordinate can also be determined by the number of arithmetic and logical units in an array processor. Thus ILLIAC IV is represented by (64,64).

Thus Feng's classification does not allow to distinguish between multiprocessors like C.mmp and array processors. This caused Enslow [7] to represent C.mmp in "gang" mode by (16,256). But C.mmp in gang mode can be regarded as similar to ILLIAC IV, with 16 ALU's executing a single program, which would give the point (16,16) which is the same as when gang mode is not used. The classification also does not distinguish between autonomous processors which execute programs and ALU's which execute operations, i.e. it does not distinguish between processing levels.

The TIASC (Texas Instruments Advanced Scientific Computer) is represented as (64,2048). The number 2048 os obtained from the 4 pipelines each consisting of 8 stages with 64 bits. However the number 2048 can be obtained in many ways, e.g. 8 pipelines, 8 stages, 32 bits. Thus the classification cannot represent a multiple pipeline structure like the TIASC accurately.

It is also not possible to represent the pipeline structure at the program level of PEPE. PEPE is characterized as (32,16), and the fact that each set of data (up to 288, each representing a flying object) is processed successively in three different ways is not represented. This is performed in three separate series of ALU's, and we can regard this as a three stage macropipeline (cf. section 3.3).

The lack of a rigorous definition of pipelining in the context of Feng's classification scheme leads to difficulties in classifying structures containing both pipelining and parallelism. Thus the scheme is not entirely satisfactory for the computer architect either.

2.3 Definition of Multiprocessing

Similarly to classification schemes, if definitions are too narrow, some viable computer structures may be excluded from consideration.

The American National Standards Institute [5] defines a multiprocessor as:

"A computer employing two or more processing units under integrated control." Manufacturers of systems containing two to four processors did not find themselves in conflict with this definition. The definition did not exclude future developments in computer architecture, but does not seem to have had any impact on contemporary architecture. Subsequently Enslow suggested a more detailed definition in his excellent book [6] which included

1. two or more processors, having access to a common memory, whereby private memory is not excluded,

2. shared I/O,

3. a single integrated operating system,

4. hardware and software interactions at all levels,

5. the execution of a job must be possible on different processors,

6. hardware interrupts.

We will concentrate on the first characteristic:

A common memory is mandatory. Such a structure is shown in fig. 3. It is easily seen that as the number of processors increases the congestion in the access to the common memory will also increase. Thus Enslow's definition seems to exclude systems containing very large numbers of processors. Microprocessors costing a few dollars are now available, so that systems containing thousands of processors are now possible. Some of the more progressive pro-

jects of computer architecture such as PRIME [9] are also excluded. On the other hand some structures which satisfy Enslow's definition are subject to severe limitations on their expandibility and application due to their use of an expensive crossbar switch [10].

Thus Enslow's definition does not either satisfy the requirements of contemporary computer architecture.

2.4 The Influence of Classification Schemes and Definitions

We have tried to show in the previous sections that definitions and classification schemes have their limitations and can prove a hindrance beyond a certain point. The computer architect should recognize when this point has been reached, and consider whether an entirely new classification scheme or definition is needed, which will ideally include all existing structures within a particular area and also all structures which will be considered in this area in the future. There is no doubt that one should consider very carefully the consequences of introducing a new classification, because of its possible educational and normative effects.

3. The Erlangen Classification Scheme

3.1 Introduction

The Erlangen classification scheme (ECS) was developed mainly in order to avoid the drawbacks of existing classification schemes, as outlined in section 2.

The basic requirements are

1. the objects to be classified should not be unnecessarily restricted. Any kind of computer system - in particular parallel processors, array processors, multiprocessors, pipeline processors must be classifiable in the scheme;

2. the classification must be sufficiently fine to express those differences between the objects considered important;

3. the classification must be unambiguous.

The classification scheme developed was also found to be a useful technique in computer architecture, in the sense that:

4. Composed computer configurations can be described by using operators which are applied to primitive elements of the scheme.

5. It can be used in evaluating architectural configurations, in particular with reference to cost.

6. It provides a measure for the flexibility of a system.

7. It provides a starting point for scheduling of flexible structures.

The objects of the classification are not necessarily computers only. This will be amplified below. The flexibility mentioned in 6. above is connected with the fact that a computer can be represented by more than one point in the classification. The various points which represent a computer will be referred to as modes. The more modes a computer has, the more choice of mode it has for a particular application, and so the greater is its flexibility.

The classification scheme can be used for algorithms as well as for computers, and demonstrates the inherent partitioning of the algorithm into parallel sections and pipeline stages. The classification of algorithms must then be related to the classification of the computers on which they are to be run. In general, jobs must be investigated to identify the classes of the algorithms contained, and matched to the classes of the computers on which they are to be run. A more detailed discussion of this question will be given in another paper.

3.2 Parallelism

Our classification aims at characterizing the parallelism and pipelining present in a computer system. The connections between the processors and the memory blocks are not included in the classification. It is assumed that the connections can carry the expected traffic and provide the required availability. In such a case the performance of the system is mainly determined by the processors, including their capability to transfer information.

The classification is based on the distinction between three processing levels:

1. Program control unit - Using a program counter and some other registers, and, in most cases, a microprogram device, the PCU interprets a program instruction by instruction.

2. Arithmetic and logical unit - The ALU uses the output signals of a microprogram device to execute sequences of microinstructions according to the interpretation process performed by the PCU.

3. Elementary logic circuit - Each of the microoperations which make up the microoperation set initiates an elementary switching process. The logic circuits belonging to one bit position of all the microoperations are called an ELC.

A computer configuration can include a number of PCU's. Each PCU can control a number of ALU's all of which perform the same operation at any given time. Finally, each ALU contains a number of ELC's, each dedicated to one bit position. The number of ECL's is commonly known as the word-length.

If we disregard pipelining for the moment, the number of PCU's, ALU's per PCU, and ELC's per

ALU form a triple, written

t (computer type) = (k, d, w).

We give some examples of the triple, where we assume that the reader is familiar with at least some of the computers:

t(MINIMA) = (1,1,1)

The "classical" serial computer. Some early European computers were of this form.

t(IBM701) = (1,1,36)

An example of the early "parallel" (on the 3rd level) Princeton computers.

t(SOLOMON) = (1,1024,1)

The historical concept of an array processor.

t(ILLIAC IV) = (1,64,64)

The famous array processor developed at the University of Illinois (without PDP 10).

t(STARAN) = (1,8192,1)

The well-known associative array processor (without host and sequential control processor) fully extended (32 frames of 256 bits each).

t(C.mmp) = (16,1,16)

The Carnegie-Mellon University mulit-mini project using 16 PDP-11's.

t(PRIME) = (5,1,16)

The University of California, Berkeley, project in which time-sharing is replaced by multiprocessing.

The different systems exhibit different kinds of parallelism, which is uniquely attached to one of the three levels. The numbers which make up the triple show this directly.

At first sight, the triples are able to classify all viable structures, particularly in regard to parallelism. But although parallelism is the most important phenomenon in contemporary computer architecture, pipelining must also be considered. The examples above exhibit parallelism but not pipelining. In the next section the classification is extended to include pipelining.

3.3 Pipelining

Pipelining can also be implemented at the three levels described in section 3.2, i.e. 1. PCU, 2. ALU, and 3. ELC.

For example level 3 pipelining is the well-known pipelining of the arithmetic unit used in the CD STAR-100 and the TIASC. The STAR-100 uses a four stage pipeline and the TIASC an eight stage pipeline.

An arithmetical pipeline can be regarded as a "vertical" replication of ECL's, compared with the "horizontal" replication used in a parallel ECL. It is therefore reasonable to multiply the number of ECL's, w, by the number of stages in the pipeline, w', to characterize the ALU. For the TIASC we have then

t(TIASC) = (1,4,64×8).

The multiplication sign will be used at all levels to separate the number representing the degree of parallelism from the number representing the number of stages in the pipeline.

The next higher level of pipelining is instruction pipelining. This involves the existence of a number of function units which can operate simultaneously to process a single instruction stream. It is based on the inspection of instructions prior to execution to identify those instructions which can be executed simultaneously without conflict. This is done by a scoreboard, in the terminology of Control Data. These instructions are executed as soon as a suitable function unit is free. This technique is referred to as "instruction lookahead", "instruction pipelining", or "parallelism of function units".

A classical example of this kind is the CD 6600 computer. Disregarding for the moment the input-output section (i.e. the peripheral processors), the internal structure of CD 6600 with 10 function units becomes:

t(CD 6600 central proc.) = (1,1×10,60).

The 10 units in this case are highly specialized (e.g. floating point multiplication, integer addition, incrementation, etc.) and therefore a gain of a factor of 10 cannot be achieved. The real factor depends on the special program actually running. An average of 2.6 is a typical figure according to information available from Control Data. A combination of several function units of the same type seems to be quite reasonable regarding the better utilization of equipment on the one hand and the now available large-scale integration technology on the other hand. These latter considerations nevertheless are not directly a subject of this paper.

Finally, we have to consider the pipelining concept of level 1, which is so far not very known. This concept can be called "macro-pipelining"[12]. Assuming that a data set has to be processed by two different tasks sequentially, then it can be performed in two different processors, each one processing one task. The data stream then passes the first processor (1. task), is stored in a memory block, which the second processor also has access to, and will then pass the second processor (2. task). Since both processors can work at the same time (on different data), the effective processing speed can be in an ideal case doubled in comparison with the use of only one processor.

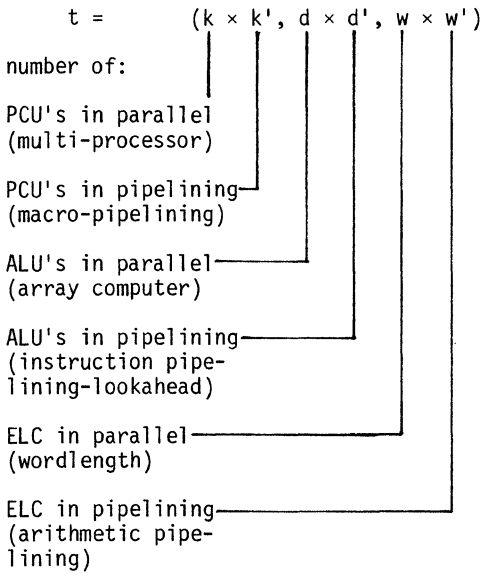
In such a way stepping from processor to processor data are 'refined' [12] on one hand or are 'integrated' [13, 8] in the case of ordinary differential equations on the other hand.

The PEPE array (without the host installation) then is characterized as

t(PEPE) = (1×3,288,32) (3-fold macropipelining).

Summarizing now, the triple has been extended to a sixtuple to incorporate pipelining. Nevertheless, we keep calling it a triple because the three levels of consideration (as introduced in 3.2) suggest that we think in three terms, which have to be extended in some cases by an additional term, attached to the other value (of the same level) by using the sign ×.

The triple now reads as follows:



All entities are independent of one another. All combinations therefore can appear.

Regarding the 'completeness' we claimed in section 3.1, we would have to prove that, apart from the three levels mentioned in section 3.2, no essential other level can be defined, and that there are also no phenomena apart from parallelism and pipelining. This is not pointed out in detail here, because this paper centers on another point, the impact of classification schemes on computer architecture. But there is some evidence regarding the completeness of our classification. While there are some modifications in details, how the level 2 pipelining is designed, there are no doubts about the other levels. With respect to parallelism and pipelining there is an exclusive duality as is known from other fields of science where parallelism and serialism also appear.

Regarding the triple notation, we introduced the following simplifications:

k=1, or k'=1, or d=1 etc. mean, respectively, the simple cases, in which no parallelism or pipelining appear;

we write then

$$(1\times k', d\times d', w\times w') = (\times k', d\times d', w\times w') \quad \text{if } k' \neq 1$$

$$(k\times 1, d\times d', w\times w') = (k, d\times d', w\times w')$$

$$(k\times k', 1\times d', w\times w') = (k\times k', \times d', w\times w') \quad \text{if } d' \neq 1$$

$$(k\times k', d\times 1, w\times w') = (k\times k', d, w\times w')$$

$$(k\times k', d\times d', 1\times w') = (k\times k', d\times d', \times w') \quad \text{if } w' \neq 1$$

$$(k\times k', d\times d', w\times 1) = (k\times k', d\times d', w)$$

If there is any form of pipelining then the character × is preserved in the corresponding level. In the case of no pipelining the triple degenerates to

t(MODEL) = (k,d,w).

This convention contributes to the clearness considerably as well as to the transparency of notation. Therefore we will use this convention in the following.

3.4 Operations on Triples

As a triple characterizes a computer structure of a certain homogeneity, a combination of triples connected by an operator can denote

a) a more complex computer structure (as given e.g. by a special I/O section of processors or by a special host, which are connected to a specific computer configuration);

b) a selection of operation modes of a structure, which can be used alternatively, fitting to different needs, according to the algorithmic nature of different applications.

It should be noted in connexion with b) that for any application there can exist a number of algorithms, each one fitting a different computer structure. E.g. one algorithm which is a solution to a given problem can be highly suited for execution on a conventional Princeton type computer, while another may be better suited for a parallel or pipelining computer.

The forementioned computer CD 6600 would read its complete structure, using a multiplication sign ×:

t(CD 6600) = (10,1,12) × (1,×10,60).

The first term on the right hand side of "="

denotes the existence of ten processors of a simple structure with a wordlength of 12 bits. The second term is the characterization of the nucleus of the CD 6600, as it was given earlier. The multiplication sign visualizes the fact that all algorithms (programs) must be forwarded through the peripheral processors first, in order to be processed then in the central processor (1,×10,60).

Another example of contemporary computer architecture is PEPE (Parallel Element Processor Ensemble). Its host is one CD 7600 with the characteristic

$$t(\text{CD 7600}) = (15,1,12) \times (1,\times 9,60).$$

PEPE then becomes

$$t(\text{PEPE}) = (15,1,12) \times (1,\times 9,60) \times (\times 3,288,32)$$

where the last term (×3,288,32) corresponds to the actual PEPE structure. As, in this example, a certain flow of information penetrates the three structures, the sign × is used between the corresponding terms.

The structures characterized by the primitive terms in these examples are very different. Therefore a further condensation of the presentation is not suggested. A further decomposition can be indicated, e.g. by the use of other operators, for instance in the special case of a CD 7600 by

$$(15,1,12) \times (1,\times 9,60) =$$

$$\underbrace{[(1,1,12) + (1,1,12)+\ldots+(1,1,12)]}_{\text{15 times}} \times (1,\times 9,60),$$

where $(n,d,w) = \underbrace{(1,d,w)+(1,d,w)+\ldots(1,d,w)}_{\text{n times}}.$

We note that the operators × and + again reflect parallelism and pipelining in a certain sense. The last example shows 15 equal processors allocated in parallel. A given job (or task) will be forwarded to the central processor. It may also be necessary to allocate processors serially, if there are different tasks to be performed one after another. This is supported by the use of functionally dedicated processors, specialized to the respective task.

The last operator we have proposed so far is the 'alternative' operator v, which is to be understood as an 'exclusive or'. For the C.mmp project which can be used in three different kinds of operation modes, an expression becomes:

$$t(\text{C.mmp}) = (16,1,16) \text{ v } (\times 16,1,16) \text{ v } (1,16,16).$$

Similarly, the EGPA project (4×4 array of processors, 32 bits each, described e.g. in [13]) reads

$$t(\text{EGPA } 4\times 4) = (16,1,32) \text{ v } (\times 16,1,32) \text{ v } (1,16,32) \text{ v } (1,512,1).$$

The last term of this expression denotes the operation mode "vertical processing" in which the 16 processors are used, each as if it consisted of 32 one-bit processors working in parallel. 16 processors then result in an ensemble consisting of 16×32 one-bit processors. Information then is oriented to one-bit vertical streams (items) and the machine-word of the memory becomes what is called a 'bit-slice' in associative processors.

The operator v visualizes alternatives regarding the processing modes which can basically be used. An extended operator + can be used for a further partitioning of a system in which the ensemble is working. Scheduling algorithms have to be developed which have to centre on the best utilization of the system with respect to a given set of jobs. The scheduling problems, however, are not covered by this paper.

Yet, a remark on the 'flexibility' should be added. The number of available processing modes of a system seems to be a reasonable measure for its flexibility. Therefore we define (F=Flexibility):

$$F(t(\text{MODEL})) = \left| (k_1 \times k_1', d_1 \times d_1', w_1 \times w_1') \text{ v } (k_1 \times k_1', d_2 \times d_2', w_2 \times w_2') \text{v} \ldots \right|$$

where || gives the number of triples connected by the v sign.

For the examples presented above we have:

$$F(t(\text{C.mmp})) = 3 \text{ and } F(t(\text{EGPA } 4\times 4)) = 4.$$

In this section we wanted to show that a classification scheme becomes operable if it is carefully chosen.

Nevertheless, it is not the aim of this paper to introduce ECS[a] completely. We have used it as a further example of the discussion about the 'impact of classification schemes on computer architecture'.

4. Summary and Outlook

Some things which can be done with ECS (chapter 3), cannot be done with any of the systems mentioned earlier (chapter 2). Although we do not claim that ECS is the only possible classification scheme, we have found it useful for evaluating computer structures, throughput, flexibility etc.

In this respect ECS seems, as briefly presented here, to be an approach which can become a viable design tool. It classifies enough objects and it does not limit too seriously the set of objects.

[a] A rigid and more formal presentation of ECS is under preparation.

The only limitation we perceive so far is the inherently binary nature of the definition of w (wordlength). If a computer is based on another modulo-number system, then we would have to slightly modify the ECS as presented.

If, for historical reasons, we have to, for example, include the old mechanical calculating machines of Ch. Babbage, then it would be necessary to extend ECS. Also excluded from ECS are computers of the analogue type. But this limitation seems to be quite natural in that analogue data processing is quite different.

The only criticism which at this time can be made within the aims of this paper could center on the number of levels we introduced in chapter 3. There we defined a triple according to three processing levels. If perhaps in a later step of evolution a level above the program interpretation level will be created, then we would have to extend the triple to a quadruple.

But just this step to achieve a new level of computer structure is a real evolution step we are searching for at present. It was exactly for this that the classification scheme has been developed as a tool. About such an evolutionary step a decision cannot be made in advance. It is rather the ECS classification scheme and the operations defined on the elements (triples) which seem to be the appropriate starting point for investigations of that kind. We hope that ECS will not limit too narrowly a future development, for it includes all structures which so far have been proved as viable examples of computer architecture.

Acknowledgement

The assistance of Mr. R. K. Bell and Dr. V. Sigmund in the preparation of this paper is gratefully acknowledged.

References

[1] M. Flynn, "Very high computing systems", Proc. of the IEEE 54 (1966), 1901-1909.

[2] T. Feng, "Some characteristics of associative/parallel processing", Proc. of the 1972 Sagamore Comp. Conf., Syracuse University, 1972, 5-16.

[3] W. Händler, "On classification schemes for computers in the post-von-Neumann era", in: D. Siefkes (ed.), GI-4.Jahrestagung, Berlin, Okt. 1974, Lecture Notes in Computer Science 24, Springer, Berlin (1975), 439-452.

[4] W. Händler, "Zur Genealogie, Struktur und Klassifizierung von Rechnern", Parallelismus in der Informatik, Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, 9 (1976)/8, 1-30.

[5] "Vocabulary for Information Processing", American National Standard, X.3.12-1970.

[6] Ph. E. Enslow, Multiprocessor and Parallel Processing, John Wiley and Sons, New York, 1974.

[7] P. H. Enslow, "Multiprocessors and other parallel systems - an introduction and overview", W. Händler (ed.), Computer Architecture, Workshop of the GI, Erlangen, May 1975, Springer Verlag Berlin (1976), 133-198.

[8] W. Händler, "Aspects of Parallelism in Computer Architecture", Proc. of the IMACS (AICA)-GI-Symposium on Parallel Computers, Parallel Mathematics, Munich (March 1977), North Holland Amsterdam, to appear.

[9] H.B. Baskin, Borgerson and Roberts, "PRIME - a modular architecture for terminal-oriented systems", SJCC 1972, pp. 431-437.

[10] W.A. Wulf, C.G. Bell, "C.mmp - A Multi-Mini-Processor", AFIPS Conf. Proc. FJCC 1972 41, pp. 765-777.

[11] B. Whorf, "Language, Thought and Reality", M.I.T. Press, Cambridge, Massachusetts, 1963.

[12] W. Händler, "The concept of macro-pipelining with high availability", Elektronische Rechenanlagen 15 (1973), pp. 269-274.

[13] W. Händler, F. Hofmann, H.J. Schneider, "A General Purpose Array with a Broad Spectrum of Applications", W. Händler (ed.), Computer Architecture, cf. [7].

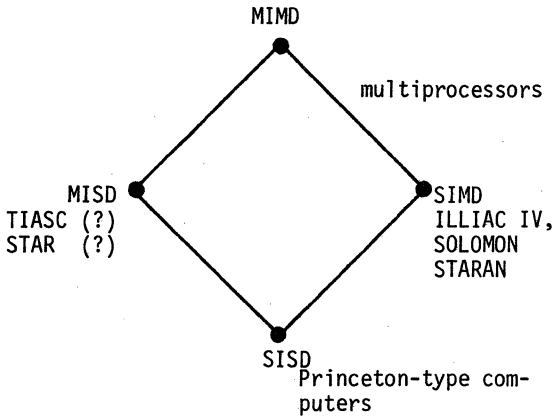


Fig. 1: M. Flynn's classification with some examples

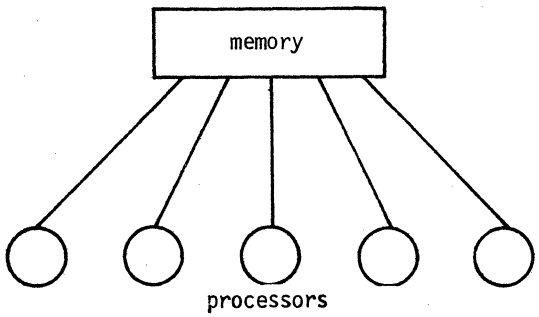


Fig. 3: P.E. Enslow's definition of a multiprocessor leads to "one common memory block" (private memory blocks, owned by a processor exclusively, are not excluded by the definition).

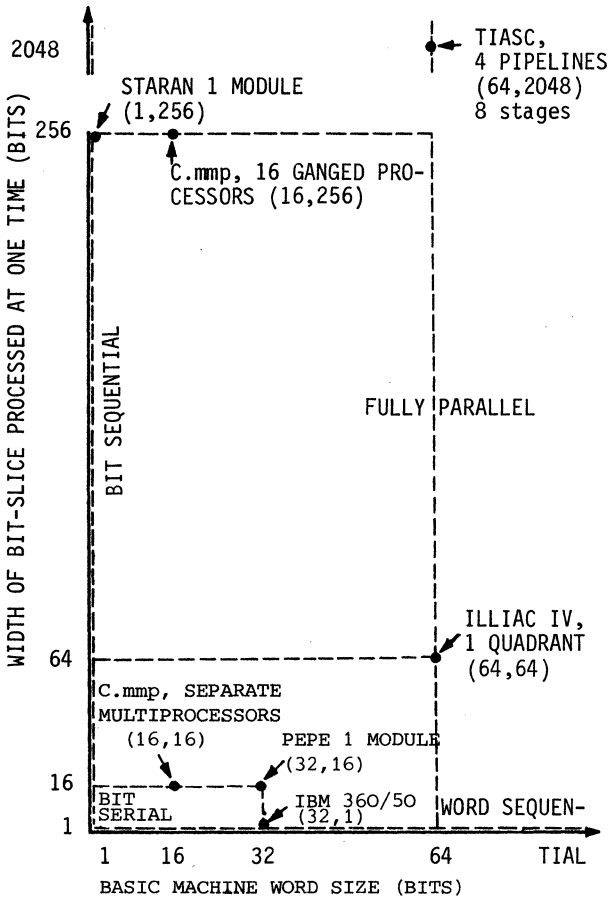


Fig. 2: T. Feng's classification with some examples

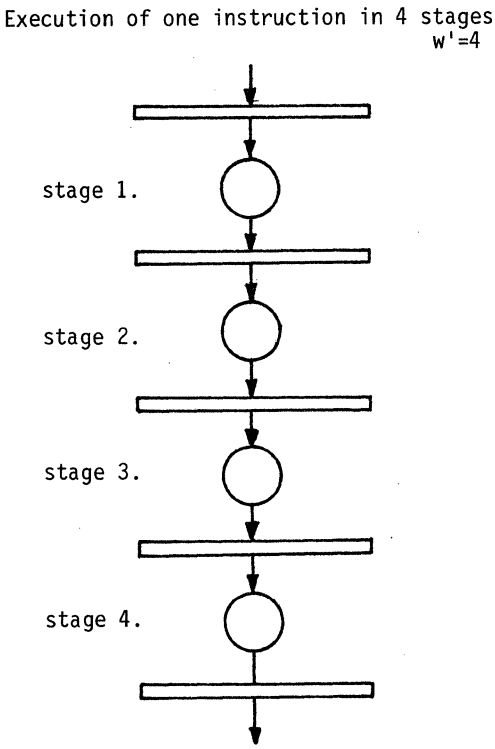


Fig. 4: Arithmetic pipelining (level 3 pipelining)

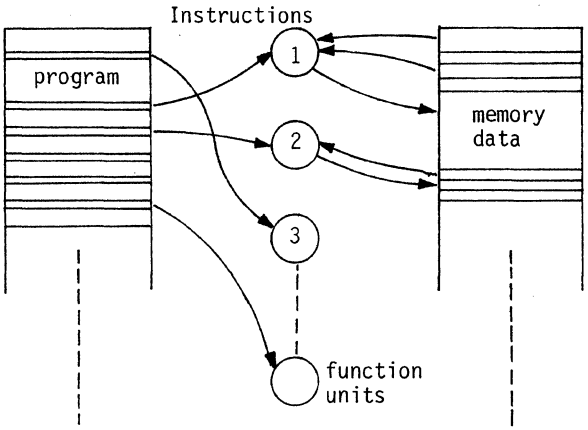


Fig. 5: Instruction-pipelining (level 2 pipelining)

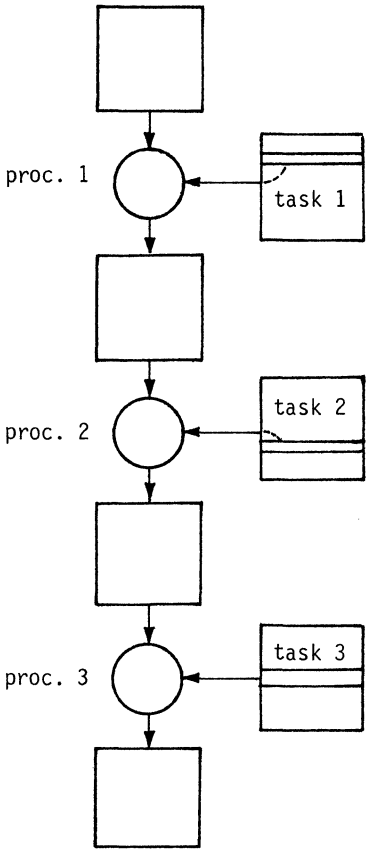


Fig. 6: Macropipelining (level 1 pipelining)

PARALLEL COMPILED INTERPRETATION

V. Sigmund
Institut für Mathematische Maschinen und Datenverarbeitung (III)
Universität Erlangen-Nürnberg, Martensstrasse 3, D-8520 Erlangen
Federal Republic of Germany

Abstract -- Program execution in some processors (analog, array, pipeline, data-flow, single-assignment, etc.) reflects the structure of compound operations described in the user program. However, the original user description of these operations has to be first transformed (transported, translated, collected, interpreted) before the actual execution can begin. The structure of compound operations in this transformation can also be exploited (parallel or pipelined data transfer by I/O-devices and channels), and, under certain conditions, even in transformation and execution together (overlapped instruction fetch/ execution in lookahead processors). An extensive application of this concept in the successive interpretation of (very) high-level languages is suggested by the current trend of hardware prices.

1. Introduction

The exploitation of parallelism in computers has been preceded by the recognition of common structural features of computations, at least at some levels. For example, the need for the repeated transport of programs and data sets from the periphery to main memory resulted in the use of multiple independent I/O-channels to the main processor, which perform this transport in parallel. Another example is the repeated transport of inctructions from the main memory to the processor, which resulted in the overlapping of the instruction fetch and execution phases. There is also quite often a need for the repeated execution of similar operations on elements of data arrays, which led to the construction of array and pipeline processors.

One feels that the types of parallelism in these three examples are somehow different, but it is difficult to characterize this distinction using Flynn's classification [1] according to whether instruction streams and data streams are processed simultaneously. It is even sometimes not clear whether pipelining can be considered as MISD, i.e. multiple instruction stream/single data stream processing, and if it is so, why (Enslow [2]). The classification scheme introduced by Händler [3],[4],[5] which consists of the numbers of parallel and pipelining function units simultaneously active at the three main processing levels, bit operation level, machine instruction level and program level, possibly together with similar numbers for I/O-, front-end or other coupled special processors, gives us much more information about the computer. This measure is quantified, similarly to the parallelism measure introduced by Feng [6], thus enabling one to compare different machines according to their degree of parallelism, and also giving us a much more detailed picture of the machine structure, which is important for our intentions here. One should always explicitly state what level is considered in studies of parallelism, etc.; speaking of the "parallelism of the ILLIAC IV computer" or of the "serial processing of the von Neumann computer" is of little value.

We find it useful to consider the computations together with the functions which they should implement. The user is interested only in the functions he wants to have computed by the machine, i.e. in the external behaviour of his program; from his point of view the machine has been constructed in order to execute these functions. Regretably, the machine has much more to do than this execution. The user program and data are mostly placed in some user space, e.g. a terminal, disc or tape, but the machine can perform the execution only when the instructions and data items involved have been brought into the machine execution space. After the execution another transport is necessary in order to give the results back to the user and to free the execution space for the forthcoming execution. We would like to speak of a transformation rather than transport or transfer, since the action can also involve some encoding of instructions or data items; in a broader sense also, for example, program compilation or subprogram collection belong to this category. Thus, although the user's only aim is the execution of his program upon his actual data, or more precisely the execution of the functions composing the external behaviour of his program, the machine has to perform both the execution and transformation. Sometimes also the transformation may be explicitly programmed by the user so that its description constitutes a part of his program, but the characteristic of the transformation is that it does not influence the external behaviour of the user program.

Now, considering the cases of parallel computations in a machine, we can divide them according to the category into which the implemented functions belong. In the first example above the parallel action of multiple I/O-channels forms a part of the transformation. In the second example the instruction fetch belongs to the transformation while the operation performed by the instruction execution upon the actual data items

often forms a part of the external behaviour of the user program. Operations performed during the parallel execution in an array processor belong in most cases to the external behaviour of the user program. We could speak therefore of transformation parallelism, transformation/execution parallelism and execution parallelism, respectively. We look closer at the transformation and execution and at the potential inherent in the extensive exploitation of their common structure in the second part of the paper (sections 4,5). If the transformation is a transport, examples of parallel transformation and execution at several levels in computers can be given: job execution overlapped with the transfer of the next jobs of the job stream from the periphery to the main memory, overlapped fetch/execution in lookahead processors, etc., and proposals for its exploitation have also been made across the whole storage hierarchy (cf. e.g. Dennis [7], Madnick [8]). The aim is to achieve the maximal possible execution speed with only very small run-time storage requirements, by having additional processing capacity to perform the transport overlapped with the execution. The price of processing elements relative to memory has fallen rapidly. But the same principle could also be exploited across the whole hierarchy of the successive interpretation of (very) high-level languages. Some proposals in this direction have been made e.g. by Miller and Cocke [9]. Consistent with the usage of the terms "compilation" and "interpretation" in processing of high-level languages, one could then speak of parallel compiled interpretation. The aim here could be characterized as, in addition to that above, saving peripheral and mass storage, for their prices remain also relatively high. Of course, the processing elements for the "on-fly" compilation must be more intelligent than for a simple transport, but this should be no problem today. A sufficient condition for the parallel transformation/execution is that the transformation preserves the program ordering (section 5).

However, before turning to the transformation and execution, we consider in more detail their common structural features (sections 2,3). The term "parallelism" is namely by no means exhaustive for alle that can be observed in computations (nor even in the papers presented at this conference, so that the words "parallel processing" in its name are partially misleading). We start with the most natural computations which are performed in the evaluation of compound operations, as described by algebraic expressions and as has been exploited by man for several thousand years in analog devices, and later e.g. in combinational circuits (section 2). We prefer to use standard terminology although it became quite modern in some places to speak about "transitions", "tokens" and "firing". Then we look at the structure of programs and machines. As for the notion of "structure", mentioned almost everywhere today, we find as its best explanation its usage in mathematics: "The fundamental structure problem of algebra is that of analyzing a given algebraic system into simpler components, from which the given system can be reconstructed by synthesis." (Birkhoff [10], p.55).

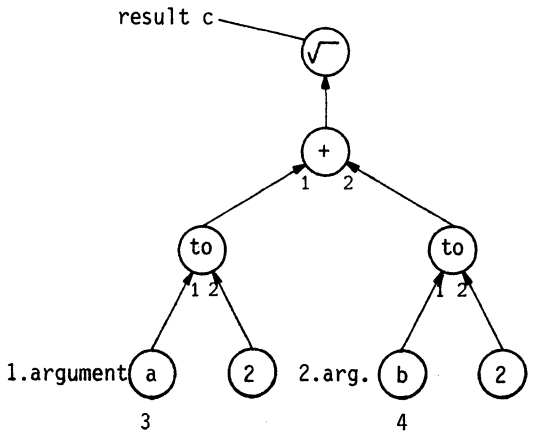
We begin with the synthesis of complex programs and machines from simple ones based on the exploitation of similarity of their components. By "complex" we mean here and in the following "more intelligent". This synthesis has been mostly motivated by economical factors, e.g. at the time of the first electronic computers the number of memory cells and function units required for the execution of a computation had to be kept small. The synthesis can be roughly characterized as "trading space for complexity", sometimes also as "trading space for time". Steps in the other direction, i.e. towards the analysis of complex machines into simpler ones can be observed in the recent work in computer architecture towards distributed processing, parallel processing (e.g. array and pipeline processors), or, as we prefer to say, structured processing (e.g. data flow machines and single assignment machines, cf. Tesler and Enea [11], Dennis [12], [7], Dennis and Misunas [13], Rumbaugh [14], [15], Plas et al [16], search mode and interconnection mode configurable computers, cf. Miller and Cocke [9], macropipelining, cf. Händler [17], and many other designs described as reconfigurable, restructurable, varistructured, variable, etc.). One could characterize this roughly as "trading complexity for space", when e.g. a complex central "Allesköner" is replaced by a number of simpler distributed processing elements, and sometimes also as "trading time for space", when e.g. execution time is saved by the use of a greater number of processing elements, in accordance with the recent developments of hardware prices and the growing need to reduce execution time (section 3).

2. Compound Operations and Related Computations

The use of compound operations and their description by expressions is widespread not only in mathematics. Consider the very well know description (1) of how to get the length of the

$$c = \sqrt{a^2+b^2}, \qquad a = 3,\ b = 4 \qquad (1)$$

hypotenuse of a right-angled triangle, given the lengths 3 and 4 of the sides adjacent to the right angle. Perhaps a more suggestive picture is (2).



(2)

Note that (1) and (2) are essentially two drawings of the same graph where in the first drawing some details such as edges, circles for nodes and the ordering of the argument nodes are omitted for reasons of economy (but are implicitly present), and where the shape of (1) is dictated by the typographic needs of machine print.

In order to be able to execute the described computation for the given arguments 3 and 4 (or, as algebraists may prefer to say, to evaluate at the point (3,4) the compound operation

$$R^+ \times R^+ \to R^+ : (a,b) \mapsto c \qquad (3)$$

corresponding to the expression $\sqrt{a^2+b^2}$, cf. Grätzer [18]), we must first have learned at school that the operators denote certain operations on nonnegative real numbers R^+, i.e. we must know what specific algebra we are dealing with, cf.[18].

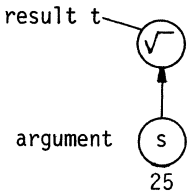
For example $\sqrt{\ }$ denotes the square root operation

$$R^+ \xrightarrow{sq.r.} R^+ : s \mapsto t$$

which sends a nonnegative real number s to that nonnegative real number t for which $t^2 = s$. Given the number 25 as the argument in the following simple computation description

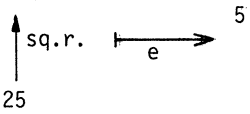
$$t = \sqrt{s}, \quad s = 25$$

or, more explicitly, (4), the execution of the described computation (the evaluation of the operation square root at the point 25) means that we determine the number 5 using our knowledge of the operation square root and having the argument 25. (The evaluation map



(4)

$e : R^{+R^+} \times R^+ \to R^+ : (\text{square root}, 25) \mapsto 5$ is very important in mathematics, cf. Mac Lane [19], p.18, 61, 96, 216). We depict this in (5).



(5)

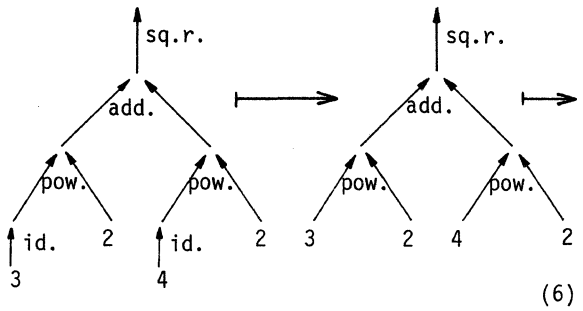
The evaluation of a compound operation described by an expression such as (1) or (2), is defined inductively over the height of an operator occurrence in the expression, i.e. the length of the maximal path from the leaves to the corresponding node labelled by this operator in the tree such as (2).

In detail: Let $f' : A^n \to A$ denote the operation corresponding to the n-ary operator f of the algebra under consideration. Then the value of the compound operation corresponding to an expression in variables $x_1,x_2,\ldots,x_k$ at the point $(a_1,a_2,\ldots,a_k) \in A^k$ is defined by:

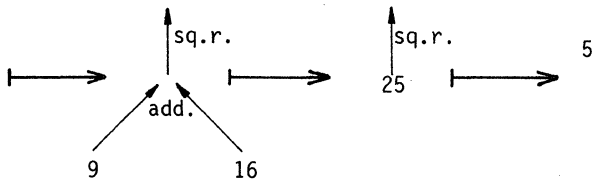
(i) the value at a leaf-node labelled by x_i is a_i
" " " " " " by a 0-ary operator $b \in A$ is b

(ii) if an n-ary operator f is the label of a node of height $h \geq 1$ in the tree and $b_1,b_2,\ldots,b_n$ are the values at its argument nodes, then $f'(b_1,b_2,\ldots,b_n)$ is the value at this node.

In the above case, the induction proceeds as shown in (6).

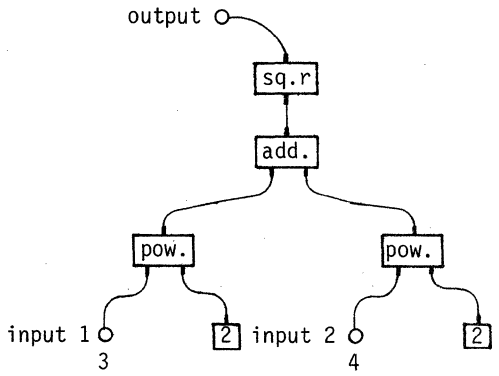


(6)



For illustration, in these induction steps the values of the intermediate results are transformed by the operations and moved along the tree from the leaves to its root.

The above interpretation and evaluation of (1) is as old as the usage of the expressions itself. However, quite usual, and, in fact, also very old, is a physical implementation of this concept. If we have functional units for the required operations, the function (compound operation) described by (1), (2) can be implemented in the combinational network (7), for example by moving and

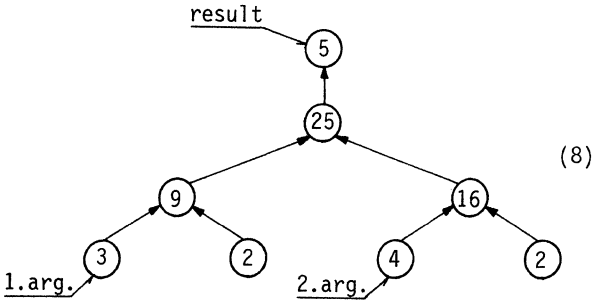


(7)

transforming electrical signals along this tree from the leaves to its root.

We could call the expression (1) (or the directed graph in (2)) a program scheme or machine scheme and the corresponding directed graph in (6), (7) a program or machine implementing the function (3), consistently with the common usage of these notions (cf. e.g. Arbib and Give'on [20]).

The term "computation" is usually used for the sequence of intermediate results or configurations (consisting of the intermediate result and the state) of the corresponding program or machine for a given argument value (cf. e.g. [20], Elgot and Robinson [21], Elgot [22]), in accordance with the intuitive meaning of this notion. More appropriate would perhaps be computation run, thus leaving the term computation to denote the set of all computation runs for all allowed argument values, similarly to the term "function" which can be interchangeably used for the set of all corresponding ordered pairs "(argument, result)". Considering the ordered pair consisting of the first and the last element in the sequence of the intermediate results of each computation run, and the set of these pairs corresponding to the computation (i.e. to the set of all computation runs), we get precisely the function implemented by the computation, sometimes called the external behaviour of the computation. In our case of unary and binary operators in (2), the computation run for the arguments (3,4) is the directed graph (8) of



(8)

the intermediate results (instead of a sequence as in the case of only unary operators) giving the assignment (3,4) ↦ 5 as its external behaviour. For the set $R^+ \times R^+$ of all admissible arguments we get a set of similar graphs as the corresponding computation, and their respective argument and result nodes give us precisely the required function (3) as the external behaviour of this computation (cf. Arbib and Give'on [20]).

We remark that the only ordering of operator occurrences in (1), (2) and of the intermediate results in each computation run such as (8) is that which is induced by their argument-result relation, directly shown by the arrows in (2) and (8). Operator occurrences which are incomparable according to this partial order, as for example the two nodes labelled with the power operator "to" in (2), are often said to be inherently parallel.

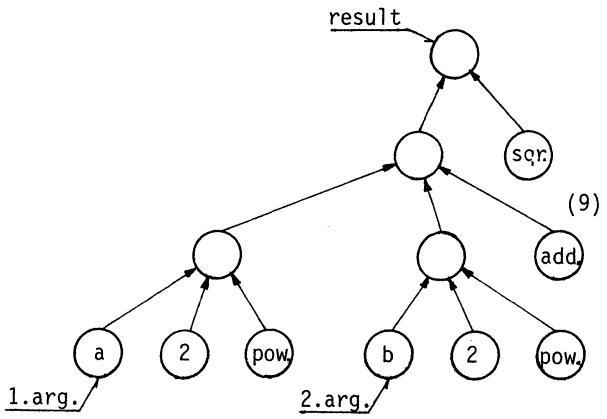
It was our intention to use standard terminology for well-known phenomena such as expressions, their interpretation and evaluation. This is not only convenient but, moreover, it enables one to exploit results already known (cf. Give'on and Arbib [23] for a study of a structure of the compound operations described by a given operator set). Some authors prefer to use notions like "tokens which contain values", "an actor with a token on each of its input arcs" which is "enabled and sometime later will fire" and quite often they speak of "data driven execution" in a similar context.

We note that the program (machine) scheme and the corresponding program (machine) have similar graph structures (cf. (2) and (7): the underlying directed graphs are indeed isomorphic). Because these graph structures are our main concern in the following, we shall feel free to use the notions scheme, program and machine interchangeably, as convenient, hopefully without causing any confusion.

3. Structure of Computations and Machines

The notions computation run and computation, used in the last section, can be considered as the prior and most natural concepts from which the notions of "operation","operation composition", "compound operation" and the corresponding "operator" and "expression" are obtained by synthesis. Concepts such as "interpretation" and "implementation" represent steps in the other direction, i.e. analysis. To this extent we can speak of the structure of computations which is reflected in the most simple programs and machines such as (2) and (7) of section 2. However to explain this in more detail is outside the scope of this paper so that we study only the structure of programs and machines in the following, starting with the simple machines of the last section.

We depict in (9) once more the program (machine) implementing the compound operation $R^+ \times R^+ \to R^+$, corresponding to the expression $\sqrt{a^2+b^2}$. The situation is now more symetric



(9)

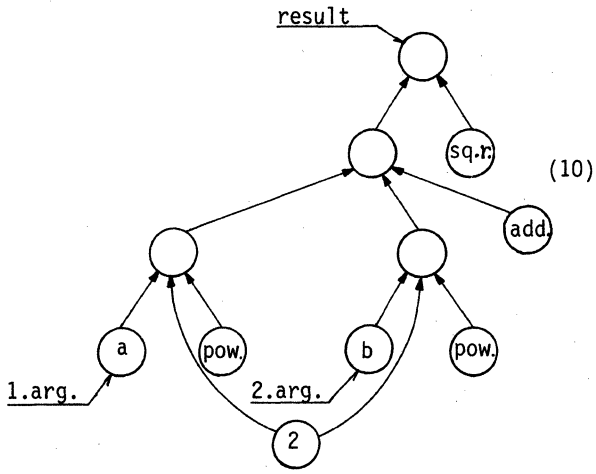
with respect to data items and operators, since operators can also be treated as data, e.g. they can be changed. The only operation acting upon all

the data items and operators is the evaluation of section 2, which we left anonymous. We call the nodes such as those labelled a,b,2 or without label in (9) data nodes and the nodes labelled with operators operator nodes. Every programmer would probably be inclined to speak instead of the data locations or variables and the (operator code part of) instruction locations, but our data nodes can actually be memory data locations as well as general purpose registers or data lines of a bus, and our operator node can be the operator code part of the location in memory of an instruction, as well as a function unit implementing the corresponding operation or a control signal line of a bus.

The implementation (description) of computations by the simplest machines (programs) such as (7), (9) is not always economically feasible. For large computations this would require too many data and operator nodes. In the case of machines this means that the number of registers and function units is too big; for programs their size is too large, and the schemes (expressions such as (1)) become clumsy. However, looking at these simple machines (programs, schemes) we observe the similarity of certain parts: the same data items or operators occur repeatedly at different nodes, somtimes even rather large identical, or at least very similar parts of the machine occur repeatedly. In what follows we describe several quite common cases of synthesis, where such similar parts of machines (programs, schemes) are "coalesced". This happens at the cost of simplicity, since some new mechanisms such as control flow, subtroutine call must then be introduced into the machines (programs, schemes). Sometimes the saving of nodes is outweighed by the introduction of a new dimension into the machines (programs, schemes), the time. For each case of synthesis we show also examples of analysis of the complex machines into a greater number of simple ones, motivated by the recent development of hardware prices and by the increasing need to reduce execution time. The external behaviour of the corresponding computations remains in all cases unchanged, i.e. both the synthesis and analysis are "semantics preserving".

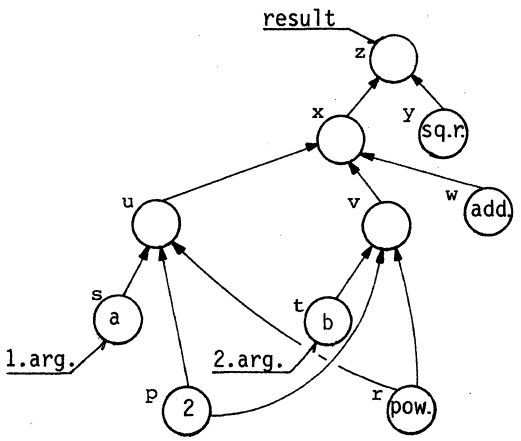
Multiple use of data nodes. Looking at the example of the machine (9) we see that the two occurrences of the data item "2" could be coalesced, thus obtaining the machine (10). The underlying directed graph is then no more a tree (cf. Arbib and Give'on [20]). This technique is quite common in programming. Such a saving of data nodes costs increased complexity of the machine: the data item "2" must be available for two references to it; increased execution time may also result if, caused by technical circumstances, one such multiple reference must wait for the completion of another.

Analysis example: the replication (broadcasting) of an argument with multiple references used in the packet communication architecture (cf. Dennis [7]).



(10)

Multiple use of operator nodes. The two nodes in (10) bearing the label "pow." can also be coalesced, cf. (11). A typical example is the use of function units in a centralized processor: there will be only one function unit for the operation "pow." which will satisfy all references to it.



(11)

Compared to the function units of (9), this happens at the cost of increased complexity of the single function unit in (11) which must be able to resolve multiple references; increased execution time may also result if one such reference must wait for the completion of another. A second example is the use of array operators in programming languages.

Analysis example: the provision of multiple function units for more frequent operations, e.g. two increment units in CD 6600. Another example is the provision of multiple function units for operations on data arrays in array processors, or the replication (broadcasting) of an array operator when executed on an array processor.

Re-use of data nodes. In the labelled graph (11) we have used explicitly the letters p, r, s, t, u, v, w, x, y, z for the nodes. Another possible representation of this graph is (12) which shows in another form the (finite) maps of the labelling

p=2, r=pow., s=a, t=b, u=r(s,p), v=r(t,p),
w=+, x=w(u,v), y=$\sqrt{\ }$, z=y(x) (12)

and assignment of nodes to edges depicted in (11). Now, observing the chain s→u→x→z of data nodes in (11) with the property that none of them has other immediate successors, we can use four copies (1,s), (2,s), (3,s), (4,s) of the same node s, with the chain ordering of 1→2→3→4, instead of the chain s→u→x→z. (An algebraist would say we take the direct product of {t} with the chain 1→2→3→4 together with the product order relation.) We get the description (13). If we call the chain 1→2→3→4 a "time sequence", we can say that the

p=2, r=pow., (1,s)=a, t=b, (2,s)=r((1,s),p),
v=r(t,p), w=+, (3,s)=w((2,s),v), (13)
y=$\sqrt{\ }$, (4,s)=y(3,s)

three data nodes u, x, z have been saved by re-using the data node s at three other time instants. The price to be paid is increased complexity of the machine, because we need a clock giving the time impulses 1→2→3→4; increased execution time may also result if the clock is slower than the gate times and conducting delays along the path s→u→x→z. Note that the program (machine) (13) cannot be depicted as a graph like (11) without introducing some new description conventions. Nor can it be implemented as a combinational network, since the clock functions as a delay unit between two successive uses of the data node s; we get a sequential network (cf. for example Hennie [24], chapter 1, for a discussion of combinational network/sequential network dichotomy in finite automata implementation).

We can save yet more data nodes by extended re-using of data nodes (14). Three data nodes, p, s and t, are sufficient instead of the seven data nodes in (11). However, the price to be paid is increased execution time by additional ordering of operators which were inherently parallel in(11).

1. p=2; 2. r=pow.; 3. s=a; 4. t=b;
5. s=r(s,p); 6. p=r(t,p); 7. w=+; (14)
8. s=w(s,p); 9. y=$\sqrt{\ }$; 10. s=y(s).

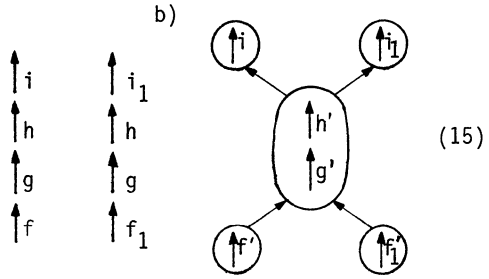
We use in (14) the convention that the value of the new ordering parameter 1→2→3→4→5→6→7→8→9→10 is written in front of a node on the left-hand side of "=" and the implicit assumption that an occurrence of a node on the right-hand side of "=" should be indexed with the last previously occuring index at this node; e.g. the full description of the fifth assignment would be (5,s) = (2,r) ((3,s),(1,p)).

Analysis example: the iterative array implementation of sequential circuits (cf. Hennie [24]). This is to some extent the principle employed in pipeline processors, data flow machines (cf. [7], [12], [13], [14]), single assignment machines (cf. [11], [16]), interconnection mode configurable machines (cf. [9]) and macro-pipelining (cf. [17]).

Re-use of operator nodes. Because of the symmetry of the programs (machines) with regard to the data and operator nodes, the same reasoning holds here as above. Typical examples are redefinable operators in some interpretive programming languages, e.g. Snobol, or microprogrammable function units in some computers.

Analysis example: the replacement of the all-purpose ALU of the CD 6400 by the dedicated function units in the CD 6600.

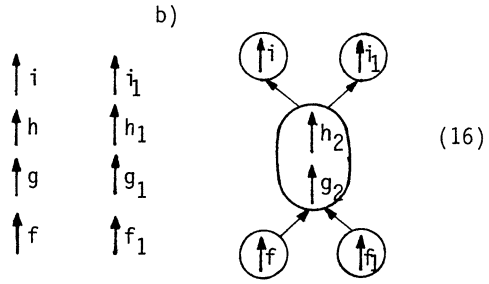
Shared use of description parts (branching). Let the two programs (machines) in (15a) contain identical parts consisting of g h. Then we can join them to the single program (machine) (15b), thus saving data and operator nodes. The price to be paid is increased complexity, viz. the introduction of a new mechanism, the control flow into the program (machine).



(15)

The operators f, f_1, g, h in (15a) have then to be extended to f', f_1', g', h' in which a new control flow parameter is taken into account. Examples can be found in ordinary programming.

Consider another case of the two programs (16a) where the domains as well as the value sets

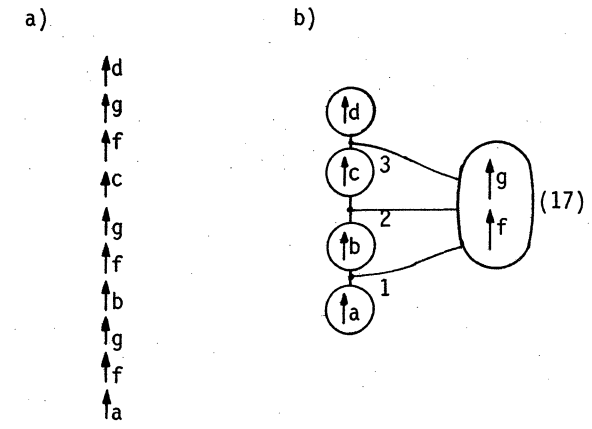


(16)

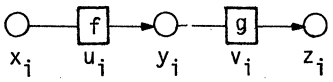
of both pairs of the operations g, g_1 and h, h_1 are disjoint. Then we can use the single program (16b) with g_2, h_2 being unions of the above pairs and with h_2 additionally producing a truth value for the control flow branch (cf. Elgot [25]).

Analysis example: the step from centralized to distributed control, e.g. from centralized "star" or bus interconnection to decentralized full interconnection of computer modules (cf. Anderson and Jensen [26]).

Re-use of program parts (subtroutine calls). Let a part consisting of the operations f g occur repeatedly in the program (17a).

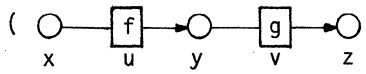


Then we can save data and operator nodes if we instead use multiple copies ($\frac{f}{g}$, i) (i=1,2,3) of the repeatedly occuring part (see (17b)), distinguished by a new parameter i. Again, calling 1→2→3 "time" we can say that we use the same program part in three different time instants. In detail: if the repeated occurrences are



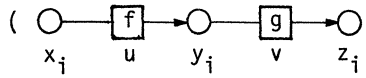
(i=1,2,3) (18)

with data nodes x_i, y_i, z_i and operator nodes u_i, v_i, then the use of (19) instead of (18)



(i=1,2,3) (19)

represents normal subroutine calls while (20) represents re-entrant subroutine calls. The price to



(i=1,2,3) (20)

be paid is increased complexity, viz. the necessary introduction of a new mechanism into the program, the subroutine call. Increased execution time may also result if one of the multiple calls of the same subroutine must wait for the completion of another.

Analysis example: some compilers generate "inline code" for each call of an intrinsic or library function subroutine, i.e. they generate full object code of the subroutine at each place corresponding to a subroutine call in the source program.

4. Transformation and Execution of Programs

In the last section we considered generally the structure of computations. In this section we want to make a difference between the computation implementing the external behaviour of the user program on his data, which we call simply execution in the remainder of the paper, and the computation performing the transformation of the user program and data, as outlined in section 1.

There exist different methods to perform the transformation necessary for the execution of a program upon its actual data.

Global program/global data transformation: First the whole program and all initial data are transformed, then executed, and then the final results are transformed back. The following are some significant features: Positive: (a) Because the whole transformation is performed in one piece, before and after the execution respectively, it is possible to analyze the transformation, as described in section 3, to the extent this is economically feasible (e.g. several I/O-channels which can transport simultaneously several data files needed for one program). (b) Because the whole execution is performed in one piece, it is possible to analyze the execution, as described in section 3, to the extent this is economically feasible (e.g. array processors, associative processors, arithmetic pipelining). Negative: (c) Large storage capacity is required for program and data in the (expensive) execution space (e.g. main memory required to store a large compiled program and the large data arrays to be processed by this program). (d) Too much transformation is performed on programs where only a small part of all instructions (operators) and data items actually occur during execution (e.g. loading of a segment of a segmented executable program when it is activated, where the whole program segment is transported into the main memory although possibly only a very small part of the segment will actually be executed; or a paging machine where a whole page of data is transported although possibly only a few data items of the page will actually be processed).

Local program/local data transformation: for a single user program statement that has actually received control: first the instruction (operator) and data items which are the arguments are transformed, then the corresponding function is evaluated, and finally the results are transformed back.

Significant features: Positive: (a) Small storage capacity is required for program and data in the execution space (e.g. a simple machine with a few registers into which the function code and arguments are loaded for evaluation). Negative: (b) Because transformation and execution are interleaved in small slices, they both can be analyzed to only a very small extent. (c) For the same reason as (b), the transformation delays the execution (e.g. a high-level language interpreter). (d) Too much transformation is performed on programs in which many of the instructions (operators) and data items occur repeatedly during execution (e.g. a high-level language program performing a large number of iterative computing steps, which is interpreted by a language interpreter).

There are also intermediate or mixed methods of performing transformations which try to exploit some advantages and avoid some drawbacks of the two extreme methods given above.

Global program/local data transformation: the whole program is transformed before and after the whole execution, while data items are transformed only when required for the current execution, and then transformed back again (e.g. an executable program processing direct access disc files).

Combined global/local transformation according to the assumed number of occurrences of instructions (operators) and data items during the execution: the most frequent are transformed globally into the execution space before and after the whole execution, while the remaining are transformed locally only when required for the current execution (e.g. parts of an operating system or a hierarchy of subprogram libraries in general purpose applications). A modification of this method is the dynamic global/ local transformation: instructions (operators) or data making up the globally transformed items in the execution space will not stay there for the whole execution but may be exchanged for locally transformed items, e.g. if they have not been involved in execution for a long time (e.g. usage of general purpose registers of a processor by executable programs which load them with some data more frequently required for execution and replace them later by others; another example is the cache memory, or the throw-away compiling, cf. Brown [27]).

Blockwise transformation: Program and data are partitioned into blocks; any of these blocks is transformed into the execution space whenever an item it contains is required for execution, and it is transformed back when an item outside of this block is required (e.g. segmented loading of programs, paging machines).

These intermediate methods depend heavily on the specification of the portions of programs or data which are to be transformed locally and globally, respectively. If badly specified, they can result in much worse overall performance than in the first two methods (e.g. columnwise processing of a large matrix on a rowwise paging machine).

5. Common Analysis of the Transformation and Execution

The global program/global data transformation is an extreme case which allows very fast execution but requires very much storage in the execution space. If the storage capacity does not suffice, one has to use some of the intermediate methods. In these, however, the interleaved transformation delays the execution, as is seen most clearly in the opposite extreme case of the local program/local data transformation. If the machine were able to transform locally the items required for the next execution during the current execution, and simultaneously to transform back the items which have been involved in the foregoing execution, it could achieve an execution speed comparable to execution after global transformation. This would mean analyzing the transformation and the execution together, as described in section 3. However, for the transformation of the items which will be involved in the next execution the machine must first decide which these will be. It is not always possible to give a precise answer. The following method offers a rather ad hoc but easy implemented solution:

Neighbourhood: One expects that the items in some neighbourhood of those involved in the current execution will be required for the next execution and transforms them (cf. blockwise transformation, section 4) into the execution space, simultaneously with the current execution (e.g. hierarchical storage reorganisation, Madnick [8]). Thus very fast execution may result, but in the least favourable case the execution speed can be worse than in the local transformation method, while much more storage is required in the execution space.

However, there is a lot of information in the user program about the possible next items. Let X be the set of user's variables, L set of user's labels, and assume that

$$
\begin{array}{l}
\vdots \\
l;\ f:\vec{x}\to(\vec{y};l_1,\ldots,l_p) \\
l_1;f':\vec{x}'\to(\vec{y}';\ldots\ ) \\
\vdots \\
l_p;f'':\vec{x}''\to(\vec{y}'';\ldots\ ) \\
\vdots
\end{array}
$$

is a part of his program, with labels $l, l_1, \ldots, l_p \in L$, where $\vec{x},\vec{y}$ are sequences $(x_1,\ldots,x_m)$ and $(y_1,\ldots,y_n)$ of variables of X. The user's reference manual interpretation, which we denote by I, then assigns to f some (partial) function If: $A_{t1} \times \ldots \times A_{tm} \to (A_{u1} \times \ldots \times A_{un}) \times p$ as its meaning, where p denotes the set $\{0,1,2,\ldots,p-1\}$. The letters $t_1,\ldots,t_m,u_1,\ldots,u_n$ denote some elements of the set T of the allowed data types and $A_{t_1}, \ldots, A_{u_n}$ the underlying sets, i.e. the sets of possible values on which the function If operates. Thus If assigns to arguments $a_1,\ldots,a_m$ of the required types in the domain of If some $b_1,\ldots,b_n$ of specified types as its results, and a truth value j as one of its possible p outcomes. In accordance with the reference manual interpretation I, the instruction labelled with l would be decoded as follows: If $a_1,\ldots,a_m$ are the respective values of $x_1,\ldots,x_m$ and if the value of If at $(a_1,\ldots,a_m)$ is $(b_1,\ldots,b_n;j)$, then assign to the variables $y_1,\ldots,y_n$ the values $b_1,\ldots,b_n$, resp., and for the next action refer to the label l_j.

Thus each program statement specifies all its possible direct successors. If the transformation preserves this partial ordering, the possible direct successors of the current executed program statement of the transformed program can also be specified.

Without going into further details, we call

the transformation order preserving, if it consists of a map F of programs and a map φ of data with the following properties: F maps the data types, $T \to T' : t \mapsto t'$, into the data types of the transformed programs and φ maps bijectively the data items, $A_t \to A'_{t'}$, for each $t \in T$, into the transformed data items; F sends each function name (instruction, operator) of a type $(t_1 \ldots t_m, u_1 \ldots u_n, p)$ into a function description (program) f' of the type $(t'_1 \ldots t'_m, u'_1 \ldots u'_n, p)$ which is to be interpreted by the interpretation I' over the sets $A'_{t'}$, $(t' \in T')$ of the transformed programs so that always $(\varphi^n \times 1_p) \circ If = I'f' \circ \varphi^m$; finally we require that F maps for each user program P injectively the user variables, $X \to X' : x \mapsto x'$, and labels, $L \to L' : l \mapsto l'$, into the variables and labels of the transformed program P', respectively, and sends each statement $l;f:\vec{x} \to (\vec{y};l_1,\ldots,l_p)$ of P into $l';f':\vec{x}' \to (\vec{y}';l'_1,\ldots,l'_p)$ in P', with $\vec{x}'=(x'_1,\ldots,x'_m)$ if $\vec{x}=(x_1,\ldots,x_m)$. These conditions on (F,φ) ensure that each user's program statement and data item can be transformed independently of other items and that the transformed program will process under the interpretation I' the transformed data as the user expects, considering his source program, data, and interpretation I only. In order preserving transformations one can apply lookahead methods for the transformation and execution.

Partial lookahead: Some of the possible direct successor user program statements are chosen and the corresponding items are transformed during the current execution. If the current execution has another outcome than expected, execution is delayed and the actually required items have to be transformed first (e.g. lookahead processors). The next method seems to be the most promising.

Total lookahead: For alle possible direct successor user program lines the corresponding items are transformed simultaneously with the current execution (e.g. a user sitting at a demand terminal and waiting a long time for the outcome of the currently executed job control command, who already pretypes onto the screen the job control line for each of the possible outcomes). Of course, if the transformation is so complicated that it lasts longer than the execution, then more advanced possible successors must also be taken into account.

From the above conditions on the transformation it follows that the transformation preserves the "shape" of the program and data (cf. Goguen [28]), as in the case of their transport (transfer), or "refines" the program instructions and operators as subprograms, and data items as data constructs in a lower-level language, what we call a successive interpretation. If we do not insist that the transformation transforms instructions and operators f of the source program "pointwise", but allow f to be a source program subroutine, than the transformation also involves, for example, the analysis of a complex program into its "unfolded" version, as described in section 3 (cf. e.g. Ramamoorthy and Gonzalez [29] for a survey of some techniques).

6. Conclusion

We have explored the common structure of the transformation and execution of programs. The notion of transformation is general enough to include not only transport but also high-level language translation, subroutine calls, emulation, hardware implementation of functions, etc. Most of machine data processing consists of a hierarchy of successive transformations where lookahead methods can be applied, if these transformations preserve the ordering of the instructions and operators in a source program. Our aim was to attract more attention to the potential inherent in the successive interpretation of (very) high-level languages and the possibility of the exploitation of lookahead methods for the compiled interpretation, in accordance with the recent developments of hardware prices.

Acknowledgement. I am obliged to Prof. Dr. W. Händler and Mr. R. K. Bell, MA, for much valuable advice and useful comments on this paper.

References

[1] Flynn, M.J., "Very high-speed computing systems", Proc. of the IEEE 54 (1966). 19o1-19o9.

[2] Enslow, P.H.,"Multiprocessors and other parallel systems - an introduction and overview", in: Händler, W.(ed.), Computer Architecture, Workshop of the GI, Erlangen, May 1975, Springer, Berlin 1976, 133-198.

[3] Händler, W., "On classification schemes for computers in the post-von-Neumann-era", in: Siefkes, D. (ed.), GI-4.Jahrestagung, Berlin, Okt. 1974, Lect. Notes in Comp. Sience 24, Springer, Berlin 1975, 439-452.

[4] Händler, W., "Zur Genealogie, Struktur und Klassifizierung von Rechnern", in: Parallelismus in der Informatik, Erlangen, Juni 1976, Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, 9 (1976)/8, 1-30.

[5] Händler, W., "Impact of classification schemes on computer architecture", this conference.

[6] Feng, T., "Some characteristics of associative/parallel processing", Proceedings of the 1972 Sagamore Comp. Conf., Syracuse University, 1972, 5-16.

[7] Dennis, J.B., "Packet communication architecture", Proc. of the 1975 Sagamore Comp. Conf. on Parallel Processing, IEEE, New York 1975, 224-229.

[8] Madnick, S.E., "INFOFLEX-hierarchical decomposition of a large information management system using a microprocessor complex", AFIPS Conf. Proc. 44 (1975), 581-586.

[9] Miller, R.E. and Cocke, J., "Configurable computers: A new class of general purpose machines" in Ershov, A.P. and Nepomniaschy, V.A. (eds.), Internat. Symposium on Theoretical Programming, (1972), Lecture Notes in Comp. Sci. 5, Springer, Berlin 1974.

[10] Birkhoff, G., Lattice Theory, 3rd Ed., American Mathematical Society, Providence 1967.

[11] Tesler, L.G. and Enea, H.J., "A language design for concurrent processes", AFIPS Proc. of the Spring Joint Comp. Conf. 1968, 32, 4o3-4o8.

[12] Dennis, J.B., "Programming generality, parallelism and computer architecture", Information Processing 68, North-Holland, Amsterdam 1969, 484-492.

[13] Dennis, J.B., and Misunas, D.P., "A pipelining architecture for basic data-flow processor", Proc. of the Second Annual Symposium on Comp. Architecture, (1974), IEEE, New York 1975, 126-132.

[14] Rumbaugh, J., "A data flow multiprocessor", Proc. of the 1975 Sagamore Comp. Conf. on Parallel Proc., IEEE, New York 1975, 22o-223.

[15] Rumbaugh, J., "A data flow multiprocessor", IEEE Trans. on Computers C-26 (1977)/2, 138-146.

[16] Plas, A., et al., "LAU system architecture: a parallel data-driven processor based on single assignment", Proc. of the 1976 Int. Conf. on Parallel Processing, IEEE, New York 1976, 293-3o2.

[17] Händler, W., "The concept of macro-pipelining with high availability", Elektronische Rechenanlagen, 15 (1973)/6, 269-274.

[18] Grätzer, G., Universal Algebra, D. van Nostrand, Princeton 1968.

[19] MacLane, S., Categories for the Working Mathematican, Springer, New York 1971.

[20] Arbib, M.A. and Give'on, Y., "Algebra automata I: Parallel Programming as a prolegomena to the categorical approach", Information and Control 12 (1968), 331-345.

[21] Elgot, C.C., and Robinson, A., "Random-access stored-program machines, an approach to programming languages", J. Assoc. Comp. Mach. 11 (1964), 365-399.

[22] Elgot, C.C., "The external behavior of machines", Proc. 3rd Hawaii Int. Conf. on System Sciences 1970, 447-45o.

[23] Give'on, Y., and Arbib, M.A., "Algebra automata II: The categorical framework for dynamic analysis", Information and Control 12 (1968), 346-37o.

[24] Hennie, F.C., Finite-State Models for Logical Machines, Wiley, New York 1968.

[25] Elgot, C.C., "Remarks on one-argument program schemes", in: Rustin R. (ed.), Formal Semantics of Programming Languages, Courant Computer Science Symp. 2, 197o, Prentice-Hall, 1972.

[26] Anderson, G.A. and Jensen, E.D., "Computer interconnection structures: Taxonomy, characteristics, and examples", Computing Surveys 7 (1975)/4, 197-213.

[27] Brown, P.J., "Throw away compiling", Software Practice & Experience 6 (1976)/3, 423-434.

[28] Goguen, J.A. Jr., "On homomorphisms, correctness, termination, unfoldments, and equivalence of flow diagram programs", J. Comp. Syst. Sciences 8 (1974)/3, 333-365.

[29] Ramamoorthy, C.V. and Gonzales, M.J., "A survey of techniques for recognizing parallel processable streams in computer programs", AFIPS Proc. of the Fall Joint Computer Conference, 1969, 1-15.

SCHEDULES FOR GENERAL MONITOR SYSTEMS WITH A MINIMAL NUMBER OF PROCESSORS

Dirk Hennings, Sigram Schindler, Michael Steinacker
Fachbereich 20 (Informatik)
Technische Universität Berlin
Berlin, Germany

Abstract -- The paper summarizes previous and contains new solutions for the problem to construct schedules for a set of independent tasks to be executed on several processors. For each task requestline and deadline for execution and the computation time required on any processor are known in advance.

1. Introduction

A general monitor system, GMS, consists of a finite set of independent tasks, $\underline{T}$, each task having an individual requestline, RL(T), an individual "hard" deadline, DL(T), and an individual computation time, CT(T), where $0 < CT(T) \leq DL(T) - RL(T)$ is assumed.

The problem is to construct schedules for a GMS and a minimal number of identical, independent processors of finite speed. If only one processor is available the general solution for a GMS is derived in [S1]. Restricted monitor systems were investigated previously by Liu/Layland [LL] and by one of the authors [S2], the latter considering the case of several processors. Another special case was studied by Labetoulle [La], assuming a single processor system.

The first result of this paper is
- a method for calculating the minimal number of processors required for executing a given GMS without violating constraints;
- a scheduling scheme which describes the class of all preemptive schedules for a given GMS and a given number of processors.

We are interested in "classes of schedules" in order to be able to care for additional constraints imposed on solutions by reality (see [SL] for further explanations).

Basically a scheduling scheme consists of two algorithms (see figure 2):
- the first algorithm computes the set of all "admissible" assignments of the GMS;
- the second algorithm computes the maximal running time for the admissible assignment from this set selected for execution.

The second result of this paper is a scheduling algorithm for a GMS having some relatively weak additional property. [H] is the full version of this paper including all proofs of correctness of the algorithms derived.

2. Notions and definitions

A general monitor system, GMS:= $(\underline{T},RL,DL,CT)$, is defined to be a finite set $\underline{T}$ of tasks and three mappings for the requestlines, deadlines and computation times

$$RL : \underline{T} \to R$$
$$DL : \underline{T} \to R$$
$$CT : \underline{T} \to R$$

The computation time, CT(T), of a task T gives the time required to execute T completely on any of the available processors. The processing of a task T cannot begin before its requestline, RL(T), and must be completed before its deadline, DL(T). Because of the graphical representation of the GMS chosen in this paper it is sometimes reasonable to speak about lengths of tasks instead of computation times of these tasks (see figure 1).

A processor system consists of a finite set of independent and identical processors which are able to process the tasks with a constant, positive, and finite speed. The units of length and time are determined such that one processor reduces the length of a task by one in one time unit. We exclude that several processors simultaneously execute one task or that one processor execute several tasks simultaneously.
$\underline{X} \subset \underline{T}$ is called an assignment if all tasks of $\underline{X}$ are processed simultaneously for some time, the running time t^X of the assignment $\underline{X}$. We say a preemption occurs if a processor executing a task is interrupted before the length of the task has been reduced to zero.

$GMS_t := (\underline{T}_t, RL_t, DL_t, CT_t)$ denotes the remaining GMS after a finite sequence of assignments with total running time t has been executed.

A schedule S for a given GMS and for a given number of processors is determined by a finite sequence of assignments $\underline{X}$ and the respective running times t^X, compatible with the requirements of the GMS.

A schedule for a GMS and M processors is called

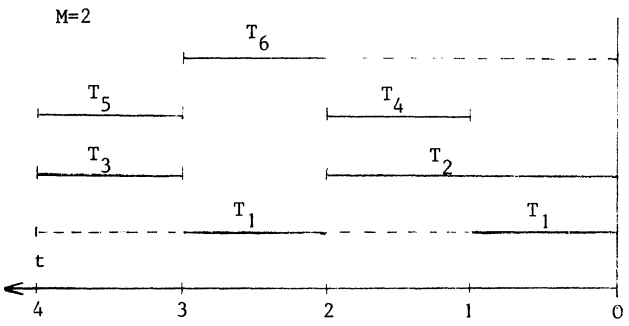


Figure 1: Representation of a GMS, processing takes place from right to left. The requestlines, deadlines and computation times are the following:

$RL(T_1) = 0,\ DL(T_1) = 4,\ CT(T_1) = 2$
$RL(T_2) = 0,\ DL(T_2) = 2,\ CT(T_2) = 2$
$RL(T_3) = 3,\ DL(T_3) = 4,\ CT(T_3) = 1$
$RL(T_4) = 1,\ DL(T_4) = 2,\ CT(T_4) = 1$
$RL(T_5) = 3,\ DL(T_5) = 4,\ CT(T_5) = 1$
$RL(T_6) = 0,\ DL(T_6) = 3,\ CT(T_6) = 1$

Note that $Y = (T_2, T_6)$ is not contained in the set of admissible assignments at time t=0, because if executed for an arbitrarily small time, $t=\varepsilon$, the GMS_ε is over-critical in the k-interval $[1,2] \cup [3,4]$. Thus T_1 and T_2 must be assigned at first.

optimal iff there is no other schedule for this GMS and less than M processors.

The following definitions are required for the description of the algorithms.

A single interval begins at some requestline and terminates at some deadline of some task(s). A multiple interval (or k-interval, $k \in N$) is the union of a finite number of single intervals. The minimal load, MINLOAD (GMS,A), of a k-interval A of a GMS is given by the sum over those parts of tasks of the GMS which at least must be processed inside of A, because they cannot be executed outside of A without violating constraints defined by the GMS (see figure 1).

For a given number of processors, M, a k-interval A of a GMS is called critical iff the condition

$$\text{MINLOAD}(GMS,A) = M \cdot \text{length}(A)$$

holds. It is called over-critical iff the condition

$$\text{MINLOAD}(GMS,A) > M \cdot \text{lenght}(A)$$

holds.

The length of a GMS, L(GMS), is defined to be the length of the interval between its first requestline and its last deadline. Then a GMS is called adjusted (with respect to M processors) iff the condition

$$\sum_{T \in \underline{T}} CT(T) = M \cdot L(GMS)$$

is fulfilled and none of its k-intervals is over-critical. Obviously the first property can be obtained in a trivial way. For an adjusted GMS an assignment is called admissible iff executing it for an arbitrarily small time t, $t > 0$, implies GMS_t is adjusted. The longest running time of an assignment, $\underline{X}$, with this last property is called maximal running time t^X_{max}.
Note: Obviously an adjusted GMS contains at least M requested tasks.

The basic idea of this paper is the following:
- In order to determine M, the minimal number of processors for executing a given GMS completely, consider the minimal load density, defined by MINLOAD(GMS,A)/length(A) for all k-intervals, A, and calculate the maximum. The next higher integer is M.
- In order to construct an optimal schedule for the GMS and M processors control the minimal load density of all k-intervals of the remaining GMS_t such that none of them exceeds this bound M, by choosing appropriate assignments and running times.

Any scheduling algorithm obeying this principle generates optimal schedules. Moreover, all optimal schedules can be described in this way.

3. Results for the general case

Theorem 1:

Let an adjusted general monitor system, GMS_t, at time t and an M-processor system be given. Then the subsequent algorithm A1 computes the non-empty set of all admissible assignments.

A1:

Input : GMS_t,M

Step 1: Compute the set, $\underline{T}^1_t$, of all requested tasks, T_t, fulfilling the condition $DL(T_t) - CT(T_t) = t$.

Step 2: For each critical k-interval, not beginning at t, compute the set of all requested tasks making it over-critical, unless assigned immediately. The union of all these sets is called $\underline{T}^2_t$.

Step 3: For each critical k-interval, beginning at t, compute the set of all requested tasks contributing to its MINLOAD. The intersection of all these sets is called $\underline{T}^3_t$.

Output: Set of admissible assignments, $\underline{AA}_t$, defined by

$$\underline{AA}_t := \{(\underline{T}^1_t \cup \underline{T}^2_t \cup \underline{Y}) \mid \underline{Y} \subset T^3_t \wedge |Y| = M - |\underline{T}^1_t \cup \underline{T}^2_t|\} \quad \text{i.e.}$$

$$\underline{AA}_t := \{\underline{X} \mid |\underline{X}| = M \wedge \underline{T}^1_t \cup \underline{T}^2_t \subset \underline{X} \subset \underline{T}^3_t\}$$

end Theorem 1

Remarks

For an adjusted GMS the interval $[t,L(GMS_t)]$ is always critical at time t. Thus all requested tasks may belong to admissible assignments. Moreover, both definitions of AA coincide because $\underline{T}_t^1 \cup \underline{T}_t^2 \subset \underline{T}_t^3$.

Theorem 2:

Let an adjusted general monitor system, GMS_t at time t and an M-processor system be given and let X be arbitrarily chosen from the set of admissible assignments, $\underline{AA}_t$, calculated by algorithm A1. Then the respective maximal running time, t^X_{max}, can be computed by the subsequent algorithm A2.

A2:

Input : GMS_t, X, M

Step 1 : For each task, T_t, not assigned by X, compute the time t', $t' = DL(T_t) - CT(T_t)$.

Step 2 : For each k-interval not beginning at t compute the next point in time, $t' > t$, at which the tasks not assigned by X would make it over-critical.

Step 3 : For each critical k-interval beginning at t compute the next point in time $t' > t$ at which a task assigned by X would no longer contribute to its MINLOAD.

Step 4 : Compute the minimum, t^X_{max}, of all the above t' and of the computation times of the tasks assigned in X.

Output : Maximal running time t^X_{max}.

Remarks

end Theorem 2

For each k-interval the t' from steps 2 and 3 can be computed by an easy and computational efficient algorithm, omitted here because of its notational complexity. But note that the number of k-intervals may be exponential in the number of tasks.

Theorem 3:

Let a general monitor system, GMS, and the number of processors, M, be given such that

$$M \geq \lceil \max \{ \text{MINLOAD}(GMS,A) / \text{length}(A) \text{ such that } A \text{ is k-interval of the GMS} \} \rceil .$$

Let SA(GMS,M) denote the set of scheduling algorithms for the GMS and M processors obtained from the scheduling algorithm scheme in figure 2 by all deterministic interpretations of the starred lines, i.e. by all choice algorithms replacing these two lines.

Let S(GMS,M) denote the set of all schedules for the GMS on M processors obtained by applying all $SA \in \underline{SA}(GMS,M)$ to the GMS.

Then S(GMS,M) is the set of all optimal schedules for the GMS on M processors.

Let $S \in \underline{S}(GMS,M)$ be obtained by a scheduling algorithm from SA(GMS,M) always choosing $t^X = t^X_{max}$ for all assignments. Then S consists of at most $2 \cdot N^2$ assignments.

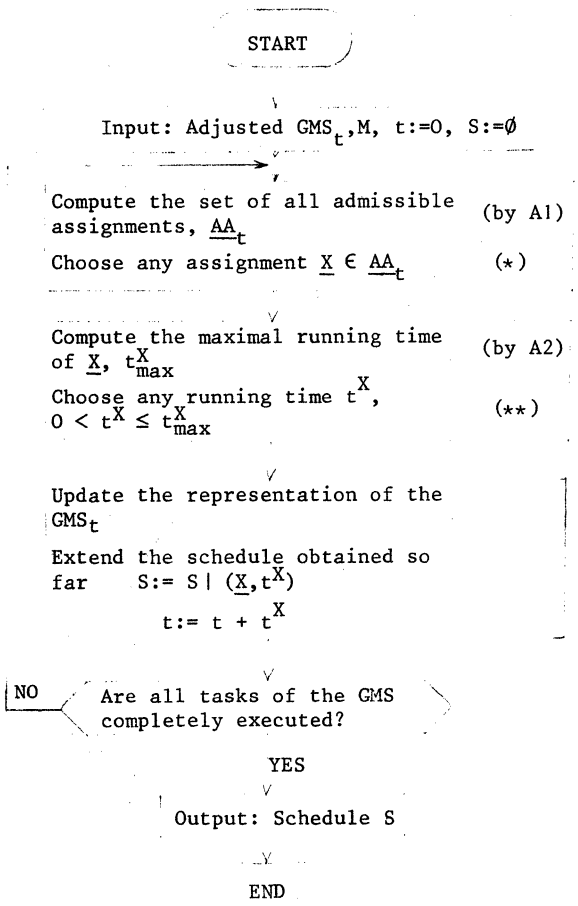


Figure 2: scheduling algorithm scheme

end Theorem 3

Remarks

The optimal schedules of an adjusted GMS are characterized by the adjustment of the remaining GMS_t at any point of time t. This also implies that the admissible assignments are those assignments consisting of requested tasks which can be executed for an arbitrarily small time without increasing M in order to be able to execute the remaining GMS completely.

4. Scheduling algorithms for the case MINLOAD ≥ 1

[H] contains various special cases, characterized by additional assumptions about the given GMS, allowing us to find an efficient scheduling algorithm. One of them is briefly discussed in this paper subsequently.

For a given GMS let $\{I_i, i=1,\ldots,i0\}$ denote the set of disjoint intervals in [0,L(GMS)] defined by all requestlines and deadlines as boundaries of the I_i; let I_i be located lower than I_j if $i < j$. The GMS is called q-simple iff $MINLOAD(GMS,I_i) \geq q \cdot length(I_i)$ for all $i=1,\ldots,i0$. (For the rest of the section we additionally assume that q is the largest such number. This additional assumption is done for simplicity of presentation but without any deeper relevance and can be omitted easily.)

For increasing q the property of a GMS to be q-simple obviously becomes more and more restrictive. For q=1 figure 3 shows that this assumption does not exclude many technically interesting problems; especially for M=2 it is relatively weak. In the sequal we describe an efficient scheduling algorithm based on this assumption allowing to reduce the M-processor problem to a single processor problem (it is not difficult to see that this assumption can be further weakened without loosing this reducibility).

In order to explain this reduction process we start with an M-processor system, a GMS being adjusted and nowhere over-critical (with respect to M) and being (M-1)-simple. From the (M-1)-simplicity we see that uniquely defined parts of tasks of the GMS mut be processed in uniquely defined intervals I_i (defining a set POT_i) and that the total length of these pieces of tasks in I_i (i.e. of the pieces in POT_i) is equal to $(M-1)\cdot length(I_i)$, $i=1,\ldots,i0$.

A short moment's reflection shows how to derive a GMS^{M-1} and a GMS^1 from the given GMS:

- The GMS^{M-1} consists of the POT_i to be processed in I_i, $i=1,\ldots,i0$,
- GMS^1 consists of those pieces of tasks in GMS not contained in a POT_i restricted by the original requestlines and deadlines; if a piece of a task is in a POT_i then for the remainder of this task in the GMS^1 there is an exclusion interval, $EI_i \subset I_i$, where this remainder may not be processed.

The GMS^{M-1} may be scheduled for M-1 processors by a trivial scheduling algorithm [C, page 76]. The following observation is now important: We can always determine a particular schedule for the GMS^{M-1} on M-1 processors such that its E_i's do not exclude the schedule for the GMS^1 on the remaining processor derivable by the DDEI-scheduling algorithm (defined below). This follows from the open I_i's property that they do not contain any requestline or deadline.

In order to schedule the GMS^1 for the remaining processor we use a modification of the deadline driven scheduling algorithm cited in [S2], such that the exclusion intervals of all tasks are taken into account. We call this modification DDEI-scheduling algorithm (for deadline driven with exclusion intervals) and define it by means of the tasks' modified deadlines. Given exclusion intervals for a GMS^1 at any instant, t, the modified deadline of T from GMS^1, MDL(T), is defined to be DL(T) - SLEI(T) - t, where SLEI(T) denotes the sum of the lengths of all exclusion intervals for $\overline{T}$ in $[t,DL(\overline{T})]$. The DDEI-scheduling algorithm now simply prescribes to schedule at any time a task (from the set of all tasks being requested and not yet completely scheduled and not entering into an exclusion interval) with smallest modified deadline (for all tasks from this set).

5. Conclusion

Obviously the GMS scheduling problem can be considered as a special graph scheduling problem; thus the former problem is simpler than the latter one.

For the preemptive case and the graph scheduling problem presently finite scheduling algorithms are not known for M > 2 (the proof of polynomial completeness by Ullman refers to a somewhat different problem, [4]).

For the preemptive case and the GMS scheduling problem we presented finite scheduling algorithms of complexity $O(2^N)$, producing the class of all schedules for arbitrary M (choosing maximal running times for all assignments their number is bounded by $O(N^2)$ in each schedule).

For arbitrary M we were not able to derive polynomial bounded algorithms nor were we able to show the polynomial completeness of the problem. In order to extend the knowledge about this problem it thus seems reasonable to look for "suboptimal" heuristic scheduling algorithms for a GMS or additional assumptions about GMS's, reducing the complexity of the problem. An example of how such additional assumptions may look like and how weak they may be is discussed in section 4. Both approaches surely will be successful if it is possible to reduce the number of k-intervals to be considered in A1 and A2 to be bounded by a polynomial in N. Such results obtained by additional assumptions can be found in [H]. Presently we investigate scheduling algorithms obtained by choosing feasible subsets (of polynomially bounded cardinality) of the set of all k-intervals.

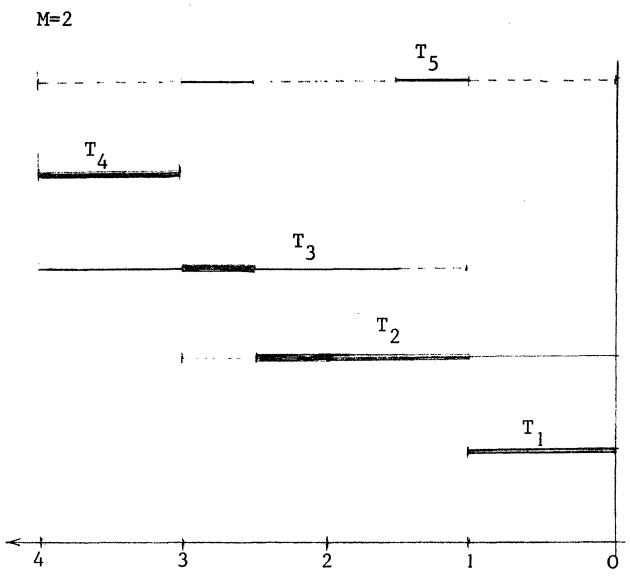


Figure 3a: Representation of a GMS given by the requestlines, deadlines and computation times, respectively, of the tasks:
$T_1 \simeq (0,1,1,)$; $T_2 \simeq (0,3,2.5)$;
$T_3 \simeq (1,4,2.5)$; $T_4 \simeq (3,4,1)$;
$T_5 \simeq (0,4,1)$.

The intervals to be checked are
$I_1 = [0,1]$; $I_2 = [1,3]$; $I_3 = [3,4]$.

The parts of the tasks to be executed in such an I_i on the first processor are shown as bold lines. The thin lines show the remainder for the GMS^1 to be executed on the second processor.

The GMS is 1-simple.

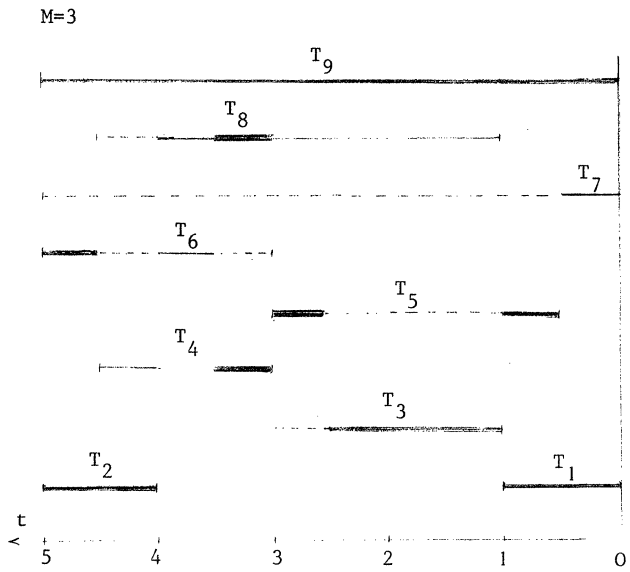


Figure 3b: Representation of a GMS as in Figure 3a.
$T_1 \simeq (0,1,1)$; $T_2 \simeq (4,5,1)$;
$T_3 \simeq (1,3,1.5)$; $T_4 \simeq (3,4.5,1)$;
$T_5 \simeq (0.5,3,1)$; $T_6 \simeq (3,5,1)$;
$T_7 \simeq (0,5,0.5)$; $T_8 \simeq (1,4.5,3)$;
$T_9 \simeq (0,5,5)$.

$I_1 = [0,0.5]$ $I_2 = [0.5,1]$
$I_3 = [1,3]$ $I_4 = [3,4]$
$I_5 = [4,4.5]$ $I_6 = [4.5,5]$

The GMS is 2-simple.

References

[C] E.G. Coffman, Jr.: Computer and job/shop scheduling theory, Wiley 1976

[H] D. Hennings: Doctoral Thesis, Fachbereich 20, Technische Universität Berlin, under prep.

[La] J. Labetoulle: Some theorems on real-time scheduling, Proceedings of the International Workshop, August 12-14, 1974, Rocquencourt, North-Holland Publ. 1974

[LL] C.L. Liu and J.W. Layland: Scheduling algorithms for multiprogramming in a hard-real-time environment, JACM, Vol. 20, No. 1, January 1973, pp 46-61

[S1] S. Schindler: Scheduling general monitor systems, Proceedings of the Ninth Hawaii International Conference on System Sciences, Honolulu, January 1976

[S2] S. Schindler: Scheduling algorithms for monitor systems on M-processor systems, $M \geq 1$, Proceedings of the Eighth Hawaii International Conference on System Sciences, Honolulu, January 1975

[SL] S. Schindler and H. Lüdtke: Eine Diskussion verschiedener Zugänge zum deterministischen Scheduling Problem, 6. Jahrestagung der Gesellschaft für Informatik 1976, Informatik-Fachberichte Nr. 5, Springer-Verlag 1976

[U] J.D. Ullman: Polynomial complete scheduling problems, Operating Systems Review, 7.4, 1973, pp 96-101

SCHEDULING TWO-PROCESSOR SYSTEMS

M. Steinacker, D. Hennings, S. Schindler
Fachbereich 20 (Informatik)
Technische Universität Berlin
Berlin, Germany

Abstract -- The paper summarizes previous and contains new solutions for the problem to construct time-optimal schedules for a set of tasks to be executed on a two-processor system. We assume that arbitrary precedence rules for the execution of the tasks and the tasks' computation times on any of the two processors are known in advance.

1. Introduction

Consider a finite, acyclic, weighted, directed graph, G (fawd-graph), with N nodes and E edges. G represents a task system; a node is a task, the weight of a node is the processing time of the task, and an edge (v,w) means that task v must be finished before task w can be started.

The problem is to construct time-optimal (minimal-length) schedules for a fawd-graph and for a system of two identical processors (see [C], page 84).

A well known special solution is obtained by the algorithm of Muntz/Coffman [MC] which computes a preemptive schedule in $O(N^2)$ steps. If we restrict attention to a system in which all tasks require the same processing time the algorithm of Coffman/Graham [CG] computes a nonpreemptive schedule in $O(Na(N) + E)$ steps where a(N) is an almost constant function of N [Se].

It is obvious that due to the various constraints imposed on real task systems (being not considered in this paper) a set of schedules is much more desirable in general than a single schedule because in many cases at least one of the schedules from that set will be compatible with such constraints. That means: as soon as different and/or varying costfunctions are to be considered (i.e. reality shall be approximated) the approach to the problem via classes of time-optimal schedules [SchL] seems to be adequate.

Predicates provide the best description for the class of all time-optimal schedules. Being true during processing of a fawd-graph on a multiprocessor system these predicates guarantee the time optimality of the respective schedule. By this method for a certain type of task systems (characterized by: arbitrary task lengths, precedence structure being a forest or an antiforest, arbitrary number of processors, preemptions are allowed) one of the authors [Sch1] constructed not only one special solution but the class of all time-optimal schedules, i.e. the general solution of this problem. Due to this general solution the authors [HSS] were able to derive a fast scheduling algorithm (where the number of steps is linear in N).

The result of this paper is

- a scheduling algorithm scheme which describes (by means of efficient algorithms) the set of all time-optimal preemptive schedules for a fawd-graph on a two-processor system (i.e. the general solution of this problem)[a]
- a special time-optimal scheduling algorithm of complexity $O(N^2)$ (i.e. the same complexity as the Muntz/Coffman algorithm) generating schedules with at most 3N preemptions (whereas Muntz/Coffman's schedule may have $O(N^2)$ preemptions).

A short report about preliminary efforts to obtain these results is given in [SchS].

Basically, the scheduling scheme consists of two efficient algorithms to be applied repeatedly to G until it is completely scheduled:

- The first algorithm, A1, computes the set of all admissible first assignments, $\underline{AA}$. That means: A1 computes the set, $\underline{AA}$, of all those assignments the tasks of which can be executed by the two processors for some time t, $t > 0$, without loss of optimality of the whole schedule.
- For an arbitrary admissible first assignment $\underline{X}$ the second algorithm, A2, computes its maximal running time, t^X_{max}. That means: after selecting aribtrarily any assignment X from $\underline{AA}$, A2 computes a time t^X_{max} such that the tasks of $\underline{X}$ can be processed for time t, $0 < t \leq t^X_{max}$, without loss of optimality of the whole schedule and t^X_{max} is the largest such number.

The correctness of the algorithm is proved. A full version of this paper including all proofs will appear [St].

[a] Note that we do not talk about the set of all time-optimal scheduling algorithms but about the set of all time-optimal schedules.

2. The class of all time-optimal schedules

2.1 Notions and definitions

Let G be a finite, acyclic, weighted, directed graph, (fawd-graph G), where weights belong to the nodes. The length of a path of G is defined as the sum of the weights of the nodes on this path. The height H(G) of the graph is defined by the length of a longest path in G. |G| denotes the sum of all weights.

Each node represents a task, T, the weight of a node represents the length $\ell(T)$ of task T. The directed edges between the nodes represent the precedence relations $>$ between the tasks. The set of all tasks is denoted by $\underline{T}$. A fawd graph is called task-graph.

The two processors are able to process tasks with equal, constant, positive, finite speed, i.e. the length of a task being processed is reduced by one unit of length per one unit of time. If a task is reduced to length zero, it is deleted from the graph. A task may be executed if it has no predecessors. A processor executing a task can be interrupted before the length of the task has been reduced to zero. This is called a preemption.

$\underline{X} \subset \underline{T}$ [(a)] is called an assignment if all tasks of $\underline{X}$ are processed simultaneously for some time, t^X, the running time of $\underline{X}$.

A schedule S for G is determined by a finite sequence of assignments $\underline{X}$ and the respective running times.[(b)] This shortest sum of running times is denoted by $t_{opt}(G)$. In an optimal schedule all assignments are called admissible. The longest running time of an admissible assignment, $\underline{X}$, is called maximal running time, t^X_{max}.

In this paper the graph G is drawn in the so-called stripe-representation D, using a cartesian coordinate system. (See figures 1a - 1c.) A task T is represented by a vertical bold line of the length $\ell(T)$ which might be partitioned into several parts connected by descending dashed arcs (see figure 1c). If T precedes T' this is expressed by a non-ascending dashed arc from the bottom of the representation of T to the top of the representation of T'. Arcs may be omitted in cases as (T_1,T_4) in figure 1b. Two simple representations are D^{ℓ} and D^h, where all tasks are positioned as low as possible and as high as possible , respectively, in the interval [0,H(G)]. The horizontal line through the top (bottom) of a task of G in representation D is called start-line (end-line) of this task. Any horizontal line is called height-line.

The load density, ld, of G in D between two height-lines is defined to be the sum of the lengths of parts of tasks between these height-lines divided through the distance between these height-lines. If G has a representation such that its ld ≥ 2 everywhere in [0,H(G)], then G is called adjustable and this representation is called adjusted representation (or a-representation).

Let us denote by L(G,D) the length of the union of all intervals in [0,H(G)] in which ld < 2. Let LMING:= min { L(G,D)| D is representation of G in [0,H(G)]} . Then obviously

$$t_{opt}(G) = \frac{1}{2}(|G| - LMING) + LMING = \frac{1}{2}(|G| + LMING) .$$

Finally we sometimes extend G (without changing notation) by a so-called zero task, which is unrelated to any task in G and which has length LMING; Then $t_{opt}(G)$ is not changed but G is adjustable.

The subsequent solution is derived from the principle - proved in [St] - that a schedule for an adjustable task-graph is optimal iff at any point in time the remaining graph is adjustable.

2.2 Algorithms

The scheduling algorithm scheme we are going to investigate is of the structure represented graphically in figure 2. From this scheme a scheduling algorithm is obtained by assigning an interpretation to the starred lines. The scheme consists mainly of four parts

- checking whether G is adjustable
- determining the set of all admissible first assignments, AA
- determining the chosen assignment's, $\underline{X} \in \underline{AA}$, maximal running time, t^X_{max}
- updating G's representation such that this step of execution is displayed.

The first three parts are based on the subsequent algorithm AO and slight modifications of it. AO starts from G in D^h and a partitioning of D^h into p levels, defined by the end-lines of the tasks of any chosen longest path. We try to adjust G level by level from the bottom (level number = 1) to the top (level number = p). As we have on any level the task of the longest path chosen we only have to check whether we have within the level to be adjusted additional tasks for adjusting this level (we call this level a-level); if there are not enough such tasks, the next higher level(s) is (are) considered (we call this level c-level) for a task to be moved down in order to adjust the a-level. This changes G's representation. Tasks of G in the current representation which may be moved down to the a-level without violating the precedence constraints are called a-candidates. We finally denote by ald the load density of the a-level in the current representation of G.

(a) underscores denote sets

(b) S is called time-optimal or optimal if there is no other schedule with a shorter sum of running times.

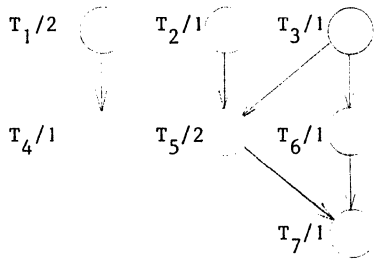


Figure 1a: Traditional representation of a fawd-graph

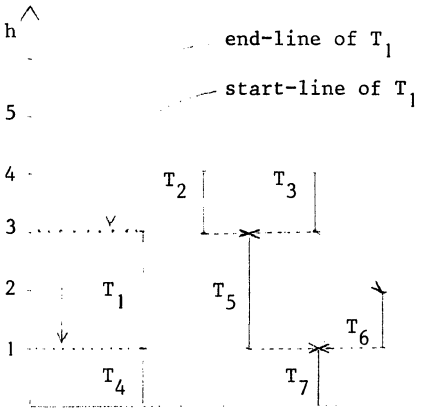


Figure 1b: Stripe-representation (ℓ-representation) where start- and end-lines are only drawn for T_1. The others are omitted in order to simplify the representation.

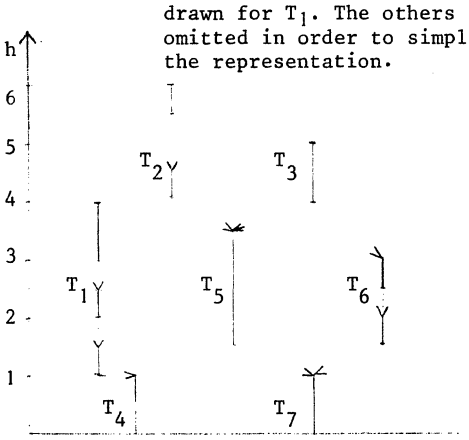


Figure 1c: Stripe-representation
Note: H(G) does not depend on G's representation.

Theorem 1:

Let G be a task graph. Then the subsequent algorithm AO computes LMING

```
AO:
begin
input G in D^h ; LMING:= 0
determine the p-levels of any longest path
for a-level from 1 step 1 until p
do c-level:= a-level + 1
   while ald < 2 ∧ c-level ≤ p
   do while ald < 2 ∧ ∃ a-candidate on c-level
      do move at most ((2 - ald) * length of a-level)
         units of length of any a-candidate from
         the c-level down to the a-level
      od
      c-level:= c-level + 1
   od
   LMING:= LMING + (2 - ald) * length of a-level
od
ouptput LMING
end AO.
```

end Theorem 1

Remember that with LMING we know $t_{opt}(G)$, too.

For the calculation of the set of all admissible first assignments, AA, we use a modification of the algorithm AO denoted by A1.

Theorem 2:

Let G be an adjustable task graph in D^h and let a longest path (defining p levels) be chosen; let T^p denote its highest task. Then the set of all admissible first assignments, AA, is obtained by applying algorithm A1. A1 computes a special a-representation D^{A1} of G. Let p-set denote the set of all tasks without predecessors in level p of G in D^{A1}. Let pld denote the load-density of level p. Then AA is defined as follows

```
AA:= { (T,T') | T = if pld = 2
                    then T^p
                    else arbitrary from p-set,
                T'= arbitrary from p-set}
```

A1 is defined as follows.

```
A1:
begin
input G in D^h
determine the p-levels of any longest path
for a-level from 1 step 1 until p
do while ald < 2
   do
   determine E and ES, E < ES, where E is the
   height of the lowest end-line of a-candidates
   and ES is the next higher such end- or start-
   line, and determine k, the number of a-candida-
   tes with end-line E. Move y units of length of
   each of these k a-candidates down to the a-
   level, where
   y = min {(2 - ald) * length of a-level / k, ES-E}
   od
od
output AA
end A1
```

end Theorem 2

For a task graph G the next theorem gives the maximal running time t^X_{max} of an arbitrarily chosen admissible assignment $\underline{X} \in \underline{AA}$.

Theorem 3:

Let G be an adjustable task graph and let $\underline{X} := (T,T') \in \underline{AA}$ be an arbitrarily chosen admissible first assignment. Let G^X denote the task graph obtained from G by reducing the lengths of T and T' by $\min\{\ell(T),\ell(T')\}$. Let D^X denote the representation D^h for G^X. Let the representation D' for G be defined such that all tasks of G^X are located as in D^X and T and T' are located on top of D^X.

By applying A2 to G in D' we obtain t^X_{max}.
A2 is defined as follows.

A2:
begin
input G in D'
determine the p levels of any longest path in G^X
for a-level from 1 step 1 until p
do
:
: see A1
:
od
t^X_{max}:= minimum of the pieces of T and T' still beyond of $H(G^X)$
end A2

end Theorem 3

The next theorem is the main result of the paper; it makes use of theorems 1-3.

Theorem 4:

Let $\underline{SA}(G)$ denote the set of scheduling algorithms for a task graph G obtained from the scheduling algorithm scheme, shown in figure 2, by all deterministic interpretations of the starred lines. Let $\underline{S}(G)$ denote the set of all schedules for G obtained by applying all $SA \in \underline{SA}(G)$ to G. Then $\underline{S}(G)$ is the set of all optimal schedules for G.

end Theorem 4

Unfortunately the approach taken to derive this general solution is of no help if more than two processors are to be scheduled. But as presently two processor systems are of great technical importance and no general solution for m-processor systems, $m > 2$, can be expected, a separate investigation of this special case is surely justified.

2.3 The computational complexity of the algorithms

Let G have N tasks and E arcs. Then the input procedure has the complexity $O(N+E)$. See figure 3 for an overview.

For computing LMING, the set of all admissible first assignments and the maximal running time for

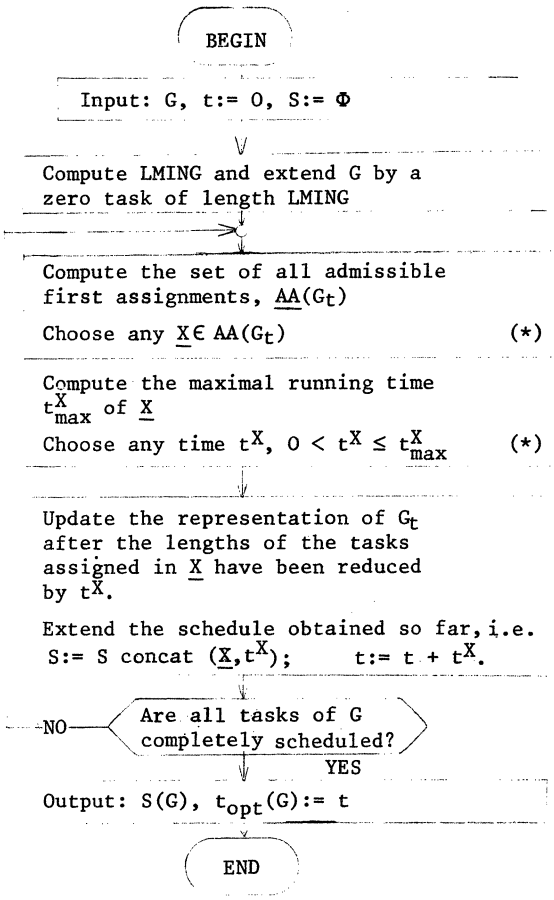


Figure 2: Scheduling algorithm scheme

G_t here denotes the part of G not yet scheduled at time t.

a chosen admissible assignment the algorithm AO and its modifications, respectively, are used. All three algorithms have the complexity $O(N^2)$. Because of their similarity it suffices to consider AO in order to obtain this bound.

Applying AO basically requires two steps:

1. Establishing the appropriate initial representation for execution of AO, i.e. bringing G_t into D^h. This requires $O(N+E)$ steps.
2. Execution of AO, requiring $O(N^2)$ steps. This low bound seems to be achieveable, observing that each of the N tasks can be cut in N parts at most; it actually is achieveable by an appropriate implementation as shown in [St].

The updatings of G_t, $\underline{S}$ and t obviously can be done in $O(N)$ steps.

If always the largest running time, t^X_{max}, is chosen, then the number of assignments is bounded by $O(N^2)$ because no assignment can occur twice. Thus the complexity of the output procedure is $O(N^2)$, too.

If moreover the choice of an $\underline{X} \in \underline{AA}(G_t)$ is always done in at most $O(N^2)$ steps then the complexity of the scheduling algorithm (obtained by this interpretation of the scheduling algorithm scheme) is bounded by $O(N^4)$.

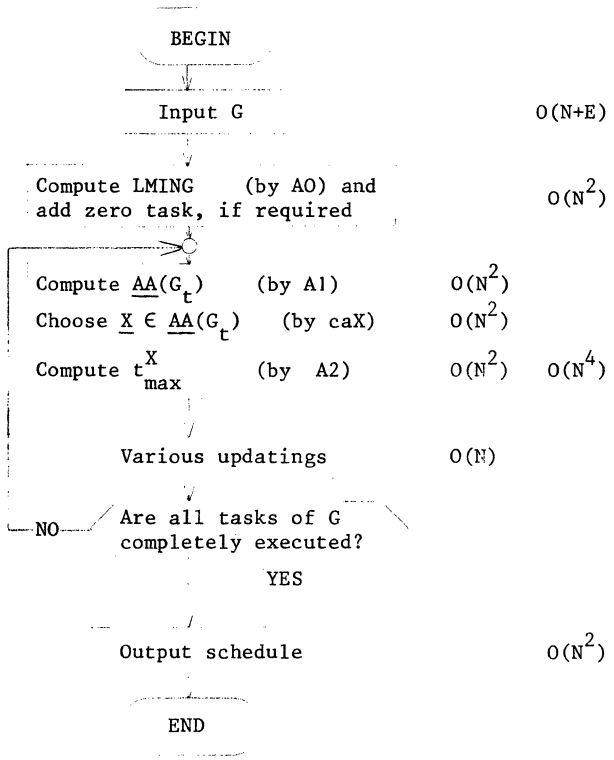


Figure 3 Computational complexity of any scheduling algorithm derived from figure 2 by interpreting the starred lines as follows: $t^X := t^X_{max}$ and for determining an $\underline{X} \in \underline{AA}(G_t)$ we have some choice algorithm, caX, of complexity $O(N^2)$, at most.

3. An efficient scheduling algorithm

In this section we give a scheduling algorithm, LNP, generating schedules with a low number of preemptions. LNP has the computational complexity $O(N^2)$, i.e. in the general case it is not faster than the algorithm with $O(N^2)$ steps given by Muntz/Coffman[(a)]. But the total number of preemptions of schedules generated by LNP is bounded by 3N, whereas the M/C algorithm may generate $O(N^2)$ preemptions, as can easily be seen [Sch2].

[(a)] For the special case $\ell(T) = 1$ for all $T \in \underline{T}$, i.e. all tasks have the same length, LNP is of complexity O(N+E), if we have a computer with multi-level indirect addressing [T, p.84], like e.g. the PDP-10, or an instruction determining the number of leading zeros in a word [H,p.245], or something similar.

It is quite interesting to see that the LNP scheduling algorithm is not derived from the scheduling algorithm scheme; although the schedules generated by the LNP algorithm may be generated by the scheduling algorithm obtained by a suitable interpretation of the scheduling scheme. Nevertheless it probably would not have been possible to construct the LNP algorithm and it is hard to see how to prove its correctness without the analysis required for the scheduling algorithm scheme.

The algorithm LNP starts from an adjustable G in D^ℓ. It begins with the highest tasks of G in D^ℓ and procedes to the lowest tasks. At any time t during scheduling G we denote by G_t the part of G not yet scheduled. The abbreviations introduced subsequently all refer to G_t in D^ℓ, unless stated otherwise. Let HEL denote the highest end-line HEL and let HEL-tasks denote the set of all tasks with start-lines higher than HEL. Let T^ℓ be a task with the lowest end-line of all tasks in HEL-tasks and let T^H be a task with start-line $H(G_t)$. HEL-tasks' is obtained from HEL-tasks by removing T^ℓ and T^H from it. Let $T^{\ell\ell}$ be a task with a lowest end-line of all tasks in HEL-tasks', if it is notempty. HEL-tasks" is obtained from HEL-tasks' by removing $T^{\ell\ell}$ from it.

Let the initial current representation, D^c of G_t, be defined such that
- all tasks not in HEL-tasks' are located as in D^ℓ
- the end-lines of the tasks of HEL-tasks" have the height HEL and
- a piece of $T^{\ell\ell}$ of length z is located beyond HEL, the remaining piece of $T^{\ell\ell}$ being located as in D^ℓ, where

$z := \min \{ \ell(T^{\ell\ell}),$ sum of the lengths of the pieces beyond HEL of all tasks of HEL-tasks $\setminus \{ T^{\ell\ell} \}$ of G_t in $D^c \}$.

Let the current inadjustment of G_t in D^c be defined as

CIA:= $2 * (H(G_t) - HEL)$ - sum of the lengths of the pieces beyond HEL of all tasks of G_t in D^c.

As long as CIA > 0 we must change D^c once more by moving another task (or a piece of it) up into a position beyond HEL. For this purpose we take a task with the highest end-line of all tasks of G_t in D^c starting not above HEL, and which may be moved up beyond HEL without violating the precedence rules in G_t (this may imply moving up a task, which is located beyond HEL, until its start-line becomes $H(G_t)$). Let this latter task (required for reducing the current inadjustment of the part of G_t in D^c beyond HEL) be denoted by T^c.

As soon as CIA becomes zero, the pieces of the tasks beyond HEL may be scheduled by a simple algorithm, e.g. the "packing" algorithm from

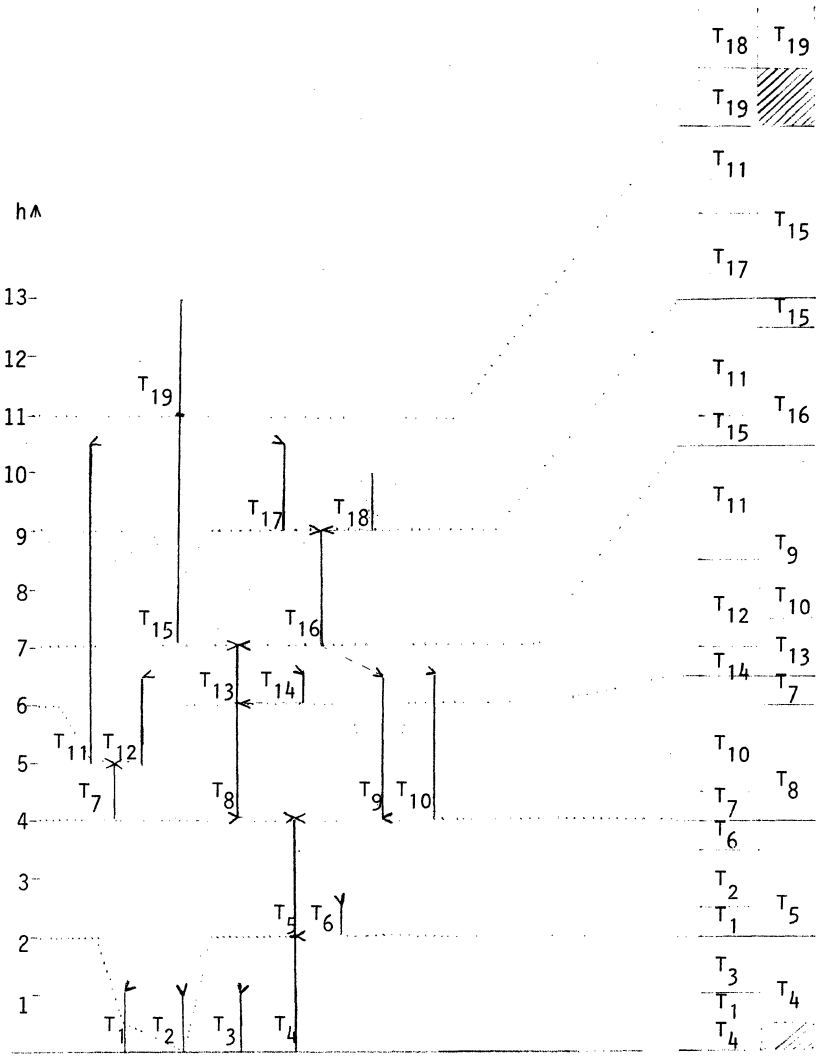


Figure 4: An example of the application of the algorithm LNP. Note the following pecularities of this example

- the highest and the lowest level cannot be adjusted
- the second lowest level has a current inadjustment which can be reduced to zero
- in the second highest level the task with the second lowest end-line ($T_{15} = T^{\ell\ell}$) is taken only partially; moreover, $T^{\ell\ell} = T^{H}$.

[C], page 76.

Note that obviously the computation of HEL and the associated adjusting takes place at most N times and that each time at most 3 preemptions are required.

Theorem 5:

Let G be a task graph in D^{ℓ}. Then the application of the algorithm LNP (defined below) generates an optimal schedule, S, for G in at most $O(N^2)$ steps and S contains at most 3N preemptions.
The algorithm LNP is defined as follows.

```
LNP:
begin: Input G in D^ℓ
while G not completely scheduled
do in G_t determine
   HEL, HEL-tasks", T^H, T^ℓ, T^ℓℓ.
   move the tasks from HEL-tasks" and a piece of
   T^ℓℓ up beyond HEL such that G_t is brought into
   its initial D^c
   while CIA > 0 ∧  ∃ T^c
   do determine T^c
      locate a piece of T^c of length min{ℓ(T^c),CIA}
      beyond HEL
   od
   schedule the pieces of G_t in D^c beyond HEL
   by applying the packing algorithm
od
output S
end LNP
```

end Theorem 5

Because of its similarity to the algorithm AO the algorithm LNP terminates after $O(N^2)$ steps. Figure 4 gives an example of the application of the algorithm LNP to a G such that the various cases to be considered do occur.

References

[C] E.G. Coffman Jr., ed.: Computer and job/shop scheduling theory, Wiley 1976

[CG] E.G. Coffman and R.L. Graham: Optimal scheduling for two-processor systems, Acta Informatica 2, 1972

[H] A.N. Habermann: Introduction to operating system design, Science Research Associates, Inc., 1976

[HSS] R.D. Hennings, S. Schindler, M. Steinacker: The complexity of preemptive scheduling algorithms for multiprocessor systems, TR 74-20, December 1974, Technische Universität Berlin, Fachbereich 20

[MC] R.R. Muntz and E.G. Coffman: Optimal preemptive scheduling on two-processor systems, IEEE Transactions on Computers, C18, No.11, 1969

[Sch1] S. Schindler: Classes of optimal schedules for multiprocessor systems, 2. Jahrestagung der Gesellschaft für Informatik, Karlsruhe 1972, in: Lecture Notes in Economics and Mathematical Systems, Vol.78, Springer-Verlag 1973

[Sch2] S. Schindler: Quantitative aspects of optimal schedules for multiprocessor systems, TR 73-10, July 1973, Technische Universität Berlin, Fachbereich 20

[SchL] S. Schindler and H. Lüdtke: Eine Diskussion verschiedener Zugänge zum deterministischen Scheduling Problem, 6. Jahrestagung der Gesellschaft für Informatik, Stuttgart 1976, in: Informatik-Fachberichte, No.5, Springer-Verlag 1976

[SchS] S. Schindler and W. Simonsmeier: The class of all optimal schedules for a two-processor system, Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems, 1973

[Se] R. Sethi: Scheduling graphs on two processors, SIAM Journal
Vol.5, No.1, March 1976

[St] M. Steinacker: Doctoral Thesis, Technische Universität Berlin, Fachbereich 20, under preparation

[T] A.S. Tanenbaum: Structured computer organisation, Prentice-Hall, Inc., 1976

ANALYSIS OF STRUCTURES FOR PACKET COMMUNICATION*

Robert G. Jacobsen
David P. Misunas
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- In a system utilizing packet communication techniques of message transmission, all communication between the units comprising the system is through discrete blocks of information conveyed in packets. Interconnection structures in such systems can range from bus and crossbar structures to complex routing networks. A comparative analysis of a number of interconnection structures for packet communication systems is presented and tradeoffs between the various structures in terms of cost and performance are analytically examined.

Introduction

The increasing popularity of multiprocessor systems and the corresponding necessity for efficient interprocessor communication means has spurred the study and development of communication paths for use in such systems. One means for interprocessor communication which is gaining popularity is that of packet communication. In a system with packet communication architecture, the units comprising the system communicate through the transmission of discrete information packets [2].

Classical approaches to the design of communication paths have included such structures as busses and crossbar switching networks. These structures are necessarily small, due to the small number of interconnected units and due to the speed requirements placed on the structure. As the number of interconnected units increases, these structures become cumbersome both in size and processing capability.

More recently, a new interconnection structure, the routing network, has been presented and used in the design of a new type of parallel computer [3]. This structure is capable of simultaneously conveying many packets to their destinations in the processor and has a slower growth rate than the crossbar structure.

The tradeoffs between the various interconnection structures are not clearly understood. In the case of the routing network, little analysis has been performed at all. Detailed studies have examined such structures as the bus and crossbar [5, 9]. Some network structures have been studied [1, 8], particularly in the context of telephone switching networks [6, 7, 8]. However, these studies have generally considered only fixed connection circuits, rather than packet switching circuitry.

In the analysis of the present paper, we examine the characteristics of three communication structures: the bus, the crossbar, and the routing network. The cost and performance of each structure is analyzed to yield results as to the various tradeoffs involved in the choice of one structure over another. The analysis of these interconnection structures is supported through simulation results obtained on a packet communication simulation facility.

System Architecture

The design of a system interconnection structure is a difficult and poorly-understood problem, generally relying heavily on the experience of the system architect. There are no rules or guidelines for one to follow in such an exercise, merely a few general philosophies. In the following paragraphs, we will examine this situation more closely in the context of a packet communication system.

A packet communication system generally has some structure similar to that shown in Figure 1. The units comprising the User of Figure 1 may be processors, memories, functional units, or any other devices capable of message transmission or reception. The Communication Network of the system provides a path between the various units of the User. This interconnection structure may provide a path from every unit to every other unit, from groups of units to groups of units, or from each unit to one or several of the others. For the purposes of this discussion, we will assume the most general case; that is, every unit of the User can communicate with every other unit through the Communication Network. Other interconnection schemes can be considered as being composed of a number of embodiments of this more general case.

*This research was supported by the National Science Foundation under grant DCR75-04060 and by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract number N00014-75-C-06661.

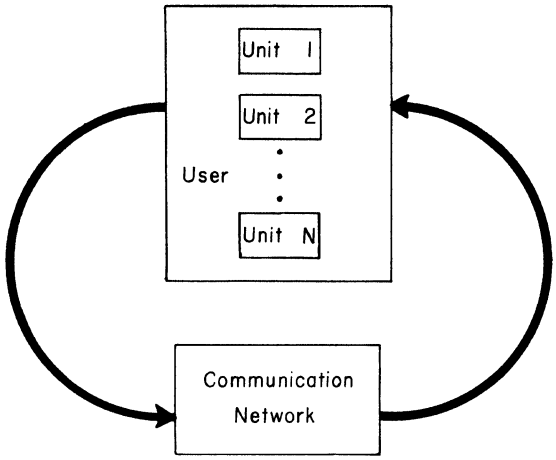


Figure 1. System Structure

Presumably, the designer of a packet communication system has an application area in mind for the system and has some idea of the amount of traffic which will pass over the communication medium. Thus, through some analysis, one should be able to generate a curve corresponding to the solid line of Figure 2. Such a user load curve expresses the number of packets generated as a function of the time required for an individual packet to transit the communication network and should always have a non-positive derivative, indicating that interunit communication will generally occur less frequently as the communication times increase.

On the other hand, the dashed curve of Figure 2 represents the load characteristics of the Communication Network and always has a non-negative derivative. The slope of the Communication Network load curve demonstrates that the load on the communication medium increases, the delay through the medium should eventually increase.

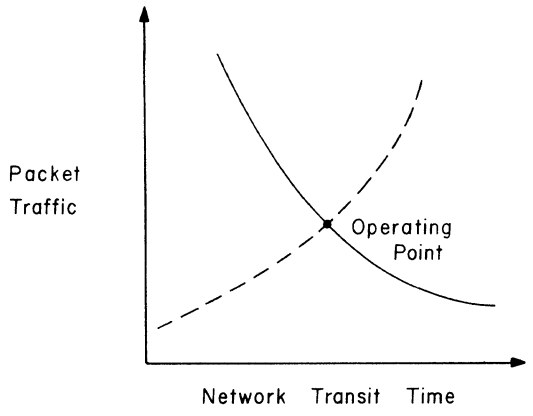


Figure 2. System Operating Characteristics

Generally, the two curves intersect at a point which will be the operating point of the system. Clearly, the system is only stable at the operating point and any digression from that point is countered by forces which tend to return the packet flow to the operating point.

Were it possible to empirically derive the User and Communication Network curves of Figure 2, the analysis and synthesis of packet communication systems would be greatly simplified. If there existed curves for the various types of interconnection structures, a designer need only develop the characteristic curve of his proposed User structure, choose a desired operating point on that curve, and match the appropriate Communication Network curve to yield the best cost/performance at that operating point.

Such a scheme may seem impractical, however, methods similar to this have been derived for many other branches of engineering, and there is no explicit reason why it is not possible to do so for aspects of computer design.

The remainder of this paper describes some preliminary results which were achieved while trying to generate load curves for various Communication Network structures. Whereas the achieved results do not yield rules for processor design, they provide a first step in that direction through the analysis of packet flow in the structures

Network Representation

The communication networks of the present study are formed of arbitration units and switch units. Each arbitration unit accepts the first packet to arrive at any input and passes the accepted packet to its output. In the case of conflict, one packet is arbitrarily selected and passed to the output before the other(s). Each switch unit transfers a packet on its input to one to its outputs, generally controlled by some switching specification contained in the packet.

The bus module of Figure 3 comprises an arbitration unit followed by a switch unit. Similarly, models for a crossbar and a routing network are shown in Figures 4 and 5. A network such as that of Figure 4 which is composed initially of switch units followed by arbitration units is called a distribution network, and a crossbar is one configuration of such a network. Similarly, a network which contains an initial stage of arbitration as that of Figure 5 is called an arbitration network.

The networks under study are structured as a number of stages connected in sequence. Each stage of a network is composed exclusively of either arbitration or switch units and is characterized by the log to the base N of the fanout/fanin ratio:

$$\log_N \frac{\text{(Number of Outputs)}}{\text{(Number of Inputs)}}$$

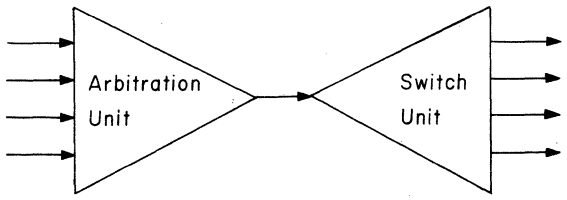


Figure 3. Structure of a Bus

This means of characterization has been chosen for two reasons. First, the size of the individual arbitration and switch units comprising each stage is clearly specified. Second, such a characterization represents a constant network architecture, regardless of the number of inputs and outputs.

The bus structure of Figure 3 (and all bus structures) is characterized by (-1, 1). Similarly, all crossbar structures are characterized by (1, -1). The "square-root" arbitration network of Figure 5 has the characterization (-1/2, 1/2, -1/2, 1/2).

Note that for an NxN communication network, the sum of all numbers in the network characterization must be equal to 0. Furthermore, in order for every input of a network to be able to communicate with every output, the sum of the absolute values of the numbers comprising the network characterization must be at least two. If the sum is greater than two, the network contains redundant paths.

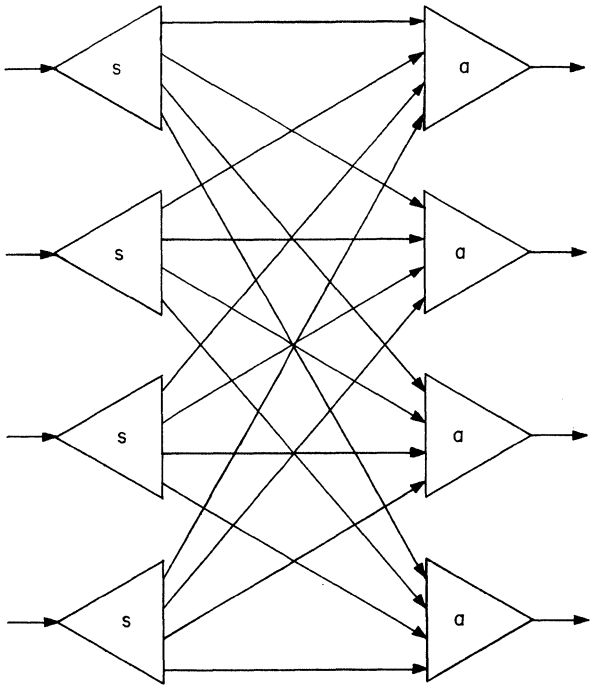


Figure 4. Structure of a Crossbar

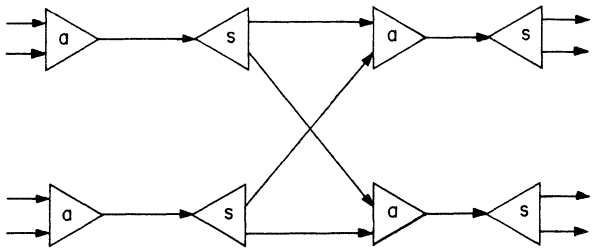


Figure 5. Structure of a Routing Network

At this point, we shall further restrict the networks under analysis to constant geometry NxN communication networks which can be characterized by a positive integer fraction f, where the network characterization is (-f, f, -f, f, ...) for an arbitration network or (f, -f, f, -f, ...) for a distribution network. The number of occurrences of f in each characterization is equal to the number of stages in the network, that is, to 2/f. Bus structures, crossbar structures, and simple power networks are examples of networks with such a characterization.

This restriction does not necessarily preclude the consideration in our model of networks which do not have alternating stages of arbitration and switch units. Without loss of generality, adjacent stages of the same type can be considered as one stage with a characterization which is equal to the sum of the characterizations of the two stages. However, the model described herein is only applicable to networks which can be characterized by a constant fraction f once reduction of identical adjacent stages has been performed.

Performance Analysis

For the purposes of finding the characteristic curve of a communication network, we need to make two simplifying assumptions. First, we consider the cost of a device proportional to the speed of the device times the number of wires connected to it. This assumption is not precisely accurate, but close enough for the purposes of this discussion.

Second, we assume that the packet distribution on the inputs of a communication network is even and Poisson and the distribution through any cross section of the network is even.

The communication networks under study are composed of an interconnection of one basic unit type, called a tie and consisting of an arbitration unit and a switch unit. The bus of Figure 3 is composed of one such tie. The network of Figure 5 can readily be seen to comprise a number of ties. Although the topology of a distribution network is slightly different than that of the networks in Figures 3 and 5, such a structure can be analyzed in a similar fashion.

We wish to examine two variables within each communication structure, a delay derater D and a loading representation F. D represents the average transit time for the network divided by the minimum transit time and can assume values ranging from one to infinity. D=1 signifies that the transit time through the communication network is only the hardware delay, whereas larger values of D indicate the presence of conflict in the structure.

F represents the fraction of the network that is not is use, that is, the free capacity of the network divided by the total capacity. In the following study, we examine D as a function of F to achieve each network characterization. The communication network load curve of Figure 2 represents a graphical depiction of a function similar to (1-F) vs. D. We have made this modification to the axis of the graph for the purposes of simplifying the analysis and the involved mathematics.

Representing the interarrival time on each input of an n-input tie by I and the service time by T, we find that a packet will arrive every I/n and hence:

$$F_{tie} = 1 - nT/I$$

Generalizing to all the units of a stage, a packet can be transmitted to the next stage at most every $T(n/N) = T(N^f/N) = T/N^{(1-f)}$. Thus:

$$F_{stage} = 1 - (T/N^{(1-f)})/(I/N)$$
$$= 1 - N^f T/I$$

Since all stages in this type of network are similary constructed:

$$F_{network} = F_{stage} = 1 - N^f T/I$$

The application of queueing theory techniques to the performance analysis of one tie, considering each tie as a queue and assuming Poisson arrival rates, yields the result:

$$D = 1 + (1-F)/4F$$

All ties in the network operate at the same F. Hence, overall, we can say:

$$D_{network} = 1 + (1 - F_{network})/4F_{network}$$

Simulation Results

Utilizing a packet communication simulation facility, a number of bus, crossbar, and routing network structures were simulated to see if actual performance followed the D = 1 + (1-F)/4F formula. The simulation results are depicted in Figure 6.

The solid line of Figure 6 represents the graph of D = 1 + (1-F)/4F, and the points resulting from the simulation appear to observe this characteristic for the three structures under study.

The simulation modelled each network input as an independent source with a Poisson distribution and given interarrival time. The discrepencies of the simulation from the model for small values of F are due to the fact that the model contained infinite queues between the sources and the input ports, whereas such is impractical in the simulation, eventually causing the input queues to back up and affect operation of the sources.

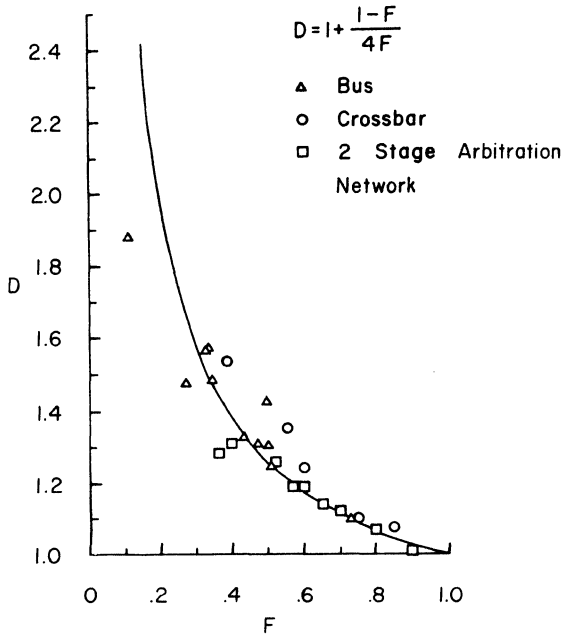


Figure 6. Simulation Results

Network Selection

The cost analysis for an arbitration network such as that of Figure 5 can be represented as follows, where C_{AN} is the cost of the network:

$$C_{AN} = (\text{number of stages})(\text{cost of each stage})$$
$$= (1/f)(\text{speed} * \text{number of wires})$$
$$= (1/f)([N^f/f] * N)$$
$$= N^{(1+f)}/f^2$$

In this case, speed is equal to N^f/f to maintain a constant average delay through the network with changes in f. The term N^f compensates for the increased loading of arbitration units due to the compression by N^f. The 1/f arises from the need for each stage to operate faster in networks with more stages.

In the case of a distribution network:

$$C_{DN} = (\text{number of stages})(\text{cost of each stage})$$
$$= (1/f)(\text{speed} * \text{number of wires})$$
$$= (1/f)(1/f) * (N^{(1+f)})$$
$$= N^{(1+f)}/f^2$$

A distribution network has a greater number of wires because each input wire of a stage of such a network is expanded to $N^{(1+f)}$ wires. Due to this

expansion, the component speed in a distribution network is only affected by the number of stages, that is, by 1/f.

Thus the linear cost assumption has led us to the conclusion that for some fixed performance, the arbitration network of Figure 5 costs the same as the distribution network of Figure 7. This result is non-intuitive at first, however, consider an arbitration network of complexity N. The units comprising this network have speed N due to the initial compression factor. The complexity of an equivalent distribution network is N^2, but the additional parallelism allows the network to be constructed of components with speed 1. Hence, the cost of the two networks is equivalent.

The minimum of the network cost $N^{(1+f)}/f^2$ occurs at

$$1/f = (1/2) \ln N$$

where 1/f is the number of stages. Hence, for the linear cost assumption of the model, the following structures are best suited for the specified number of inputs for either arbitration or distribution network:

N	Structure
7	1-stage networks (bus and crossbar)
50	2-stage networks
400	3-stage networks
3000	4-stage networks

An interesting result which arises from the performance computations is the determination of the optimal value of n, that is, the number of inputs to each arbitration unit and outputs of each switch unit. As we have seen, the minimum cost occurs when f = 2/ln N. Thus, these expansion and compression ratios should be:

$$N^f = N^{2/\ln N} = e^2 \cong 7$$

To utilize the previously described results in the design of a packet communication system, one first determines the load curve of the units to be interconnected. The architecture of the communication network utilized in the system is specified by the number of units. With these specifications in mind, there are a number of design choices which can be made.

The load curves of the communication network consist of a family of curves which are parametric with cost. To design for a specific cost or technology, the intersection of that member of the family with the user load curve yields the performance which can be achieved.

Conversely, to structure the system for a specific performance, the desired operating point on the user curve is specified and the network curve which passes through that point determines the cost and speed necessary in the component parts.

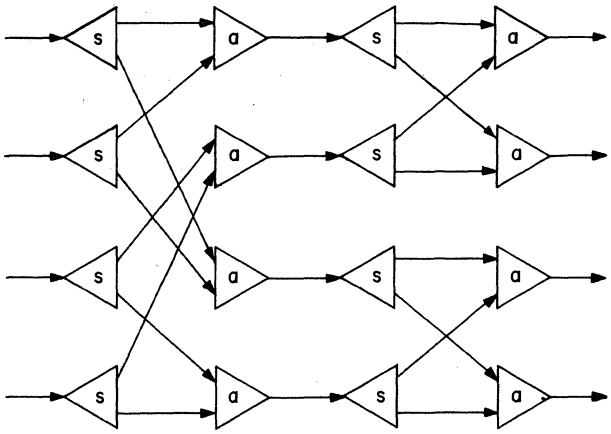


Figure 7. Structure of a Distribution Network

The choice of either an arbitration network or a distribution network must take into account important factors such as the available technologies. While these factors are not included in the model, they will dictate actual use of any results achieved therefrom.

Concluding Remarks

This attempt to probe the interconnection problem for packet communication systems has left many questions unanswered. The model utilized has a number of deficiencies and remains to be made more exact and extended to structures other than certain NxN power networks, such as asymmetric networks and concentration networks. Further refinement of the model and addition of other structures should provide much information useful in the synthesis of processor structures for packet communication. Despite its deficiencies, the model provides a first attempt to analyze such packet communication interconnection structures and yields some interesting insights into their behavior.

References

[1] Davidson, I. A., and J. A. Field, "Design Criteria for a Switch for a Multiprocessor Computing System," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, (August 1975), pp. 110-114.

[2] Dennis, J. B., "Packet Communication Architecture," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, (August 1975), pp. 224-229.

[3] Dennis, J. B., and D. P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing," Proceedings of the ACM 1974 National Conference, ACM, New York, (November 1974), pp. 402-409.

[4] Marcus, M. J., New Approaches to the Analysis of Connecting and Sorting Networks, Research Laboratory of Electronics, M.I.T., Cambridge, Mass., Technical Report 486, (March 1972), 54 pp.

[5] Pearce, R. C., and J. C. Majithia, "Upper Bounds on the Performance of Some Processor-Memory Interconnections," preprint.

[6] Pippenger, N., The Complexity Theory of Switching Networks, Research Laboratory of Electronics, M.I.T., Cambridge, Mass., Technical Report 487, (December 1973), 51 pp.

[7] Pippenger, N., "On Crossbar Switching Networks," IEEE Transactions on Communications COM-23, 6 (June 1975), pp. 646-659.

[8] Thurber, K. J., "Interconnection Networks -- A Survey and Assessment," AFIPS Conference Proceedings 43, AFIPS Press, Montvale, New Jersey, (1974), pp. 909-919.

[9] Thurber, K. J., et. al., "A Systematic Approach to the Design of Digital Bussing Structures," AFIPS Conference Proceesings 41 Part II, AFIPS Press, Montvale, New Jersey, (Fall 1972), pp. 719-740.

INTRODUCING THE CONCEPT OF DATA STRUCTURE ARCHITECTURES

W. K. Giloi and H. Berg
Computer Science Department
University of Minnesota
Minneapolis, Minnesota 55455

Abstract -- In the paper, the concept of data structure architectures is developed as a solution to the problem of providing increased hardware support for the basic task of computing, viz. the creation and processing of data structures. As a starting point, a uniform algebraic description of data structures is presented. Consequently, the necessity for a management of the two fundamental types of data entities, ordered sets and general sets, is recognized. In order to allow a machine to handle the various data structures by a standardized hardware, an intermediate data structure, called the basis and managed by hardware, is introduced. The programmer creates arbitrary data structures in terms of basis elements which are, in turn, mapped by the hardware onto consecutive storage. The processing of basis elements in response to a single machine instruction is based on the referencing of basis element descriptors and implemented by pipelined processors.

Keywords: computer architecture, performance architecture, general-purpose computing, data structures, ordered sets, data model, descriptor-referenced allocation, tagged architectures, hardware execution.

1. Introduction

The basic organizational concept of most computers presently being used or being marketed is still the 30 years old concept as developed by von Neumann, Burks, and Goldstine [1]. In our opinion, the reason for the so amazing longevity of the von Neumann principle is its unique combination of simplicity and flexibility. The von Neumann concept may be epitomized as a concept of minimal hardware resources: The basic von Neumann machine encompasses one central processing unit, one main memory, and one input/output channel.

This concept of hardware minimality, which was perfectly adequate at a time when the hardware of a computer was the major cost factor, has meanwhile turned into the major factor that will obsolete the von Neumann architecture. In the age of dramatically decreasing cost of standardized LSI componentry, concepts are needed which allow increased hardware expenditures in order to achieve certain design objectives such as an increase in performance or availability or both. Such a multiplication of hardware resources implies the abolishment of the most severe performance-limiting feature of the von Neumann machine, namely that it manipulates the content of only a single memory location at the time, in favor of the simultaneous accessing and processing of a set of values, i.e., in favor of parallel processing.

Most of the existing parallel processing architectures, however, were initially designed for special purposes rather than for general-purpose computing, and many of these architectures do not lend themselves very well to a generalization. Whereas it is a rather straightforward task to design the architecture of a special purpose computer, based on a homogeneous class of algorithms (e.g., for solving partial differential equations or for processing a matrix of radar data), it is not possible to define such distinguished classes of algorithms after which a computer architecture could be modelled if the universe of all possible algorithms is considered. However, in the search for a class of architectures for general-purpose computing, i.e., architectures which can really replace the von Neumann architecture, the whole domain of computing must be taken into account.

The most general definition of computation is that of "a sequence of transformations which transform an initial representation through a sequence of intermediate representations into a final representation" [2]. A representation is a transforming function and its data. As it is not possible to identify patterns in the universe of all possible transforming functions which could render the blueprint for a class of general-purpose architectures, the only possibility left is the structuring of the data or, more precisely, the processing of appropriately structured data entities.

In the von Neumann machine, data are totally unstructured, i.e., the only data entity of the machine is the scalar. In real-world computation, we find always structuring relationships between the data of a program which constitute the basis for data retrieval and processing. An architecture which supports the representation and processing of arbitrary data structures by hardware shall be called a data structure architecture (DSA). It need hardly be emphasized that a data structure architecture should be complete and minimal, i.e., it should allow for the representation of any desired structure, and it should employ for this purpose a minimal number of standardized tools.

2. A Formal Definition of Data Structures

Knuth [3] defines data structure as "a table of data including structural relationships". Formalizing this, we define a data structure as a pair

$$(S, \rho)$$

where $S = \{s_1,\ldots,s_n\}$ is a set of data objects

and $\rho = \{R_1,\ldots,R_r\}$ is a set of binary relations such that $\bigwedge_{1 \le i \le r} : R_i \subseteq S \times S$.

By specifying certain properties of the relations in ρ, different structure types are obtained. These are basically the following four types [4].

$$(S,\rho) = (S,\{\trianglelefteq\}) \quad , \tag{1}$$

where $\trianglelefteq$ denotes a relation that is reflexive, antisymmetric, and transitive, and satisfies the additional condition that for any two objects s_i, $s_j \in S$ at least one of the two propositions $s_1 \trianglelefteq s_2$ or $s_2 \trianglelefteq s_1$ is true. Thus $\trianglelefteq$ denotes a linear ordering. This relation defines an ordered set $\{S[1],\ldots,S[n]\}$ of data objects $S[i] \in S$ which are identified by an ordinal number specifying their relative position in the set. This structure is usually called a linear list.

A simple generalization of a linear list is a two-dimensional or higher-dimensional array of data objects. In a rectangular two-dimensional $m \times n$ array we have the linear row lists $(R_i, \{\trianglelefteq_{R,i}\})$, $i \in [1:m]$, with $R_i = \{R_i[1],\ldots,R_i[n]\}$ and the linear column lists $(C_j,\{\trianglelefteq_{C,j}\})$, $j \in [1:n]$, with $C_j = \{C_j[1],\ldots,C_j[m]\}$. These linear lists are orthogonally connected such that the linear ordering of the row lists implies the same ordering of the column lists and vice versa. Hence, a two-dimensional $m \times n$ array is defined by the pair (the definition can be easily extended to any arbitrary higher dimension)

$$(S,\rho) = (\bigcup_{i=1}^{m} R_i,\ \{\trianglelefteq_{R,1},\ldots,\trianglelefteq_{R,m},\ \trianglelefteq_{C,1},\ldots,\trianglelefteq_{C,n}\}).$$

$$(S,\rho) = (S,\{q_1,\ldots,q_{p-n}\}) \quad , \tag{2}$$

where $\rho = \{q_1,\ldots,q_{p-n}\}$ is a set of relations that are reflexive, symmetric, and transitive, i.e., equivalence relations. The equivalence relation q_1 defines a partition $P(S) = \{S_1,\ldots,S_m\}$ of the set S, and the remaining equivalence relations in ρ define refinements of this partition. If we assume that P(S) is refined until n singleton sets, $\{s_1\},\ldots,\{s_n\}$, are obtained, each one containing exactly one of the elements of S, the result is a collection of nested sets [3] $C = \{C_1,\ldots,C_p\}$ such that each equivalence relation $q_k \in \rho$ defines a partition $P(C_k) \subset C$, $k \in [1:p-n]$, whereas the remaining n sets in C are the elements of the set $\{\{s_1\},\ldots,\{s_n\}\} = \{C_{p-n+1},\ldots,C_p\}$ (any partition generates a collection of nested sets; but not any collection of nested sets constitutes a partition). Such a data structure is called a tree.

The definition of the collection of nested set $C = \{C_1,\ldots,C_p\}$ does not imply an ordering of the equivalence classes $C_j \in P(C_k)$, $1 \le k \le p-n$, but only indicates the ancestor-descendant relationship among the equivalence classes $C_i \in C$. Trees which are equivalent to a collection of nested sets are called oriented trees, since only the relative orientation of the nodes is being considered. An ordering of all equivalence classes $C_j \in P(C_k)$ implies an ordering $\trianglelefteq$ on the data objects $s_i \in S$. Thus, an ordered tree is defined by a pair

$$(S,\rho) = (S,\{\trianglelefteq,q_1,\ldots,q_{p-n}\}) \quad .$$

$$(S,\rho) = (S,\{R_1,\ldots,R_r\}) \quad , \tag{3}$$

where the relations $R_i \in \rho$ are defined in certain pre-defined subsets $A_i, B_i \subseteq S$, i.e., $R_i \subseteq A_i \times B_i \subseteq S \times S$, with the additional constraint for the range $\mathcal{R}_i$ of a relation R_i that $\mathcal{R}_i = B_i$ (we call such a relation range-total). No constraint is given for the domain $\mathcal{D}_i$, that is $\mathcal{D}_i \subseteq A_i$. Furthermore, we have

$$S \supseteq A = \bigcup_{1 \le i \le r} A_i \quad \text{and} \quad S = B = \bigcup_{1 \le i \le r} B_i = \bigcup_{1 \le i \le r} \mathcal{R}_i \, .$$

Adopting Knuth's terminology, we call such a data structure a List.

The set ρ of relations $R_i = \{(s_j,s_k)/p_i(s_j,s_k)\} \subseteq \mathcal{D}_i \times \mathcal{R}_i \subseteq A \times S \subseteq S \times S$ may be defined by a set $\pi = \{p_i,\ldots,p_r\}$ of propositions. For each relation $R_i \in \rho$, we call the elements $a_j \in A_i$, with $\mathcal{D}_i \subseteq A_i \subseteq S$, the reference elements of R_i. Then, a relation $R_i \subseteq \mathcal{D}_i \times \mathcal{R}_i \subseteq A_i \times S$ generates for each $a_j \in A_i$ a subset of $\mathcal{R}_i$ that shall be denoted $\mathcal{R}_i/a_j$ (read: "the subset of $\mathcal{R}_i$ with respect to a_j"), such that

$$\mathcal{R}_i/a_j = \{s \in \mathcal{R}_i / p_i(a_j,s)\} \quad .$$

$\mathcal{R}_i/a_j = \emptyset$ if $a_j \notin \mathcal{D}_i$ and $\bigcup_{a_j \in \mathcal{D}_i} \mathcal{R}_i/a_j = \mathcal{R}_i$. The relations $R_i \in \rho$ define a set $N = \{\mathcal{R}_i/a_j / i \in [1:r] \wedge a_j \in A_i\}$ such that $\{s_1\},\ldots,\{s_n\} \in N$. The nodes of a List represent the sets $\mathcal{R}_i/a_j \in N$. The definition of some ad hoc ordering $\trianglelefteq$ on the data objects $s_i \in S$ implies an ordering of the nodes representing the sets $\mathcal{R}_i/a_j \in N$ in all sub-Lists of a List L which is defined by a pair

$$(S,\rho) = (S,\{\trianglelefteq,R_1,\ldots,R_r\}) \quad .$$

$$(S,\rho) = (S,\{R_1,\ldots,R_r\}) \tag{4}$$

where the relations $R_i \in \rho$ are defined in subsets $A,B \subseteq S$ such that

$S \supseteq A = \bigcup_{1 \le i \le r} \mathcal{A}_i$ and $S \supseteq B = \bigcup_{1 \le i \le r} \mathcal{R}_i$ and $S = A \cup B$.

No constraints are imposed on the relations $R_i \in \rho$. We call such a data structure an associative structure. The elements $a_j \in A \subseteq S$ are called domain elements and the elements $b_k \in B \subseteq S$ are called range elements. The set ρ of relations $R_i = \{(a_j,b_k)/p_i(a_j,b_k)\} \subseteq \mathcal{A}_i \times \mathcal{R}_i \subseteq A \times B \subseteq S \times S$ is defined by a set $\pi = \{p_1,\ldots,p_r\}$ of propositions. Thus, the data structure under consideration may be specified by the triad (A,π,B) [5].

3. The Necessity of a Machine Data Model

In order to store a data structure $(S,\rho) = (\{s_1,\ldots,s_n\},\{R_1,\ldots,R_r\})$ in a computer memory, the information content of that data structure, i.e., the set $S = \{s_1,\ldots,s_n\}$ of data objects and the set of structuring relations $\{R_1,\ldots,R_r\}$ must be represented in an appropriate form. That is, a memory representation of a data structure must retain the set-element relationships defined by the relations $R_k \in \rho$. The relations $R_k \subseteq S \times S$ define subsets $S_j \subseteq S$ which are represented by the nodes of the corresponding data structure (S,ρ). If all singletons $\{s_i\}$, $s_i \in S$, are uniquely identified by the relations $R_k \in \rho$ in connection with reference elements $s_j \in S$, then the definition of a linear ordering of the data objects $s_i \in S$ implies an ordering of all the nodes of the corresponding data structure. Otherwise, the nodes of the corresponding data structure represent (unordered) sets $S_j \subseteq S$ of data objects. Therefore, a data structure architecture must provide hardware support for the management of ordered sets and general sets, as well as an adequate set of operators defined on these fundamental types of data entities.

Physical memory can be either location-addressable or content-addressable (associative). Hardware-associative memory is ruled out for two reasons: Firstly, its cost is prohibitive and, secondly, it is not needed, as will be shown subsequently, if the purpose is to store and access structured sets of data rather than unstructured, general sets. In the case of location-addressed memory, the most fundamental mode of storing the data items of a data structure is the consecutive storage in the form of a data vector. The algebraic definition of a data structure can here be substituted by the "semantic" definition

<data structure> =

(<data vector>,<structure specification>) .

In the von Neumann machine, the mapping from a data structure to its data vector is performed (by software) in one step. However, such a mapping can be greatly facilitated if the data structure is, in a first step, mapped onto an appropriate 'intermediate' data structure which, in turn, is then mapped in a second step onto the data vector. We call the first mapping a structure definition and the second mapping an addressing function. The advantage of this approach lies in the fact that a standardized intermediate structure can be found that is necessary and sufficient for the representation of all data structures defined in section 2, whereas the data vector represents only the data of those structures.

Let $\mathbb{N}$ denote the set of non-negative integers and let M be the set of memory addresses. A data vector that is physically represented by sequential memory location is defined by the mapping

$$\nu : \mathbb{N} \to M \quad .$$

Let B be a set of r-dimensionally ordered sets, i.e., an element of B is defined by

$$\sigma : \mathbb{N}^r \to \mathbb{N} \quad .$$

We call B the basis of the data structure architecture. The positions in the r-tuples $(n_1,\ldots,n_r) \in \mathbb{N}^r$ are called the coordinates of the r-dimensionally ordered set, and r is called its rank (dimensionality). An element of $\mathbb{N}^r$ is called an index r-tuple. The index r-tuples are unique identifiers of the elements of an r-dimensionally ordered set, as the function σ maps index r-tuples into indices which specify the relative position of the identified element in the data vector. Hence, the mapping from the thus defined basis into a physical data vector, based on a sequential memory allocation, is accomplished by a composition of the functions σ and ν into a function

$$\alpha : \mathbb{N}^r \to M$$

which we call the addressing function.

Sequential allocation is characterized by the linear ordering of the memory locations. The addressing function for sequentially allocated r-dimensionally ordered sets is

$$\alpha(n_1,\ldots,n_r) = \beta + (\sigma(n_1,\ldots,n_r)-1)\cdot m \quad ,$$

where $\beta \in M$ is the base address, and m is the number of memory words occupied by each data item. The limitation of the basis to multi-dimensionally ordered sets thus allows the use of a rigorously standardized addressing function -- an absolute must if the addressing function is to be executed by hardware. Hence, we consider a class of computer architectures where we have multi-dimensionally ordered sets as the standardized internal data structure, called the basis and handled by the hardware of the machine. Fig. 1 presents a general diagram of such a data structure architecture.

Of course, a data structure architecture shall process at the hardware level not only multi-dimensionally ordered sets but any of the structure types as defined in section 2. To this

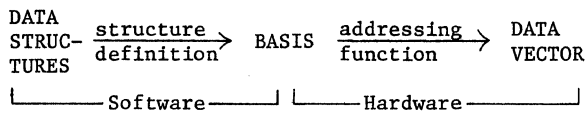


Fig. 1 General Concept of Data Structure Architectures

end, other data structures must be mapped through an appropriate structure definition on multi-dimensionally ordered sets, i.e., on the basis of the data structure architecture. Therefore, a mechanism for structure definitions must be developed, and it must be proved that all types of data structures can be defined in such a way. These stipulations can be satisfied by introducing an appropriate machine data model. A machine data model defines legitimate data types and structuring relations which are applicable for the definition of arbitrary data structures in terms of the basis. In order to mitigate the restriction that only rigorously standardized physical structures can be used for a hardware realization, a machine data model must be more general than the conceptual data models which were developed for generalized data-base management [6], [7].

4. The Linear Data Model

As a basis for the design of data structure architectures, we define the linear data model.
DEFINITION: The linear data model is based on the linear ordering as the only structuring relation. Identifiers of basis elements are a data type of the linear data model.
Unlike a pointer, an IDENTIFIER does not represent a reference to the identified basis element but the basis element itself [8]. The data items of a basis element are stored in a data vector. Hence, the linear data model defines basis elements as ordered sets of data vectors. Consequently, multi-dimensionally ordered sets are the only basis structures permitted by the linear data model.

Let A be an r-dimensionally ordered set whose components are denoted $A[n_1;...;n_r]$. With the definition of the admissible ranges of the index values n_i, $i \in [1 : r]$, in all index lists $[n_1;...;n_r] \in \mathbb{N}^r$ such that $n_1 \in [1 : d_1]$ and $n_j \in [1 : d_j[n_1;...;n_{j-1}]]$, $j \in [2 : r]$, the linear lists $(A[n_1;...;n_{i-1};1;n_{i+1};...;n_r],...,A[n_1;...;n_{i-1};d_i[n_1;...;n_{i-1}];n_{i+1};...;n_r])$, $i \in [1 : r]$, define cross sections of A which are denoted $A[n_1;..;n_{i-1};n_{i+1};..;n_r]$. We call $d_i[n_1;...;n_{i-1}]$ the dimension of the linear list $A[n_1;..;n_{i-1};n_{i+1};..;n_r]$.

The introduction of the linear data model as the fundamental notion for the design of data structure architectures is based on the
THEOREM: The linear data model is necessary and sufficient for the definition of linear lists, arrays, trees, generalized lists, and associative structures in terms of multi-dimensionally ordered sets of linear lists.
Proof. (1) Necessity: Linear orderings constitute the simplest possible structuring relations with respect to the representation of data structures in location-addressed memories. Linear lists are the fundamental basis elements, as they are identical with the structure of the underlying data vectors.

(2) Sufficiency: An r-dimensionally ordered set A can be represented by an (i-1)-dimensionally ordered set whose components are the (r-i+1)-dimensionally ordered sets $A[n_1;...;n_{i-1}]$, $n_j \in [1 : d_j[n_1;...;n_{j-1}]]$, $j \in [1 : i-1]$. In the notation $A[n_1;...;n_{i-1}][n_i;...;n_r]$, the second index list $[n_i;...;n_r]$ specifies the components in the (r-i+1)-dimensionally ordered set $A[n_1;...;n_{i-1}]$ as defined by the index list $[n_1;...;n_{i-1}]$. Thus, the components $A[n_1;...;n_{i-1}][n_i;...;n_r]$ represent the components $A[n_1;...;n_r]$ of the r-dimensionally ordered set A. By forming cross sections of the components $A[n_1;...;n_{i-1}][n_i;...;n_r]$, an r-dimensionally ordered set A can be defined as an (i-1)-dimensionally ordered set of linear lists

$$A[n_1;...;n_{i-1}] = (A[n_1;...;n_{i-1}][1],...,A[n_1;...;n_{i-1}][d_i[n_1;...;n_{i-1}]]),$$

such that the second index list [k], $k \in [1 : d_i[n_1;...;n_{i-1}]]$, specifies the (r-i)-dimensionally ordered sets $A[n_1;...;n_{i-1};k]$ which are the components of the linear lists $A[n_1;...;n_{i-1}]$. Obviously, for i = r, the above derivation defines an r-dimensionally ordered set A as an (r-1)-dimensionally ordered set of linear lists. It is readily recognized that the recursive application of the above definition leads to representations of r-dimensionally ordered sets as orthogonal interconnections of linear lists. Moreover, the definition of the data type IDENTIFIER allows the representation of any set containment in an r-dimensionally ordered set in the form of linear lists $A[n_1;...;n_{i-1}]$ whose components $A[n_1;...;n_i][k]$ may represent arbitrary identifiers $A[m_1;...;m_j]$ which, in turn, represent (r-j)-dimensionally ordered sets $A[m_1;...;m_j]$, $j \in [1 : r]$. Obviously, with the above definition of the data type IDENTIFIER, the r coordinates of an r-dimensionally ordered set correspond to r levels of substructure containment. Therefore, a linear list $A[n_1;...;n_{i-1}]$ of data type IDENTIFIER may represent a node at the (i-1)st level of a hierarchical structure and is thus a "parent" of components $A[n_1;...;n_{i-1}][k]$ which may represent nodes

$A[m_1;\ldots;m_j]$ at any level of the hierarchical structure. In addition to the predecessor-successor relationships defined by linear orderings, the introduction of the data type IDENTIFIER hence allows the definition of arbitrary parent-child relationships.

A tree structure can be defined by the specification of an r-dimensionally ordered set, such that the components $A[n_1;\ldots;n_{i-1}][k]$ of the linear lists $A[n_1;\ldots;n_{i-1}]$ of data type IDENTIFIER exclusively represent the (r-i)-dimensionally ordered sets $A[n_1;\ldots;n_{i-1};k]$. As the components $A[n_1;\ldots;n_{i-1}][k]$ of linear lists $A[n_1;\ldots;n_{i-1}]$ may represent identifiers of arbitrary (r-j)-dimensionally ordered sets $A[m_1;\ldots;m_j]$, it is obvious that generalized lists can be defined by the linear data model.

The linear ordering of the memory locations in a location-addressed memory implies an ordering of the elements of general sets in memory representations. Thus, linear lists are adequate logical structures for the representation of general sets in location addressed memories. Possible nestings of general sets are also easily manageable through linear lists of data type IDENTIFIER. The latter property of the linear data model, and the ability to arbitrarily link multi-dimensionally ordered sets through their identifiers, may efficiently be applied for the definition of associative structures (q.e.d.).

The above discussion of the efficiency of the linear data model shows that a hardware-associative memory would not facilitate the storage of basis elements,for components of multi-dimensionally ordered sets are uniquely identified through its index list $[n_1;\ldots;n_r] \in \mathbb{N}^r$. Therefore, the multi-match capabilities of an associative memory cannot be exploited.

5. The Internal Information Structure

5.1 A Proposed Standardization of the Basis Elements

So far, we assumed basis elements to be r-dimensionally ordered sets. In section 4 it is proved that one-dimensionally ordered sets (linear lists) are sufficient for the representation of arbitrary data structures. However, we propose two-dimensionally ordered sets, given in the form of homogeneous, rectangular arrays (matrices) as the standardized basis element. Such a structure has the following desirable properties:

(i) The dimension of the linear lists in the two coordinates of a matrix are the same, i.e., a basis element is fully specified by a dimension vector $D = (d_1,d_2)$, where d_1 and d_2 are the column dimension and the row dimension, respectively.

(ii) The addressing function is given by the simple expression

$$\alpha(n_1,n_2) = (n_1-1)\cdot d_2 + n_2 - 1 + \beta \ .$$

(iii) Matrices are the most important data structure in practical applications. The existence of several structuring relations within a linear list (as usually represented by multi-linked structures) can be represented by a single basis element of data type identifier.

5.2 Memory Representation of the Basis

The standardized basis elements are represented by variable descriptors which contain the parameters of the addressing function α: $\mathbb{N}^2 \to M$. The general format of these variable descriptors is defined by the triple

$$VD = (a,s,b) \ ,$$

with a = variable attributes (including data type specification), s = structure specification, and b = base address of the data vector. Hence, the data definition of two-dimensional arrays is obtained in the form of standardized variable descriptors

$$VD = (\langle\text{attributes}\rangle,(\langle\text{column dimension}\rangle,\langle\text{row dimension}\rangle),\langle\text{base address}\rangle).$$

Variable descriptors of this format can have a uniform length of one memory word. Thus, data definitions can be stored as named variable descriptors, such that descriptor identifiers are equated with the memory locations which contain the associated variable descriptors. It is readily recognized that identifiers of basis elements as defined by the linear data model correspond with descriptor identifiers. In contrast to the von Neumann machine, where a machine variable is defined by a pair <variable> = (<location>,<value>), we have the following machine variable structure

$$\langle\text{variable}\rangle = (\langle\text{name}\rangle,\langle\text{value}\rangle)$$
$$\langle\text{name}\rangle = \langle\text{descriptor identifier}\rangle$$
$$\langle\text{value}\rangle = (\langle\text{data vector}\rangle,\langle\text{structure specification}\rangle).$$

This machine variable structure implies a two-stage value reference scheme through variable descriptors. The components of the data vector are accessed by executing the addressing function α for the structure specification given in the variable descriptor. This value reference scheme also applies to multi-dimensionally ordered sets which are represented by two-dimensional arrays of data type IDENTIFIER. According to the definition of the data type IDENTIFIER, references to components of data vectors of data type IDENTIFIER are automatically replaced by references to the identified variable descriptors. This indirect reference scheme can be nested to any arbitrary depth, resulting in an iterative application of the standardized two-stage value reference mechanism.

With the equivalence of coordinates of multi-dimensionally ordered sets and the levels of sub-structure containment (cf. section 4), we obtain a correspondence of n-1 nested references of descriptor identifiers in two-dimensional arrays with a (2n)-dimensionally ordered set. Let A be a two-dimensional array of data type IDENTIFIER which represents a (2n)-dimensionally ordered set with components $A[m_1;m_2][m_3;m_4]\ldots[m_{2n-1};m_{2n}]$. The components of this (2n)-dimensionally ordered set are the components of all two-dimensional arrays $A[m_1;m_2]\ldots[m_{2n-3};m_{2n-2}]$ which are accessed through n-1 levels of descriptor references. The descriptor references are defined by the descriptor identifiers $A[m_1;m_2]\ldots[m_{2i-1};m_{2i}]$ of the two-dimensional arrays $A[m_1;m_2]\ldots[m_{2i-3};m_{2i-2}]$ of data type IDENTIFIER, $i \in [1 : n-1]$ (for i=1, $A[m_{-1};m_0] = A$). That is, the components $A[m_1;m_2][m_3;m_4]\ldots[m_{2n-1};m_{2n}]$ are accesses through n iterative executions of the addressing function

$$\alpha(m_{2j-1};m_{2j}) = (n_{1,A[m_1;m_2]..[m_{2j-3};m_{2j-2}]}{}^{-1}) \cdot d_{2,A[m_1;m_2]..[m_{2j-3};m_{2j-2}]} + n_{2,A[m_1;m_2]..[m_{2j-3};m_{2j-2}]}{}^{-1} + \beta_{A[m_1;m_2]..[m_{2j-3};m_{2j-2}]} ,$$

$j \in [1 : n]$. The base addresses $\beta_{A[m_1;m_2]..[m_{2j-3};m_{2j-2}]}$ and the dimension vectors $d_{A[m_1;m_2]..[m_{2j-3};m_{2j-2}]} = (d_{1,A[m_1;m_2]..[m_{2j-3};m_{2j-2}]}, d_{2,A[m_1;m_2]..[m_{2j-3};m_{2j-2}]})$ are specified in the variable descriptors of the two-dimensional arrays $A[m_1;m_2]\ldots[m_{2j-3};m_{2j-2}]$. Each two-dimensional array $A[m_1;m_2]\ldots[m_{2i-3};m_{2i-2}]$ defines the ordering of the two-dimensional arrays $A[m_1;m_2]\ldots[m_{2i-1};m_{2i}]$, $i \in [1 : n-1]$, within the coordinates with indices 2i-1 and 2i. Hence, in accordance with the definition of the data type IDENTIFIER, the machine variable A completely defines the ordering of all components $A[m_1;m_2]\ldots[m_{2n-1};m_{2n}]$ within all 2n coordinates.

We call the above memory allocation scheme for basis elements a descriptor referenced allocation. With the specification of all necessary variable attributes and of the dimension vectors in the variable descriptors of the two-dimensional arrays, $A[m_1;m_2]\ldots[m_{2i-1};m_{2i}]$, variable definitions are completely self-descriptive. That is, descriptor referenced allocation allows for modular variable definitions through variables of data type IDENTIFIER. The descriptor identifiers bind the variable definitions of basis elements, and hence, the descriptions of multi-dimensionally ordered sets. The automatic replacement of descriptor identifiers by the referenced descriptors builds up a complete structure specification by selecting the appropriate parameters for the iterative execution of the addressing function α. Hence, the addressing function for two-dimensional arrays is the only tie between multi-dimensionally ordered sets of basis elements representing arbitrary data structures and components of these data structures.

The descriptor reference mechanism for variables of data type IDENTIFIER does not prescribe a uniform data type for all components. Rather, the data type of the components is described by their variable descriptors. Hence, heterogeneous data structures can be defined. Furthermore, the unique definition of the coordinate dimensions of multi-dimensionally ordered sets by the variable descriptors at the different reference levels allows the construction of irregular data structures. As shown in [9], the descriptor referenced allocation scheme can also be exploited to arbitrarily restructure multi-dimensionally ordered sets without modifying or copying the underlying data vectors. To this end, the addressing function is extended into a generalized storage access function. In addition to that, the dimension vector in the variable descriptor of a restructured variable is replaced by a description of appropriate structure functions. Restructured variables reference the variable descriptor of the variables from which they were generated through restructuring. With the automatic replacement of descriptor identifiers, the generalized storage access function then maps a components S[i;j] of a two-dimensional array S onto the data vector of a two-dimensional array A, from which S was generated through restructuring. That is, the execution of the generalized storage access function comprises the execution of the addressing function α and the execution of the stored structure functions.

The descriptor referenced allocation of data structures is a refinement of the concept of self-identifying information components in tagged architectures [10], [11]. In tagged architectures, self-identification provides the possibility to uniquely associate with each category of variable specifications dedicated control routines. Contrastingly, descriptor referenced allocation defines self-descriptive data entities through a standardized basis which can be managed by a standard set of control routines. Basis elements are self-identifying. However, there is no need for a self-identification of different types of data structures, as they are uniformly constructed from self-descriptive components. The invokation of the appropriate standard control routines is completely described by the variable attributes in the self-identifying basis elements and the ordering of descriptor identifiers in variables of data type IDENTIFIER.

The modularity of the internal information structure, as implied by the above implementation of the linear data model, suggests the separate

storage of three basic information components [12], [13]. These are

-- an instruction list IL,
-- a variable descriptor list VDL, and
-- a data list DL.

Hence, the internal information structure is defined by the triple

(IL,VDL,DL)

5.3 Machine Language Instructions

Machine language instructions exclusively reference variable descriptors, i.e., we have the general instruction format (Ψ,VD_3,VD_2,VD_1). Ψ is the operation code and VD_3 is the descriptor identifier of the result variable, whereas VD_2 and VD_1 are the descriptor identifiers of the operand variables. Hence, a data structure architecture processes basis elements in response to single machine instructions. 3-address instructions are a prerequisite for the processing of ordered sets in a streaming mode. The machine language instructions may be grouped into the following categories [9]

-- Scalar Operations
-- Reductions
-- Inner Products
-- Structuring Operations
-- Transfer Operations
-- Queries
-- Jumps
-- Declarations and I/O Operations

The first three groups are value-transforming operations which generate a new variable descriptor and a new data vector for the result variable. Structuring operations create a new structuring of existing data, i.e., they are solely performed on descriptors, not on data. Transfer operations primarily perform parameter transfers 'by reference' and 'by value' to and from subroutines. Queries apply to the basic components of the internal information structure, i.e., to variable descriptors and data vectors. Jumps constitute the program flow control operations.

The self-descriptiveness of stored data structures allows the creation of complete variable descriptors as part of the execution of assignment statements. Hence, variables are dynamically declared at run time. Consequently, the machine language is to a large extent declaration free, except for input operations. Variables which are created by input operations must be declared as to their data type and coordinate dimensions.

Normally, a sequential storage of data is inefficient if such data vectors are to be manipulated dynamically. In data structure architectures, this problem is circumvented by the capability to manipulate variable descriptors through structuring operations. Furthermore, with descriptor referenced allocation, unnecessary copies of data vectors can be avoided by the definition of different basis elements on the same underlying data vector. The mechanization of the conversion of basis elements into data vectors achieves physical data independence. Hence, the reference of self-descriptive variables in machine language instructions is not affected by the representation of data objects in the data vectors. A high degree of logical data independence is achieved by the fact that changes of data definitions through the creation of variables of data type IDENTIFIER do not affect other existing data definitions.

Conclusion

Attempts have been made before to provide hardware support for the generation of data structures. One such example is the SYMBOL machine [14]. However, while the SYMBOL concept provides a mechanism for building structures, it offers no means for processing them. Ultimately, we may only then speak of a certain data structure of a machine if it comprises operators to perform transformations on the structure. Other authors [15,16] have recognized the necessity for data structure architectures but do not present a general solution.

The concept of data structure architectures, as introduced in the paper, represents a novel approach that is radically different from most endeavors as yet so typical in computer architecture. The typical approach has been to multiply certain hardware resources (e.g., processors, memories, etc.) and arrange these modules into organizational structures which reflect certain task patterns. Contrastingly, our approach is to start from a general requirement of computing, the ability to create and process data structures, and develop a standardized logical model. It is shown in the paper that this is generally feasible, and the resulting information structure is described. Its modularity implies a high degree of orthogonalization of the hardware, thus lending itself in a natural way toward parallel processing. In our opinion, the concept of data structure architectures presents a genuine alternative to the von Neumann concept in the realm of general-purpose computing.

References

[1] A. W. Burks, H. H. Goldstine, J. Von Neumann, Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, (Part I, vol. 1), report for the U.S. Army Ordnance Department, 1946, in A. H. Taub (ed.), Collected Works of John von Neumann, Vol. 5, The MacMillan Company, New York, (1963), pp. 34-79.

[2] P. Wegner, Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, London, (1971).

[3] D. E. Knuth, The Art of Computer Programming, Vol. 1, Chapter 2, Addison Wesley, 3809, Second Edition, (1975).

[4] W. K. Giloi and H. Berg, A Uniform, Algebraic Description of Data Structures, Computer Science Dept., Univ. of Minnesota, Tech. Report 76-15.

[5] J. A. Feldman and P. D. Rovner, "An ALGOL-Based Associative Language," CACM 12,8 (August, 1969), 439-449.

[6] CODASYL Data Base Task Group, April 1971 Report, ACM, New York, (1971).

[7] D. S. Tsichritzis and F. H. Lochovsky, "Hierarchical Data Base Management," Computing Surveys, Vol. 8, No. 1, (March, 1976), pp. 105-123.

[8] W. K. Giloi, "BEYOND APL - An Interactive Language for the Eighties," Proc. ICS 77, North Holland Publ., (1977).

[9] H. Berg, A Computer Architecture Based on Ordered Sets as Primitive Data Entities, Ph.D. Thesis, Computer Science Dept., Univ. of Minnesota, (1977).

[10] J. K. Illiffe, Basic Machine Principles, American Elsevier Publishing Co., New York, (1968).

[11] E. A. Feustel, "On the Advantages of Tagged Architecture," IEEE Trans. on Computers, Vol. C-22, No. 7, (July, 1973), pp. 644-656.

[12] W. K. Giloi and H. Berg, "STARLET - A Computer Based on Ordered Sets as Primitive Data Types," Proc. 2nd Annual Symposium on Computer Architecture, Houston 1975, pp. 201-206.

[13] W. K. Giloi and H. Berg, "STARLET - A Contribution to the Computer Architecture of the Post von Neumann Era," Computer Science Dept., Univ. of Minnesota, Tech. Report 75-21.

[14] W. R. Smith, et al., "SYMBOL - A Large Experimantal System Exploring Major Hardware Replacement of Software," Proc. AFIPS SJCC 1971, 601-616.

[15] Y. Chu, "Architecture of Hardware Interpreter," Proc. 4th Annual Symposium on Computer Architecture, IEEE Computer Society Catalog No. 77CH1182-5C, 1-9.

[16] K. J. Thurber and P. C. Patton, Data Structures and Computer Architecture, Lexington Books, Lexington, Massachusetts, (1977).

A MULTI-MINICOMPUTER APPROACH TO CONCURRENT COMPUTATION FOR INTERACTIVE ON-LINE SIMULATION OF COMPLEX BIOSYSTEMS[a]

Ivan R. Neilsen, Ted C. Park, and
C. Duane Zimmerman
Department of Biomathematics, School of Medicine
Loma Linda University
Loma Linda, CA, 92354

It is the main purpose of this paper to describe our experience in designing and implementing an all-digital simulation system with the problems partitioned to run on a tightly-coupled complex of arithmetic processors. These arithmetic processor modules are, in fact, modern high-speed minicomputers. More particularly, we describe a new modular computing resource currently being developed specifically to meet the needs of the biomedical modeler. A computer system well suited to the needs of this environment may be equally appropriate for use in the simulation of other complex systems and the approach taken in designing a simulation resource for biomedicine is described for the general interest of the computer science and engineering community.

The presentation follows in two principal parts. The first part is a discussion of the rationale for the development of a new multicomputer simulation system with a consideration of alternative approaches and associated trade-offs. This is then followed by a description of the overall system architecture and of the hardware and software that have been assembled and integrated into the now operational MMCS (MultiMiniComputer System). In addition it seems appropriate to consider some of the factors, economic and technological, that make such a system especially attractive at this time.

It seemed clear to us that the machines typically used to support common modeling languages were less than ideal for this modeling task and that a multicomputer system could be devised that would be a much better match to the requirements of the modeling process. In particular in order to provide the compute power needed to work with complex models it seemed highly reasonable to provide parallel computing to better match the parallel nature of the systems being simulated. The system, as initially conceived, would be made up of a number of modern high-speed minicomputers operating concurrently. It was anticipated that such a multicomputer system could retain many desirable features and capabilities typically found in other modeling or simulation systems at a much improved cost-effectiveness level while providing a number of other significant advantages.

[a]This work was supported in part by a Biotechnology Resource grant RR 00276 from the Division of Research Resources of the National Institutes of Health.

The principal hardware components making up the currently operating MMCS are as follows:

1 Mapped Eclipse S/200 with 192K bytes of memory and hardware floating point
3 Eclipse S/200 with 64K bytes of memory and hardware floating point
1 Floating-point array processor (AP120B)
1 Mapped Nova 3/12 with 128K bytes of memory
2 80 megabyte disk drive with controller
5 MCA (Multi-Communications Adaptor, allows memory to memory data transfer for all machines)

The development of system software for a multicomputer system can be an enormous task involving many man-years of effort. Our initial approach was to use Data General Corporation's ARDOS operating system. To run programs on the satellite computers the load (link-edit) processes for ARDOS was modified so that a small psuedo-operating-system (approximately 400 bytes) is inserted into each load module (core image file). This change has the far-reaching effect of allowing a load-module produced by any of the language processors to be executed on any of the computers whether or not an operating system is present.

The user software available to accomplish concurrency consists of a few primitives which may be called as subroutines from the various language processors. The primitives allow such functions as sending to or receiving from any other processor, reading or writing common memory, and testing or setting common flags. All concurrency is controlled directly by the programmer.

The great generality of our hardware and software configuration allows not only the traditional forms of concurrent processing but promotes the use of pipelining techniques as well. Our experience thus far shows pipelineing to be a much more widely applicable and useful technique than we had previously anticipated.

Most of the biosystem simulations we have undertaken have been written in FORTRAN. In the interest of freeing the modeler from some of the coding tedium and numerical analysis aspects of working at this level there is a role for high-level simulation languages. The first major simulation language implemented on MMCS is DAREP (developed at the University of Arizona). DAREP is a lanaguage for describing systems of first order differential equations. The package includes hardcopy and CRT graphic capabilities.

ARRAY TYPE VARIABLE TOPOLOGY MULTICOMPUTER SYSTEMS*

Yakup PAKER
Universite de Rennes
UER Mathematique et
Informatique
35031 Rennes Cedex,France

On leave from:
Polytechnic of Central London

Muslim BOZYIGIT
Polytechnic of Central London
115 New Cavendish Street
London WL, England

Summary

A system architectural concept called Variable Topology Multicomputer (VTM) is proposed to implement large networks of low cost computers linked with serial communication paths which can be reconfigured according to the needs of each computation [1]. VTM consists of N computer pairs called nodes interconnected with duplex lines. Each node contains a local computer, a communications computer and an inter-computer message handler. The local computer executes user programmes whereas the communications computer is totally dedicated for message handling between the nodes. The inter-computer message handler contains the input and output terminations so as to enable links to be established with other nodes.

VTM utilizes a synchronous communication scheme where message carrying packets are transmitted during each fixed transmit time T_t repeated every main period time T_m , common throughout the system so that all nodes send and receive messages at the same time. Transmission efficiency is defined as $\rho_t = T_t/T_m$.

In a VTM system topology can be varied in two levels: physical and logical. By connecting wires between various nodes a desired physical network topology can be obtained. Over a given physical network, it is possible to establish logical connections between nodes with no direct link between them, by means of one or more intermediate nodes, using a packet switched or cicuit switched scheme.

Organizing the VTM nodes as a two dimensional mesh yields an array structure. Such a configuration has interest because of suitibility in many important fields of applications. A simulation model of the VTM system has been developed for performance evaluation [2]. Extensive studies have been carried out on an 8x8 VTM array structure. Boundary nodes have been connected so as to obtain a closed toroid. The routing matrix is computed by using a modified Floyd's algorithm for even load distribution [3]. The characteristics of four typical topologies that have been tried are listed in Table 1. The hexagonal and cubic topology are also included for comparison.

(*) This work is supported in part by an US Army, European Research Office, research grant (No. DAJA 37-36-0401).

Performance measures of message delay time, total system throughput, and buffer lengths are simulated under various topology conditions to study the influence of say additional lines on message delay times. The transmission efficiency is a measure of channel capacity in the system. Its increase provides more slots for message transmission. This, however, reduces the period during which the local processing takes place and hence requests for transmission. For large ρ_t values the average delay time nears the average path length times T_m. For smaller ρ_t delays due to queueing start to accumulate. For very small ρ_t values congestion starts building up. Throughput depends very little on topology for large ρ_t values and goes through a maximum as ρ_t is decreased. For small values of ρ_t the effect of topology is clearly seen. Determining the maximum value of ρ_t is called "tuning" where the message generation rate is best matched with the message transmission capacity. The simulation results have indicated that the toroidal organization of 8x8 mesh with alternate diagonal connections has interesting properties to make it a powerful candidate for a general purpose multicomputer structure.

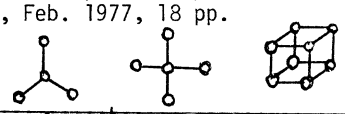
References

1 Y. Paker, M. Bozyigit,"Variable Topology Multicomputer Systems" ,Euromicro 1976, North-Holland, pp.141-151.

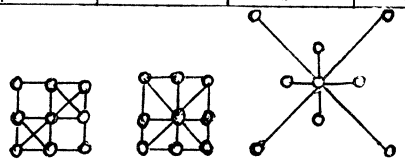
2 Y Paker, Variable Topology Multicomputer, Final report, US Army, European Research Office, Grant DA-ERO-124-74-60079,Nov. 1975, 98 pp.

3 M. Bozyigit, Routing Problem in Uniform Large Networks, Internal Report, Polytechnic of Central London, Feb. 1977, 18 pp.

Table 1



Connectivity	3	4	6
No. of links	96	128	192
Av. path length	4.29	4.06	3.04
Longest path	7	8	6



6	8	8
192	256	256
2.82	2.73	2.31
4	4	3

VIRTUAL INSTRUCTION SETS IN AN MIMD MICROCOMPUTER NETWORK

Melvin M. Cutler
Computer Systems Laboratory
Hughes Aircraft Company
Culver City, CA 90230

Summary

While technology advances have greatly reduced the cost of simple computing devices, it is not clear that a network of such devices operating in parallel provides a cost-effective solution for complex tasks. An ongoing research activity to define and evaluate microcomputer architectures for effective network implementation has characterized the generic features of state-of-the-art microcomputers, identified those features which impair network implementation, and proposed improvements [1]. A result of this effort, the implementation of virtual instruction sets within physical clusters of microprogrammed microcomputers, is summarized here.

The contemporary computer-on-a-chip is too limited and slow for effective networking. Thus, an assumption of the research is that a high-speed MIMD (Multiple-Instruction, Multiple-Data) network is implemented by microprogrammed microcomputers using bit-slice CPUs. Two generic features of such microcomputers serve as the impetus for our design: narrow (typically 16 bits) instruction words and a CPU minor cycle which is two or three times as long as the microprogram memory cycle itself. A 16-bit format places a premium on operation code field width; thus, microcomputer instruction sets are small and general-purpose. A fast microprogram memory means that it is under-utilized by a single CPU. The solution we propose is to share a microprogram memory among a number of CPUs. This particular approach offers three advantages:

- Execution speed is not affected
- Arbitration logic is not needed
- Hardware savings are converted to software and reliability savings

The first two advantages are achieved by a "barrel switch" which allocates one microprogram memory access to each CPU during each of the CPU's minor cycles. Thus, if the CPUs operate with their minor cycles "out of phase" from one another by one microprogram memory cycle, there is no change in execution speed. Regularity of microprogram accesses insures that no conflicts occur between CPUs, and that no arbitration hardware is required. Added cost is the access switching mechanism and the faster basic clock, which now runs at the rate of microprogram memory cycles rather than the rate of CPU cycles. The third advantage is a result of using the net hardware savings to expand the number and capability of (macro-level) instructions implemented in the shared microprogram memory. Thus, each cluster of microprocessors will have access to a large and powerful "real" instruction set. This set might be indexed using an 8-bit "real" operation code, while the "virtual" operation code would be the microcomputer operation code, which might be 6 bits wide.

For any task, the applications programmer selects a 64-instruction subset of the 256-instruction set, either on an individual instruction basis, instruction group basis, or functional instruction set basis. During execution, when this task is assigned to a microcomputer, the executive constructs the mapping from virtual instruction code to real instruction code for the particular microcomputer; further information on executive implementation and protection can be found in [1]. Each shared microprogram memory implements this mapping via a table addressed by a field which consists of a CPU ID code followed by the virtual opcode. The contents of this table is the 8-bit real operation code; the table look-up is done once per instruction. The microprogram memory contains one address register for each CPU in its cluster; while this system is less modular, its addressing is completely in the "real" space except for instruction sequencing.

It is hoped that the above brief description of the implementation of virtual instruction sets is sufficient to convince the reader that this particular approach is appropriate to low-cost microcomputer networks. A popular approach, that of using writable control store, is far more costly because it requires the addition of low-density read-write microprogram memory (for each CPU) as well as data paths and control for reading into them. An alternative approach, implemented in the Burroughs B1700/B1800 [2], is an intriguing and low-cost implementation of virtual instruction sets via interpretation. However, for a microcomputer network with limited memory, the B1700's use of distinct interpreters for each (perhaps only slightly) different task would be wasteful of program memory space, and requires extensive sharing of common program memory (for interpreters). In summary, the proposed design is uniquely suited to providing a network of microcomputers with powerful instruction set capability and with flexibility for degraded mode operation at virtually (sic) no additional cost.

References

[1] M. M. Cutler, An Improved Microcomputer Architecture for Network Implementation, Hughes Aircraft Co., IDC 7531.30/1349, (March, 1977).

[2] W. T. Wilner, "Design of the Burroughs B1700," Proc. FJCC (1972), pp. 489-497.

EXPRESSION OF PARALLELISM AND COMMUNICATION IN DISTRIBUTED NETWORK PROCESSING

NG.X. DANG & G. SERGEANT
ENSIMAG - Institut National Polytechnique
B.P. 53 - 3804I Grenoble Cedex, FRANCE

This work was supported by the French D.R. M.E. under contract.

Summary

In distributed network processing (I) the control and the functions of a distributed application are performed by many geographically dispersed sites. To define and to implemente such an application, one needs the existence of a Logical Network Machine and its operating language which take in the network the same part as a basic software of a given general computer. This machine, named SIGOR, would supply the users with a set of tools necessary to facilitate the definition and the implementation of distributed applications in an heterogeneous environment. These tools are represented by a transportable and interpreted language (2) which is able to run on all the machines of the network. This language defines the set of objects and basic functions linked to the design of distributed applications (3) : transport of algorithms (remote process initialisation, control of the algorithm's trans port, control of the distributed execution), expression of parallelism (by using a variable of mode event and the following instructions : wait, post, multiple wait of n events among p, check), communication between processes (implicit communication of information, explicit transfer of information). The Logical Network Machine SIGOR is realized on a multiprogramming support which conforms to the basic principles of a teleprocessing system (4).

The operating language of SIGOR is a procedural type language (3). The procedure is the basic unit used for transport. Except in the case of explicit transfer of information and explicit synchronization, all the functions as communication and expression of parallelism between processes, which interprete user's procedure algorithms, are done implicitly; tree of hierarchical processes, inter-process protocols, finite states automata, queue to stack requests are defined in order to perform these functions.

It is hoped that this summary will served as a gateway to increase appreciation of the Logical Network Machine SIGOR and to its probable descendant : a high level network command language allowing users to define distributed algorithms in a network environment.

References

(I) Andy Van Dam, Transcripts of the two Distributed Processing Workshop,(Aug. I976 and Aug. I977), Brown University, Providence, R.I., 02912

(2) M.N. Farza, G. Sergeant, Machine Interpretative pour la mise en oeuvre d'un langage de commande sur le reseau CYCLADES, These de Doctorat de 3e cycle, Universite de Toulouse, (I974).

(3) Ng.X.Dang, Systeme et Langage Portable pour le traitement des applications reparties, These de Doctorat de 3e cycle, USMG, INPG, Grenoble, (1977).

(4) Ng.X. Dang, V. Quint, J. Seguin, G. Sergeant, Presentation et Definition de SYNCOP, un sous-systeme de commutation de processus pour la tele-informatique et les reseaux d'ordinateurs, ENSIMAG, Rapport de Recherche no 64, (I977), 54 pp.

(5) M. Elie, H. Zimmermann, Transport Protocol. Standard end-to-end protocol for heterogeneous computer networks, IFIP WG6.I, INWG 6I, (May I975), 33 pp.

(6) N. Wirth, Modula : a language for modular multiprogramming, Software - Practice and Experience, (I977).

(7) Brinch Hansen, Concurrent Pascal Report, Californie Institute of Technology, (June I975).

(8) Hoare, C.A.R., Monitors : an operating system structuring concept, Communications ACM I7, I0, 549-557, (Oct. I974).

ON THE CONSTRUCTION OF MICROPROCESSOR-ORIENTED OPERATING SYSTEMS†

Martin Freeman††
Walter W. Jacobs
Department of Mathematics, Statistics and Computer Science
The American University
Washington, D.C. 20016
and
Leon S. Levy†††
Department of Computer and Information Sciences
University of Pennsylvania
Philadelphia, Pennsylvania 19174

Summary

Microprocessors and semiconductor memories are becoming faster and cheaper. As this situation progresses, the constraint imposed on the number of pins available on these components will force us to consider more carefully the functionality of such components and their interconnection.

In this regard, two approaches immediately suggest themselves as design philosophies for constructing microprocessor systems: (1) provide a general interconnection network among microprocessors where data paths, control paths and communication protocols are already specified, and try to map a (software) solution onto such a system; or (2) start from the general system functional specifications (e.g. system requirements) and refine them into a logical design which provides a basis, in an implementation phase, for determining the (hardware/software) functionality of specific microprocessors and a suitable interconnection structure.

In this paper we take the latter approach and describe a model (i.e. a conceptual framework) which forms the basis for the design and implementation of microprocessor systems.

References

[1] B.W. Arden, and A.D. Berenbaum, "A Multi-Microprocessor Computer System Architecture," Proc. Fifth Symp. Operating Syst. Principles, (Nov., 1975), pp. 114-121.

[2] R.S. Barton, "Ideas for Computer Systems Organization: A Personal Survey," Software Engineering, Academic Press, (1970).

[3] R.B. Bunt, and J.N. Hume, "Self-Regulating Operating Systems," Canadian J. Operational Res. and Inform. Processing, (June, 1972), pp. 232-239.

[4] Y. Chu, High-Level Language Computer Architecture, Academic Press, (1975).

[5] P.J. Denning, "Fault-Tolerant Operating Systems," ACM Comput. Surveys, (Dec.,1976), pp. 359-391.

[6] E.W. Dijkstra, "The Structure of the T.H.E. Multiprogramming System," CACM, (May, 1968), pp. 341-346.

[7] M.J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Trans. Comput., (Sept., 1972), pp. 948-960.

[8] C.C. Foster, "An Unclever Time-Sharing System," ACM Comput. Surveys, (Mar., 1971), pp. 23-48.

[9] M. Freeman, W.W. Jacobs, and L.S. Levy, A Model for the Construction of Operating Systems, Dept. Math. Stats. and Comput. Sci., The American Univ., TR-100 (draft), (Aug., 1977), 25 pp.

[10] M. Freeman, W.W. Jacobs, and L.S. Levy, "On the Construction of Interactive Systems," (in preparation), 22 pp.

[11] G. Goos, "Some Basic Principles in Structuring Operating Systems," Operating Systems Techniques, Academic Press, (1972).

[12] P.B. Hansen, Operating System Principles, Prentice-Hall, (1973).

[13] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," CACM, (Oct., 1974), pp. 549-557.

[14] J.J. Horning, and B. Randell, "Process Structuring," ACM Comput. Surveys, (Mar., 1973), pp. 5-30.

[15] W.W. Jacobs, "A Structure for Systems that Plan Abstractly," AFIPS Conf. Proc., (1971) pp. 357-364.

[16] W.W. Jacobs, "Control System in Robots," Proc. ACM 25th Anniv. Conf., (1972), pp. 110-117.

[17] W.W. Jacobs, "How a Bug's Mind Works," Cybernetics, Artificial Intelligence, and Ecology, Spartan Press, (1972).

[18] G.J. Lipovski, and J.A. Anderson, "A Virtual Memory for Microprocessors," Proc. 2nd Ann. Symp. Comput. Arch., (Jan., 1975), pp. 80-84.

[19] B. Liskov, "The Design of the Venus Operating System," CACM, (Mar., 1972), pp. 144-156.

[20] B. Liskov, and S. Zilles, "Specification Techniques for Data Abstractions," IEEE Trans. Software Eng., (Mar., 1975), pp. 7-19.

† This work was funded by NSF Grant MCS 76-07682.

†† Present address: Digital Systems Laboratory, Stanford University, Stanford, California 94305.

††† On leave from the University of Delaware, Newark, Delaware 19711.

A PIPELINED DYNAMO COMPILER*

Wing Huen, Ossama El-Dessouki, Eugene Huske, Martha Evens
Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois 60616

ABSTRACT - The design of a pipelined DYNAMO compiler which produces parallel code segments for a network computer is described. The network computer is dedicated to execution of a single job at a time. Phases of the compilation process, residing on separate computers in the network, cooperate to process an input source stream in a pipelined style but are constrained not to access global tables or intermediate files. The object code is partitioned automatically into clusters by the compiler and the clusters are allocated to constituent computers for run time execution. Problems raised by the constraints are discussed and design alternatives to these problems are examined.

Introduction

A pipelined DYNAMO compiler which produces parallel code segments has been designed for the TECHNEC, a network computer at Illinois Institute of Technology. The TECHNEC will be a ring network of twelve LSI-11s. It is called a network computer rather than a computer network because the whole network will be dedicated to the execution of a single job at a time.

The design aims to make full use of the parallelism provided by the network computer. At compile time, the compiler itself is organized in the form of a pipeline. Stages of the pipeline execute in parallel and cooperate by passing statements in a conveyor belt style. But the communication is asynchronous between stages of the pipeline. The generated object code is partitioned automatically by the compiler into clusters which are to be executed in parallel at run time on the network computer.

This paper is concerned with the problems we have encountered and the alternative solutions to the problems. The deciding factors in solution selection are the efficiency of the solution and the degree of parallelism exploited.

Goals of the DYNAMO Project:

a. Compilation on a Network of Microcomputers

This project aims to investigate the problems inherent in implementing a high-level language compiler which is distributed on a network of microcomputers. The compilation process is treated as a single task and partitioned into cooperating subprocesses on the network. Each microcomputer is relatively slow compared with larger computers and the primary memory on each microcomputer is restricted in size. But a network of microcomputers as a whole serves as a powerful computing device by exploiting parallelism on the network.

Microcomputers have been predominantly used to control real-time processes and languages available on microcomputers are usually assembly languages. This attempt to implement a compiler distributed on a network represents an exploration of new applications of microcomputer networks.

b. Pipelined Compilers

The compiler will be in the form of a pipeline each stage of which carries out an individual phase of compilation.

Each computer on a network has a local primary memory but does not share any common global memory with another computer. A computer thus can only communicate with other computers in the network by means of data messages. The TECHNEC on which the compiler is to be implemented is in the form of a unidirectional ring in which any computer may communicate with another by circulating a message around the ring. Each phase of the compiler receives a statement in the form of a message, converts it to some internal form and passes the converted statement to the next stage as a message.

c. Partitioning a Distributed Program

Parallelism is to be exploited by executing the compiled object code in parallel on the computers of the network. The generated code is partitioned automatically by the compiler into code segments called clusters. This partitioning involves tradeoffs between

* This work was supported by National Science Foundation under grant MCS 76-01310.

speedup due to parallel execution of clusters and the amount of message passing necessitated by communication between dependent clusters.

d. Synchronization Between Clusters

The compiled object code of the DYNAMO compiler will be in the form of program clusters which communicate by data messages. A communication mechanism has to be provided between the clusters. The style of synchronization for the clusters is an important problem.

Choice of Language

Simulation of parallel processes is a basic concern of the Network Research Group at Illinois Institute of Technology because we view simulation as a fundamental part of the future development of networks. While simulation is often carried out on single processors there are obvious conceptual advantages in simulating parallel processes on a network of parallel processors. Clearly this kind of simulation is a most natural and appropriate task for a network computer.

The decision to focus on continuous rather than discrete simulation was motivated by the concern of the Network Research Group with control processes. This group aims at investigating a style of control developed in [3] in which complex tasks requiring accurate coordination of many variables are performed by distributed controllers, each handling a stage of rough computation. The TECHNEC system [4] provides a hardware/software environment for experimenting with this style of control. A control process is to be programmed as a collection of control tasks each responsible for controlling a subset of the variables. The whole TECHNEC is to be dedicated for the execution of a single control process at a time. These control processes will be studied with the help of simulation models. This design leads to the implementation of a continuous simulation language. An appropriate language tool must lend itself easily to problem decomposition.

DYNAMO [2,5] is a well-known continuous simulation language. While DYNAMO presents serious complications in the areas of sequencing and partitioning, it is easy to parse and its only data structures are simple variables and 1-dimension arrays. Thus implementation of DYNAMO seemed to be a feasible step in the development of network software.

A continuous simulation model is often represented by a set of differential equations. DYNAMO [5] models a system with a set of variables called LEVELs and their rates of change called RATEs. The differential equations are solved by determining the value of each LEVEL at regular time points and the corresponding RATE in an interval between adjacent time points. The value of a LEVEL at a simulation time point is expressed as an integration of the corresponding rate over a regular time interval. A very simple integration scheme (the Euler or rectangular method) is used. The scheme is very efficient when no need for great accuracy exists. There are also AUXILIARY variables to help specify the relationship between variables especially in nonconservative systems.

DYNAMO differs from procedural languages in that statements of a DYNAMO program are not sequential, i.e., they can be written in any order without affecting the outcome of the program. No goto nor conditional statements are provided. In a traditional implementation for single processor systems, there is an implicit order of executing LEVEL variables as a group first, followed by AUXILIARYs and finally RATEs in one cycle of simulation. There is still considerable freedom available in varying the order of execution of LEVELs since they are independent of one another. Similarly all RATEs are independent of one another. This independence among LEVELs and RATEs gives rise to opportunities in parallel processing. In a network computer such as the TECHNEC on which a single DYNAMO program is distributed, much more parallelism can be exploited. An AUXILIARY equation can be executed in parallel with a LEVEL or a RATE equation allocated to a different processor as long as they are independent.

The state of a model is computed at regular time points. The length of the constant interval is designated by the symbol DT. The size of DT is chosen by the user. DYNAMO adopts the convention of attaching one of the symbols J, K, JK, or KL as subscripts to a variable to indicate the timing. The value of level ABC at the instant at which calculations are being made is referred to as ABC.K. Its value at the previous instant is ABC.J. The interval just passed is called the JK interval; the interval coming up is the KL interval. Since RATEs hold over an interval, their subscripts are either JK or KL while other variables have J or K as subscripts. It is not necessary to attach a subscript to constants.

Most DYNAMO statements are assignment statements whose right hand sides

(RHS) are arithmetic expressions. We will use the term 'equation' interchangeably with 'statement' since the assignment statement defines the value of the variable on the left hand side (LHS) at a particular time point. The type of the variable on the LHS is indicated by the first character in the statement. Thus in

```
L     ABC.K=ABC.J+DT*R.JK
```

ABC is defined as a LEVEL. There are seven equation types: level (L), auxiliary (A), rate (R), supplementary (S), initial value (N), given constant (C) and table (T). Each variable is defined exactly once with at most one corresponding initial value (N) equation.

To summarize DYNAMO has been adopted as a research vehicle for several reasons. First of all, simulation of parallel processes is a central problem in network development and a most important application for network computers. DYNAMO raises the central problem of partitioning tasks in an urgent and immediate fashion. Second, the Network Research Group needs a continuous simulation language to model control processes. Third, the simplicity of the syntax and data structures of DYNAMO make it a good starting point for compiler development on networks.

TECHNEC System Overview

We shall present a brief introduction to the hardware configuration and software facilities available on the TECHNEC. The emphasis is on the interface between the available software facilities and the DYNAMO Compiler.

Hardware Configuration

The TECHNEC [3] is a ring network of five nodes initially (Figure 1) with 12 nodes planned in the second year of the project. Each node consists of a COSMAC (called the Ring Interface Unit - RIU) and an LSI-11 (called the Micro Processor Unit - MPU). COSMACs are linked together by I/O ports to form a ring. Each MPU is attached to a corresponding RIU. All user tasks reside in MPUs. The RIUs are responsible for message communication among the nodes.

Each MPU is a 16-bit LSI-11 with at least 12K words of RAM and floating point hardware. One of the MPUs has an RX-11 dual floppy disk and serial I/O interface. This node will be designated as the system node. A system console is attached to this node. The network will be connected to other computers on campus via modems.

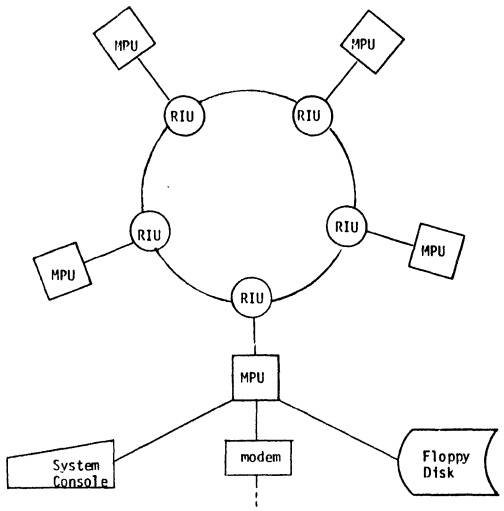


Figure 1. Structure of TECHNEC

The RIU is an 8-bit microprocessor with 1K bytes of RAM and three sets of I/O ports. One set of ports implements the message communication path between adjacent RIUs. The message communication between RIUs is byte parallel and unidirectional. The other two sets of ports are used for communication with its corresponding MPU. One set implements a control/status and data buffer register interface and the other set serves as a DMA (Direct Memory Access) interface between the RIU and the memory of the MPU. An RIU may interrupt its MPU (but not the other way around) and it can access the 12K RAM of its MPU in the DMA mode.

Software Facilities

The operating system includes a multitasking executive called SEXTECH which allows multiple tasks to reside in one MPU and schedules user tasks in a simple round robin fashion.

The TECHNEC supports two modes of message communication between tasks. One is the broadcasting mode in which one task passes a message around the ring via a 'channel' and all tasks which are opened to receive messages at this specific channel may receive the message. The channels are virtual because no physical links are established between the tasks. A channel is simply an identifier tagged to each message. This is a one-to-many communication mode. The identity and the location of the receivers are not known to the sender. The other mode is point-to-point transmission in which a task transmits a message to exactly one receiver via a channel and the receiver

is identified by a 'subchannel.' The location of the receiver need not be known to the sender.

A collection of facilities such as the console management routine, file management, and debugging facilities are available on the system node. The console management routine allows the system console to interact with user tasks via messages. The file management routine provides storage and retrieval of files resident on one of the floppy disks. The debugging routine provides functions such as suspension of a task, resumption of a task, modifying contents of a location, display of status, and breakpoints. A loading routine exists to load programs from the floppy disk or other external computers to the TECHNEC.

Structure of the Compiler

The compiler is structured in the form of a pipeline with the various phases of the compilation process distributed over the ring network. Each phase resides as a module on a separate computer of TECHNEC. A module receives one statement in the form of a message at a time from the previous module, performs one compilation phase on the statement, and passes the statement to the next module. Statements of the source program, originating from the system node, thus pass through the phases in order, with no feedback required. So one may consider the compiler to be pipelined in the same sense as pipelined arithmetic units. The code generated will return to a file at the system node. The system node behaves both as a source and a sink for the pipeline.

Several severe constraints are imposed on the design of the compiler. First, no intermediate files exist between the phases. Each phase can be considered to be processing a statement in the statement stream through a window. Once a statement is processed and passed to the next phase, neither the original nor the modified form of the statement will be available to the phase. Secondly, the individual phases cannot access global tables. Ideally information derived by each phase should be embedded in the internal code which is routed to successive phases. Thirdly, the memory available to each phase is limited.

The first two constraints are not due to theoretical or physical limitations but are based on performance considerations. A file or information tables at any node could be made accessible to any process on TECHNEC via the interprocess communication mechanisms. It is felt that message communication involves too much overhead in parallel computation on TECHNEC. No feedback is allowed in the compiler out of a desire to keep the pipeline as full as possible and to reduce message communication. The compiler as a whole is a one pass compiler without the benefit of global tables. Moreover it is distributed on multiple computers. The compiler is composed of eight modules organized as in Figure 2:

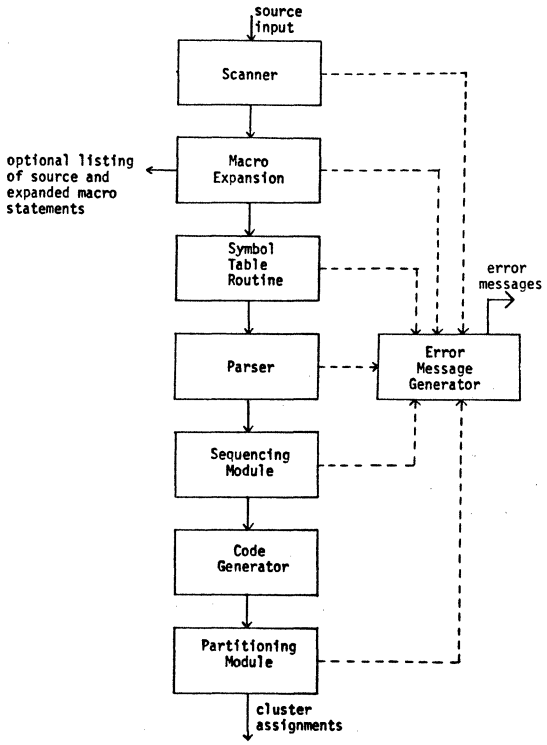


Figure 2. Structure of the Pipelined DYNAMO Compiler.

Error reports, originating from one of the first seven modules, bypass intermediate modules to reach the Error Message Generator which produces symbolic error messages.

Figure 2 represents the structure of the compiler in our current design. Initially we placed the Code Generator after the Partitioning Module. We realized, however, that if the Code Generator preceded the Partitioning Module, the latter would have more accurate estimation of the execution time and storage requirements of statements in forming clusters.

A statement is passed between modules in an internal form of a string of tokens. A token has two fields: type and value. The value field indicates the

symbol represented by the token (e.g., identifier, subscript, statement, etc.). The interpretation of the value field is dependent on the type field. For example the value field of a subscript token indicates the subscript type (J, K, JK or KL). The value field of a statement token denotes the statement type (level, auxiliary, rate, supplementary, constant, initialization, table, etc.). The value field of an identifier or a real number points to the original symbolic representation in a character string which follows the string of tokens. Organization of the string and information in the token vary from phase to phase.

Scanner

The input to the Scanner is the DYNAMO source program. The function performed by the scanner is to transform the text input into an internal form of tokens.

Scanning a DYNAMO statement is a simple matter. A routine is first called to scan the statement identifier (L, A, R, SPEC, etc.). All statements except PRINT, PLOT, NOTE, RUN and title statements are scanned by the same routine. There are only four kinds of symbols that need to be dealt with: quantity names, subscripts, numeric constants and delimiters/operators. A token is created for each symbol and stored in the output message. When the statement is completely scanned, the message is sent to the Macro Expansion program.

Macro Expansion

The macro expansion module expands macro calls into one or more DYNAMO statements, produces the source listing (optionally listing expanded statements), and assigns to each statement a unique number.

The language requires that a macro definition appear before a call to it is made. This is important for a one pass compiler. The tokens for statements within a macro definition are stored in a table as the statements are received from the Scanner. Special tokens for occurrences of local variables, formal parameters, and macro names replace the normal identifier tokens to speed up macro expansion. The macro and a pointer to the macro definition are stored in a second table.

At expansion time, the macro call is replaced by a compiler-generated identifier. Each statement in the macro definition is processed by replacing local variables and occurrences of the macro name by compiler generated identifiers and replacing formal parameters by the actual parameters. As each statement in the macro body is processed, tokens are transferred to a message to be sent to the symbol table program. Macro calls are allowed in macro definitions and in actual parameters. These nested macro calls get expanded in the same way.

The source listing is produced by this module because it is felt that listing of expanded statements should be a user option and it is desirable to perform the source listing as early as possible to reduce message passing overhead.

The number assigned to a statement is printed on the listing and stored in the internal form statement for the purpose of associating an error message (both compile time and run time) to a particular statement.

Symbol Table Manipulation Routine

The Symbol Table Manipulation Routine is the third module in the pipeline. The function performed by this module is to replace the value field of quantity name identifier tokens by an index into the symbol table. It also checks subscripts in a given statement type, checks for multiple definitions of a quantity name, checks for undefined quantity names, and searches for conflicting use of subscripts.

Converting value fields of an identifier token to a symbol table index is straightforward. When a message is received from the Macro Expansion Module, each identifier is looked up in the symbol table. If the identifier is not found in the table, a new entry is made in the symbol table and the value field set to the table index.

More work is required to achieve subscript checking. The nonsequential nature of DYNAMO gives rise to the problem of verifying a subscript appearing on the RHS of an equation. It should be emphasized that the pipeline does not permit a second pass through the source. One solution is to keep in the symbol table entry for each quantity name a bit to indicate whether or not the identifier is defined. If defined, a field indicates the equation type (L, A, R, S, N, C, CP, T or TP) in which the quantity name is defined; in this case the subscript is immediately verified. If the quantity name is not yet defined, the equation type field is a pointer to a linked list. Each node in the linked list contains a field for the statement number, a field indicating the equation type and a field indicating the subscript used. When the definition of the

quantity name is encountered, the subscripts for the previous references to the quantity name are verified for correctness. The Symbol Table Routine notices inconsistencies in use of subscripts.

Parser

The Parser is the fourth stage in the pipeline. Separate routines parse assignment statements, print and plot statements, and specification statements. The main function performed by this module is to transform expressions from infix to Polish suffix notation. Polish suffix notation was chosen as internal form because the LSI-11 provides stack operations. For this kind of stack machine, code generation from Polish suffix form is particularly simple.

A transition matrix is used by the parser to handle arithmetic expressions. Transition matrix parsing has the advantage of being a particularly robust parsing method. It also facilitates the production of good error messages. The main disadvantage of this parsing method is the space required for the matrix. Fortunately, DYNAMO expressions are so restricted in form that the matrix for this language is of reasonable size. Operator precedence parsing is awkward for DYNAMO because it allows two operators to appear next to each other; A*-B and (A+B)(A-C) are both legitimate.

Sequencing Module

The nonsequential characteristic of DYNAMO by no means implies that DYNAMO statements can be executed in any order. Determination of data dependency, sequencing of statement execution, and initialization of the model are essential tasks of a DYNAMO compiler. Each DYNAMO assignment statement can actually be considered the defining equation of its LHS variable. Moreover, these statements define a partial ordering relation between variables in the program in which the LHS variable is a "successor" of each variable in the RHS. A topological sorting algorithm can be used to produce a linear sequence consistent with the partial ordering relations.

As was mentioned in Section 3, there is an implicit order of execution (or sequencing) of statements during each simulation cycle, that begins with LEVEL equations followed by AUXILIARY equations, and finally RATE equations. This sequencing will be referred to as the "LAR looping sequence." The LEVELs can be computed in any order in the beginning of a cycle, and also the RATEs can be executed in any order at the end of the cycle, [5, p. 25]. The sequencing of AUXILIARY equations must be determined by the compiler.

A DYNAMO compiler must provide the initial values for those auxiliaries and rates that have no explicit (i.e., user-defined) initial value equations (N-equations) by treating the auxiliary or rate equation as an N-equation [5, p. 25]. This actually means that, the first simulation cycle is to produce the initial values for all quantities in the model. This is done by beginning with those quantities that have a user-defined initial value and executing the appropriate statements in the model in the correct order to provide initial values for other quantities. This sequence is referred to as the "initialization sequence." In general the initialization sequence may differ from the LAR looping sequence (in fact they are different in most practical examples). Hence, the Sequencing Module (SM) can be schematically represented by Figure 3. The input is fed to the sequencing module a statement at a time. Data dependency information is extracted from that statement and the statement is passed untouched to the next stage. When a RUN statement is encountered, SM begins processing the accumulated information.

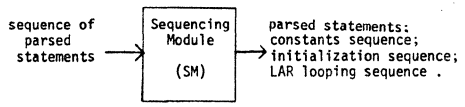


Figure 3. Input and Output of the Sequencing Module

Constant (C) and table (T) equations actually may be evaluated in any order at the very beginning in the initialization sequence. Moreover, constants and tables are the only quantities that may be redefined in case of reruns. It is, therefore, expedient to group constant and table equations as a separate sequence, referred to as "Constants Sequence" in Figure 3.

The Sequencing Module can only determine sequencing after examining the complete program. Again the three constraints mentioned at the beginning of this section come into play. To avoid a second pass and to keep the next stage busy while SM is functioning, each statement written by the programmer will be eventually converted to a subroutine by the Code Generator. The SM will produce a sequence of subroutine calls.

The Sequencing Module can be described functionally by the flowchart

in Figure 4. There are two logical parts in SM. The first part, consisting of modules M1, M2, and M3, builds the data structure and the second part produces the sequences. The data structure used to convey the data dependency information in the SM consists of a set of linked lists. Each linked list, shown in Figure 5, corresponds to a variable in the program and contains the data dependency in both the defining equation of the variable and the user-defined initial value equation, if available. The data dependency information is conveyed in the form of a COUNT field, indicating the number of predecessors to the variable, and a successor list for that variable.

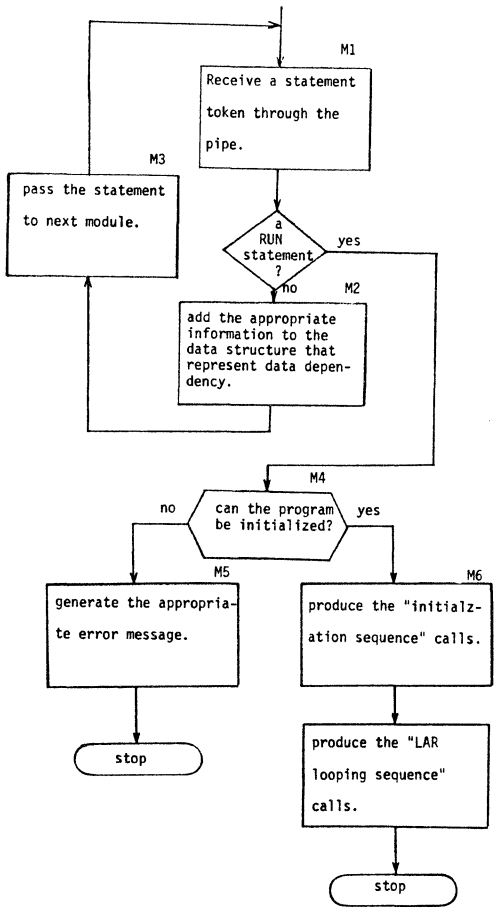
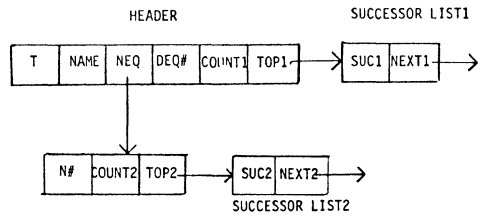


Figure 4. Logical Flow of the Sequencing Module

The SM then checks whether the model can be initialized and simulated properly. The conditions for proper execution are:

a. All LEVELs are initialized using user-defined N-equations. This is checked using the T and NEQ fields.

b. All variables are properly defined. This is indicated by a nonzero DEQ# field.



Fields are interpreted as follows :

T is a type field (indicating the type of the variable)

NAME is the symbol table entry for the LHS variable

NEQ is a pointer to the list representing the corresponding N-equation supplied by the user (0 if none)

DEQ# is the defining equation number of the variable

COUNT1 is the number of predecessors of the variable in DEQ
COUNT2 is the number of predecessors of the variable in NEQ

TOP1 is a pointer to the first member of the SUCCESSOR LIST1
TOP2 is a pointer to the first member of the SUCCESSOR LIST2

NEXT1 is a pointer to the next entry in the SUCCESSOR LIST1
NEXT2 is a pointer to the next entry in the SUCCESSOR LIST2

SUC1 is the symbol table entry for the successor in LIST1
SUC2 is the symbol table entry for the successor in LIST2

N# is the user-supplied N-equation number for the variable

Figure 5. Data structure used in the Sequencing Module

The "LAR looping sequence" calls can be generated directly from the "initialization sequence" calls if no AUXILIARY variable has a user-defined N-equation. In this case, the LAR looping sequence is simply calls to LEVELs in any order, followed by calls to AUXILIARYs in the same order as in the initialization sequence (same equation numbers), followed by calls to RATEs in any order.

On the other hand, if N-equations are supplied for some AUXILIARY equations, the data structure is searched for entries with zero NEQ fields. For each entry, a SUCCESSOR LIST2 is built as required by the language as a copy of SUCCESSOR LIST1 with N# equal to DEQ# and COUNT2 equal to COUNT1 field. This is necessary because the topological sorting program has to be run twice in this case.

In general, the topological sorting program incorporated in the SM searches for a zero COUNT field, produces the call for the corresponding equation number, and decrements by 1 the COUNT field in the header of each variable appearing in the SUCCESSOR LIST. To produce the "initialization sequence" calls, in case a above, this procedure is applied iteratively using COUNT2 and SUCCESSOR LIST2 for those entries with nonzero NEQ fields and COUNT1 and SUCCESSOR LIST1 for the others. In case b, COUNT2 and SUCCESSOR LIST2 are used for all variables. If the model is consistent and an evaluation sequence can be found, the SM produces

calls in the correct order. Otherwise, the number of calls produced does not check with the number of statements processed and a "simultaneous equations" error message is generated from SM that contains those variables for which no calls were produced.

Code Generator

The Code Generator receives from the SM parsed statements. It extracts the variables from a statement and sends them as a list to the PM which will need predecessor-successor relationship among variables. The parsed statement is converted by the Code Generator to a subroutine in assembly language and sent to the code file. PRINT and PLOT statements are handled differently since routine checking and formatting are necessary. The Code Generator finally generates for the PM execution time and storage requirements of each statement.

Partitioning Module (PM)

One of the main objectives of the DYNAMO project is to exploit parallelism by executing the compiled object code in parallel on the computers of the network (goal c). The partitioning module is to receive from CG lists of variables in each statement that is used to build a data structure representing the dependency between variables. After encountering the RUN statement, the module processes the data structure and produces a "processor assignment list" that specifies the statements to be executed on each processor in the network, using the statistics provided for execution time and storage requirements of each statement. This can be represented schematically as in Figure 6. The PM also is supposed to insert the required communication primitives between variables in different partitions. Although the data structure required in PM has many similarities with that of the Sequencing Module, since both of them reflect some sort of connectivity relation, a main difference exists in subscript treatment. In PM the initialization cycle is completely ignored, because it occurs only once. Supplementary equations are also ignored at this point because they are only executed during special print or plot cycles. The main objective is to produce partitions that will reside on different processors at run-time in order to achieve the fastest execution of the program. Hence, it is clear that a relation between the initial value of two variables V1, V2 is not as important as a recurrence relation between them.

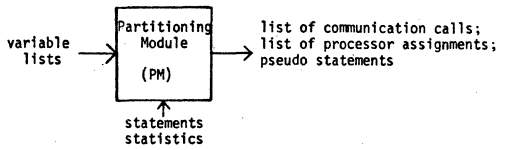


Figure 6. Input and Output of the Partitioning Module.

Consequently, a directed graph, representing two successive simulation cycles of a DYNAMO source program, is built as follows:

(i) All C, N, S, PRINT, PLOT, SPEC, T statements are ignored. An appropriate initializing process will be provided to initiate constant sequence and initialization sequence calculations and send the results to different partitions before the main iteration operation begins.

(ii) A variable with a J or JK subscript is represented by an entry node with no predecessors.

(iii) Every variable V is represented by two nodes N1, N2:

N1 representing V.J (or V.JK)
N2 representing V.K (or V.KL)

As mentioned in (ii) N1 is an entry node.

(iv) An arc from node Ni to node Nj represents a precedence relation, i.e., Ni is a RHS variable of an equation that has Nj in its LHS.

The main algorithm for automatic partitioning is the subject of a study parallel to this project. The main points investigated in automatic partitioning [1] can be stated as follows:

(1) Two basic approaches are being studied.

a. An optimal partitioning approach based on an integer programming model that produces partitions of DYNAMO code that takes minimum time to run on the network computer (taking into consideration the communication overhead).

b. A heuristic approach that investigates different partitioning policies that can be incorporated easily at compile time.

The tradeoff between the two approaches together with comparative studies for different heuristics, is the main theme of that study. The study had reached the stage of completing the formulation of the problem as a Mixed-Integer-Linear-

Programming (MILP) model of a reasonable size. Test runs using sample DYNAMO programs are being attempted using a standard package (FMPS) for producing solutions on a UNIVAC 1108 processor. In addition four heuristics have been suggested and are being tested on the same sample programs. In all these algorithms an important assumption has been made. Namely, every two nodes N1, N2 representing the same variable V at different time points, are grouped together in one computer. This implies that a variable is assigned to one processor during the whole simulation period.

(2) In the MILP model a combination of synchronous and asynchronous modes (see the section on Run Time Synchronization of clusters) is assumed. This assumption does not affect the resulting solution, but mainly influences its optimality.

(3) The MILP model generates not only processor assignment, but also the optimal starting-time-values for each variable. This can be used to generate the LAR looping sequence instead of the scheme described above. The latter produces a feasible but not necessarily optimal sequence.

The Symbol Table, the Sequencing Module, and the Partitioning Module all generate data structures involved with connectivity between statements which are related but not identical. The first two constraints imposed on the design motivate these distributed data structures. Some duplication does occur, but each data structure is tailored specifically for the phase and is thus more efficient. Moreover the phases are executing in parallel. A single central, general data structure can only be accessed sequentially.

Error Message Generator

The error message module is the last logical step in the pipeline. This module differs somewhat from the other modules in that there is more than one input source. Error messages are received from any of the other modules except the code generator module.

Each error message contains an error message number, a line number, and, optionally, one or more identifier tokens and/or text tokens. The error message number is used to retrieve an error message from disk. The line number is printed with the error message to indicate where the error occurred. Variable text, e.g., a subscript name, is simply inserted where required. After the error message is formatted, it is sent to a print program.

Error Recovery and Error Correction

A compiler must be able to discover as many errors as possible before terminating. This implies that a good error recovery scheme must be incorporated in every good compiler.

When it comes to error correction, however, there is serious conflict, particularly in regard to DYNAMO subscript errors. We believe on principle that a compiler should not correct user errors. This is a complicated process which may lead to unpredicted and unreliable results. What is more, user errors may signal defects in the model. We can serve the user better, we think, by flagging errors and forcing him to correct them. On the other hand we also believed in language standardization and we have tried to follow the DYNAMO manual as closely as possible. According to the DYNAMO manual, all subscript errors are considered nonfatal [5 p. 53-54].

Run Time Synchronization of Clusters

A cycle of simulation consists of execution of LEVEL statements, AUXILIARY statements if any, RATE statements and in certain specified cycles SUPPLEMENTARY statements for PRINT and PLOT statements. Due to dependency between statements, values may also be passed between cycles.

The evocation of cycles and statements within a cycle can be performed synchronously or asynchronously. By synchronously we mean statement execution of cycle initialization are evoked by a signal given by a central process. In the asynchronous mode, no timing mechanism exists to control the timing of evocation. Each process evokes its logical successor.

There are a number of ways to synchronize partitions:

Option 1. (Synchronous Mode): Partitions are evoked by a global signal at the beginning of a cycle and evoked to send and receive messages at the end of the cycle. This mode stipulates that no messages be passed between partitions in the same cycle. It implies that values required for the execution of a statement are either available at the beginning of the cycle or generated by the partition itself. Only intercycle data dependency is taken care of. This mode imposes a serious constraint on partitioning.

Option 2. (Synchronous Mode): An evocation signal is provided for each class of statements of the same type and a signal

in between classes for message passing. This approach allows more freedom in partitioning and message passing between L-A, A-R, L-R, L-S, A-S, R-S pairs. The price paid is additional signals and reduction in speed. The execution time of a cycle is the sum of maximum execution times for each class of statements plus the maximum transmission times. Moreover a message cannot be sent once the value of quantity is available but must wait for the synchronization signal.

Option 3. (Asynchronous Mode): In the completely asynchronous mode, each statement is executed once all the required values on its right hand side are available. It is conceivable that one partition may run a number of cycles ahead of another. Data messages may have to be tagged by cycle numbers or a FIFO queue is needed between partitions that communicate with each other.

Option 4. (A Combination of Synchronous and Asynchronous Modes): Cycles are evoked by a global signal. The broadcast message facility is used advantageously for this purpose. Intracycle messages are sent asynchronously. A partition may send a value needed by another using the point-to-point communication scheme. A partition may also pause to wait for a data message. Each partition may inform the signaling mechanism of its readiness to start a new cycle which implies completion of all execution and intercycle data transfers. When all partitions are ready, the signaling device generates a signal for the new cycle. If n partitions exist, then n messages plus one broadcast are necessary. A broadcast message from the signaling mechanism to poll each partition's readiness is a more efficient solution. But a good estimate on the cycle execution time is important to avoid multiple pollings.

The fourth option is favored for run time synchronization.

Summary

We have described the design of a pipelined DYNAMO compiler to be implemented on a network computer. The goals are to make use of parallelism available both at compile time and run time. At compile time the compiler itself is organized in the form of a pipeline. Each stage of the pipeline executes in parallel and communicates asynchronously. The object code is automatically partitioned into clusters by the compiler so that the clusters execute in parallel on the constituent computers of the network computer. The problems raised by the objectives and the constraints of the environment are discussed and alternative solutions to these problems are examined.

References

1. O. El-Dessouki and W. Huen, "Partitioning and Processor Assignment on a Network Computer," Technical Report 77 - 2, Department of Computer Science, Illinois Institute of Technology, Chicago, Ill. 60616.

2. J. Forrester, Industrial Dynamics, MIT Press, Cambridge, Mass., 1961.

3. P. Greene, "Proposed Organization of Hierarchical Multicomputer Control Systems," to appear in International Journal of Man-Machine Systems.

4. W. Huen, P. Greene, R. Hochspring, and O. El-Dessouki, "TECHNEC, A Network Computer for Distributed Task Control," to appear in Proc. of the First Rocky Mountain Symposium on Microcomputers: Systems, Software and Architecture 1977.

5. A. Pugh, III, DYNAMO User's Manual, MIT Press, Cambridge, Mass., 1975.

A COMPARISON OF VARIOUS METHODS FOR DETECTING AND UTILIZING PARALLELISM IN A SINGLE INSTRUCTION STREAM

by

Henry D. Shapiro
Computer Architecture Department
Sperry Research Center
Sudbury, Massachusetts 01776

Abstract -- By analyzing the data dependency graph of a program it is possible to determine the potential for program speedups by simultaneous execution of logically independent operations. When concurrent execution of instructions in existing programs on a given machine is attempted, efficient detection of data independence during execution is a central difficulty. Simulation, using actual program traces, has been used to evaluate the effectiveness of several approaches to detecting the presence of logically independent operations as a function of the number of processing elements. The results indicate that simple conflict detection algorithms perform about as well as more complex detection algorithms if the number of processing elements is six or less. The complex algorithms continue to show performance improvements as the number of processing elements increases, whereas, performance levels off if the simple algorithms are used. The rate of this increase indicates that the additional improvement achievable probably does not justify the increased cost of the complex detection mechanisms and the additional processing elements.

Introduction

The idea that program speedups can be obtained by simultaneous execution of logically independent instructions has received the attention of numerous researchers and practitioners [6,9,10, 11]. While some authors have reported that utilizing potential parallelism can give program speedups of a factor of 50, computer manufacturers have settled for actual performance improvement in the 1.5-3 range. There are two main reasons for this:

1) Theoretical work has tended to ignore the fact that the dependency graph, on which the more optimistic estimates are based, must be constructed during run time from a program stored linearly in main memory. If a computer utilizing the potential parallelism inherent in the dependency graph is to be cost effective, the hardware to detect the data dependencies present in the code must be fast, yet it cannot overshadow the multiple execution units in cost.

2) The problem of effectively handling conditional branches has not been solved. In pipelined machines, as described in M_4 below, both the next sequential instruction and the instruction branched to if the jump is taken can be conveniently prefetched. In machines with this type of architecture the test to abort the inappropriate branch can be made before any instructions along that branch have reached a point where recovery of the correct state is difficult. The conditional execution of (several) instructions along either path after a branch, before the test can be resolved, can lead to a large quantity of state information, in fact the amount of state information can grow exponentially since the instructions along the paths may themselves be branches.

Many researchers have felt that the conditional branch problem is the main reason that the potential parallelism in code is not better utilized. In this paper we analyze the effects of the problem raised in point one: How much parallelism is actually present in existing code, and how much does the technique used to detect and utilize this parallelism degrade performance from the ideal?

In the next section we shall review the theory associated with concurrent execution of logically independent instructions, pointing out a number of problems and subtleties not previously noted in the literature. After that we will present several machine models which detect and utilize parallelism in a single instruction stream in different ways. Some of the models embody the theoretically ideal data dependency detection mechanism, while still incorporating the realistic limitations of non-zero instruction decode time and main memory fetch time and an addressing structure similar to those found on many current computers. An empirical upper limit on performance improvement can be obtained for a given piece of object code by executing it on a (simulated) machine employing the ideal data dependency mechanism. Other of the models are based on data dependency detection mechanisms that do not fully exploit the parallelism inherent in the code, and thus are not as complex to implement. In the final section of the paper, simulation results and an interpretation will be presented.

Theory of Concurrent Instruction Execution

The abstract theory of program speedup by concurrent execution of logically independent instructions is well documented [5, 6, 9, 10]. In fact by appropriate interpretation the theory can be applied at a number of levels.

Definition: Let $T_1, T_2, \ldots, T_n$ be a sequence of elemental operations, each with a well defined set of input variables and output variables. We define an ordering relation $\textcircled{<}$ on the elemental operations as follows:

$T_i \textcircled{<} T_j$ if and only if $i < j$ and at least one of the following three conditions holds

(i) an input variable of T_j is an output variable of T_i

(ii) an input variable of T_i is an output variable of T_j

(iii) T_i and T_j have an output variable in common.

The transitive closure of $\textcircled{<}$ defines a partial ordering, $<$, on the set of elemental operations. From this definition it is possible to construct a data dependency graph (see Figure 1). It is customary to include only those arcs that cannot be deduced by transitivity. If execution times are associated with each node of the graph, we have the following:

Principle of Optimality: Given unlimited resources, the minimal execution time of a program, sequentially specified as $T_1, T_2, \ldots, T_n$, is equal to the length of the longest path in the dependency graph (the length equals the sum of the execution times of the nodes along the path), and this minimal execution time can be realized by starting an elemental operation as soon as all its predecessors (in the partial ordering) have completed.

This model has been specialized in a number of ways. Graham [4] and Coffman [2] have interpreted "elemental operation" as a job and have considered scheduling interrelated jobs on a multiprocessor system (with limited resources). Brinch-Hansen [1] has treated "elemental operation" as a procedure or begin block, allowing the user to specify parallelism in a higher level language. At the other end of the spectrum, Tsuchiya and Gonzalez [12] have performed automatic optimization of horizontal microcode within the constraints imposed by the dependency graph that results from considering as "elemental operations" logically indivisible sub-instruction functions.

In the research reported below we will be adapting this abstract model to execution of the instructions in a single user program. A number of subtleties arise in this case. It should be noted that the points presented below can be incorporated into the abstract model, by either slightly modifying the definition of ordering relation, $\textcircled{<}$, or by carefully defining the input and output variables. As with many other simulation problems the difficulty in building an accurate model is determining what aspects of the problem are the most relevant.

The first class of subtleties deals with why the principle of optimality does not really produce the minimum possible execution time.

1) Changing the elemental operations. By changing the choice of elemental operations the total execution time may be reduced. Formally we have

Definition: A refinement of a sequence of elemental operations, $T_1, T_2, \ldots, T_n$, is a sequence T_{11} $T_{12}, \ldots, T_{1m_1}$, $T_{21}, T_{22}, \ldots, T_{2m_2}, \ldots, T_{n1}$, $T_{n2}, \ldots, T_{nm_n}$, such that

(i) for $i \neq j$, if $T_{iu} < T_{jv}$ then $T_i < T_j$,

(ii) for $i \neq j$ if $T_i < T_j$, then there exists u and v with $T_{iu} < T_{jv}$, and

(iii) the longest path (where the length of a path is defined to be the sum of the execution times associated with the nodes along that path) in the subgraph $T_{i1}, T_{i2}, \ldots, T_{im_i}$ equals the execution time for node T_i, for all i, i.e. the execution time for the subgraph into which T_i is decomposed equals the execution time of T_i when viewed as a whole.

It is not difficult to prove that the minimal execution time of a refinement is less than or equal to the minimal execution time of the original sequence. Intuitively, the subsequence, $T_{i1}, T_{i2}, \ldots, T_{im_i}$ performs the same task as T_i. This condition can also be formally stated, but a precise statement of this condition is not important here.

The notion of refinement is relevant to the current discussion since there are two natural choices for elemental operations: Machine instructions, like load accumulator number five from the main memory location symbolically labeled I (L A5,I), or subinstruction functions, like compute an address, fetch an operand, etc. Figure 2 shows the same program segment as Figure 1, but with a different choice of elemental operations. Because of the environment within a computer, detecting dependencies at the subinstruction level is not more difficult than at the machine instruction level. In the simulation results reported later subinstruction functions are used as the elemental operations.

2) Restrictive instruction format. Even if a computer contains an unlimited supply of arithmetic-logic units, a rigid instruction format or lack of a sufficient number of general registers may introduce dependencies in the machine code not implied by the higher level language statement of the program. Inefficient use of general registers or poor code generation by a compiler can also create such dependencies. Dependencies introduced for these reasons normally manifest themselves as dependencies due to conditions (ii) and (iii) of the basic definition of the ordering relation. The arcs marked with asterisks in Figures 1 and 2 represent such dependencies. Keller [5] discusses the technique of "virtual registers" which can be used to eliminate these dependencies, and thereby, (potentially) reduce the minimal execution time. When viewed theoretically, the technique amounts to having an infinite number of input/output variables available for use with the elemental operations, and using each variable for output only once (it can subsequently be used for input indefinitely.) Practical implementation of the virtual register technique may be quite costly and the necessarily non-zero time to use the additional hardware may negate any expected performance improvement. Careful examination of code from machines with numerous general registers and register-memory and register-register instructions (UNIVAC 1100 series and IBM 360 series are typical) indicates that careful register allocation makes the potential gain from the use of virtual registers quite small in most real applications. At the other extreme, in machines with one accumulator this problem is so severe that almost no program speedup is possible without using virtual registers.

3) Alternate program formulations. The sequence $T_1, T_2, \ldots, T_n$ may be able to be replaced by another sequence $W_1, W_2, \ldots, W_m$, which accomplishes the same job. Kuck [6] and Lamport [7] have investigated speedups obtainable by semantic analysis of FORTRAN programs. Careful analysis of the algorithm being employed, with subsequent recasting to take advantage of vector/array features of the hardware can produce dramatic improvements. A recent paper by Giroux [3] reported a speedup factor of 25 for code carefully reworked for the CDC-STAR. The reprogramming effort took several years, however. Such techniques will not be investigated here.

The points mentioned above demonstrate that unless precautions are taken, determining potential program speedups from the dependency graph of a program can yield results that are too low. There are also a number of precautions that must be taken to avoid overly optimistic estimates of performance improvement. We discuss one here, since it is relatively abstract in nature; others are discussed in the next section.

4) Unresolved addresses. This problem is not apparent if the elemental operations are machine language instructions, and is easily overlooked if the dependency graph is generated from an assembly language listing containing symbolic addresses. On many third generation computers absolute addresses are not part of the instruction, but are computed during run time by adding a displacement contained within the instruction to the contents of a base register and, possibly, the contents of an index register. This addressing scheme permits, amongst other things, shorter instructions, greater run time flexibility in managing storage and makes array referencing more natural. Run time computation of absolute addresses poses a significant hazard for the potential for concurrent execution however. Consider the symbolic code segment:

```
      .
      .
      .
S AO,I
      .
      .
      .
L A2,J
      .
      .
      .
```

The fetch from memory location J cannot be safely initiated until it is known that there are no uncompleted (including uninitiated) stores to memory location J, preceeding this instruction. This implies that the real address symbolically represented by I will have to be computed and this address and the real address symbolically represented by J will have to be compared for possible conflict before the fetch can be safely initiated. Note carefully, we have a dependency between computing the real address of I and fetching from the memory location addressed by J. NOT between the actual store into I and the fetch from J. In fact no fetches or stores past the S AO,I instruction can be safely initiated until after the address of I is known. In the situation just described, where the base register is implied (and not modified frequently), and no indexing is performed, no delay will actually occur. The reason for this is that by the time the real address symbolically represented by J is known, so the fetch could be initiated, the real address symbolically represented by I will also be known. However, if a store is being made into a location whose address is computed using an index register, like S A1,A(XO), then the computation of the address symbolically represented by A(XO), can be delayed a long time due to a dependency on XO. Thus every fetch past the S A1,A(XO) will be (indirectly) delayed, waiting for an earlier load index register instruction to complete, even though there may be no conflicts over actual data. There is no a priori way of determining how much this indirect effect will lower the potential for program speedup. In practice

this problem arises naturally in two ways: 1) in scientific code, where indexing is a common occurrence, and 2) in the object code of programs written in ALGOL-like languages, where an index register is used as a base register to address variables whose scope is global to the currently active block. To the best of the author's knowledge nobody has investigated the effects of this problem on potential program speedup.

Machine Models

From the points discussed in the last section, determination of the potential for performance improvement by concurrent execution of logically independent elemental operations must be done with care if the results are to have credibility. A major additional problem remains when we consider building a computer to utilize all or some of this potential. How can the parallelism inherently present in a single instruction stream be efficiently utilized? The importance of this question cannot be stressed too much, since the data dependency detection mechanism creates an overhead cost in addition to the increased cost due to the presence of the multiple arithmetic-logic units needed to perform the computation. The (non-zero) time the detection mechanism takes to function must also be considered.

Four machine models were developed and a simulation was performed to determine how much of the potential program speedup could be realized by each one. The models differ primarily in the way they detect parallelism and in how they utilize the parallelism once found. The philosophy has been to determine the maximum possible potential parallelism within the constraints of each model. Toward this end the following properties are common to all four models.

1) The elemental operations are subinstruction functions. These include compute an address (see the discussion below), fetch/store from/to memory or a register, perform the basic algebraic or logical operation, and fetch and decode an instruction. The time it takes to perform each elemental operation is a parameter of the simulation. The rate at which instructions stored sequentially can be fetched can be set faster than main memory speeds, effectively simulating a high speed instruction buffer.

2) The virtual register technique is used to eliminate data dependencies arising from conditions (ii) and (iii) of the definition of the ordering relation.

3) Memory bandwidth is assumed adequate to handle the requests generated. Delays due to memory bank conflicts or cache misses (if memory cycle time is set sufficiently low as to imply a cache memory is being used) are ignored.

4) When a conditional branch is encountered the correct path is traversed, even before the test is completed. The simulator can perform in this manner because actual program traces are utilized, so the next instruction actually executed is available. This attitude is essentially the one taken by Riseman and Foster [9] in their earlier simulation experiments. Some factors which negate the clearly too optimistic nature of this approach are discussed below.

The goal of achieving the maximum possible potential parallelism must be tempered by reasonable constraints if the models developed are to produce meaningful results. The following properties, which are restrictive in nature, are common to all the models.

5) The addressing structure of the underlying instruction set reflects that used on a number of widely available machines. In all four models the final memory address is computed by adding a displacement to a base register and to an optional index register. As was discussed in the last section the computation of final addresses during execution impacts potential performance improvement by indirectly inhibiting all future fetches.

6) The too optimistic estimates developed by assuming foreknowledge of the way in which the test in a conditional branch will resolve is ameliorated by two constraints imposed on the models:

- The time between recognizing that an instruction is a conditional branch and the decoding of the next instruction to execute is greater when the branch is taken than when the next instruction to execute is fetched from the next sequential location. This reflects the fact that the "jump to" address must be computed before the instruction can be fetched.

- No stores into registers or memory are permitted after a conditional branch, until the test is made. The philosophy here is that no irrevocable actions should be taken until it is guaranteed that they will occur.

These two points, coupled with a non-zero instruction fetch time and the observed average distance between successive conditional branches in actual code, help to explain why the speedup factor of 50 reported by Riseman and Foster [9] is not confirmed by this research.

7) The amount of hardware that is dedicated to performing instruction execution (as opposed to hardware for detecting data dependencies) can be limited.

Definition: Suppose the instructions in the program trace are numbered consecutively (in the order they executed), and suppose a number N, known as the window size, is given. An instruction numbered i, is called active if and only if instructions 1, 2, ..., i-N have completed. In other words, the first uncompleted instruction and the N-1 instructions following the first uncompleted instruction are active.

Note that an active instruction need not actually be making forward progress; all of its uncompleted subinstruction functions may be waiting on data dependencies. The window size is a parameter of the simulation. When the window size is set to one all four models reduce to a common denominator--a computer which executes one instruction at a time, using parallelism within the instruction, and overlapping instruction fetch and decode of the next instruction with execution of the present instruction. The simulated execution time on this machine is used as the basis for computing program speedups when several processors are employed. This is a realistic model for both speedup and costing estimates, for it corresponds to many present day medium scale machines.

Within the constraints listed above, it is still possible to have wide variation. Determining which elemental operations can be started at any time is quite complex. To be valid a data dependency detection mechanism must not start an operation in violation of the partial ordering imposed by the dependency graph; it need not however, start an operation that logically can begin. In order to keep hardware costs within bounds the designer of a computer may choose to use a data dependency detection mechanism which does not utilize the full potential for concurrent execution.

Theoretical Ideal - M_1.

The purpose of this model is to establish an upper bound on possible performance improvement. For active instructions, subinstruction functions are started as soon as the necessary input data is available. In particular, dependencies created by the addressing structure are ignored, and only dependencies on actual data are considered. In terms of the addressing structure used in the simulation, displacement plus contents of a base register plus (optional) contents of an index register, the dependencies caused by unresolved addresses, as discussed in the last section, are ignored. This model, not only establishes an empirical upper limit, it also allows us to gauge the effect of two measures designed to increase parallelism potential:

- Fetch operands before the possibility of conflict over data is resolved and restart an instruction if a data dependency is later discovered.
- Increase the word length, allowing programmers and compilers to use the added bits to contain data dependency information derivable from the symbolic (assembly) listing, but lost in translating into absolute machine code.

Fully Parallel Computer - M_2

The data detection mechanism is quite similar to that used in M_1, a subinstruction function for an active instruction is begun as soon as it can logically be started. The difference is in the way dependencies due to the addressing structure are treated. In M_2 the (indirect) dependencies are not ignored as in M_1. Since elemental operations can proceed in an order quite different from that implied by the sequential program statement, the name "fully parallel" is justified. It is very important to note the complexity of the algorithms used to detect data independence in M_1 and M_2. Since any active can be dependent on any other active instruction which precedes it in the sequential program statement, if the active instruction window is of size N, the number of comparisons which must be made to determine all startable elemental operations is $O(N^2)$. Thus the overhead cost of the detection mechanism grows at a faster rate than the hardware performing the actual instruction execution.

Concurrent Execution/Sequential Detection - M_3

One way to prevent the overhead cost of the detection mechanism from growing at a rate faster than the rate of growth of the arithmetic-logic units is to use a data dependency detection mechanism that processes requests to fetch or store an operand in a sequential manner. In model M_3 the data dependency mechanism provisionally approves a fetch or store if

(1) the address of the fetch or store is known and

(2) there is at most one dependency which prevents complete approval of the request, (e.g. the only reason a store to an address cannot be approved is that there is a fetch from the same address).

The arithmetic-logic units in model M_3 are assumed to be sufficiently complex that they can detect resolution of the one dependency. If the address of the fetch or store is not known or there are several dependencies then the request is held up, as are all requests following this request. When the request can be provisionally approved, processing of requests resumes. In addition to the general strategy just described, if an instruction fetches a value from a location and later stores an (updated) value back into the same location this is not counted as a dependency. This is perfectly safe since no value to store can be computed until

after the fetch is complete. This, apparently minor addition is necessary if two address machines are not to be unduly penalized by their addressing structures. (Note the earlier discussion of overly restrictive instruction formats.) The actual execution of instructions goes on concurrently in multiple execution units in M_3.

Mechanisms of the type described here have been used on a number of computers. Certainly the most widely known is the scoreboard on the CDC-6600 [10].

Overlapped Computer - M_4

The philosophy behind this design is that the execution of an instruction can be divided into phases, which the instruction progresses through sequentially. On code with no dependencies an instruction is in phase N, the next sequentially specified instruction is in phase N-1, the next sequentially specified instruction is in phase N-2, etc., and at the end of every machine cycle all instructions advance to the next phase. N is called the degree of overlap. Checking for dependencies is quite simple, since the activity in each phase is proscribed. Uneven execution times and data dependencies cause delays by not permitting an instruction, and all instructions that follow it, to advance to their next phase. The arithmetic-logic unit can be divided into several phases (pipelined) if this seems appropriate. Even many modest computers are overlapped to some degree, instruction fetch and decode is overlapped with instruction execution. In this simple case the only dependencies possible are due to self modifying code (prefetched instruction no longer correct) or branches taken (wrong instruction prefetched). Normally a delay is encountered while the correct instruction is refetched. The UNIVAC 1100/80 and IBM 370/168 are examples of machines in which a high degree of overlap is used to gain significant speedups.

Results and Conclusions

A simulator for the models described in the preceding section has been developed and run on a number of benchmarks. The benchmarks, written in FORTRAN and executed on a UNIVAC 1106 available to the staff at the Sperry Research Center, were chosen from programs currently being used in unrelated research areas. The programs studied in depth were a constrained minimization problem with integer variables, a model of a physical problem which uses double precision floating point instructions, and a system print routine. The simulator uses program traces generated by actual program executions. The results of a number of runs are summarized in the graphs at the end of the paper. (Figures 3-5). The horizontal axis represents the number of active instructions. The vertical axis represents the potential speedup factor due to concurrent execution of logically independent elemental operations. The speedup factor is computed relative to a computer (with the same component speeds) that executes instructions in a strictly sequential manner, except that instruction prefetch is performed.

The Effect of Sophisticated Addressing Structures on Potential Performance Improvement

The need to be concerned about the indirect effects of unresolved addresses has been mentioned in several places in this paper. There was no a priori way of determining the degree to which these dependencies would affect performance. Comparing the simulation results of M_1 and M_2 allows us to conclude that this problem is relatively minor. While the reason the performance degradation is so small is not completely understood several factors appear to contribute :

- index registers are frequently loaded in advance of need, so no delay is encountered when an address using indexing is computed.
- when a fetch is delayed in M_2 because of the presence of an unresolved address it is often simultaneously dependent on the actual data as well. This dependence on the data causes a delay in M_1. (This was discovered by monitoring some of the queues internal to the simulator.)

Several consequences of the small nature of the performance degradation should be noted. The two techniques for improving performance discussed with model M_1, recording within the instruction information about data dependencies derivable at compile time and prefetch of data with refetching if a dependency is found, do not provide noticeable improvement in speedup potential. An important conclusion can be drawn in the area of system security by naturally extending these results. The use of numerous base registers to allow implementation of small segments and rapid segment switching does not appear antithetical to the goals of using parallelism to gain high performance. The reason for this is that while changing the contents of a base register is done more frequently in an environment that supports flexible use of small segments, it is not done as often as loading or modifying an index register; thus having little additional impact.

Instruction Starvation and Differential Execution Time

The continued gradual rise in the potential performance of M_1 and M_2 as a function of the number of processors, after the performance of the simpler designs of M_3 and M_4 has leveled off might make it attractive to those users demanding the highest possible speeds. The sensitivity of the curve of performance versus number of active instructions to changes in the relative speeds of some of the basic operations becomes an interesting question. Two experiments were performed: In one the relative speed of sequential instruction fetch time to data fetch time was varied. In the other the relative time to perform a long instruction

(e.g. double precision floating point multiply) to that of a short instruction (e.g. add a fixed point value to a register) was varied. The results are summarized in Figures 6 and 7.

Figure 6 shows that unless a very fast instruction buffer is provided the performance improvement that can be expected from use of M_1 is limited due to instruction starvation. When the observed average instruction execution time is less than the rate at which instructions can be supplied, performance will level off at about the number of processors needed to maintain this average instruction execution time. The leveling off is gradual, of course, because the amount of parallelism in the instruction stream varies.

Figure 7 indicates that obtaining large speedups is difficult for those users who desire it the most, users with large simulation programs that use many double precision floating point operations (e.g. weather prediction programs). When an instruction with a long execution time falls on the critical path of the dependency graph, the activities that can be performed simultaneously with the execution of this instruction are soon completed and the system must wait for the instruction to complete.

Practical Data Dependency Detection Mechanisms

The evaluation of the effectiveness of practical mechanisms for detecting subinstruction independence was a major goal of this research. The graphs at the end of the paper show that sequential (provisional) approval of fetches and stores, as described in machine model M_3 and the overlapped approach as described in model M_4 limit the amount of parallelism that can be utilized. In order to validate the performance improvement recorded for model M_3, the author investigated the experiences of users who have run large FORTRAN codes on both CDC-6600 and CDC-6400. The two machines have the same instruction set. The CDC-6600 corresponds closely to model M_3, while the CDC-6400 corresponds to the base for comparison, a computer with only one active instruction. After taking into account the differences in component speeds, the performance improvement observed for the CDC-6600 over the CDC-6400 is similar to that obtained in this simulation [8]. The widening gap between M_1 and M_3 - M_4, as the number of active instructions increases beyond six, indicates that it is difficult to find algorithms to detect parallelism while keeping the cost of detection hardware down.

It appears that for all but specialized scientific problems that can be formulated in terms of vector or array operations, cost effective program speedups by architectural techniques cannot be pressed much beyond current implementations.

Some reasons for this are:

- The cost of the mechanism for detecting dependencies used in M_1 would be much greater than that used in M_3 or M_4, if it could be designed at all.

- The regularity introduced into M_3 and M_4 by the natural sequencing of certain elemental operations permits more even utilization of memory resources, keeping down the cost of the memory interface. It appears that more realistic modeling of the memory interface would impact M_1 more than M_3 or M_4.

- The benefits of M_1 over M_3 or M_4 are most significant when the number of active instructions is large. As the number of active instructions grows the problem of handling conditional branches becomes more acute.

- The sensitivity of M_1 to a number of factors (e.g. sequential instruction fetch times and execution time for long instructions) implies that a computer developed around model M_1 might very well show performance lower than anticipated from the simulation results.

- The rise of the potential performance curve for M_1 is gradual after six processors. The slow rate of increase means a marginal return for each processor added (at more or less constant cost).

It is not possible to conclude whether the M_3 or the M_4 design is more cost effective, in general. The effect of minor variations in the addressing structure, the instruction set and the job mix imply that a detailed analysis is needed in each individual case.

References

(1) Brinch-Hansen, P., "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, SE-1 no. 2, June 1975 pp. 199-207.

(2) Coffman, E.G. Jr. and R. L. Graham, "Optimal Scheduling for Two Processor Systems," Acta Informatica, 1972, pp. 200-213.

(3) Giroux, E.D., "A Large Mathematical Model Implementation on the Star-100 Computers", presented at the Symposium on High Speed Computer and Algorithm Organization (proceedings to appear).

(4) Graham, R.L., "Bounds on Multiprocessing Timing Anomalies.", SIAM Journal of Applied Mathematics, Vol.17, no.2, March 1969, pp. 416-429.

(5) Keller, R. M. "Look-Ahead Processors", Computing Surveys Vol. 7, no. 4, December 1975, pp. 177-195.

(6) Kuck, D. J., Y. Muraoka, and S.C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-like Programs and their Resulting Speedup", IEEE Transactions on Computers, C-21, no. 12, December 1972, pp. 1293-1309.

(7) Lamport, L., "The Parallel Execution of DO Loops", CACM, Vol. 17, no. 2, February 1974, pp. 83-93.

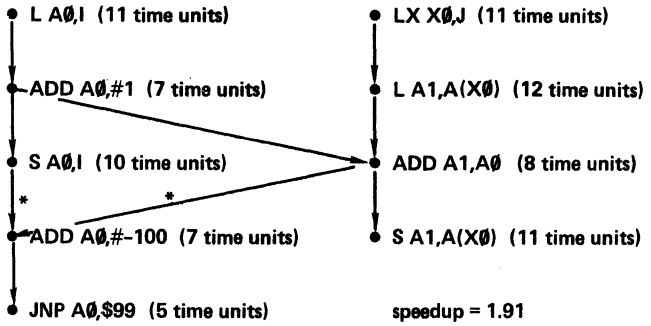
(8) Link, B. Sandia Laboratories. Private Communication.

(9) Riseman, E.M. and C.C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, C-21, no. 12, December 1972, pp. 1405-1411.

(10) Thornton, J.E., Design of a Computer--The Control Data 6600, Scott, Foresman and Company, Glenview, Illinois, 1970.

(11) Tjaden, G. S. and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions", IEEE Transactions on Computers, C-19, no. 10, October 1970, pp. 889-895.

(12) Tsuchiya, M. and M.J. Gonzalez, "Toward Optimization of Horizontal Microprograms", IEEE Transactions on Computers, C-25, no. 10, October 1976, pp. 992-999.



FORTRAN		Sample Machine Code		Key	
I=I+1	(1)	L	AØ,I	L	= load accumulator
	(2)	ADD	AØ,#1	ADD	= fixed point addition
	(3)	S	AØ,I	S	= store accumulator
A(J)=A(J)+I	(4)	LX	XØ,J	LX	= load index register
	(5)	L	A1,A(XØ)	JNP	= jump non-positive
	(6)	ADD	A1,AØ	#i	= immediate operand
	(7)	S	A1,A(XØ)	$99	= label
IF (I .LE. 100)GOTO 99	(8)	ADD	AØ,#-100	Ai	= accumulator i
	(9)	JNP	AØ,$99	Xi	= index register i
				A(Xi)	= address computation using indexing (array A is of type INTEGER)

FIG. 1 A Typical Dependency Graph

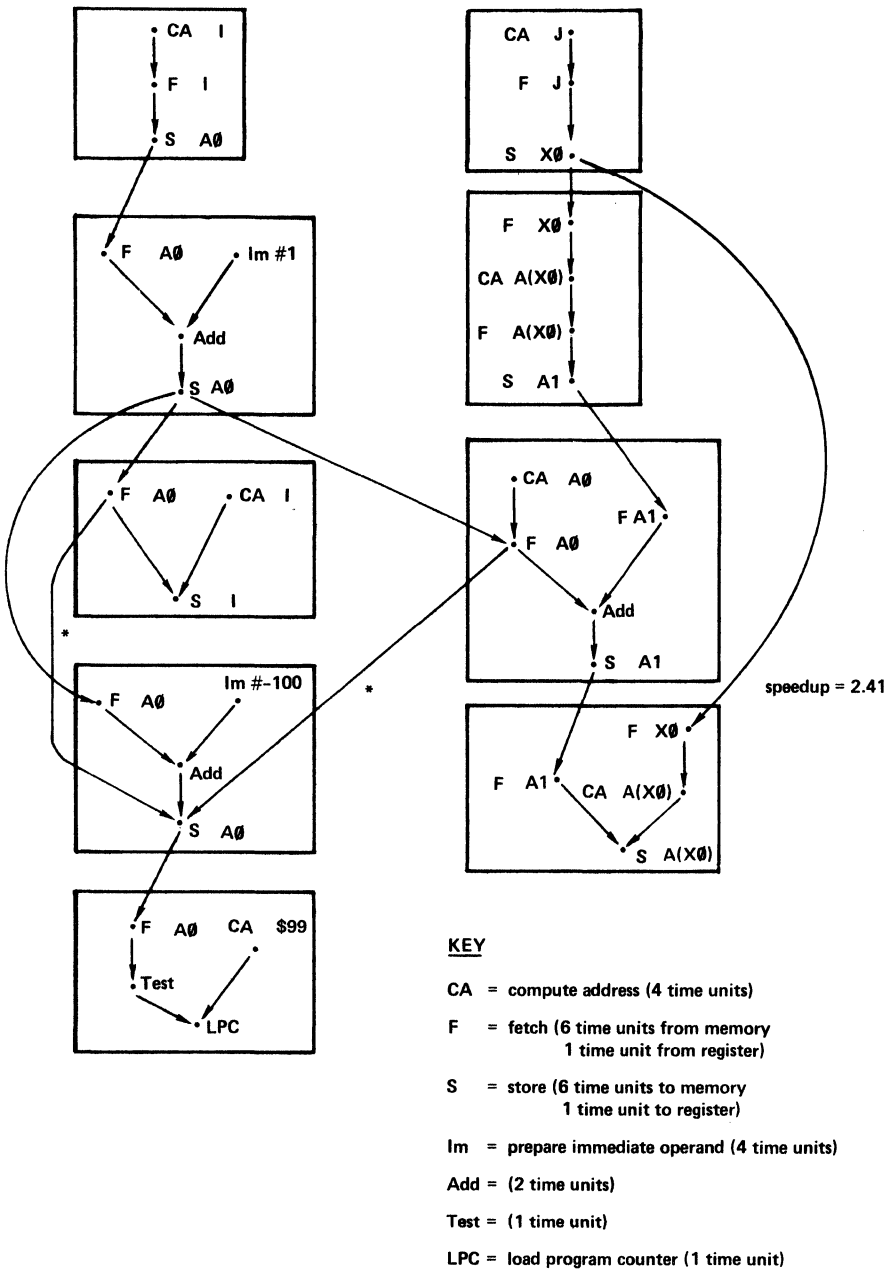


FIG. 2 A Refinement of a Dependency Graph

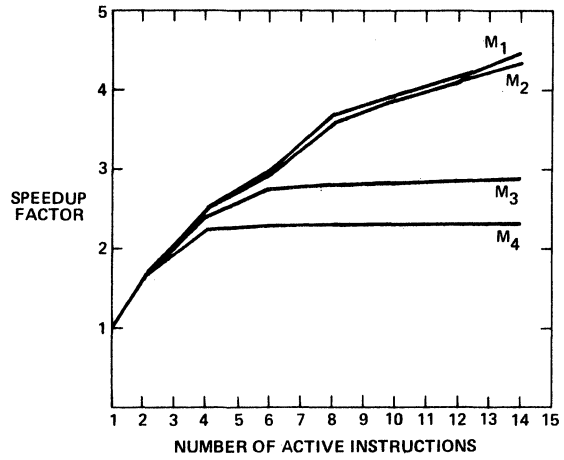


FIG. 3 Parallelism Potential — Constrained Minimization Problem

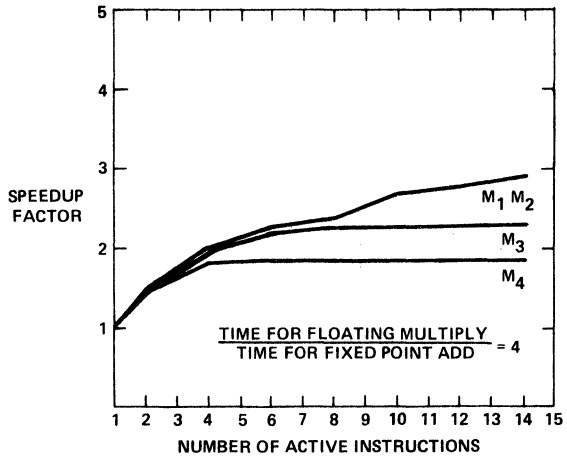


FIG. 4 Parallelism Potential — Simulation of Electron Scattering

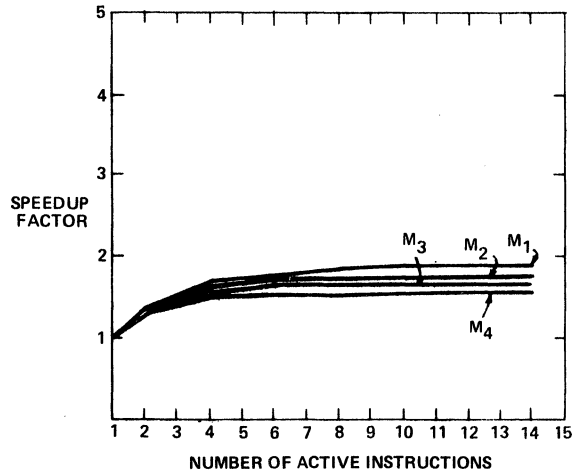


FIG. 5 Parallelism Potential — FORTRAN Print Routine

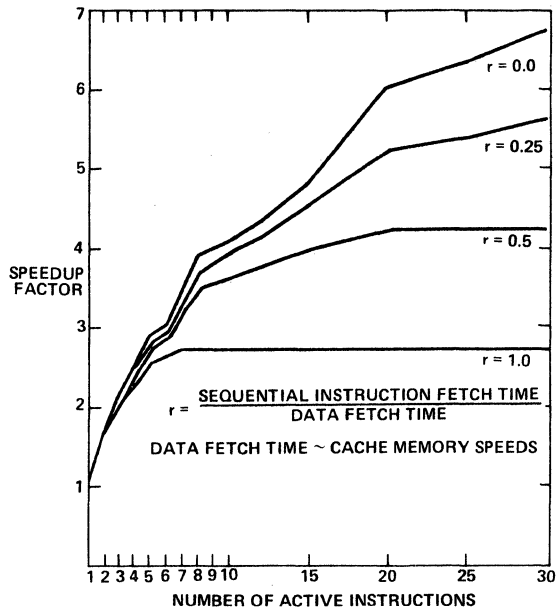


FIG. 6 The Effects of Instruction Starvation

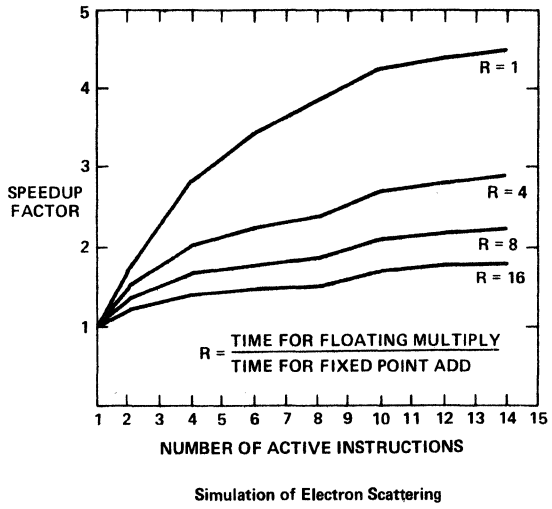


FIG. 7 The Effects of Instructions with Long Execution Times

Implementation of Procedures on a Class of Data Flow Processors

Glen Seth Miranker *
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- We present a machine structure which has as its base language, encodings of a general class of parallel programs known as ***data flow schemas***. The machine is unique in that it supports a rich subset of the schema class that includes procedures. The procedure implementation scheme is particularly novel in that the creation, execution and termination of procedure activations are distributed over the machine. The additional hardware required to handle procedures is small and smoothly incorporated into existing data flow machine designs. Furthermore, the execution overhead is low. Fundamental to the scheme is ***selective copying*** of active parts of an invoked procedure. ***Runtime renaming*** of the copied parts of the procedure is used to maintain the identity of distinct activations.

1. Introduction

1.1 Motivation

One general architectural approach to the task of improving machine performance has been the design of parallel processors. The traditional approach has been to design ***stream processors*** [25, 26] . These processors tend to divide into two general groups. There are those that achieve modest performance improvement but are quite general purpose, typical examples being the IBM 360, a SISD machine (Single Instruction, Single Data Stream), and the CDC star, a MISD machine (Multiple Instruction, Single Data Stream) [26] . There are also those that achieve striking performance improvement for specific types of computations, most notably the SIMD (Single Instruction, Multiple Data Stream) machines, such as ILLIAC IV or STARAN [9].

These approaches to parallel computation are limited by the traditional view that a computation is a sequentially ordered set of operations to be performed on a set of data. A more natural notion is that any operation can proceed when its operand values are available, and the destination of the results of the operation are able to receive them. The class of programs called ***data flow schemas*** captures this idea.

1.2 A Parallel Representation of Computations

A ***program schema*** in general is an abstract program, constructed from a precisely specified set of primitives according to a prescribed set of composition rules. Data flow schemas (DFS) are graphical models of programs that embody the idea that a component operation can be carried out as soon as all its operands are available. More generally, control flow can be (and in DFSs is) completely determined by flow of data. The most important property of DFSs is that the computations modeled by DFSs though highly parallel are ***determinate*** [21] . That is, the outputs of any DFS program that terminates on a set of inputs, are functions only of the particular set of inputs and not the past history, or the particular computation sequence.

Furthermore, DFSs in addition to exhibiting large amounts of parallelism, are syntactically modular, side-effect free, and quite general in expressive power. It has been shown (constructively) that DFSs are equivalent to the class of flowchart schemas which are well known to be able to model ALGOL-like programs [15].

Thus DFSs are a well understood model of parallel computation which exhibits a number of properties one might desire to have in a language executable on a new (***parallel***) machine architecture. Consequently, if a machine existed which could execute instructions that were an encoding of a DFS, and executed them in the manner prescribed by the data flow model (i.e. flow of control is determined by flow of data) we would have a parallel machine which had as its "base language" a data flow language [3] . Such a machine would exhibit many of the properties of the programs themselves, e.g. a high degree of parallelism and determinate computations.

A number of the architectures proposed to date that directly execute encodings of data flow programs cannot handle procedures efficiently - or at all [2,7, 22,24] . In this paper we will present an extension to the data flow machine of Dennis and Misunas [7] which can execute data flow programs as procedures in a general, efficient and architecturally simple manner. Section 2 of the paper is a review of the base language of the data flow machine. Sections 3.1, to 3.3, review those apsects of the machine's operation that are central to our procedure implementation. The remainder of the paper discusses the implementation scheme itself.

2. The Base Language

2.1 Introduction

The base language for the new machine, DFM, is a subset of a data flow language. DFL is itself just the set of interpreted DF schemas that have been thoroughly discussed in [5, 10, 11, 13, 15] . We will not introduce the general schema class but briefly review the language.

DFL is a generalization of the idea of execution of an operation as soon as its operands are available. A DFL program is a bipartite, directed graph whose nodes are

1. Actors - sites of action

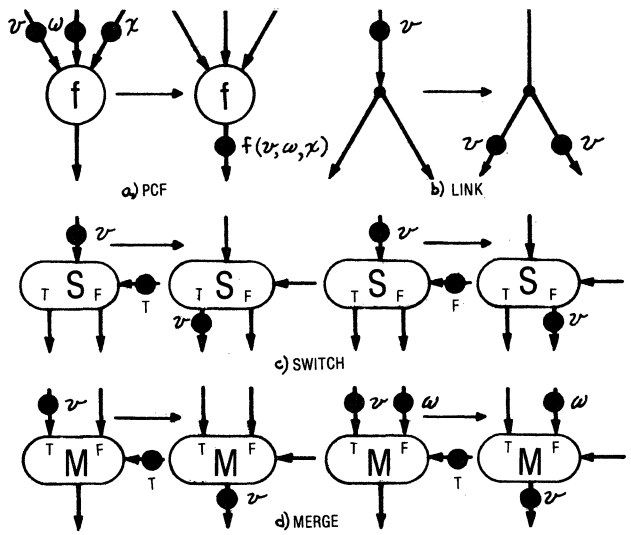
and

2. Links - conveyors of values.

All the roots and leaves of the graph are links. We regard values as being associated with tokens which are placed on the input and output arcs of the links. An arc may hold at most one token.

A node's ***behavior*** or ***firing rule*** is characterized by a simple algorithm which depends on whether it is a link, or which of the four classes of actors it is a member - primitive computational function (PCF), SWITCH, MERGE gate or APPLY. The firing rules for links and the first three kinds of actors are depicted in figure 1. We will defer the discussion of the APPLY actor until later.

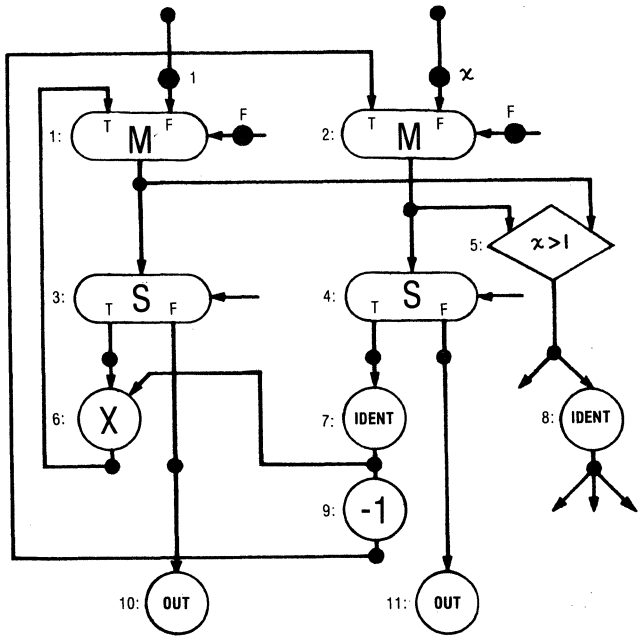
*The author is currently supported by the Hertz Foundation through a graduate fellowship.



(analogously for "F"tokens)
Firing Rules
Figure 1

In general a node fires by absorbing a token from each of its input edges, performing a transformation on them, and placing a single token on the appropriate output edge. A node may fire only if the ouput edge to receive the value is empty, and all of the input edges of the node are occupied by tokens. An exception to this general rule is the MERGE gate. It observes the above output restriction but needs only two of its inputs to fire, as shown above.

Given these primitives and the composition rules, we can construct fairly elaborate programs. Figure 2 is a data flow program that computes **factorial(x)** (x > 0).



A Data Flow Program Computing Factorial
Figure 2

The actor IDENT performs the identity transformation, the × actor's output is the product of its inputs, and the -1 actor decrements its input argument.

2.2 Data Flow Procedures

In sequential programming languages, the abstraction obtained by using procedures is a useful one. One would expect the same sorts of advantages to accrue in a parallel programming language. We will incorporate them into DFL by generalizing the notion of a program and adding one additional actor type. A DFL program is a ***forest*** of properly formed DFL programs called procedures. We associate a ***procedure name*** with each of the subprograms in the forest. One of the subprograms, called the ***main program***, has the name P_0 associated with it, and it is the only procedure to receive tokens on its input arcs from an "outside agent". That is, it is the entry point of the program in the sense that it receives the input values.

We introduce a new actor which we call an APPLY actor. It has m inputs, n outputs and is labelled with a procedure name P_i. The APPLY actor when enabled to fire, (conceptually) substitutes for itself a copy of the procedure whose name matches that of the actor. This action taken by the actor is defined only if a procedure exists with name P_i and the procedure has the same number of inputs and outputs as the actor.

To completely understand how the APPLY actor works, we must define the enabling condition, the mechanism for transmitting input values to the copied procedure, and the return mechanism for results. There are two alternatives we consider:

1. The APPLY actor is enabled, as soon as its first argument token arrives. It then copies the procedure (a procedure copy is called an ***instantiation***) and passes argument tokens as they arrive. An argument is passed by absorbing a token from an input arc to the APPLY, and placing a copy of it onto the procedure instantiation's corresponding input link's output arc. The actor copies output values from the procedure copy as soon as they become available on the output links input arc and the corresponding link of the calling program is empty. When values from each output link have been returned the copy is destroyed.

2. The APPLY actor is enabled only when all its argument values have arrived and its output links are empty. When these two conditions are met, the procedure is copied and the argument tokens passed. When ***all*** result tokens are available they are copied by the APPLY from the input arcs of the procedures output links to the output arcs of the APPLY actor. The copy of the procedure is then destroyed.

In both cases we assume that the n input links are numbered left to right, 0,1, . . . , n-1, for both the APPLY actor and the procedure it invokes. We associate the i^{th} link of the APPLY with the i^{th} link of the procedure it invokes. We treat the m output links in a similar fashion.

The semantics of the two approaches to procedure activation are quite different. In the first approach an APPLY actor can be thought of as replaced inline by the graph of the procedure it invokes. In the second approach an APPLY actor behaves exactly like a PCF actor, except that it may have multiple outputs and computes a function that is not necessarily in the repertoire of PCF's. The first approach we call the immediate copy rule (ICR). The second is called the deferred copy rule (DCR). The DCR most closely corresponds with one's idea that a procedure is some sort of a functional abstraction, whereas the ICR is more like a macro expansion. The DCR has the advantage of simplicity of implementation. It also lends an additional homogeneity to the set of actors, since its enabling rule is that of

a PCF. However, the ICR clearly allows greater parallelism than DCR. Furthermore, it is easy to simulate the DCR for an APPLY actor using an ICR APPLY actor and gates. Consequently, we have chosen to incorporate the ICR form of the APPLY in DFL.

The ICR has one potential problem. Suppose an argument token arrives on the i^{th} link and the execution of some procedure is initiated. Consider what happens if another argument arrives on the i^{th} link before the previously invoked copy of the procedure terminates. In order to be consistent we must create another copy of the procedure and pass this newly arrived token to it. Thus the APPLY actor must "keep track of" an arbitrary number of concurrently executing instantiations of the procedure. Needless to say, this poses some serious implementation questions. If we can demonstrate for every APPLY actor $\mathcal{A}$ that

$$\forall (i, j)\ |\mathcal{N}(i) - \mathcal{N}(j)| \leq 1, \quad 0 \leq i, j \leq \text{number of inputs of } \mathcal{A}$$

where $\mathcal{N}(i)$ = number tokens that have arrived on the i^{th} input of $\mathcal{A}$

for any reachable configuration of a data flow program, then we can show for any apply actor $\mathcal{A}$ of a DFL program that at most one instantiation can exist at any time, and consquently the state information is bounded. In general, data flow programs do not exhibit this behavior. However, certain large syntactic subclasses of DFL programs have been shown to satisfy the above arc condition [19, 20]. One such class is known as ***well formed data flow programs***. Beside having the above property, a well formed data flow procedure, when it terminates, will be in its initial configuration. In particular, the only tokens left on the arcs of a terminated procedure, will be the initial "F" tokens on the gating inputs of MERGE gates of iterative loops. This property will be important for the implementation.

The introduction of procedures concludes our discussion of DFL. The reader may note the DFL language does not contain structures as a token type, nor does it have any structure manipulating actors. These features have been omitted for brevity, since they are adequately discussed in [5], and [20].

3. The Machine

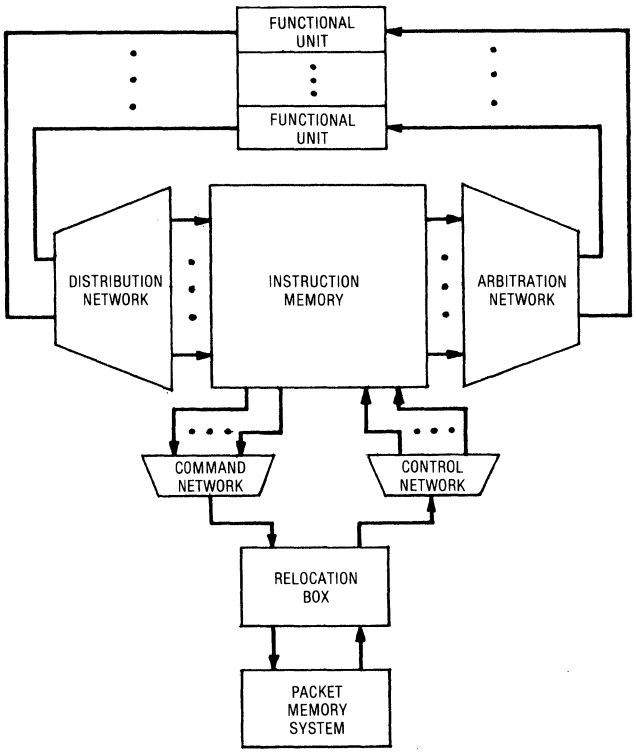
3.1 Introduction

There are obvious hardware analogs to each of the the actor types of a data flow program. This suggests that it is possible to execute an encoding of a data flow program by constructing an exact hardware realization of the program [14]. This approach has several serious liabilities [19]. Most important, the resulting machine would have much idle hardware, since any direct implementation of actors would necessitate placing substantial computational power at each hardware actor. Also, such a machine would not be readily configurable. For all these considerations we reject out of hand any direct implementation of data flow program graphs.

The description of the machine architecture we present will be at a rather abstract level, and quite brief since our main purpose is to introduce the procedure implementation scheme. The interested reader is referred to [4, 7, 19] for details. Briefly, this new machine, DFM, is a member of a larger general class of system structures known as ***packet communication architectures*** [6]. Such systems consist of a collection of interconnected subsystems, which themselves may be packet communications structures. The subsytems are interconnected via a set of one way links called ***channels***. They communicate with each other by sending messages through the channels using a well specified protocol. The messages transmitted are known as ***packets***.

3.2 The Processor Pipeline

The processor pipeline is composed of four parts, the instruction memory (IM), the arbitration network (AN), the functional units (FU) and the distribution network (DN) connected as shown in figure 3.



Data Flow Processor With Procedure Capability
Figure 3

The IM is the heart of the machine. It contains a collection of nc identical units called ***cells***. We can refer to a particular cell of the IM by a unique integer which we call a ***cell name***. With the exception of APPLY actors, there is a one-to-one correspondence between a cell of an object program and an actor of the source program. Since we are not interested in the precise representation of cells, we will schematically represent them as in figure 4.

OPCODE	
ARG_0	$DEST_0$
ARG_1	$DEST_1$
ARG_2	$DEST_2$

Abstract Cell Representation
Figure 4

The OPCODE field specifies a function to be computed according to the type of the actor the cell represents. It is set at program loading time and remains fixed throughout execution. The ARG fields are also registers. Their contents are encodings of the values that reside on the input arcs to the actor whose representation was loaded into the cell. Clearly the ARG fields change their contents during execution. Notice that by assuming three such fields, only actors with three (or fewer) inputs are representable in the machine. The DEST fields contain addresses, which are encoded as register pairs which specify an ARG

register of a cell. If a cell has fewer than three destinations, the unused DEST fields are set to a distinguished state notused. Similarly, if an OPCODE requires fewer than three arguments, the unused ARG fields are set to notused. Multiple destination fields are used to provide *fanout* of values. Since there are three destination fields, the maximum branch factor of a link of a representable program is three.

When the function specified by OPCODE (OPCODE ≠ SWITCH) is computed, the results are sent to the ARG registers of the cells named in the DEST fields. For SWITCH's, *one* of the destination will be selected on the basis of the Boolean argument of the cell. The cell encoding of the factorial program of figure 2 is shown in figure 5.

1:

MERGE	
1	3, 0
	5, 0
FALSE	NOT USED

2:

MERGE	
α	4, 0
	5, 1
FALSE	NOT USED

3:

SWITCH	
	6, 0
	NOT USED
NOT USED	NOT USED

4:

SWITCH	
	7, 0
	11, 0
NOT USED	NOT USED

5:

LESS THAN	
	8, 0
	1, 2
NOT USED	NOT USED

6:

MULT	
	1, 1
	NOT USED
NOT USED	NOT USED

7:

IDENT	
	6, 1
NOT USED	9, 0
NOT USED	NOT USED

8:

IDENT	
	2, 2
NOT USED	3, 1
NOT USED	4, 1

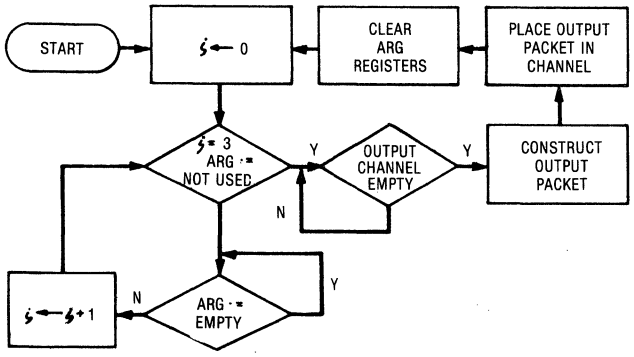
9:

DEC	
	2, 1
NOT USED	NOT USED
NOT USED	NOT USED

Cell Encoding of Factorial Program
Figure 5

By convention, we let the 0^{th} ARG field of a SWITCH correspond to the data input. The first ARG field corresponds to the control input. The 0^{th} DEST field holds the T destination address, and the 1^{st}, the F address. Similarly, the 0^{th} ARG field of a MERGE is the T input, the 1^{st} is the F input, and the 2^{nd} is the control input.

Now that we have outlined the structure of cells, we must describe their operation. Figure 6 specifies the operation of cells whose opcode field is not MERGE. The operation of cells implementing MERGE gates is an obvious modification.



Cell Operation
Figure 6

where an operation packet is a septuple of binary words

{opc, v_0, v_1, v_2, d_0, d_1, d_2}
opc = contents of opcode register
v_i = contents of register ARG_i
= null if register ARG_i is notused
d_i = contents of register $DEST_i$
= null if register $DEST_i$ is notused

Thus the cells of the instruction memory pass "messages" to the arbitration net requesting that an operation be done. Notice that cells have some processing capability - they are small finite state machines. However, this capability is quite primitive. The operation of the rest of the pipeline should be intuitively clear, and is omitted.

3.3 The Virtual Memory

Now that the basic processor pipeline has been outlined, and its major characteristics described, we turn our attention to the virtual memory part of the DFM. With iterative and conditional programs and recursive procedures (to be described later) it becomes obvious that parts of a program are more "active" than others. Therefore, it is desirable to have a *two level memory hierarchy*. One level is the instruction memory, in which we keep the most active parts of a program. The other level is some sort of *backing store*, much larger than the IM, and lacking any processing ability. Hence it is relatively inexpensive. *Cell images* are kept in the backing store. Each cell image is a sequence of binary words which completely specifies the state of a cell.

When a cell is referenced during execution that is not in the IM, its image is fetched from the backing store and placed into a physical cell (p-cell) of the instruction memory (assuming a p-cell is free). If a cell c is referenced that is not in the IM, and there are no free p-cells, a p-cell is selected for displacement. A p-cell is said to be *pristine* if it has received no operands, *or* is a MERGE cell and contains only and initial F token. A pristine, selected p-cell is simply discarded. Otherwise, the state of the selected p-cell, together with the name of the cell that is "in it" is stored in the backing store. This makes room for the cell image of c which is subsequently fetched and installed in the new, free p-cell in the IM. Notice that the set of cell names now has the size of the backing store plus the IM, rather than just the IM. Furthermore, a cell may be resident in the machine as an image either in the backing store or in a p-cell. Thus the *virtual* cell name space is much larger than nc. The command and control networks and the PMS implement the backing store [6,19] . To retreive a cell named c, a packet of the form {fetch, c} is issued by some controller in the IM to the command network. Storing is similar, but a packet tagged store is sent, and the packet also contains the state of the named cell. Retrieved cells are returned to the IM through the control network, again via a packet containing a cell name and the named cell's state.

3.4 Procedure Invocation and Activation Names

There are several ways of invoking a procedure in data flow language that are consistent with the data flow model. Consider for example, a single input, k output procedure P. The effect of a token arriving at an actor $\mathcal{A}$ labelled APPLY P is easily described. When a data token α arrives on the input arc, a copy of the data flow graph for P is made, α is absorbed from the input arc of $\mathcal{A}$ and a copy is placed on the output arc of the input link of procedure P. As each of the k outputs for this activation of P is produced, it is passed from its output link to the corresponding output link of the APPLY and hence to a successor cell of $\mathcal{A}$. (For convenience in discussion, each cell in a data flow program is assumed to have a unique name associated with it. The name of a cell will be shown in the figures as <name>: next to the representation of the cell. The name of a cell is used in

the data flow processor to identify it. For example, to route result packets to it or to retrieve it from memory.) To be syntactically correct, P must have one input link and k output links.

The heart of the procedure implementation scheme is the ***relocation box*** together with a special functional unit that can do "byte" manipulations on operation packets. Anticipating mechanisms to be proposed later, some of the functions of the relocation box (RB) will be described. It is assumed that every actor in a data flow program as represented in the processor has a unique cell name except for APPLY actors. Further, during the course of execution (where and how will be described later) a cell name may have a suffix appended to it - these suffixes will serve to distinguish activations. At any time during the execution of a program, there will be a one to one correspondence between used suffixes and procedure activations. In the following discussion we will seperate a cell name from a suffix by a ".".

The RB's operation is quite simple. Upon receipt of a ***fetch packet*** from the memory command network {fetch, $\alpha.\sigma$}, it passes the packet {fetch, α} to the packet memory system. When cell image of α is returned by the PMS to the relocation box, all the names in its destination fields are changed to have suffix σ. The RB then passes the resulting cell image back through the memory control network to the instruction memory. Finally, it is assumed that with the sole exception of the relocation box and one special functional unit, no other component of the data flow processor of figure 3 can distinguish if a cell name has a suffix appended or not. That is, if the distribution network for example, receives a packet with a destination cell name $\alpha.\sigma$ it sends a result packet to cell $\alpha\sigma$ (the dot seperating the name from the suffix is included merely as an aid to the reader's eye). The essential idea is that a complete cell name (i.e. a cell name plus an appended suffix, which we refer to as an ***execution name***) is treated everywhere but the relocation box and the distinguished functional unit as a single entity - a cell designation.

Before introducing the complete procedure mechanism, we will first demonstrate how a single input/single output procedure is invoked on the machine. The APPLY cell in particular has the format shown in figure 7.

APPLY	
P	DEST.Q
empty	NULL.Q
not used	NULL.Q

An Apply Cell
Figure 7

P is the name of the procedure to be invoked, DEST.Q is the name of the cell that is to receive the result value, and the empty ARG register is to receive the argument for the procedure call. The implementation of a single input-single output APPLY actor is straightforward. When the operand α arrives, the APPLY becomes enabled, and transmits the packet
{APPLY, P, α, DEST.Q, NULL.Q, NULL.Q} which the arbitration network routes to the special functional unit that processes procedures. The FU on receipt of an ***apply operation packet*** creates a unique suffix σ and then outputs two packets {α, P.σ} and {DEST.Q, RT.σ}. (Every packet sent to a cell must also contain ***field addresses***, that is, specifications of which register in a cell is to receive the value(s) conveyed by the packet. These will not be explicitly represented in the diagrams since it should be clear from the context which register of a cell is supposed to receive which value. Leaving out the field address will make the diagrams a bit less detailed and hence less confusing.)

The first packet to arrive at the instruction memory destined for the procedure activation P.σ, causes the cache mechanism of the instruction memory to retrieve from the packet memory system a cell with name P.σ. (Since σ is unique suffix, cell P.σ can't possibly have been in the instruction memory.) Due to the action of the relocation box, a copy of cell P will be retrieved and all its destination fields given the suffix σ. Once the cell P.σ is successfully installed in the instruction memory, actor P.σ will then receive its operand α, and become enabled. Whatever computation is specified by P (the first actor of the invoked procedure) will be carried out and the resulting values will be in packets destined for cells $D_0.\sigma$, $D_1.\sigma$ . . . $D_n.\sigma$, where $D_1, \ldots D_n$ are the contents of the destination fields of P. Clearly $D_0.\sigma$ . . . $D_n.\sigma$ will not be found in instruction memory and will be fetched from the packet memory system in the manner described above for P.σ. And so execution of the σ^{th} activation of P proceeds.

To return the value computed by P we assume that all procedures are compiled so that their output value is to be sent to a cell named RT, which is a ***reserved*** cell name. That is, rather than having an output link, programs are compiled to have a (uniformly named) output cell. Further, it is assumed that resident in the packet memory system is a cell with name RT, as shown in figure 8.

RT:

RET	
empty	NULL
empty	NULL
not used	NULL

Return Cell
Figure 8

This cell belongs to no procedure (i.e. it is a runtime support cell). It will be retrieved by the second packet {DEST.Q, RT.σ} generated by the FU as a result of the APPLY cell firing. When it is finally resident in the IM, it will appear as in figure 9.

RT.σ :

RET	
DEST.Q	NULL.σ
empty	NULL.σ
not used	NULL.σ

Return Cell in IM
Figure 9

With the compiler convention previously mentioned, when execution of the procedure is complete the packet {Ω, RT.σ} will be sent by the FU to the DN, where Ω is the output value of the σ^{th} activation of P. Thus RT.σ will be enabled and create the packet {RET, DEST.Q, Ω, NULL, NULL.σ, NULL.σ, NULL.σ} which is sent to the appropriate function unit. This FU will then output the packet {Ω, DEST.Q} thus sending the result of the σ^{th} activation of P in the correct destination cell, and returns σ to the pool of free suffixes.

The question immediately arises as to whether the machine resources are all reclaimed. Clearly the activation name σ has been recovered. But what about the "residue" from the now terminated procedure activation? We remind the reader that the procedure is a well formed data flow program. Consequently, at termination, all of its cells are pristine. Thus because of the cache displacement (described in section 3.3) algorithm ***there will be no cells with a σ suffix in the PMS or in the request queues of the PMS.*** So all the σ residue is in the IM. Since cell names (unadorned by suffixes) are all unique, this residue will be purged from the IM as room is required for new cells. No trouble arises should σ be reused as an activation name for the same procedure as its last use before that residue has been cleared because the left-over cells are all pristine.

It may be objected that the IM should be immediately

purged of σ residue to prevent displacement of "useful" cells from the memory. This can be done by having the FU emit a packet $\{\sigma, IM\}$ in addition to $\{\Omega, DEST.Q\}$ when it processes the RET operation packet. Upon receipt of this packet, the IM cache manager purges itself of all cells with a σ suffix. When it is finished, it send the packet $\{RECLAIM, \sigma\}$ to the FU, which then places σ on the free list of suffixes. Alternately, we could construct the cache manager so that pristine cells have highest priority for removal.

This procedure application scheme has several attractions. First it is simple. Overhead in terms of storage, or extra packets in the system, is almost zero. Few changes need to be made to the basic data flow processor, and those that are necessary are incorporated in a smooth and natural way. Also, notice that in this scheme the entire procedure is not copied, just the pieces of it as they become active. This is an important characteristic especially for programs with conditional constructs. For these programs, the amount of processing activity is not uniform over all program actors. In particular the predicate of a conditional program Π, will select either the "true" or "false" subgraph of Π. It will never be the case that both subgraphs (of a given activation) execute. Thus to load both of them into the instruction memory is wasteful of instruction memory space and memory control network bandwidth. Finally, procedures are compiled no differently than programs, thus allowing (without recompilation) the use of any data flow program as a data flow procedure. This will be discussed at greater length in section 3.7.

3.5 More Elaborate Schemes.

The primary deficiency of this scheme for procedure implementation is that it supports a rather primitive form of the APPLY actor - only one input and one output. If this sort of procedure call were incorporated in a data flow processor which could manipulate data structures, this deficiency would not be so bad. Then multiple input values and multiple output values could then be encoded as structures. However, such a form of procedure invocation would be undesirable because it would limit the degree of parallelism achievable. After all, there is no inherent reason for returning all the outputs of a procedure simultaneously. If a procedure produces k outputs, to wait for all of them to be computed, assemble them into a structure, return the structure to the calling routine, disassemble the structure, and then to use the resulting k components, restricts the amount of concurrency achievable in a program. It also incurs the overhead of assembly and dissassembly of structures. One would like to pass each of the k output values to its destination in the calling procedure as it is generated.

Similarly, one would like to have multi-argument functions. Passing an ℓ component structure with the ℓ argument values to a procedure as its components is undesirable. Again there is a loss of parallelism. A particular subgraph of the data flow procedure may require only a subset of the ℓ input values to start execution. Thus to inhibit passing of any argument values to the procedure until all of them are available, seriously limits the achievable level of concurrency.

The naive approach to implementing multiple input/multiple output APPLY (in the context of the previous discussion) is simply to have ℓ apply cells for a ℓ input APPLY actor each receiving one argument value:

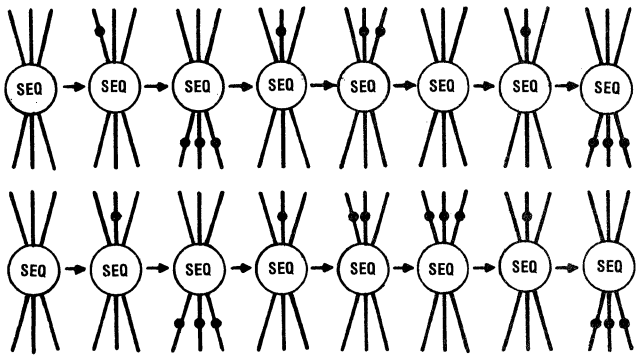
APPLY	
P_i	$DEST_0$
empty	$DEST_1$
not used	$DEST_2$

APPLY Cell for Multiple Input APPLY
Figure 10

All fields are as in figure 7 except P_j is the j^{th} input link of procedure P. The compiler is assumed to write "code" to send the j^{th} argument value to cell AP_j. We also place some number n of RETurn cells in the PMS, with names $RT_0, \ldots RT_{n-1}$. When the FU processes an APPLY operation packet it issues packets that retreive k return nodes $RT_0.\sigma, \ldots RT_{k-1}.\sigma$ and supplies each with the appropriate destination name. Unfortunately, this scheme does not work. The functional unit will assign a new activation name (new suffix) to each cell AP_i. This will guarantee incorrect operation for all but the most contrived programs. Even if this could somehow be patched, there is also no way to return output values correctly since each cell AP_j would cause a new set of RETurn cells to be fetched. Finally, there is no mechanism for freeing suffixes once a procedure instance terminates.

To correctly invoke a procedure we need to guarantee that for each set of input values to a particular APPLY actor only the first argument value causes generation of a packet which causes the functional unit to create a new activation name. Furthermore, we must make certain that the other argument values that are sent to the APPLY actor are all passed to the same procedure activation as the first. Last, we must be sure only one return mechanism is set up. To do this we introduce a new actor which we call SEQ (for sequence). It has ℓ inputs and ℓ outputs.

Operation of the SEQ actor is simple. Upon receipt of any one of its inputs, say on input arc j, it produces an output value which is a unique suffix name. This value is then passed out on each of SEQ's output arcs. No further action is taken until tokens arrive on inputs $0,1, \ldots j-1, j+1, \ldots \ell-1$. When this state occurs, one token on each of the input arcs i ($i \neq j$) is absorbed and no output token is produced. The actor (and cell) then return to the initial state and the above action is repeated. Figure 11 shows two examples of possible firing sequences of a SEQ actor.



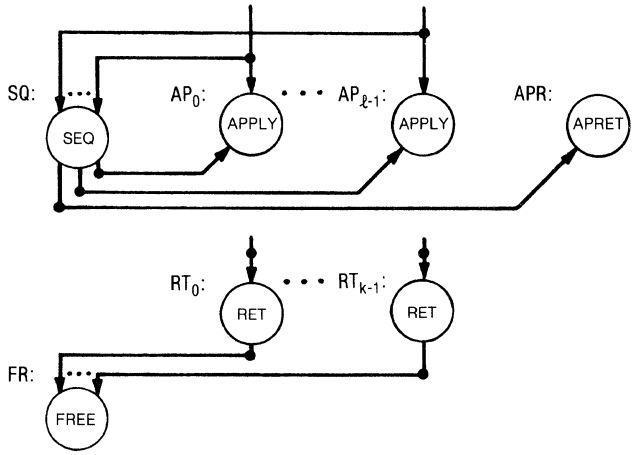
SEQ Actor Firing Rule
Figure 11

Notice that while the times at which outputs are produced are quite unusual compared to the other data flow actors, only one output is produced for each set of ℓ input values.

To allow freeing of activation names we introduce a new actor type FREE. Like SEQ, the FREE actor is *not* available to the programmer. It is a "runtime" support actor that allows proper activation name maintenance. This new actor has been introduced (just like SEQ actors) as a convenient way of showing how a procedure call and return is implemented. It is *not* an addition to DFL, but merely a way of presenting the details of the call-return mechanism that preserves a one to one correspondence between cells and actors. The FREE cell receives as inputs copies of each of the output values of the procedure instance of which it is a member. When all the output values have been produced the FREE cell is enabled. It then outputs a packet $\{FREE, \Omega_0, \Omega_1, \Omega_2, \ldots, NULL.\sigma, NULL.\sigma, \ldots NULL.\sigma\}$ which is routed to the functional unit that handles APPLYs and RETurns. Upon receipt of this

packet the functional unit frees the activation name σ or outputs the packet $\{\sigma,IM\}$ as before.

The reader should notice that unlike the RETurn cells, the FREE cell is *explicitly* included as part of the procedure application mechinism. This saves us the difficulty of determining at runtime the number of values the FREE cell must receive before becoming enabled. It may be objected that since the RETurn actors are not part of the procedure that the RETurn cells have no way to "know" what the name of the FREE cell is. Consequently, they cannot send result values to it. To fix this we send *two* destination names to the RETurn actors at runtime - the name of the cell that is to receive a result of the procedure call, and the name of the FREE cell. To ensure that these names are sent exactly once to the return cells, we add one more runtime cell, with an opcode APRET, whose function is described below. The full procedure call mechanism is shown in figure 12.



Procedure Call Mechanism
Figure 12

The ℓ input APPLY actor now maps into ℓ APPLY cells of the form in figure 10 except that they have NULL destination field, and the third argument field is used to hold a suffix name. SEQ and FREE actors are added to maintain activation names, and RET and APRET actors to handle return values. However, at the source level all the user is aware of is that he is using a single actor, an ℓ input/k output immediate copy APPLY. To see how this all works, let us suppose that the Q^{th} activation of some procedure produces the i^{th} argument a_i for an application of P, where P has ℓ inputs and k outputs. For concreteness, we depict the case of $\ell = 2$, $k = 3$ in figures 13 through 15.

The compiler will have generated code so that the actor that produces the value a_i has as its destinations the cells AP_i and SQ. Thus the packets $\{a_i , AP_i.Q\}$ and $\{a_i , SQ.Q\}$ will be produced. These will cause the two cells shown in figure 13 to be fetched into the instruction memory.

$AP_i.Q$:

APPLY	
P_i	NULL.Q
empty	NULL.Q
empty	NULL.Q

SQ.Q:

SEQ	
empty	AP_0.Q
empty	AP_1.Q
empty	APR.Q

Figure 13

The cells are depicted before the argument packets $\{a_i , AP_i.Q\}$ and $\{a_i , SQ.Q\}$ have been delivered. SQ is now enabled and will generate the packet $\{SEQ, \ldots a_i \ldots , AP_0.Q, AP_1.Q, APR.Q\}$ which is routed to the special functional unit. This FU outputs the packets $\{\sigma, AP_0.Q\}$, $\{\sigma, AP_1.Q\}$, and $\{\sigma, APR.Q\}$. The packet $\{\sigma, AP_i.Q\}$ will enable the cell $AP_i.Q$, and as a result it will output the packet $\{APPLY, P_j, a, \sigma, NULL.Q, NULL.Q, NULL.Q\}$ which is then routed to the apply functional unit. It then outputs the packet $\{a, P_j.\sigma\}$ and execution from the i^{th} entry cell σ^{th} activation procedes as described in the single input case. The other packets output by the functional unit - $\{\sigma, AP_j.Q\}$, $j \neq i$, $j \leq \ell$ - each cause cache faults, thus bringing in cells shown in figure 14.

$AP_j.Q$:

APPLY	
P_j	NULL.Q
empty	NULL.Q
empty	NULL.Q

Figure 14

When their arguments arrive, execution will proceed in the manner described for AP_i.

The returning of values from the σ activation of Π is handled in a similar fashion as that described in the previous section on single input/single output APPLY. In this case it is the APRET cell that causes the RET cells to be fetched into the instruction memory. The packet $\{\sigma, APR.Q\}$ output by the FU retrieves the cell shown in figure 15.

APR.Q:

APRET	
empty	$DEST_0$.Q
FR	$DEST_1$.Q
not used	$DEST_2$.Q

Figure 15

Notice that the APRET cell has as a constant operand, the name of the FREE cell. When it is enabled it produces the output packet $\{APRET, \sigma, FR, NULL, DEST_0.Q, DEST_1.Q, DEST_2.Q\}$ This packet in turn causes the FU to output packets $\{DEST_0.Q, RT_0.\sigma\}$, $\{FR.Q, RT_0.\sigma\}$, $\{DEST_1.Q, RT_1.\sigma\}$, $\{FR.Q, RT_1.\sigma\}$, ... Thus the RETurn cells receive as parameters the names of *both* of their destinations. When a RET operation packet is processed by the FU, both the named FREE cell and the return destination cell receive copies of the output. We see that output values are returned as before. However, processing a RET operation packet does not cause the FU to terminate the activation. Instead it is the firing of the FREE cell that instructs the FU to terminate an activation as discussed above.

3.6 Relocation Box Revisited

To simplify the previous discussion we have purposely oversimplified part of the operation of the machine. We had said that the relocation box upon receipt of a fetch packet would always pass the packet to the PMS with the suffix stripped off the cell name. This is incorrect. If the machine really operated in this fashion, then it could never retrieve cells that had been displaced from the IM into the PMS. There are a number of solutions to this problem. However, selection among them is impossible without a more detailed model of the implementation of the packet memory system and the cache mechanism. Thus a full discussion is beyond the scope of this paper. The particular solution described here was chosen for its simplicity, and should not be thought of as an "optimal" solution.

Upon receipt of a packet $\{\underline{fetch}, a.\sigma\}$ the relocation box

first checks if σ is a valid activation name. If it is not, the RB signals a runtime error; otherwise it issues *two* fetch requests to the PMS. One is for a cell with name α, and the other is for a cell with name $\alpha.\sigma$. The PMS will respond in one of several ways.

1. If the PMS returns no entry for either request, then the RB signals an error.

2. If the PMS returns a cell image for $\alpha.\sigma$, and no entry for α, then the RB signals an error

3. If the PMS returns a cell image for both, then the cell image returned in response to the request $\alpha.\sigma$ is passed back to the IM unaltered.

4. If the PMS returns a cell image for the request α, and no entry for the request $\alpha.\sigma$, then the cell image returned in response to the request α is passed back to the IM with its destination fields altered as previously discussed.

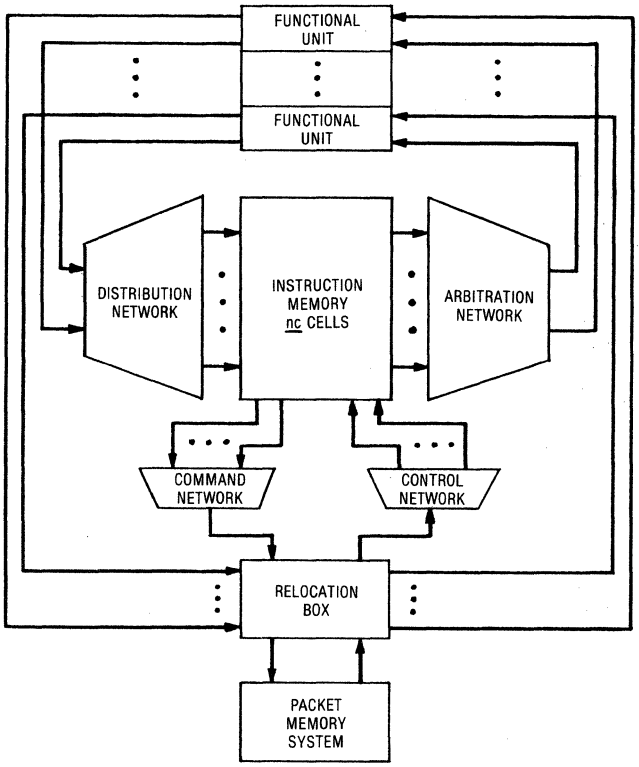
3.7 Names and Loading

Another attraction of this implementation scheme for APPLY is that it does away with the need for an elaborate linking loader for data flow programs. Consider for example, a data flow program consisting of several procedures that have been independently compiled. One may assume for the purposes of discussion that the cell names (and hence the contents of the destination fields of a cell) correspond in some direct way to the memory locations of the packet memory system (PMS). Thus when loading the component parts of the program (i.e. the procedures) into the PMS two things must be done. Assume for concreteness that a procedure is compiled into a linear block of cells numbered from 0 and that cell numbers are cell names. When loading a procedure, a number equal to the cell number into which the first cell of the procedure was loaded must be added to all destination fields of all cells that refer to cells that are part of the procedure. Then all external references in a procedure - that is entrance cell names in APPLY cells - must be set to the correct value. This value depends of course on the location into which the referenced procedure is loaded. Notice however, that return names need not be relocated since they are "constructed" at runtime. Thus no correcting of the destination addresses of an APRET cell need be done. Indeed, the RETurn cells which actually transfer return values to the invoking procedure are *not even part* of the invoked procedure. Thus the two tasks of loading - fixing (by adding an offset) of internal references is easy, and fixing of external references is greatly simplified. Only half of the job must be done before execution, since the entry but not the return points must be relocated.

The naming scheme provides a solution to establishing the "identity" of distinct activations amenable to efficient hardware implementation. Since a cell of an activation is uniquely identified by its name (assigned in the original compilation) and a single suffix, names are bounded in size (we assume of course that suffixes and simple names are of fixed maximum length).

3.8 One Last Change

Creation and returning to the "free pool" of activation names (i.e. suffixes) is probably not an appropriate activity for a FU. The primary reason is that one would like to have multiple FU's for processing of APPLY, RET, FREE and SEQ packets. Coordinating the creation and returning of suffixes to the free pool among several autonomous, asynchronously operating modules (FUs) is a messy task. It also introduces an overhead since the FUs coordination must take place through exchange of messages (packets) if we are to keep the overall machine structure consistent. Consequently, we propose the following modification to the scheme described above. We introduce an additional data path from each FU that processes applies etc. to the relocation box and from the relocation box to the those FU's. Thus the machine structure is:



Final Machine Structure Supporting Procedures
Figure 16

When j^{th} functional unit needs a new suffix, rather than generating it internally, it now sends the packet {NEWSUFFIX, j} to the relocation box. The RB responds by generating a new suffix name and sending the new name (as a packet over the new data path) to the requesting FU. When the FU wishes to free a suffix σ it sends the packet {FREESUFFIX, σ} to the RB which then returns σ to the pool of free suffix names. Thus in the new machine, assignment and freeing of activation names takes place at a central location, hence avoiding the problems of maintaining a distributed list of free suffix names. The choice of using the relocation box to perform these functions is somewhat arbitrary, the main idea being to centralize activation name management. Where it is done is not critical. Note that simply letting each apply FU manage its own activation names cannot work even if a FU's pool of names contains no elements in common with any of the others. The problem is that an additional mechanism must be provided to guarantee that a packet {FREE, σ} is routed to the functional unit that created σ. Because of this complication the above scheme is inferior to the central name allocator.

3.9 Procedure Variables

Finally, it should be observed that the naming scheme for establishing unique activations is in no way dependent on the fact that the name of procedure in an apply cell that is to be invoked is constant. (Its placement in an operand field of the APPLY cell was intentional.) Consequently, without adding any additional mechanisms to the schemes proposed, procedure variables can be handled. One merely has to compile the APPLY cells with empty entries where the entrance cell names for the

invoked procedure was. Of course some other actor of the program now must send a cell name corresponding to an entrance point of a procedure to the appropriate APPLY cell in order for it to become enabled.

One simple way to do this is to have each APPLY cell of the call mechanism receive the name of the procedure it is to invoke, rather than a node name of a particular entry point. When the i^{th} one is enabled it passes to the FU a packet of the form {APPLY$_p$, P, σ, a, NULL.Q, NULL.Q, NULL.Q}. The FU "knows" from the opcode and the first two arguments, that processing of this packet is supposed to send the i^{th} argument to the σ^{th} activation of procedure P. The FU can either

1. Look up in a table set up at loading time the name of the cell that is the i^{th} entry point of procedure P.

or

2. With a suitable compiler convention, ***compute*** the name of the cell given i and P.

Though either approach works, the former has two advantages. First, since the call structure is set up at compile time, the number of inputs and outputs to the apply mechanism is known. This information could be incorporated in the encoding of the APPLY cells and also in the table. This would allow the FU to do a runtime check to see if the named procedure has the appropriate number of input and outputs. Second, using a table allows the FU to check if procedure P was defined.

3.10 Conclusion

We have presented here several schemes of increasing capability for implementing procedure application in a large class of data flow processors. All of the schemes implemented the immediate copy rule - that is, a data flow program with an apply actor is semantically equivalent to one where the APPLY has been replaced by the program graph of the procedure it invokes. This effect can only be achieved at runtime since recursive procedures and procedure variables are allowed. The schemes presented were efficient in the sense that the overhead in terms of the number of packets required to set up and terminate an activation was small. In addition only the pieces of the invoked procedure that were active were brought into the instruction memory. This helps economize the use of instruction memory space, especially in the case of data flow procedures with conditional components.

This scheme was designed to be quite general, and minor modifications to it can achieve very different behaviour. For example:

1.
   a) Forcing SEQ to wait for all its inputs before firing
   b) Altering the compiler so that the FREE cell of a return mechanism receives it values from the same sources as the RETurn cells

changes the invocation rule to DCR.

2. We restricted programs to have only one outstanding procedure activation at each call site (section 2.2). If we make some simple modifications, the scheme will work in a DFM with acknowledgement of packets [8] and this restriction can be eliminated. Another minor modification then allows the machine to handle ***streams*** as input and output data types [27]. The details involve no major changes, but are beyond the scope of this paper.

The mechanism for procedure implementation outlined here should be regarded as a ***scheme***, and not a particular method. The things that are central to the approach are the ideas of

1. Creation of a ***virtual cell name space***, using a hierarchical associative store.

2. Creation and separation of different procedure instances through the ***runtime renaming*** of the cells that store the (encoding of the) procedure instance.

3. ***Selective copying*** of the parts of procedures as they become active.

4. Keeping the amount of ***state information*** that is necessary to characterize a procedure ***instance bounded*** and encoded in the cells by imposing ***syntactic restrictions*** on the base language.

BIBLIOGRAPHY

1. Amerasinghe, S. N. ***The Handleing of Procedure Variables in a Base Language***. S.M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA. September 1972.

2. Arvind, and Gostelow, K. ***A New Interpreter for Data Flow and Its Implications for Computer Architecture***. UCI Technical Report #72, Department of Information and Computer Science, University of California - Irvine, Irvine, CA. October 1975.

3. Dennis, J. B., and Fosseen, J. B. ***Introduction to Data Flow Schemas***. CSG Memo 81, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 1973.

4. Dennis, J. B, and Misunas, D. P. "A Preliminary Architecture for a Basic Data Flow Processor". ***Proceedings Second Annual Symposium on Computer Architecture***, January 1974.

5. Dennis, J. B. "First Version of a Data Flow Procedure Language". ***Lecture Notes in Computer Science, 19***, G. Goos and J. Hartmanis, Editors, Springer-Verlag, New York, NY, 1974.

6. Dennis, J. B. "Packet Memory Systems Architecture". ***Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing***, August 1975.

7. Dennis, J. B. and D. P. Misunas "A Computer Architecture for Highly Parallel Signal Processing". ***Proceedings of the ACM 1974 National Conference***, ACM, New York, NY, November 1974.

8. Dennis, J. B., Misunas, D. P., and Leung, C. K. ***A Highly Parallel Processor Based on the Data Flow Concept***. MAC TR134, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, August 1974.

9. Flynn, M. "Some Computer Organizations and Their Effectiveness". ***IEEE Transactions on Computers***, September 1972.

10. Hack, M. ***Analysis of Production Schemata***. MAC TR-94, Department of Electircal Engineering and Computer Science, MIT, Cambridge, MA, February 1972.

11. Karp, R. M. and R. E. Miller "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing". ***SIAM Journal of Applied Mathematics, 14***, November 1966.

12. Keller, R. M. "Look-Ahead Processors". *ACM Computing Surveys*, vol 7, number 4, December 1975.

13. Kosinski, P. R. "A Data Flow Language for Operating Systems Programming". *Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting*, SIGPLAN Notices, 8,9, September 1973.

14. Miller, R. E., and Cocke, J. *Configurable Computers: a New Class of General Purpose Machines*, Report RC 3897, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., June 1972.

15. Leung, C. K. *Formal Properties of Well-Formed Data Flow Schemas*. MAC Technical Memorandum 66, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA. June 1972.

16. Miranker, G. S. *Implementation Schemes for Data Flow Procedures*. CSG memo 138, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, May 1976.

17. Miranker, G. S. *Proving Packet Communications Architectures Correct*. CSG memo-143, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 1976.

18. Miranker, G. S. *An Approach For Proving Packet Communications Architectures Correct*. CSG note-27, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 1976.

19. Miranker, G. S. *Design and Correctness of a Data Flow Procedure Mechanism*. S.M. Thesis Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, January 1976.

20. Misunas, D. P. *A Computer Architecture for Data Flow Computation*. S.M. Thesis Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, July 1975.

21. Patil, S. S. "Closure Properties of Interconnections of Determinate Systems". *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, ACM, New York, 1970.

22. Plas, A., et. al. "LAU System Architecture: A Parallel Data-Driven Processor Based on Single Assignment". *Proceedings of the 1976 Sagamore Computer Conference on Parallel Processing*, August 1976.

23. Rodriguez, J. E. *A Graph Model for Parallel Computation*. TR-64, Project MAC, MIT, Cambridge MA, September 1969.

24. Rumbaugh, J. *A Parallel Asynchronous Computer Architecture For Data Flow Programs*. MAC Technical Memorandum 150, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, May 1975.

25. Slotnick, D. "Unconventional Systems". *Proceedings AFIPS Spring Joint Computer Conference*, 1967.

26. Thurber, K., and Wald, L. "Associative and Parallel Processors". *Computing Surveys*, vol. 7 number 4, December 1975.

27. Weng, K. *Stream Oriented Computation in Recursive Data Flow Schemas*. MAC Technical Memorandum 68, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, October 1975.

PIPELINING, PARALLELISM AND ASYNCHRONISM IN THE LAU SYSTEM

J.C. SYRE, D. COMTE, N. HIFDI
O.N.E.R.A - C.E.R.T., Department of Computer Science
B.P. 40-25, 31055 TOULOUSE CEDEX, FRANCE.

Abstract : This paper presents the hardware specifications and figures of a parallel multiprocessor system, currently under construction. The LAU system philosophy comes from the Single Assignment software concept and data-directed expression of problems. Up to now, a high level language, a machine language and a paper machine have been defined. A compiler and a simulator gave us significant results enough to start the actual implementation of a prototype processor. The paper focuses on the advantages of data-directed control mechanisms and register-independent instruction/data formats throughout the different parts of the processor : maximally efficient pipelining, full parallelism and asynchronism will be shown at the different stages of an instruction execution, with accompanying figures in speed, complexity and cost of their implementation.

Introduction

This paper presents the hardware specifications of a parallel multi-processor architecture, called "LAU system", whose control and sequencing mechanisms are directed by the computation of the program data. The data-driven control leads to some interesting properties and advantages in the design of the system, in terms of "independent" pipelining, parallelism and asynchronism.

Since 1973, when the LAU Project started [1, 2] , the "Single Assignment" software concept [3, 4] has led to the definition of a high level language, a machine language and a paper machine. A compiler and a simulator gave us significant results enough to allow the actual implementation of the system. Much like other principles, mainly stressed by J.B. Dennis, [5, 6, 7] , all control mechanisms are provided by the "readyness" of data. In short, a statement in the (high-level- or machine -) language is executable as soon as its operands have their 'unique' values computed. Instruction sequencing is readily achieved by the data flow itself, and no longer by the relative order of statements. The single assignment rule, combined with these properties, guarantees program determinism, and leads to a "maximal parallelism" together with some interesting hardware design advantages. To be a little more acquainted with the LAU system philosophy, the reader is invited to follow an instruction stream, from its generation by the compiler to its execution in the different parts of the machine. In the first section of this paper, we present the high level software and hardware characteristics of the LAU system, some of them being still open problems. The second section outlines the architecture of one processor and the specific mechanisms implementing the data directed flow of control. The third section gives more details on each of the functional parts of a processor. At each step will be given some figures (technological choices, cycle time, complexity) and the design properties which take full account of parallelism, concurrency and asynchronism derived from the single assignment and data-driven sequencing approach.

High levels system principles

The single assignment approach

Beside methods trying to discover parallelism in programs and others which transform sequential code into pseudo-parallel executable sequences, some radically different ways have emerged which start from the problem statement itself. The parallel program schemata and the data flow approach fall in this category. The single assignment rule is another concept [3, 4] that applies from the problem analysis to its execution on a parallel processor system. This rule states that :

> an object may be assigned a value at most once during program execution

Implications of single assignment are immediate :

1. a statement, say X = A + B, is "executable" as soon as its operands (A, B) are computed
2. it may be executed at any time later, in a way totally independent from its location in the program.

The parallelism expressed by the single assignment rule (S.A. rule) corresponds to the inherent parallelism as stated in the problem. Due to the S.A. rule, instructions are guaranteed to deal with the unique values of operands, which insures program determinacy.

However, executable instructions may be performed in any order; this is a definite advantage for an implementation which will not be concerned with instruction sequencing at the hardware level.

LAU System level 0 : the general system (paper made only)

An S.A. program may be viewed as a collection of tasks whose activations are data-driven : when input data are evaluated, a task supervisor may decide to route the task onto an idle processor. Outputs of the task allow the detection of the task termination and the activation of other tasks. The following software/hardware diagram describes the LAU system at this level :

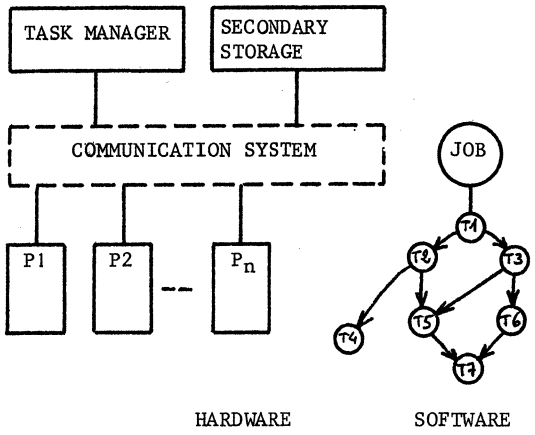


Figure 1 : level 0 LAU system architecture

Independent tasks may run concurrently on different processors; the task manager has only to keep track of the task execution by examining the data directed tree produced at compile-time. Notice that this scheme might be extended to a set of different jobs, whose task trees would be controlled by the task manager. A processor P_i, enters the active state when loaded by the data and instructions of a task. As data values are unique, the execution time is irrelevant for determinacy. When task outputs are computed, the processor informs the task manager which stores data produced by the task, and checks for new tasks to be activated. This higher level of the LAU system has not yet been implemented nor studied in full detail. Open problems are still to be evaluated : the expression of problems by single assignment data flow tasks, the compilation mechanisms that could break a program into tasks suitable in the memory space of a processor, the workload on the communication system and the task manager.

- LAU system level 1 : the LAU processor -

The high level language :

From now on, we come to the real things with the software and hardware at the processor level.

As for the software part of the system, a high level language has been defined. Statements in the language are syntactically classical, but semantically different from usual ones in sequential languages : every statement is an assignment statement, and produces values for each of the objects to be computed within its scope. In the following program :

```
S1 INPUT (INMATX);
S2 N = 128
S3 EXPAND/16 I = 1, N :
    LOCAL : TEMP1; TEMP2;
    TEMP1 = INMATX (I) - INMATX (I - 1)
    TEMP2 = INMATX (I) + INMATX (I - 1)
    MAT(I) = TEMP1 * TEMP2;
  END EXPAND;
S4 MATS = VSUM (MAT FROM 1 TO N);
S5 OUTPUT (MATS);
```

S1 is the unique statement assigning INMATX, while S3 is the only assignment statement of MAT. To be executable, S3 has to wait for the computation of N, INMATX, i.e. the completion of S1 and S2. Notice that S1 and S2 are "ready" instructions, which will be generated by the compiler. The first block will compute MAT (1), MAT (17), ..., MAT(113) the sixteenth one MAT (16), MAT (32), ..., MAT (128). Once activated, each block runs concurrently with its 15 brothers, and parallelism will range between 16 and 32. When S3 is terminated (i.e when MAT is computed), S4 will be activated, and finally S5.

Open problems in the high level language include some properties that can be found in other languages : very high level data structures, creation of user defined types. The reason is that we wanted a compiler as soon and as efficient as possible.

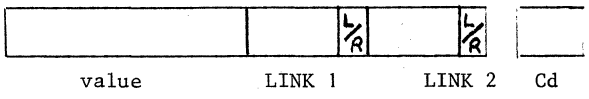
The LAU machine language

It is a single assignment language, too, and must imbed the data flow control expressed at the upper level:

- an instruction may enter its execution cycle as soon as its operands are ready
- the result object is the only link between the instruction computing it and those using it as an operand.

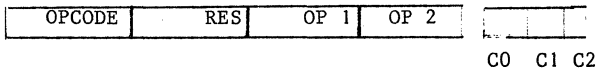
Ready instructions should run independently and free from hardware constraints (other than those imposed by the technology at the logic gate level), such as the organization and the number of processing units, management of requests, register assignment and control, data organization in memory. All these problems are totally irrelevant in our high level approach. The definition of the machine language has led to the following formats :

- Data format :



LINK 1 and LINK 2 refer to instruction addresses using the data as an operand. Cd is a control tag bits indicating the computation of the data.

- Instruction format (for simple computational instructions)



Further details on the machine language can be found in [2,9] . These features imply the following comments :

- the instruction format is large (64 + 2 bits) but corresponds to approximately three classical machine instructions (LOAD OP1, ADD OP2, STO RES) An instruction will occupy one memory word. The data format is larger than a classical one which does not contain instruction links.

- the instruction has a three-address format, yelding two interesting properties for its execution and the design of the machine : an instruction may be executed in any one of the processing units of the machine, and, more, the machine may be built with any number of processing units that can be increased, decreased, or support degradation without program change at the compile time.

THus, the machine language preserves the parallelism expressed at the high level programming stage, and does not introduce hardware constraints for the definition of the machine.

The LAU processor architecture

The LAU processor consists of three functional boxes, interconnected by data and control busses as follows :

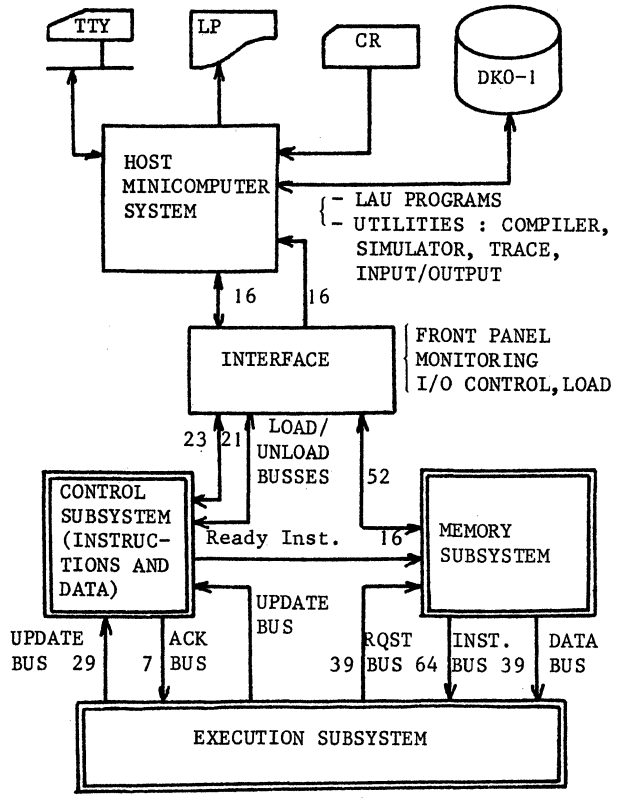


Figure 2 - Functional diagram of an LAU processor

The Control sub-system is the truly original part of the processor : the Von Neuman program counter is replaced by two memories; the instruction control memory (ICM) and the data control memory (DCM). ICM handles the three tags bits Co, C1, C2 of the program instruction, while DCM will take care of the Cd tag bit of program data.

The Memory Subsystem manages input requests and delivers memory operations to the other units. Special interest will be given to this box that may be the bottleneck of the processor.

The Execution subsystem consists of N elementary processors connected on various busses. Each processor is totally independent from its neighbours, except for requests to external busses.

We find it useful to explain the lower levels of the LAU processor by following the trip of machine instructions throughout the different parts of the system. The machine instruction (let's call it MI) has the following initial assignment :

Co C1 C2				
1 0 1	+	RES	OPER	2

RES and OPER are data addresses
2 is an immediate operand.

Co C1 C2 are located in the Instruction Control Memory at the same address as MI in main memory. Co has been set to 1 at compile time, and means that MI is not nested in a control instruction. C1 denotes that the object OPER is not yet computed at this time, and C2 = 1 corresponds to the constant 2 in the second operand field.

M I Magic Mystery Tour

Let us come now to the micro-functional level of the design, and follow M I.

Extracting M I address from the Control Unit

The machine instruction, computing OPER, is being performed in some processing unit. It sends the Control Unit a signal indicating the computation done.

The UPDATE module will set the C1 bit to 1 at M I address. Now M I is virtually executable. In parallel, with UPDATE an Instruction Fetch Processor (IFP) is permanently looking for "111" configurations in ICM (simulating a not yet available associative mechanism). IFP examines only the ICM portion where there is a chance to find out ready instructions, and it is capable of sending their addresses every 120 ns. Thus M I will eventually be checked as a ready instruction. All things happening elsewhere in the machine will never affect M I, due to the unique just computed value of OPER guaranteed not to change. IFP sends M I address to either a FIFO

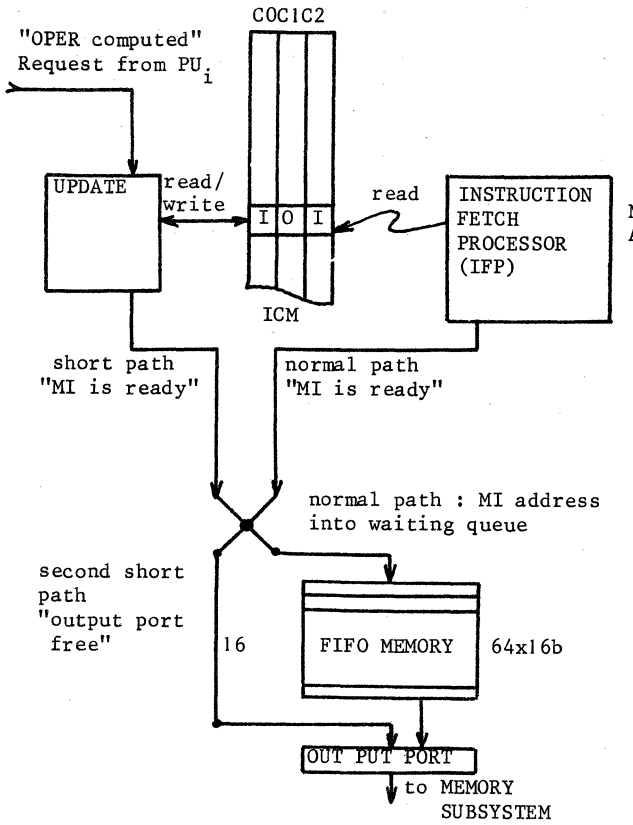


Figure 3 - Instruction Control Memory and Instruction Fetch.

file buffer, or directly, to the Instruction Output Port of the Control Unit, using a short path, thus eliminating systematic buffering of requests. The File is a 64 x 16 bit FIFO stack and may deliver an instruction address every 120 ns. When full, meaning 64 instructions waiting for execution, the file inhibits the Instruction Fetch Processor. Notice that when this occurs, IFP will not make access to ICM : this will speed up the UPDATE device, and consequently, the servicing of processing units which will become idle sooner, thus will accept new instructions sooner, and so accelerating requests for ready instructions located in the file. Beside that, the short path is possible because ready instructions are fully independent from each other, and ordering them is irrelevant. This property allows a straightforward implementation of the instruction fetch mechanisms. Here parallelism is not perturbated by logic constraints but only at level of accesses to the Instruction Control Memory.

Reading M I in the Memory Subsystem

The Instruction Output Port of the Control Unit is one of the three possible inputs to the Memory Subsystem, organized as follows :

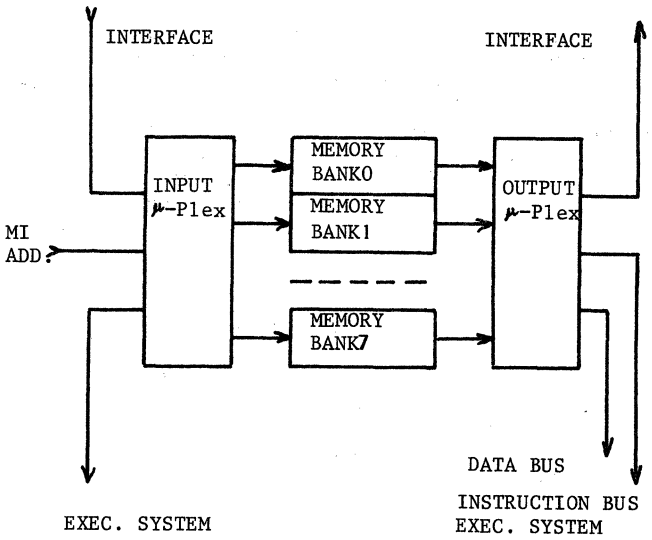


Figure 4 - Functional organization of memory

The input policy gives priority to the execution subsystem (accesses to data operands from processing units), which may send requests every 60 ns. When there is a gap in this flow, M I address is processed by the Memory Input Manager, whose job is to translate the request into a standard memory request (one of 16) and then to drive the request to the corresponding Memory Bank, and it lasts 60 ns to do so. The activated Memory Bank stores the request into a buffer of waiting operations. Still here ordering is not relevant and a LIFO or FIFO scheme may be adopted. Overflow in the buffer is prevented by the following :

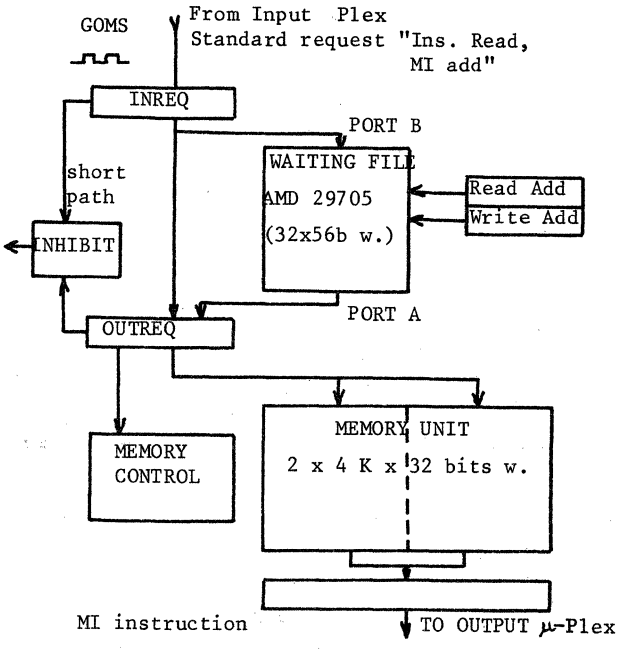


Figure 5 - Micro-functional description of a Memory bank

The Memory unit which stores program and data instructions, is organized in two independent 4 K x 32 bits memory blocks allowing read and write operations on each block in one 480 ns cycle.

Thus, MI is pushed on to the buffer, or directly conveyed to the request register of the Memory Unit. An instruction read operation is performed, the result stored into the Output Memory Latch 8 minor cycles after the cycle initiation. The Output Memory Manager looks for requests coming from the 8 Memory banks, and will take MI within the next 480 ns. This request is decoded , translated and sent on to the appropriate output bus (here, the Instruction Bus).

In the Memory subsystem, parallelism is achieved by the interleaved organization of the Memory : 8 requests every 480 ns can be completed making a 60 ns - minor cycle (and, for those who like large numbers, 1 066 10^9 bits/second, parity bits not included). Asynchronism can be seen at the level of the acceptance of input memory requests : read, write, instruction read operations are not to be ordered to guarantee a correct program execution. However, priority and clock synchronization are necessary at the logic level, to deal with all requests without loss of information.

M I enters the Execution Subsystem

The Instruction Bus is connected to one of the six controlers managing the execution subsystem.

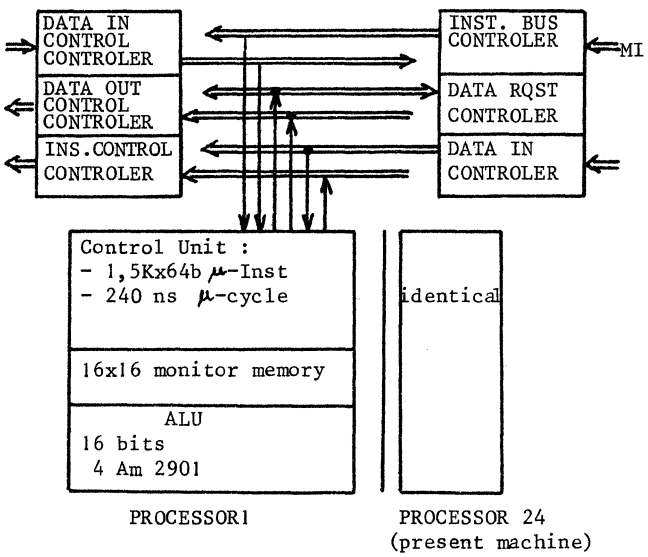


Figure 6 - Organization of the Execution Subsystem

Any number of elementary processors may be attached to the six busses whose functions are given in the diagram. The instruction bus controler is composed of input file containing instructions to be processed, and once more, a short path permitting an instruction to be driven directly onto the internal instruction bus. The instruction controler checks for idle processors and allocates the bus to the first one which will accept M I; M I is stored into the Instruction Register and execution starts. An elementary processor is built from AMD 2900 series bit-slice microprocessors. This 16-bit data processor has no local memory (other than the 16 internal registers), but contains a special memory used for monitoring and workload evaluation , and runs microprograms corresponding to the whole instruction set. An extensible micro-assembler has been developped and is used to generate 64-bit micro-instructions composed of 20 fields. Some of these fields are dedicated to monitoring and spying operations that can be performed in parallel, thus not affecting the actual execution of instructions. Once in the processor, M I is interpreted as follows :

- Decode CODOP
- Make a data operand access request, corresponding to OPER on the data out request bus.

- As M I uses an immediate value as second operand, the constant 2 is prepared in a register

- When the request is performed (it may occur at any time later, and depends on the load of the data out controler, the memory input manager, the Memory Block interested, etc...) The result (i.e OPER value) comes back to the data In Request Controler which signals it to the processor on which M I is being performed.

Then operation proceeds : RES is computed, and special control primitives are performed :

- Send a write-read request to memory at address RES : the value of RES will be stored in, and the link fields will be brought back to the processor

- in parallel, send a write request to Data Control Memory : the Cd tag bit will be set to 1, and the Data Control Unit will send back an acknowledgement.

- Once the link fields are present in the processor, send update requests to the Instruction Control Memory. Each link field consists of an instruction address using RES as an operand and one bit indicating a left or a right operand. The C1 or C2 bits will be set to one and eventually will make new "111" configurations in the ICM.

There are other devices in the Data Control/ Unit which run in parallel with the Update

processor. They implement the control primitives (creation of data descriptors, descriptor checking single assignment verification) that we shall not discuss here.

In the execution subsystem :

- processors are identical and independent from each other. They perform instructions given by the Instruction Bus controler in parallel, and work asynchronously. They are managed by the Controlers for their requests to the outside. The number of processors can be extended; though not yet much studied, it should be interesting to discuss the capabilities of fault tolerant or degradated processing in this part of the system.

- pipelining of requests makes no problem and may be short-circuited in some cases, thus speeding up the actual throughput.

- within a processor, the microprogram may send simultaneous requests to the output busses However, every request has to be acknowledged before the processor may be considered as idle.

Simulation results showed that the major problems lie in the memory subsystem, and not in the execution subsystem. The prototype will thus be built with 16-32 elementary processors, with a 240 ns microinstruction cycle.

Its connection to a host system will allow the use of peripheral devices. In particular, a library of LAU programs, located on a disk will be managed by the interface unit which will also give facilities for tracing, monitoring and data input/output.

Conclusion

The LAU system is an overall hardware/software system which makes use of data-driven sequencing principles at all levels : a high level language, implemented and a general multi-processor architecture have been studied. A machine language and a compiler producing executable code from user's programs have been defined, together with a data driven processor structure. The processor is composed of any number of processing units running in parallel, and a special control unit implementing the data driven mechanisms. Parallelism is securely achieved by the independence of data flows in the machine, due to the single assignment rule. Use of pipelined mechanisms allows a bufferization of requests when needed and a better throughput in the system when saturation occurs in some functional part of it. However, the pipeline mechanisms can always been short-circuited in case of low workload, without any trouble for the determinism of computation. Asynchronism is the main feature of the system : instruction fetch, memory operations for any number of processors, instruction execution and control updating functions are achieved concurrently. Only logic constraints may sometimes affect the operation in the system. These characteristics allowed us to be interested in the actual problem of implementation at the chip level and to have a straightforward design period. The LAU system waffle 1 is expected to be operational by the end of 1978 and application programs will be evaluated at that time.

References

1 J.M. NICOLAS, J.C. SYRE, TEAU 1-4 : Techniques et exploitation de l'assignation unique, Rapports SESORI 73-038, 1972, 1973

2 D. COMTE, G. DURRIEU, O. GELLY, A. PLAS, J.C. SYRE : TEAU 5-8, Rapports SESORI 74-167, 1975, 1976

3 L.G. TESLER and H.J. ENEA, : A language design for concurrent processing, Proc AFIPS, SJCC 68, Vol 32, p.403-408, 1968

4 D.D. CHAMBERLIN, Parallel implementation of a single assignment language, Ph D. Thesis, Stanford U. TR 19, Jan. 71

5 J.B. DENNIS, First version of a data flow procedure language, MIT, Lab. for Computer Science, MIT/LCS/TM-61, May 75

6 J.B. DENNIS, D.P. MISUNAS, : Design of a highly parallel computer for parallel processing application, MIT, Project MAC, TR 101, 1974.

7 J.E. RUMBAUGH, : A parallel asynchronous computer architecture for data flow programs, Ph D. Thesis, MIT, MAC-TR-150 May 75

8 ARVIND and K. GOSTELOW : A new interpreter for data flow schemas and its implications for computer architecture, TR 72, ICS, University of California, Irvine, Oct 75

9 A. PLAS and al : LAU system architecture, a parallel data driven processor based on single assignment, 1976 International Conference on parallel processing, Aug 76, pp. 293-302, IEEE n° 76CH1127-0 C

10 D. COMTE, G. DURRIEU, O. GELLY, A. PLAS J.C. SYRE : TEAU 9 (Vol 1,7) Final reports Contract SESORI 74 167, Sept. 1976 (vol. 7 is a summary in English)

11 Advanced Micro devices, The Am 2900 Family Data Book, Sunnyvale, California 1976

12 Texas Instruments, the TTL Data Book for Design Engineers, 1976.

A DISTRIBUTED COMPUTER SYSTEM USING A DATA FLOW APPROACH*

by

Michael A. Schroeder
Robert A. Meyer

Electrical and Computer Engineering Department
Clarkson College of Technology
Potsdam, New York 13676

Summary

This paper presents the design of a highly asynchronous distributed computer system which employs a data flow approach for handling parallelism in programs.

The proposed system is a hierarchically connected network of modules, each of which operates in an asynchronous manner. The main components of the system are a set of Computation Activation Processor (CAP) Modules, each executing an individually assigned procedure on the input data it receives. A Scheduler module regulates the sequencing and dispatching of all CAP procedures and data flow instruction packets.

The data flow approach is based on direct initiation of each operation simply by the presence of the required operand values. In a 1974 paper Dennis [1] first proposed a very basic version of a data flow language in which instruction execution was limited only by the data dependencies of the program. Dennis and Misunas [2] did preliminary work into the design of a computer based on this language. Rumbaugh [3] has expanded and improved the earlier version of the data flow language proposed by Dennis and has developed a multiprocessor architecture consisting of N identical Activation Processors. Each processor is capable of executing in a pipeline manner several data flow instructions at a time.

The system we propose is a new, simpler implementation based on their work. We have partitioned our system into a number of asynchronously operating modules, each of which consists of a controlling processor and its associated memory structures. Our system consists of five major module types, namely an Interface Module, Assignment Module, Collection Module, Scheduler Module, and a set of N Computation Activation Processor (CAP) Modules. These five module types are interconnected to produce a system in which the total functioning of the system is spread throughout the various modules, thereby realizing a distributed architecture system. Each module is only responsible for executing a specific, preassigned portion of the total system workload, with each module functioning in an independent manner. This approach provides a far superior method of system functioning than most contemporary systems since the distributed architecture concentrates the concurrency problems inherent in parallel computer systems in the module interfaces. Our module interconnection is accomplished through the use of a set of similarly functioning bus-to-queue interfaces. These interfaces reduce the module interaction to the problem of putting items in and removing items from the queues in a non-conflicting manner.

The Interface Module accepts programs, procedures, and data input from the outside world in a high level language for later entry into the main system.

The Scheduler Module has three main areas of responsibility: 1) accepting the transformed compilation structures from the Interface Module, 2) dispatching completed operand packets to the Assignment Module, and 3) retrieving completed result packets from the Collection Module.

Each Computation Activation Processor Module contains a processor capable of performing a specific operation stored in its procedure store. Operand packets are removed one at a time from the CAP's operand packet queue, placed in its working store, and then the assigned procedure is performed on the packet. After the computation is completed, a result packet is formed and placed in the CAP's results packet queue where it awaits collection.

The distribution of the processors and memory into modules, which perform individual portions of the system's total functioning, reduces the problems of memory contention and processor synchronization. The consistent way in which the interfacing and synchronization of the various modules is handled helps to simplify the problems involved in determining system failures. Since the system design permits us to isolate faults to one or more modules or interfaces, we are easily able to correct the problem by simply replacing the faulty module.

As a final comment, we believe that our system design will allow us to implement many hardware aspects of the system with the use of low cost, currently available hardware. The individual processors may be realized using microprocessors, and the various memory structures are easily implemented using LSI techniques.

*This work was supported in part through the Post Doctoral Program by the U.S Air Force under contract F30602-75-C-0082.

References

[1] Dennis, J.B., "First Version of a Data Flow Procedure Language", Project MAC, MIT Cambridge, Mass., 1974.

[2] Dennis, J.B. & Misunas, D.P., "A Computer Architecture for Highly Parallel Signal Processing", Proceedings of the ACM 1974 National Conference, ACM, New York, pg. 402-409.

[3] Rumbaugh, J.E., "A Parallel Asynchronous Computer Architecture For Data Flow Programs", Ph.D. Thesis, MIT Project MAC, 1975.

A MULTILAYERED DATA FLOW COMPUTER ARCHITECTURE

John Gurd and Ian Watson
Department of Computer Science
University of Manchester
Manchester M13 9PL , England

Summary

It is difficult to take full advantage of the parallelism in a problem when designing a computer containing many processors. Consequently, several new types of parallel computer architecture have been proposed with the objective of allowing efficient exploitation of problem parallelism. The class of configurable computers [1], including data flow machines (e.g. [2,3]), has evoked considerable interest because of its general applicability and inherently parallel nature. More recently, the possibility of interconnecting very large numbers of microprocessors has been suggested, and data flow configurations have been proposed in this vein [4]. This paper outlines a basic ring-structured data flow architecture, and proposes an extended version (using connected, multiple layers of simple rings) in which an arbitrarily large number of processing units may operate.

The basis of all data flow systems is a graphical computational schema (e.g. [3-7]). The schema used as a base for the ring-structured system is described in detail in reference [8]. It differs from other data flow schemata in that tokens on arcs need not be maintained in first-in-first-out order. As a consequence, all tokens in a re-entrant graph must be labelled to ensure that they are distinct. A label comprises three fields which distinguish three sources of reentrancy, namely: (i) parallel data structure (array) ; (ii) iteration (sequence) ; and (iii) recursion. Specialised nodes can adjust all or part of a label when entering or leaving a reentrant section of a graph. The schema also restricts the maximum number of input (and output) arcs at a node to 2.

The single ring architecture contains a processing area and an assembler together with several stores.

The instruction store contains a (read-only) linear representation of the computational graph : each instruction defines a nodal function and up to two possible destination instructions for the results (i.e. the output arcs).

The matching store is associatively accessed and is used to hold those results (i.e. tokens) which cannot proceed to the next instruction because the necessary firing condition has not been met (i.e. because a second input result is not yet available). The name used for association consists of the destination of the token together with its label. This uniquely identifies every result in the system provided that labels are distinct.

The result queue acts as a buffer for named results between their leaving the processing area and their being dealt with by the assembler. Although called a queue, this store need not be strictly first-in-first-out.

The assembler takes results one-at-a-time from the head of the result queue and tries to form for each an executable instruction by accessing the instruction store (to find the next function and destinations) and (if necessary) the matching store (to find a matching input result for a two input function). If the last action is unsuccessful, the incoming result is saved in the matching store to await its partner : otherwise the ensuing executable instruction (i.e. function, two operands, common label and destinations) is sent to the processing area to be executed.

An input/output switch at the output of the processing area enables results to be transmitted into or out of the ring.

In the multilayered architecture, the input/output switch is extended to become a switching network connecting the outputs of the processing areas to the inputs of the result queues in the system. The switching network resembles a sorting network [9] which directs each result according to some part of its name.

Both switching network and individual rings can be constructed as pipelines : any number of rings may be connected without reducing the pipeline beat period.

References

[1] R.E.Miller and J.Cocke, "Configurable Computers : A New Class of General Purpose Machines", Lecture Notes in Computer Science, vol. 5, Springer-Verlag, (1974), p 285

[2] J.B.Dennis and D.P.Misunas , "A Preliminary Architecture for a Basic Data Flow Processor", Proc. Second IEEE Annual Symposium on Computer Architecture, (Jan. 1975), p 126

[3] J.E.Rumbaugh, "A Data Flow Multiprocessor", IEEE Trans. Comp., vol. C-26 no.2, (Feb. 1977), p 138

[4] H.Arvind and K.P.Gostelow, "A Computer Capable of Exchanging Processors for Time", Proc. IFIP Congress, (Aug. 1977)

[5] R.M.Karp and R.E.Miller, "Properties of a Model for Parallel Computations : Determinacy, Termination, Queueing", SIAM J. Appl. Math., vol. 14, (Nov. 1966), p 1360

[6] D.A.Adams, "A Model for Parallel Computations", in Hobbs (ed.), Parallel Processor Systems, Technologies and Applications, Spartan Books, (1970), p 311

[7] J.B.Dennis, "First Version of a Data Flow Procedure Language", Lecture Notes in Computer Science, vol. 19, Springer-Verlag, (1974), p 362

[8] J.R.Gurd, P.C.Treleaven and I.Watson, "A Data Flow Computer Architecture" , Draft paper, University of Manchester, (Aug. 1977)

[9] K.E.Batcher, "Sorting Networks and Their Applications", AFIPS SJCC, vol. 32, (1968), p 307

ON THE OPTIMALITY OF FIRST-FIT AND LEVEL ALGORITHMS FOR PARALLEL MACHINE ASSIGNMENT AND SEQUENCING

E.G. Coffman, Jr.
Dept. of Electrical Engineering
and Computer Science
Columbia University
New York, N.Y. 10027

Joseph Y-T. Leung
Dept. of Computer Science
Virginia Polytechnic
Institute and State University

Donald Slutz
IBM Research
San Jose, Calif,

Abstract -- The application of one-dimensional bin-packing to the problems of efficient allocation of parallel resources is studied. Two classes of fast approximation algorithms are considered for these problems, which are known to be NP-complete. Known bounds on performance are briefly surveyed; some new results are then derived for broad classes of problems for which the approximate algorithms are in fact optional.

I. Introduction

In the implementation of parallel processing, one encounters the sequencing of tasks on multiple processors and the allocation of information to parallel storage units as two common design decisions for which specific problems are defined and efficient solutions sought. With the nature of the units being assigned, allocated or sequenced being known, these problems are normally instances of general bin-packing problems. In the sequel we shall define the basic model of bin-packing, couch the above sequencing and allocation problems in these terms, and then explore two classes of approximation algorithms that have been applied to these problems. In this study of the so-called first-fit and level algorithms we shall examine questions of optimality and performance bounds relative to optimization algorithms. It will be seen that the first-fit algorithms appear more promising in the applications considered.

II. One-Dimensional Bin-Packing

Informally, we are given a collection of m "bins" $B_1, \ldots, B_m$ of equal capacity, c, and a set P of n "pieces" $p_1, \ldots, p_n$ each of which has a size not exceeding the bin capacity. The piece sizes as well as names will be given by p_i, $1 \le i \le n$. The object of any bin-packing problem is to pack the pieces into the bins so as to optimize some given measure of the packing. In applying this model to processor sequencing and storage allocation we have the following term associations (the first term applies to sequencing/scheduling while the second applies to storage allocation):

Bin: processor, storage unit (e.g. cylinder)

Piece: task, record

Packing: schedule, allocation

Capacity: deadline, capacity

We consider the following problems.

P1. Assume m is as large as necessary ($m \ge n$ will always suffice), and a fixed capacity. The object is to minimize the number N of bins required to pack P.

This problem as well as those below have many applications outside of computer science. In computer operation P1 concerns the problems of (1) minimizing the number of units (cylinders, pages, etc.) necessary for the storage of a collection of variable size records, and (2) minimizing the number of processors needed to complete all tasks by a given deadline common to all tasks.

P2. Assume m is fixed. The problem is to minimize c such that all pieces can be packed.

P2 is the classical problem of sequencing to minimize make-span, or the design problem of finding a capacity such that all records can be placed in a fixed set of equal capacity storage units.

P3. With m and c fixed maximize the number n of pieces (i.e. the subset of P) packed in the bins.

P3 is clearly the problem of maximizing the number of tasks finished by some deadline (c), or the number of records stored in a fixed collection of storage units.

P4. With m fixed and c unconstrained pack the pieces so as to minimize $\sum_{i=1}^{m} \ell_i^2$ where ℓ_i is the level* of B_i.

P4 has arisen in storage allocation applications in which the object is to minimize average access times. In these problems the records are of equal size; the "piece sizes" correspond to stationary record access probabilities.

Although P1-P4 by no means exhaust all problems of the bin-packing type they are the principal ones to which the simple approximation algorithms that we discuss have been successfully applied. For general parameter values each of P1-P4 is NP-complete; this fact has been a prime motivation for studying fast heuristics. Those having received most attention fall into two classes, heuristics in both classes assigning pieces one at a time as they are drawn in sequence

*The level of B_i is the sum of the piece sizes in B_i.

from a given list L.

III. Level Algorithms

These algorithms are distinguished by the assignment criterion: The next piece to be assigned is packed into the bin currently of lowest level. The largest-first (LF) algorithm is a level algorithm that initially puts L into non-increasing order (as with the first-fit algorithms below, it is only the assumed ordering of L that distinguishes two different level algorithms). The LF rule has been applied to P2 and P4; it is illustrated below for P2.

L=(1/2, 2/5, 3/8, 1/3, 5/16, 5/16, 1/4, 1/5, 1/6, 3/20)

m = 3

47/48	17/16	23/24	levels
	3/20		
1/6	1/5	1/4	
5/16	5/16	1/3	
1/2	2/5	3/8	
B_1	B_2	B_3	

LF Rule Applied to P2 (c_{min} = 17/16)

1	1	1	levels
1/6	5/16	3/20	
1/3	5/16	1/5	
1/2	3/8	2/5	
B_1	B_2	B_3	

An Optimum Packing (c_{min}=1)

Let us define the performance ratio for an algorithm and a given m as the worst-case ratio of the performance of the given algorithm to that of an optimization algorithm. The asymptotic performance ratio is the limiting value as $m \to \infty$. For the cases at hand performance means the maximum bin occupancy (least capacity necessary for an LF packing) for P2, and the sum of the squares of the bin levels for P4. It has been shown that the LF performance ratio is 4/3 - 1/3m for P2 and is bounded by 25/24 for P4. It is not known whether the latter is achievable, but this is conjectured not to be the case.

The shortest-first level algorithm has been applied to P3, but it has a performance ratio of m/(2m+1), which is relatively poor as we shall see later. LF level algorithms can also be applied to P1 and P3, but their performance ratios are easily shown to be inferior to those of the corresponding first-fit algorithms described below.

IV. First-Fit Algorithms

The first-fit (FF) algorithms are more explicitly goal oriented, in contrast to the level algorithms. With these algorithms, the bins are scanned in the order $B_1, B_2, \ldots$ until a bin is encountered which will accommodate the next piece to be packed. An effective FF algorithm is the first-fit-decreasing (FFD) algorithm which, like the LF algorithm, initially puts the list into non-increasing order. The FFI ("I" for "increasing") algorithm initially puts the list in a non-decreasing order. The FFD algorithm is illustrated below for P1.

L = (1/2, 2/5, 3/8, 1/3, 5/16, 5/16, 1/4, 1/5, 1/6, 3/20)

c = 1

1/10	1/24	1/120	17/20	unused capacity
		1/6		
	1/4	1/5		
2/5	1/3	5/16		
1/2	3/8	5/16	3/20	
B_1	B_2	B_3	B_4	

FFD Rule Applied to P1 (4 bins)

0	0	0	unused capacity
		3/20	
1/6	5/16	1/4	
1/3	5/16	1/5	
1/2	3/8	2/5	
B_1	B_2	B_3	

An Optimum Packing (3 bins)

Variations of the FF algorithms are the best-fit (BF) algorithms, in which the bins are scanned as before, but with the result that a piece is placed in that bin of lowest index for which the resulting unused capacity is least.

The FF, BF, FFD, and BFD algorithms have been applied to P1; the a symptotic performance ratios have been shown to be 17/10, 17/10, 11/9, and 11/9, respectively.

The FFI algorithm has a performance ratio of 3/4 when applied to P3. Improved performance for P3 is obtainable from an iterated FFD algorithm which works as follows. L, assumed to be in non-decreasing order, is scanned until a largest r is found such that $\Sigma_{i=1}^{r} p_i \leq mc$, where mc represents the total capacity. The list $p_1 \leq p_2 \leq \ldots \leq p_r$ is then scanned in reverse order and packed according to the FFD rule. If the rule fails to pack all pieces then the largest piece (p_r) is discarded and the process repeated on $p_1, \ldots, p_{r-1}$. Largest pieces are discarded iteratively until a packing of all remaining pieces succeeds. The asymptotic performance ratio of this rule is known to be in the interval (6/7, 7/8].

The same concept has been exploited in applying the FFD rule to P2. In this case, the FFD rule is iterated on the entire list in a (binary) search for a c in a bounded range. The result

is an algorithm whose performance ratio is known to be in the interval [20/17, 61/50) for $m \geq 8$; for $m = 2,3$ and $4 \leq m \leq 7$ the performance ratios are precisely 8/7, 15/13, and 20/17, respectively.

The FF rules have yet to be applied to P4, although an algorithm based on the FFD rule appears to have superior performance to the LF level algorithm.

V. Optimality Tests

There are a number of important cases when the simple heuristics we have discussed are in fact optimal or asymptotically so. For a given positive real, a, let S(a) denote the set of positive powers of a; i.e. $\{a, a^2, \ldots\}$.

Theorem 1*

If for some a, $p_i \in S(a)$ for all i, then the LF and iterated FFD rules are optimal for P2. Moreover, the FFD rule is optimal for P1.

This result verifies optimality, in particular reference to computer applications, when all task execution times, or perhaps more appropriately all record sizes, are powers of two. In addition, this is true for P2 even when an arbitrary level algorithm can still do fairly badly, as shown in the following generalization of a result due to Graham [3].

Theorem 2

Suppose the piece sizes can be normalized so that for all i, p_i is an integer. Then an arbitrary level algorithm for P2 can perform worse than optimal by a factor of up to $2 - 1/\min\{m, \max\{p_i\}\}$.

A result somewhat stronger than Theorem 1 can be proved for the FFD rule when applied to P1.

Theorem 3

Suppose p_i divides c for all i. Then for P1 an FFD packing never requires in excess of one bin more than an optimum packing.

A test for optimality existing with the LF, but not the iterated FFD, rule applied to P2 is given next.

Theorem 4

Let w_L be the maximum bin occupancy on applying the LF rule to an instance of P2 in m bins. If

$$w_L > \left(\frac{3}{2} - \frac{1}{2m}\right)\sum_{i=1}^{n} p_i/m$$

then LF is optimal.

*Results in this section are proved in the appendix.

This result states that if the LF rule is forced to do "too badly" with respect to the absolute minimum, $\sum_{i=1}^{n} p_i/m$, then it must be optimum. For example, if the LF rule ever does 50% worse than the absolute minimum, then it must be optimal.

These are useful tests for optimality which allow one to avoid expensive enumerative or approximate approaches when solutions very close to the optimum are always needed. It is obviously desirable to derive conditions that are necessary as well as sufficient; however, the nature of the problem appears to make this a very difficult question.

VI. Open Problems

Worst-case performance bounds have obvious shortcomings as criteria for algorithm selection. This problem is aggravated by the fact that level and first-fit algorithms are seldom comparable in the sense that one always does at least as well as the other. Indeed, worst-case examples for one are frequently handled optimally by the other. Thus, although worst-case bounds may point to one algorithm, there is no assurance that that algorithm is statistically best.

Worst-case examples are somewhat pathological in some cases, and rely, for example, on large variations in piece sizes; thus, it would be of considerable interest to parameterize bounds in terms of this variation. Further results concerning instances when certain algorithms are optimal would also appear possible. Simple probability models appear to be of greatest interest; with such models one can hope to quantify such statements as: "algorithm performance will be within x percent of optimum with a probability bounded by p".

The FFD algorithms exhibit anomalies in many cases. These occur in their application to P1, P2 and P3. For example, there are lists for P1 such that if a piece is removed, the number of bins required to pack (by the FFD rule) the remaining pieces becomes one greater. It would be helpful to obtain some usable estimate of the maximum effects of such anomalies.

References

P1

[1] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham, "Worst-Case Performance Bounds for Simple One-Dimenstional Packing Algorithms," SIAM J. on Computing, 3 (1974), 299-326.

[2] D.S. Johnson, "Fast Algorithms for Bin-Packing," J. of Comp. and Sys. Sci., 8 (1974)

P2

[3] R.L. Graham, "Bounds on Multiprocessing Timing Anomalies," SIAM J. on Applied Math.,

17 (1969), 416-429.

[4] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson, "An Application of Bin-Packing to Multiprocessor Scheduling," SIAM J. on Computing (to appear).

[5] R.L. Graham, "Bounds on the Performance of Scheduling Algorithms," Computer and Job-Shop Scheduling Theory, E.G. Coffman (Ed.), Wiley and Sons, 1975.

P3

[6] E.G. Coffman, Jr., Joseph Y-T. Leung, and Dennis Ting, "Bin-Packing: Maximizing the Number of Pieces Packed," Acta Informatica (to appear).

[7] E.G. Coffman, Jr., Joseph Y-T. Leung, "Efficient Sequencing and Allocation Algorithms: Maximizing the Number of Units Assigned," (to appear)

P4

[8] A. Chandra and C.K. Wong, "Worst-Case Analysis of a Placement Algorithm Related to Storage Allocation," SIAM J. on Computing, 4 (1975), 249-263.

[9] R. Cody and E.G. Coffman, Jr., "Record Allocation for Minimizing Expected Retrieval Costs on Drum-Like Storage Devices," Journal of the ACM, 23 (1976).

Appendix

Proof of Theorem 1

Consider the LF rule applied to P2 on m machines under the assumptions that $p_1 \geq p_2 \geq \ldots \geq p_n$ and each $p_i \in P$ is a positive power of some positive number a. (For this part of Theorem 1 we shall use the terminology of task scheduling.) Suppose, contrary to the theorem, that the LF rule is not optimal for P, and suppose that P is a smallest counterexample. Thus, if w_L and w_{opt} are the respective lengths of LF and optimum schedules for P, then $w_L > w_{opt}$.

Suppose p_n, the smallest task, does not terminate the LF schedule; i.e. suppose p_n does not have a latest finishing time in the LF schedule. Then we note that the LF schedule length for $P-\{p_n\}$ must be equal to w_L while the length of an optimum schedule for $P-\{p_n\}$ can not be greater than w_{opt}. Thus, $P-\{p_n\}$ provides us with a smaller counterexample, and this contradicts the assumed minimality of P. Hence, we may assume that p_n terminates the LF schedule.

Finally, observe that p_n divides p_i for all i. It follows easily from the LF rule and the observations to this point that $w_L = \left\lceil \sum_{i=1}^{n} p_i/m \right\rceil_{p_n}$, where $\lceil y \rceil_x$ denotes the least multiple of x greater than or equal to y. But $\left\lceil \sum_{i=1}^{n} p_i/m \right\rceil_{p_n}$ is a lower bound to the length of any schedule for P and hence the LF schedule is optimal.

Consider now the FFD rule applied to P1 under the above assumptions. Assuming as before that P is a smallest counterexample implies that p_n uniquely occupies the last bin. Since p_n divides all p_i, this in turn implies that every bin, except the last one, must be filled to a level of $\lfloor c \rfloor_{p_n}$. Since no packing can fill a bin to a higher level, the FFD rule must be optimal.

With reasoning similar to the above it is easily shown that the iterated FFD rule is also optimal for P2 under the assumptions of the theorem.

Proof of Theorem 2

Suppose that each p_i is an integer. We want to consider the largest possible ratio between two schedule lengths arising from two different level algorithms applied to P2 on m machines. (We shall use the terminology of task scheduling.) First, we dispose of the case when $p_1 \equiv \max\{p_i\} < m$.

Since all tasks are integral and each task is assigned in its turn to the first available processor, we have for a maximum schedule length

$$w' \leq p_1 + \left\lfloor \sum_{i=2}^{n} p_i/m \right\rfloor$$

For a minimum schedule length we have the lower bound $w \geq \sum_{i=1}^{n} p_i/m$, and hence

$$\sum_{i=2}^{n} p_i/m = \sum_{i=1}^{n} p_i/m - p_1/m \leq w - p_1/m$$

But w must be integral, and since $p_1/m < 1$ we have

$$\left\lfloor \sum_{i=2}^{n} p_i/m \right\rfloor \leq \lfloor w - p_1/m \rfloor = w - 1$$

or

$$\frac{\left\lfloor \sum_{i=2}^{n} p_i/m \right\rfloor}{w} \leq 1 - \frac{1}{w}$$

Hence,

$$\frac{w'}{w} \leq \frac{p_1 + \left\lfloor \sum_{i=1}^{n} p_i/m \right\rfloor}{w} \leq \frac{p_1}{w} + 1 - \frac{1}{w} = 1 + \frac{p_1 - 1}{w}$$

Since $w \geq p_1$ we get

$$\frac{w'}{w} \leq 2 - \frac{1}{p_1}$$

As shown in [3], 2-1/m is a general bound on w'/w; in particular, the bound holds when $p_1 > m$. Thus,

$$\frac{w'}{w} \leq 2 - 1/\min[m, \max\{p_i\}]$$

An example showing that the bound is achievable is provided by the parameters $n=2m-1$, $p_1 = k-1 (2 \leq i \leq m)$, and $p_i = 1$ $(m+1 \leq i \leq 2m-1)$ for any integer $k \leq m$. □

Proof of Theorem 3

For the FFD rule applied to P1, we assume that p_i divides c for all i. For simplicity we may assume $c=1$, and hence that the p_i are all unit fractions. Assume the FFD packing requires m bins of which $j \leq m$ have a non-zero unused capacity. Let $1/k$ be the size of the largest piece in the last bin, B_m. Thus, at least $j-1$ of the incompletely filled bins must be filled to a level exceeding $(k-1)/k$.) Consequently, since $1/k$ is a lower bound to the level of the last bin, we have

$$\sum_{i=1}^{n} p_i > (m-j)+(j-1)(k-1)/k+1/k$$

or

$$\sum_{i=1}^{n} p_i > m-j/k-(k-2)/k .$$

Clearly, each incompletely filled bin, except possibly the last, must have at least two different piece sizes in it. Therefore, since the pieces are packed in a non-increasing order, we have $j \leq k$ and

$$\sum_{i=1}^{n} p_i > m-1-\frac{k-2}{k} > m-2$$

Finally, if N_{opt} is the optimum number of bins, we have the obvious lower bound $N_{opt} \geq \left\lceil \sum_{i=1}^{n} p_i \right\rceil$.

Making use of the previous inequality, we get $N_{opt} \geq m-1$; i.e. an optimum packing requires at most one fewer bin than the FFD packing. □

Proof of Theorem 4

Using scheduling termonology we shall show that for P2 the LF rule is optimal if in the resulting schedule for P we have

$$w_L > \left(\frac{3}{2} - \frac{1}{2m}\right) \sum_{i=1}^{n} p_i/m$$

First, note that the starting time of any task must not exceed any processor finishing time. Thus, if P is assumed to be a smallest counter example for the theorem, than we have as in the proof of Theorem 1 that the smallest task, say p_n, terminates the schedule. Hence,

$$w_L \leq p_n + \sum_{i=1}^{n-1} p_i/m = p_n\left(1-\frac{1}{m}\right) + \sum_{i=1}^{n} p_i/m$$

Using the above two inequalities we obtain

$$p_n\left(1-\frac{1}{m}\right) + \sum_{i=1}^{n} p_i/m > \left(\frac{3}{2} - \frac{1}{2m}\right) \sum_{i=1}^{n} p_i/m$$

which reduces to

$$2p_n > \sum_{i=1}^{n} p_i/m$$

But this means that there are at most two tasks per processor in the LF schedule. (If a processor had three or more tasks, then conservation of $\sum_{i=1}^{n} p_i$ would require that the finishing time of some other processor precede the starting time of the third task scheduled.) Finally, it is routine to verify that an LF schedule with at most two tasks per processor is an optimum schedule. This contradiction proves the theorem. □

ON SCHEDULING ALGORITHMS FOR N-FREE TASK DEPENDENCY STRUCTURES

Edgar Nett
Gesellschaft für Mathematik
und Datenverarbeitung
5205 St. Augustin, West Germany

Abstract-- Multiprocessor scheduling strategies have been the focus of substantial research in recent years. The complexity of the general scheduling problem necessitates an abstract mathematical model to analyze scheduling algorithms with respect to their worst-case performance bounds. This paper is to study the task scheduling problem for a Task Dependency Structure which differs from the general precedence relation on tasks only by one restriction: All immediate successor tasks of a branch task must not be merge tasks. In particular, it is shown that for $m > 2$ processors the ratio of the length of an arbitrary level schedule for N-free task structures and of the corresponding optimal schedule is bounded by 3/2. Furthermore, if $m = 2$, then a level schedule is always optimal for N-free task structures.

I. Introduction

Multiprocessor scheduling strategies have been the focus of substantial research in recent years. The complexity of the general scheduling problem necessitates an abstract mathematical model to render an analysis of scheduling algorithms with respect to certain performance goals.

One of the most common models is the so-called 'General Multiprocessor System' proposed by Graham [1]. This model is defined by the following components:

1) A set of m identical processors P_i, $i = 1,\ldots,m$.
2) A set of tasks $T = (t_1,\ldots,t_n)$ which is to be processed by the P_i.
3) The general task dependency structure $<$ on T which is a binary relation that is anti-symmetric and transitive.
4) A function $\mu : T \to (0,\infty)$ which denotes the execution of each task $t \in T$.

The performance goal to be considered here is to find scheduling algorithms which minimize the total time required to execute the task set T on the processors P_i.

Ullman [4] has shown that this problem is polynomial complete even if one of the following restrictions holds:

1. All tasks $t \in T$ require one unit of time for their execution.
2. All tasks $t \in T$ require one or two units of time for their execution and there are only two processors P_1 and P_2.

This result it tantamount to showing that the problem of optimal task scheduling according to this model is computationally intractable. Moreover, worst-case investigations of the known polynomially bounded scheduling algorithms have shown that the length of the generated schedules can be approximated arbitrarily close to $(2-1/m)$ times the length of the optimal schedule. This value, however, is an upper bound on every demand schedule, which allocates an idle processor to anyone of the tasks that are ready for processing. Hence, it does not appear reasonable to apply sophisticated algorithms unless the scheduling problem conforms to a simpler model which, beyond being confined to unit time tasks, meets some additional restrictive conditions. These restrictions may be imposed on the number of processors or on the task dependency structure (TDS). In fact, it has been demonstrated that

optimal schedules can be generated by polynomially bounded algorithms for the case that the number of processors is confined to two, while the TDS, according to Graham's model remains unrestricted [2], and for the case that the TDS constitutes a tree structure while the number of processors remains unbounded [3].

Tree structures, however, do not reflect the type of tasks dependencies that can typically be found in conventional computer programs. Recently, a more adequate class of restricted TDS, the so-called series-parallel graphs, have been investigated by Goyal [6]. However, the upper bound on the length of a level schedule for these graphs is $(2 - \frac{2}{m})$ times the length of an optimal schedule. With increasing m, this value can be approximated arbitrarily close by the worst possible bound $(2 - \frac{1}{m})$.

These two examples - tree structures on the one hand and series-parallel structures on the other hand - indicate that two objectives have to be met when imposing restrictions on the TDS with regard to the generation of task schedules by polynomially bounded algorithms:

1.) The algorithm must produce at least suboptimal schedules, i.e. the upper bound on the length of the schedule must be decidedly lower than the worst case upper bound to justify the increased computational effort as compared to arbitrary demand scheduling.
2.) The class of TDS that conforms to the first objective must reflect structures that appear in real processes.

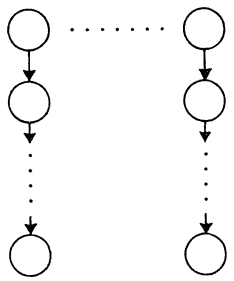
The purpose of this paper is to present TDS's which meet these objectives with regard to level schedules. In the next section, we introduce rather informally the class of the so-called N-free TDS's, and in the third section, we demonstrate that level schedules for these TDS's are bounded by $\frac{3}{2}$, which represents a considerable improvement over the worst case bound. In section 4, we show that N-free TDS's cover the structures of nested DO-loops, the Fork-Join concept, and also comply with the rules of structured programming.

II. N-free Task Dependency Structures

When trying to define a class of TDS's that yields at least suboptimal level schedules, it is, as a first approach, useful to identify those TDS-elements which may cause level schedules to become non-optimal.

Consider, as an example, the TDS in figure 1, which belongs to the class of the so-called multi-linear structures.

Figure 1:

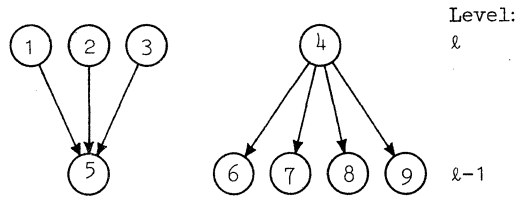


A level schedule for such a simple structure is always optimal, since every task that has been completed frees its successor task from the next lower level for execution. Hence, the tasks can be processed level by level without violating any dependency relations. This situation changes, if tasks with more than one successor and more than one predecessor are permitted.

Definition 1: A task t in a TDS G is called a branch task, if the number of its immediate successors is greater than one; correspondingly, a task t is called a merge task, if the number of its immediate predecessors is greater than one.

Now, it is conceivable to have a task dependency structure as shown in figure 2, where the tasks 1,2,3,4 belong to some level ℓ, and the tasks 5,6,7,8,9 belong to the level $\ell-1$.

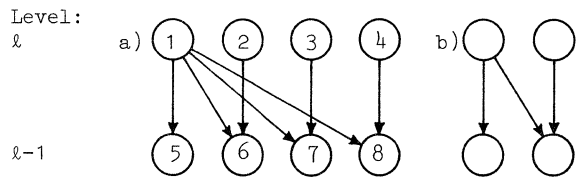
Figure 2:



We assume, that the number of processors is three. Clearly, if during the first time slot the tasks 1,2,3 are being performed, then only the tasks 4 and 5 can be executed next, leaving one processor idle. It takes another two time slots to process the remaining task 6,7,8,9; i.e. four time slots altogether. If, however, task 4 is executed during the first time slot, then all processors can be kept busy and the entire job is done in only three time slots.

Another example of a TDS-element which may be responsible for erroneous scheduling is shown in fig. 3a.

Figure 3:



Suppose the tasks 1,2,3,4 belong to the level ℓ, and the tasks 5,6,7,8 belong to the level ℓ-1. If the number of processors is three, then it is imperative that the task 1 be processed during the first time slot while two more tasks from the level ℓ can be arbitrarily selected. Otherwise, all tasks of the level ℓ-1 would remain blocked for one more time slot, during which only the task 1 could be executed. The entire job would require four time slots, compared with only three time slots for an optimal schedule. Task 1 looses its blocking effect with regard to level scheduling, if its level is lifted above the level of the tasks 2,3,4, in which case the task 1 can never be scheduled later than the tasks 2,3,4. This may be caused either by a higher level of task 5 relative to the tasks 6,7,8, or by an additional task along at least one of the edges leading from task 1 to the tasks 6,7,8.

Thus, we have:

Definition 2: A task t with level ℓ in a TDS G is called a blocking branch task, if at least one of its immediate successor with level ℓ-1 is a merge task.

The simplest structure that fulfils definition 2 is composed of only four tasks, and, as shown in fig. 3b, has the shape of an N. These N-structures constitute the most critical TDS-elements as far as level scheduling is concerned, since errors can be made with the least number of tasks involved. A large number of N-structures within the TDS may therefore cause a level schedule to rapidly approach the worst case bound $(2 - \frac{1}{m})$ [7].

Hence, it appears rather rewarding to investigate the scheduling problem for TDS's which are free of these N-structures. It is reasonable to assume that, on the one hand, these - only slightly restricted - TDS's yield at least sub-optimal level schedules and that, on the other hand, computer programs can be kept free of blocking branch tasks.

III. Worst case bounds for level schedules on N-free task dependency structures

In this section, we submit and proof the basic theorems regarding the length of level schedules on N-free TDS's.

Definition 3: A TDS G is called N-free, if it has no blocking branch task.

Theorem 1: Let G be an N-free TDS, let w_ℓ be the length of an arbitrary level schedule, w_{opt} be the length of an optimal schedule for G. Then, for $m > 2$ processors it holds:

$\frac{w_\ell}{w_{opt}} \leq \frac{3}{2}$ and this is the best possible bound.

Theorem 2: If the same assumptions apply as in theorem 1, then for $m = 2$ processors every

level schedule for G is optimal.

These two theorems may be proved by means of the following definitions and lemmata. Some of the definitions can be found in [5], most of the proofs for the lemmata are omitted since they are of a more or less technical nature.

Definition 4: Let S be a schedule for a TDS G. Let w(S) denote the length of S given by the number of time slots required to execute G according to schedule S. S(i) denotes the subset of all tasks $t \in T_G$ that are executed during time slot i. If S(i) is smaller than the number m of the given processors, then time slot i in schedule S is called an 'Incompletely Occupied Time Slot' (abbr. IOTS).

Lemma 1: Let i be an IOTS in a schedule S for G. Then the set N(S(i)) of all successors of the tasks belonging to S(i) is equal to the set of all those tasks that have not yet been executed at the end of time slot i.

It should be emphasized, that this lemma, of course, is correct only since we are dealing with demand schedules.

Definition 5: An IOTS i in S is real if there exists an optimal schedule R for G so that S(i) = R(j) and $j \leq i$; otherwise, i is called virtual.

Lemma 2: Let i be a virtual IOTS in a non-optimal schedule S for G. Then, there exists at least one task $t \in S(i)$ which in every optimal schedule R for G is executed in a preceding time slot.

Definition 6: A virtual IOTS i in a non-optimal schedule S is called primary, if there exists a task $r \in S(i)$ and a task $t \in S(\ell)$, $(\ell<i)$, so that for an arbitrary optimal schedule R for G holds: $r \in R(j)$ and $t \in R(k)$ and $j < k$.

Lemma 3: Every non-optimal schedule S has at least one primary IOTS.

Definition 7: A task r is said to dominate a task s in a TDS G, if the set of successors of s, N(s), is a subset of the corresponding set N(r) of r.

The dominance criterion requires that a task is always executed no later as those tasks it dominates.

Lemma 4: Let S be a schedule for a TDS G so that the dominance criterion is not violated in S. Then there exists no primary IOTS i in S with $|S(i)| = 1$.

Proof: Assume there exists a primary IOTS with the above property. Let $i := \min \{j \mid j$ is a primary IOTS in S and $|S(j)| = 1\}$ and let $S(i) = \{t\}$.
According to lemma 1, all tasks that are executed in S after task t belong to the set of successors N(t) of t. From the definition of a primary IOTS follows that there must exist at least one more task s, which is independent of t and which can be executed in time slot i without involving an IOTS; otherwise, i would be a real IOTS. Therefore, the following relation holds for these two tasks: $N(t) \supsetneq N(s)$ and t is executed later than s in schedule S. This is a violation of the dominance criterion and, hence, a contradiction to the assumption.

Lemma 5: Let G be an N-free TDS, let S be an arbitrary level schedule for G. Then the dominance criterion is not violated in S.

Proof: Assume the dominance criterion is violated in S. Then there exist two tasks s and t so that $N(t) \supsetneq N(s)$, and s is executed before t, i.e. : $\ell(s) \geq \ell(t)$. Then, the set of immediate successors of s, DN(s), must be a pure subset of the corresponding set DN(t). Since DN(s) cannot be empty, there exists a task r with $r \in (DN(t) \cap DN(s))$. Hence, task t is a blocking branch task. This is a contradiction to

the assumption that G is N-free.

Lemmata 4 and 5 immediately lead to the following

Lemma 6: In an arbitrary level schedule S for an N-free TDS G there exists no primary IOTS i with $|S(i)| = 1$.

Obviously, in every IOTS of a schedule S for two processors only one task is executed. Considering the foregoing lemma 6, it follows that in every level schedule for an N-free TDS G and two processors there exists no primary IOTS. According to lemma 3, in every non-optimal schedule, however, there exists at least one primary IOTS. This proves the correctness of theorem 2.

Lemma 7: Suppose, in a schedule S for a TDS G all the time slots i, i+1,...,j, (i<j), are IOTS's and at least one of these IOTS's is primary. Then all IOTS's i,...,j are primary.

Definition 8: Let the time slots i,...,j,(i≤j), be consecutive primary IOTS's in a schedule S for N-free TDS G. The neighborhood U(i,j) of these IOTS's is defined as follows:
$U(i,j) := \{u \mid \exists\ i' \leq \ell' \leq j'\ S(u) \cap R(\ell') \neq \emptyset\}$
where the time slots i' and j' are defined as follows: Let R be an optimal schedule for G then

$$i' := \min_{u'} \{R(u') \cap \bigcup_{\ell=i}^{j} S(\ell) \neq \emptyset\}$$

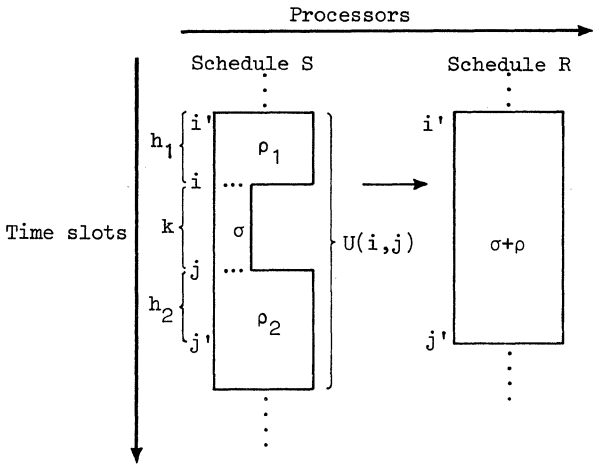
$$j' := \max_{u'} \{R(u') \cap \bigcup_{\ell=i}^{j} S(\ell) \neq \emptyset\}$$

For a better understanding of this definition and the following step in the proof see fig. 4.

(a) $\lceil \frac{\rho}{m} \rceil$ denotes the smallest integer greater than $\frac{\rho}{m}$

(b) $\lfloor \frac{\sigma}{m} \rfloor$ denotes the greatest integer smaller than $\frac{\sigma}{m}$

Figure 4:



Comment: From the definition of a primary IOTS follows, that $i' < i$ and $j' \geq j$.

The rest of the proof of theorem 1 can now be given straightforwardly:

Let $\sigma = |\bigcup_{\ell=i}^{j} S(\ell)|$. Then the following equation holds for the optimal schedule R:

$|\bigcup_{\ell=i'}^{j'} R(\ell)| = \sigma + \rho$ with $\rho > 0$. The integers k and h are defined as follows: $k := j-i+1$ and $h := i-i'+j'-j$.

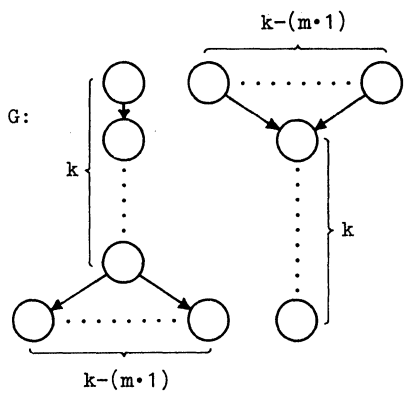
Then the number of time slots that is necessary to execute the $\sigma + \rho$ tasks in schedule R is given by $w_{opt}^{\sigma+\rho} = k+h$. To execute the same number of tasks in the level schedule S, $w_{\ell}^{\sigma+\rho} = k+\lceil \frac{\rho}{m} \rceil$ (a) time slots are necessary, where m is the number of processors. Since $\sigma+\rho \leq (k+h)\cdot m$, $2k+h-\lfloor \frac{\sigma}{m} \rfloor$ (b) is an upper bound for $w_{\ell}^{\sigma+\rho}$. Now, σ becomes minimal and, subsequently, $w_{\ell}^{\sigma+\rho}$ becomes a maximum if in every primary IOTS between i and j only the minimal number of tasks is executed. According to lemma 6, for every primary IOTS, this minimal number of executable tasks is two and, therefore, $\sigma = 2k$. Then, from lemma 2 follows that $k = h$ and, therefore,
$w_{\ell}^{\sigma+\rho} \leq 3k-\lfloor \frac{2k}{m} \rfloor \leq 3k-\frac{2k}{m}+1$.

Consequently, $\frac{w_\ell^{\sigma+\rho}}{w_{opt}^{\sigma+\rho}} \leq \frac{3}{2} - \frac{1}{m} + \frac{1}{2k}$. If $m > 2k$, then $\lfloor \frac{2k}{m} \rfloor = 0$ and therefore $\frac{w_\ell^{\sigma+\rho}}{w_{opt}^{\sigma+\rho}} \leq \frac{3}{2}$.

Since the above estimate holds for an arbitrary sequence of primary IOTS's(which includes the case of only one IOTS, since it was defined in definition 8 that $i \leq j$), it holds for every sequence of primary IOTS's in a level schedule S for an N-free TDS G. Hence, it is also valid for the sum of all such sequences in S. If there exist some tasks in S which are executed in time slots that do not belong to any neighborhood U(i,j) in S, all tasks executed in these time slots according to schedule S are executed in schedule R also in one time slot. That is, in order to extend the above estimate to the lengths of the whole schedules S and R, it is necessary to add a constant $c \geq 0$ to the numerator and the denominator of the fraction $\frac{3}{2}$. Since $\frac{x+c}{y+c} \leq \frac{x}{y}$ for $x,y,c \geq 0$, it follows: $\frac{w_\ell(G)}{w_{opt}(G)} \leq \frac{3}{2}$ and this completes the proof of theorem 1.

To verify that this upper bound is the best possible, figure 5 shows a model of an N-free TDS G for which $\frac{w_\ell(G)}{w_{opt}(G)}$ can be approximated arbitrary close to $\frac{3}{2}$.

Figure 5:

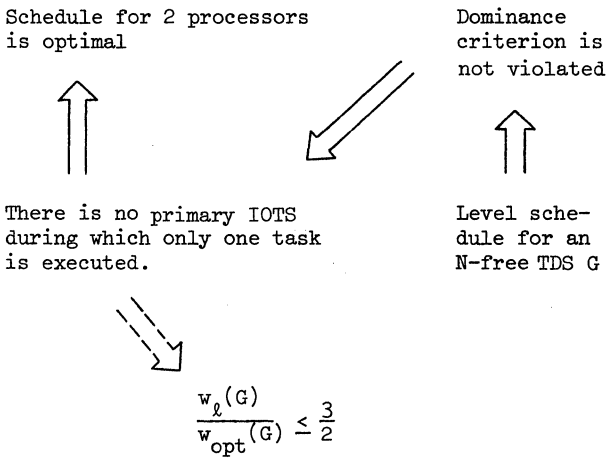


Assume $k = n \cdot m$, i.e. k is a multiple of the number of processors. Then it is easy to verify that $w_{opt}(G) = 2\ n \cdot m$ and $w_\ell(G) = 2 \cdot n(m-1)+n \cdot m$.

Hence, $\frac{w_\ell(G)}{w_{opt}(G)} = \frac{3nm - 2n}{2nm} = \frac{3}{2} - \frac{1}{m} = \frac{3}{2}$ for $m \to \infty$.

The following diagram exhibits the correlations between the results we have obtained from our investigation of level schedules on N-free TDS and, particularly, from the notions of dominance and IOTS's.

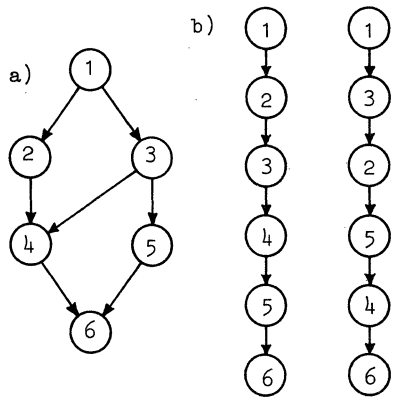
Figure 6:



IV. Relevance of N-free structures to computer programs

In the preceding section it has been shown that N-free TDS's are suited for level scheduling. In the following, we offer some representative examples which exhibit the relevance of these structures to computer programs. The TDS of a program can be considered as the representation of what we call the 'minimal control structure' of this program. This is to say that the TDS imposes only those precedence relations between tasks that have to be observed in order to accomplish the correct execution of the program. Consider, for instance, figure 7.

Figure 7:



The graph in figure 7a represents data dependencies between some tasks in a program. This TDS permits several sequential control structures for the program execution. In figure 7b we list two of them. All correct control structures for the execution of this program have in common that, for instance, task 4 is executed later than task 2 and task 3 or that task 6 is executed at least.

For multiprocessor scheduling purposes, programs (or program segments) that feature a high degree of parallelism are of considerable interest. This is particularly true for iterative (nested) DO-loop structures [8]. In a previous paper we have shown that these structures, which constitute a subset of N-free structures, even yield optimal level schedules under certain conditions [5].

FORK-JOIN constructs are known as programming tools which allow the explicit specification of parallel executable task sequences [10]. Every FORK instruction can be considered as the realization of a branch task. Moreover, the FORK-JOIN concept demands that every FORK instruction is associated with a junction point. At this junction point - represented in the TDS by a merge task - all processes that have been initiated by the associated FORK instructions must recombine. This, however, implies that the FORK-JOIN concept does not permit N-structures.

Improving program verification and program manageability are two main objectives of what is commonly known as structured programming [9]. Particularly in very large and complex program these properties are quite desirable. The general idea of structured programming is to permit only some elementary control structures within programs. These structures include:

1) a single assignment statement,
2) a conditional branch statement,
3) an interation, i.e. the repeated execution of an elementary control structure as long as a certain condition holds,
4) a sequence of elementary control structures.

Obviously, the control structure of a structured program is N-free, if each of these elementary control structures is N-free. Since we are dealing only with deterministic scheduling, the data flow in a program is assumed to be known a priori. For this reason, a conditional branch can be omitted in our considerations. Assignment statements or sequences of them are represented in the corresponding TDS as a single task. Hence, only the iteration remains to be considered. If the repeated execution of elementary control structures within an iteration can be done independently of each other, it is equivalent to a DO-loop structure the representation of which has already been treated. If, however, the single

steps of an iteration must be executed in a sequential order, its control structure is similar to the sequence of elementary control structures.

Every algorithm can be written using only simple instructions, conditional branches and interations. Consequently, every program, which is nothing but the description of an algorithm, can be written in a well structured form.

Concluding remarks

Of all the algorithms, by which the priority of tasks for scheduling purposes in multiprocessor systems can be determined, the level algorithm appears to be most suitable for practical applications because of its simplicity. So far, it has been demonstrated that the level algorithm suffices to generate optimal schedules for TDS's that feature a tree structure.

In this paper, a new type of TDS's can be presented which represents an extension of the tree structures and which reflects to a greater extent structures that appear in real processes. It has been shown that level schedules for this class of restricted TDS's provide considerably better results than for the general TDS's.

Acknowledgements

I am indebted to Dr. W. Kluge for his stimulation and help. I also like to thank Dr. K. Ecker and my other colleagues for useful discussions.

References

[1] Graham, R.L.: 'Bounds on Multiprocessing Anomalies and Related Packing Algorithms', Spring Joint Comp. Conf., 1972

[2] Coffman, E.G.: Graham, R.L.: 'Optimal Scheduling for Two-Processor Systems', Acta Informatica, Vol.1, No.3, 1972

[3] Hu, T.C.: 'Parallel Sequencing and Assembly Line Problems', Op. Res., Vol.9, No.6, Nov. 1961

[4] Ulman, J.D.: 'Polynomial Complete Scheduling Problems', Techn. Report 3, Dept. of Comp. Science, Univ. of Calif. at Berkeley, March 1973

[5] Nett, E.: 'On Further Applications of the Hu Algorithm to Scheduling Problems', Proc. of the 1976 Int. Conf. on Parallel Processing, P.H. Enslow Jr., Editor

[6] Goyal, D.K.: 'Scheduling Series-Parallel Structured Tasks on Multiprocessor Computing Systems', Dept. of Comp. Science, Washington State Univ., Pullman, Sept. 1976

[7] Chen, N.-F.: 'An Analysis of Scheduling Algorithms in Multiprocessor Computing Systems', Dept. of Comp. Science, Univ. of Illinois, Urbana, May 1975

[8] Baer, J.L.: 'A Survey of some Theoretical Aspects of Multiprocessing', Comp. Surveys, Vol.5, No.1, March 1973

[9] Dahl, O.J.: 'Structured Programming', Dijkstra,E.W. Academic Press, 1972 Hoare, C.A.R.

[10] Conway, M.: 'A Multiprocessor System Design', Proc. AFIPS 1963, Fall Joint Comp. Conf., pp. 139 - 146.

A FIXED-VARIABLE SCHEDULING MODEL FOR MULTIPROCESSORS[a]

John E. Jensen
Department of Mathematics
Texas Tech University
Lubbock, Texas 79409

Abstract -- A model is presented for a scheduler which interfaces multiprocessing hardware and software to form a complete model of the dynamic effects of scheduling on a multiprocessor system. A fixed-variable scheduling philosophy is introduced which allows both the immediate scheduling of tasks in a high-demand situation and the careful improvement of schedules through arbitrarily complex algorithms as time becomes available to the scheduler. Hardware and software are determined for the creation of a direct mechanism for the implementation of this new scheduling philosophy, although only the fixed part is actually implemented. Experiments on the model enable several conclusions to be made regarding the acyclic representation of cyclic graphs and the marginal improvements achievable by some global scheduling heuristics.

I. The Scheduling Philosophy

The problem of maximizing the efficient use of the computer's resources is hampered by the large number of system variables to consider. Even in the restricted case where task lengths and precedences are the only considerations, looking for optimal solutions becomes futile [19]. In this paper we accept the bounds on optimality presented by heuristic methods [7] and consider appropriate heuristics which might include some of the other parameters of importance. Some theoretical results [4] demonstrate the levels of complexity which are added by the consideration of the computer system as a general set of limited resources, while others [11] specifically approach the problem of including memory restrictions in the scheduling algorithm. This rapid growth seen in analytical complexity with each new system parameter suggests simulation as the most immediately viable tool for studying the dynamic effects of scheduling in realistic system models.

What we propose is a new philosophy of "fixed-variable" scheduling. The theory is based on the fact that there are certain duties the scheduler must perform as rapidly and frequently as possible in order to avoid undue delays for scheduling overhead. At the same time there are many interesting parameters we would like to include in the scheduling algorithm if it were not for the time penalty. The solution comes from the fact that the demands made upon the scheduler are not constant; both the frequency of requests for tasks by the processors and the time penalties for polling new tasks and implementing the scheduling algorithm will vary widely during the hours of operation on the multiprocessing system.

The "fixed" part of the scheduling philosophy is the implementation of a minimal scheduler in the following fashion. Static heuristics are used to pre-order the tasks such that delivery of tasks to a processor simplifies to a hardware function [8] with the only real overhead for the scheduler being the time required for polling the user programs for tasks ready to be scheduled. A high rate of polling (the asymptote being demand scheduling) requires too much time on the part of the scheduler to search for "ready" tasks. The alternative is a low polling rate which may cause the "ready list" to become empty or a high-priority task to miss an early chance at execution. This trade-off creates a decision in the polling frequency which is best made dynamically.

The "variable" part of the scheduling philosophy incorporates all of the various dynamic decisions which may be made. With the hardware producing a "reasonable" schedule of tasks at a frequency comparable to the highest rate of processor demand, the scheduling software need only "improve" the already existing schedule as time permits. That is, when the request rate for tasks is low, the scheduler executes a complicated algorithm involving a large number of dynamic variables, and when the request rate is high, the scheduler spends nearly all of its time polling in order to keep the "ready list" from becoming empty. Other decisions, such as the polling frequency, which and how many dynamic variables to consider, and the desired level of optimality in the algorithm which uses these variables, are made dynamically by the scheduler.

The difficulty with this scheduling philosophy is in determining the answers to the dynamic questions just raised. Since it is difficult to measure the effects of the algorithms in this highly variable setting, and since the overhead penalties are heavily machine and program dependent, it becomes impossible to determine the answers to the dynamic decisions the scheduler must solve. In particular, this aspect of the scheduling model could not be included in the simulation experiments to follow, due to the lack of the appropriate overhead factors and the unbounded possibilities for algorithms to test. The remainder of this paper investigates the appropriate parameters for the fixed portion of the scheduling model and leaves the variable portion of the model open to further investigation.

[a] This research was supported by NSF grant GJ-41164.

II. The System Model

Assumptions Regarding the Host System

The features which we consider as part of our model begin with a collection of independent processing units organized on a time-phased ring. All execution is handled by functional units shared by the processors, with the scheduling duties being performed by a dedicated processor similar to the one offered as part of Texas Instruments' ASC [18]. The processing units request tasks from the scheduler upon completion of their previous assignments (in concordance with the results of [5]) while the operating system polls user programs to find tasks which are ready for scheduling. When the polling routine detects that all of the predecessors of a given task have been processed, the task is placed on a "ready" list where it waits for assignment to a processor by the scheduler. The scheduling processor also assigns tasks for interrupt processing as described in [6].

The aspects of memory which influence the scheduling model are memory interference and the availability of memory for tasks about to be executed. A variety of interference models is available [2] with the simplest being to assume that memory is shared randomly. Less interference may be achieved by the "home memory" concept [17] in which instructions and data used primarily by one processor are loaded in a memory module specifically assigned to that processor. However this requires that the processor assignment be known in advance for proper loading of the task, and further complicates the treatment of interference among the various resources in the system. The availability of memory for loading tasks also appears in [3] as a part of the scheduling algorithm.

The "forking scheme" which determines the various parallel execution paths among the tasks is a function of the model used to describe the programs which are to be executed on the multiprocessor. The responsibilities of the processors, the processes, and the operating system for achieving the proper matching between tasks and processors is a matter taken up by [5]. The "ready list" of tasks about to be scheduled is managed by the scheduling processor and is constantly being updated. A special controller serves as the interrupt mechanism and delivers the next task from the ready list upon processor request. If hardware monitoring is desired to improve the scheduling efficiency, that also is interpreted by the scheduling processor.

Run-time anomalies, or variance in task attributes, is modeled in several stages. Variance in task lengths due to data dependencies, looping instructions, or IO waits within a task is determined by treating the task length as a random variable with only its mean and some assumed distribution being known to the scheduler. Variance due to contention for hardware resources is modeled analytically [10] to produce a "contention function" relating the "free" execution time of a task to its "limited-resource" execution time. Variation which occurs in the interpretation of the program graph, such as branches and loops between vertices, has been studied by Martin and Estrin [14].

Representation of Tasks

The tasks to be scheduled are considered to be vertices of a single program graph of the type discussed in [9]. The model for an individual task consists of its connecting structure in the program graph and its resource attributes. Since many of the operating system tasks may be consiered as service calls from the user program, they may be represented as subgraph modules of the user program, thus enabling the model to treat user and operating system tasks as indistinguishable. In addition, it is anticipated that future research will enable the entire operating system to be modeled by a single program graph with subgraph substitutions being performed dynamically to represent the flow of user jobs in the system. This makes the assumption in this paper that only one program graph exists in the system at a time a reasonable simplification.

The origin of the program graph and descriptions of the individual tasks is not a major concern here. We assume that all user programs require an amount of computation time warranting their decomposition into several (parallel) tasks, that they are properly compiled, checked for proper termination conditions, and linked through appropriate graph-module replacement algorithms into the operating system's program graph. Each task possesses a unique identity, a link to its creator (operating system or user job), its memory size and protection requirements, and estimates for its duration, variance of duration, and instruction mix. The instruction mix reflects the resource requirements in that it differentiates tasks with heavy usages of IO, arithmetic, or memory capacity. The variance parameter allows the task to behave anomalously during execution after the scheduler has assumed the other estimated parameters to be exact.

Several graph structures for programs to be input to the system were taken from [13] (e.g. Figure 1a). The scale of the tasks and distribution of task weights were modified, however, to resemble a more balanced weighting of tasks that might result from a compiler analysis of user programs [8]. The desirability of having the compiler balance the task weights in this fashion becomes apparent when comparing expected results from heuristics in [1] with the worst-case bounds on scheduling anomalies expressed in [7] and [19], which are attained by considering asymptotic task relationships. Transformations were also made on the original graphs to make them adhere to proper termination criteria [9] and to make the graphs acyclic, replacing the cycles by multiplying the vertex weights within the loops by the estimated loop frequencies [14].

Figure 1a - Example Program Graph

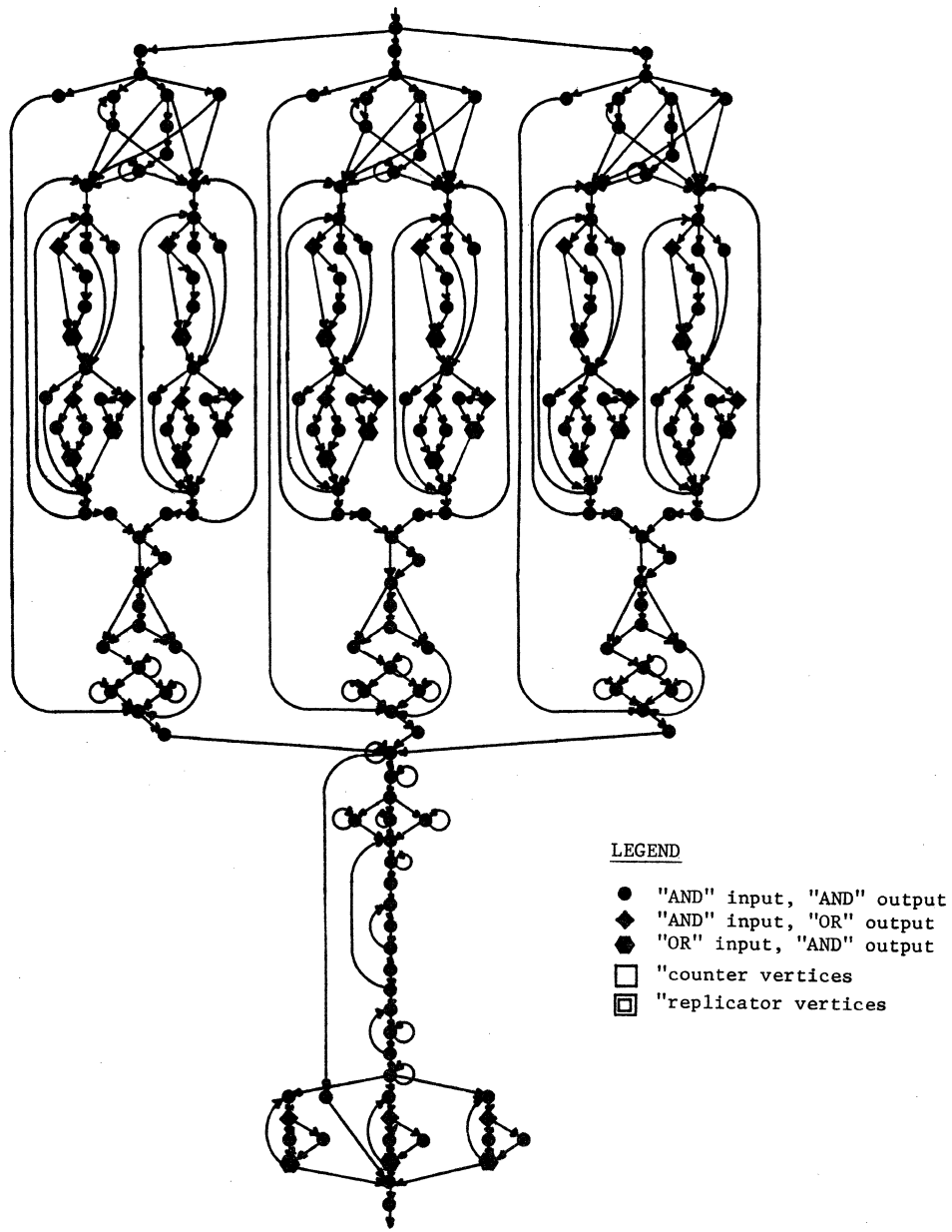
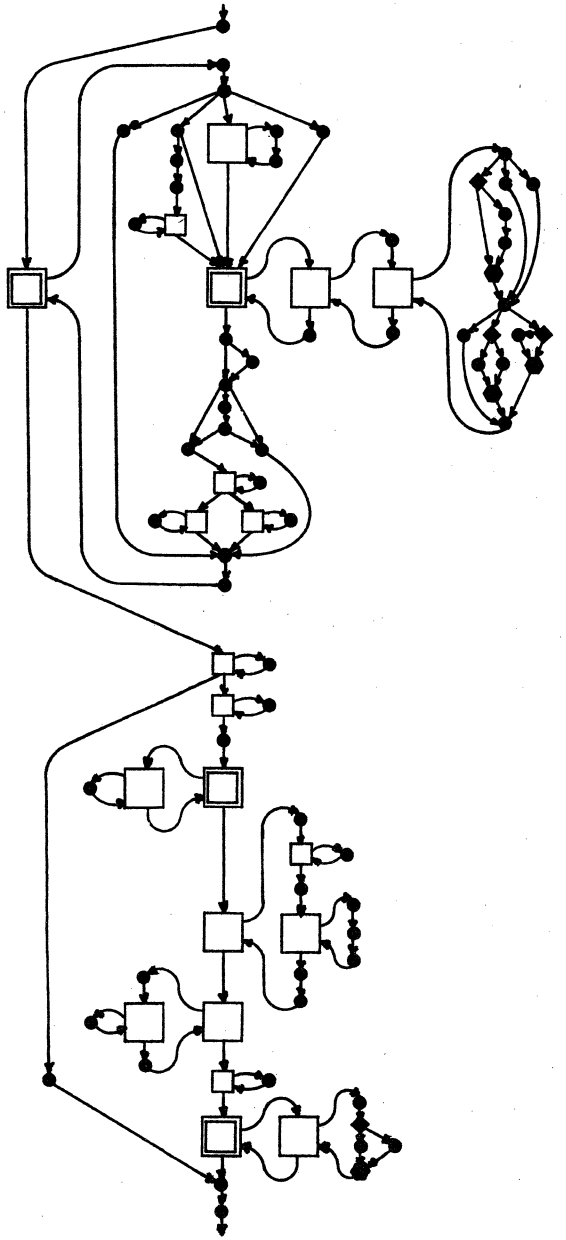


Figure 1b - Modularized Program Graph



Our purpose is to simulate both the cyclic and acyclic versions of the graphs in order to determine the ralative advantages of each modeling technique. The final transformation performed on the graphs made use of graph modules and replication vertices [9] to implement the cycles in the graphs directly. Returning to the original cyclic graphs, each loop was replaced by an appropriate graph module controlled by one of the replication vertex forms provided in the extended program model. The result may be seen by comparing Figures 1a and b. Significant savings are accomplished in both the number of tasks to be specified by tables and in the simplicity of the overall graph. Although simulation proved to be considerably longer in terms of both simulated and real execution times in the modularized case, due to the repeated simulation of each loop for each replication, control structures of this kind must be investigated for the actual implementation of real programs in real multiprocessing systems.

Modeling the Invoking of Algorithms

Several scheduling algorithms have been suggested for use in multiprocessing systems. In this paper we consider a number of heuristics based on task lengths and program structure, leaving user job priorities for later consideration. Two approaches were taken in [7], one based on longest-task-first, the other on a lexicographic ordering from a labeling process giving unique assignments beginning from the exit vertex of the graph. Chen and Epley [3] used the latest allowable starting times, while Martin and Estrin [14] used a variety of heuristics, including first-in-first-out, shortest task first, longest task first, longest path to exit, expected path length to exit, and number of successors. Adam [1] compares some of these with similar path calculations measured from the entry vertex.

Additional algorithms appear with the inclusion of memory requirements as a recognized scheduling parameter. The work reported in [3] considers both channel speed and memory size in finding schedules, while [11] introduces a "two-dimensional" scheduling strategy which uses an optimistic initial guess for the schedule completion time to initiate an iterative trial method for finding optimal schedules satisfying the memory requirement. Analysis of these and other memory-oriented task-scheduling algorithms is available in [12]. The limitation on these algorithms is that they are intended for multiprogramming systems where there are no inter-job dependencies. We wish to study multiprocessing systems where partial orderings exist between tasks due to the structure of the program graph of which all tasks are a part.

The heuristics chosen for study may be divided into three groups: local, global, and comparison standards. The local algorithms (those in which priorities are calculated from the parameters of the task alone) are most immediate successors first (MISF), longest time first (LTF), shortest time first (STF), and largest memory first (LMF). The global algorithms (those in which priority calculations require knowledge of the graph structure and the priorities of other vertices) are most total successors first (MSF), highest level first (HLF), highest weighted level first (HWLF), longest expected path length first (LPF), and longest weighted path length first (LWPF). The standards of comparison are first-in-first-out (FIFO) and random selection (RAND). The reason for having two comparison standards is that FIFO is the simplest to implement (no sorting at all), while purposely randomizing the list eliminates any bias that may have been entered by the polling algorithm.

The amount of difficulty to compute each of these heuristics is determined primarily by the same group classification. The comparison standards are essentially no work at all, while the local algorithms simply select one of the task attributes and assign it as a priority. The global heuristics, however, require more computation. The "level" of a vertex is defined as the number of vertices in the longest path from that vertex to the exit vertex (inclusive), while the "expected path length" is defined as the combinatorial average of the probabilities of each possible path to the exit, based on the output probabilities associated with each vertex. (The actual calculation of these heuristics can be done by a polynomial process [15].) For the "weighted" algorithms, the number of vertices in each path is replaced by the sum of the estimated times for each vertex in that path.

The algorithms we have discussed may be invoked either statically or dynamically. The static or list-schedule technique is to order all tasks prior to execution according to whatever heuristic is chosen, and to assign the list members to processing units on a first-come-first-served basis. More complicated schemes may do the assignment in advance as well by creating a separate list of tasks for each processing unit. The dynamic approach differs in that the task priorities may be modified at run-time as more information becomes available about each task and and the status of the program graph. This is particularly true of global heuristics, since they are based upon graph structures, where already-executed portions of the graph may be removed and the remaining task priorities recalculated. While the dynamic method produces better schedules in general, due to increased knowledge, it suffers from the problem that an inordinate amount of time may be spent recalculating the task priorities.

The basic strategies chosen for testing in this paper are invoked in a mixed fashion, in that the scheduling priorities are pre-calculated (thus static) while the tasks are assigned to processors dynamically upon processor request. The "ready" queue is maintained by the operating system polling user jobs for tasks available to be scheduled. Tasks are selected from the queue during program execution, based upon the immediate resource availability and the priority

assigned to each task by the heuristic.

III. The Experiments

Input Selection and Validation

A number of experimental machine configurations and scheduling philosophies may be represented by the model we have discussed. In addition, the class of job mixes which can be modeled for execution on this system is unbounded. We wish to describe specific configurations and mixes in order to exercise the simulation under different scheduling strategies and ascertain the effects of these strategies on the overall system. To achieve this it becomes necessary to gather data from a variety of sources in order to determine the hardware configurations, resource descriptions, and hardware instruction mixes for the machine model, and to select sample program graphs, task descriptions, and resource requirements for the programs to be executed on the modeled system.

The simulations were run on a Xerox Sigma 5 using a FORTRAN and assembly language system to implement a portion of the SIMULA 67 class concept. The simulation program operates on a discrete system-state basis wherein state-tables are modified as events such as job-entry, task completions, scheduling functions, and processor requests for tasks occur. The resource competition and configuration parameters resemble those of [16] in that lists of resources, competitors, priority rules, service times of resources, and load variance conditions describe the particular hardware being simulated by that run. An exterior event generator is used to create the program graphs and task parameters which are to enter the simulation as exogenous events. A general mix of programs, including highly parallel, intermediate, and serial jobs, is generated in the form of tables produceable by compilers, and "executed" on the simulated hardware model. The simulation is then exercised under different scheduling strategies in order to ascertain the effects of these strategies on the overall system.

The general hardware configuration studied was a 32-processor system [8]. When it was determined that sufficient program loads could not be generated within the limits of the simulation host computer, smaller numbers of processors were sampled. The hardware timing parameters were chosen on the basis of currently available hardware units, with the number and types of resource units selected empirically to fit the instruction mixes anticipated. In order to determine the instruction mix parameters, traces were taken on the XDS Sigma 5 from what were considered to be typical programs: a double-precision FORTRAN subroutine, a series of FORTRAN IO statements, the XPL/S compiler, and a sample run of the SIMTRAN simulation package. The results of the traces are reported as percentages of instructions counted in Table 1. Over 600,000 instructions were counted for each mix.

With the variety and volume of input data and model parameters available, it becomes necessary to make some preliminary studies of the data and calculations made upon it to determine their validity and appropriateness for use in the simulation model. Since the memory interference function must be re-evaluated for each new task to be executed by a processor, its behavior, particularly relating to convergence speed, was studied under a variety of parameter values to determine the numerical accuracy of the analytical solutions [10]. These calculations were also helpful in selecting the numbers of processors and memories and cache hit ratio to be included in the simulation.

Some interesting observations may be made from the duplicate runs on the cyclic and acyclic versions of the test graphs. While the cyclic representation may be considerably more accurate in representing the flow of control in the actual program, the penalty in simulation time is severe. For a loop that is replicated k times, k times as many tasks must be scheduled, resulting in many more polling and scheduling delays. In the example of Figure 1, 21.7 times as many vertices were executed with a 22% increase in elapsed time. These time increases were reflected to a much greater degree in the computation time to perform the simulations. It may be concluded from this that, depending upon the level of detail desired, either method of modeling cycles may have merit. It is necessary to be capable of modeling the cyclic execution, both for understanding of the model and for implementation of real schedulers, but the overhead makes it not worth the effort in a larger-scale view of the system, particularly if the loop count is very large.

Variance and Load Parameters

The subject of variance due to run-time anomalies is an inescapeable part of the multiprocessor system and is included as part of our model. Variance in the instructions executed within a task, variance in executing power due to resource contention, variance in decision branches between tasks, and variance from exogenous variables such as system load characteristics each cast their effect on the overall system. Specific studies of these effects have been performed previously [14] and need not be repeated here, but they are still present in the simulation and must be accounted for in a statistical sense. Results from single runs on any of the experiments reported here tend to lack credibility unless we can be certain that the range of the results is clearly within the range of the variance involved in the experiment.

Since processing time constraints on the simulation make it impossible to repeat all of the experiments sufficiently to reduce the variance in the values reported, a series of 10 runs with different random-number seeds was made on each of the sample graphs in order to establish an estimate for the range of credibility of future experiments. Simulations of one graph executed an

average of 46 vertices in 1023 time units, requiring a total of 1414 units of execution from the processors, with a variance of 14%. In the next graph simulated, 145 vertices were executed in time 590, requiring 3611 processor-time units with less than 1% variance. Simulations of a third graph completed 124 tasks in time 1974 with 8% variance and required 2376 processor-time units with 15% variance. In the final graph (Figure 1) 204 tasks were completed in time 8494 with 2% variance and 33356 processor-time units were required with only 5% variance. The results were similar or better when the modularized graphs were tested in the same manner. In the remainder of this paper, a parameter will be considered significant only when its effect exceeds the variance from these preliminary runs.

One form of variance not studied by [14] is that caused by resource contention. When the system is under sufficient load, tasks with similar instructions mixes will conflict over resource requests and cause variance in the task completion times. In order to observe this phenomenon, a full load is necessary on the system resources which cannot be achieved by a single graph, and increasing the size of the graph creates space problems in the simulation. If we limit the size of the system, considering the graph under study to possess only a portion of the total system, then the model weakens because no interaction exists with the jobs assumed to hold the remaining system resources. What is required is a means of occupying the other processors not immediately concerned with the current graph in such a way that they simulate the contention effects of having other jobs in the system.

One solution is to introduce "dummy" tasks to the system. Whenever a task is to be scheduled (found "initiable" by the token machine) a set number of "ghost" copies of the task are entered in the ready list. This not only causes other processors to become occupied with tasks which will have an effect on the resource contention calculations, but also produces a load on the scheduler and may cause the real tasks from the graph to wait for "dummies" to complete, much as they would have to in a real system containing other jobs from a general mix. The "ghost" tasks differ from the real ones in that they merely "die" upon completion of their processor-computation time and do not cause tokens to be moved as in the case of "real" tasks. The number of "ghosts" created for each "real" task is taken as a system load parameter.

The effects of the load parameter on the system are shown in Figure 2, where each run assumes all tasks to be from the same one of four instruction mixes. The idle time for each mix is graphed as a function of the number of ghost tasks produced, and as would be expected, the effects of the different mixes becomes more distinct as the load increases. The next experiment examined the effect of "balancing" the instruction mix on the system by randomly assigning a different instruction mix to each task. The result was compared with an average taken from the runs made with the separate mixes, in order to show the advantage of scheduling an equal number of tasks from each mix. By balancing the instruction mix on the system in this way, the scheduler may avoid contention bottlenecks which occur when all the processors are executing tasks which request the same resources. The difference found, however, was determined insufficient to justify including the mix parameter as part of the scheduling priority.

Another factor to consider is the source of the idle times that are being measured. We have considered a system where the operating system polls its users for tasks available for scheduling and made the frequency at which the polling occurs a simulation parameter. When at least one task has completed, a flag is set enabling the polling process-object. When the appropriate time interval expires, the graph is polled, or scanned by the scheduler. We may model demand scheduling with this model by simply setting the polling interval to an extremely small value, such that polling will occur precisely whenever the flag is set. The difference in idle time between a polled and a demand system was found to be significant. Because of this difference, the remaining simulations were performed under demand scheduling so that all of the idle time reported may be directly attributed to the scheduling algorithm and/or the configuration under study.

Comparison of Scheduling Algorithms

The idle times of the various scheduling heuristics described in Section II are compared in Figure 3 as functions of the number of processors in the system. In every case we see that the local algorithms (MISF, LMF, LTF and STF) are poor in that they show no improvement over the comparison standards (FIFO and RAND). The global strategies involving some measure of path length (HLF, HWLF, LPF and LWPF) show consistently better performance, particularly in the 4-6 processor region. The pseudo-global urgency MSF gives some improvement over the local urgencies but is not as good as the path length algorithms. Little difference can be seen between the four global strategies.

At this point it becomes necessary (for limitations in computer time) to fix the level of load on the system. This may be accomplished by selecting a specific processor range since the same job stream is used for simulating each configuration. We may observe that for lightly loaded systems (characterized by large idle times) all scheduling algorithms will tend to perform equally well because there is no real contention requiring a careful scheduling decision. For moderately-to-heavily loaded systems (characterized by smaller idle times) the distinguishing performance characteristics of the algorithms will emerge. We select the 4-6 processor range since it seems to typify the moderately-loaded situation.

Figure 2 - Comparison of the Instruction Mixes Under Load

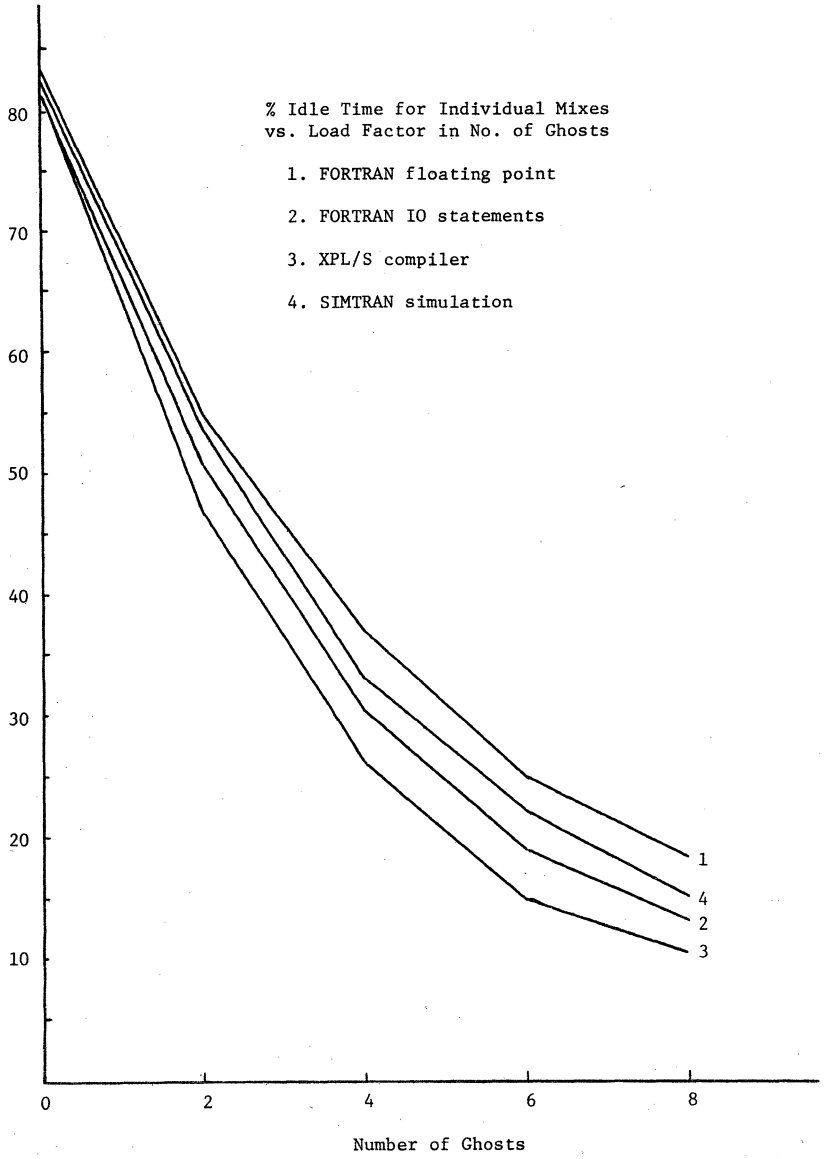
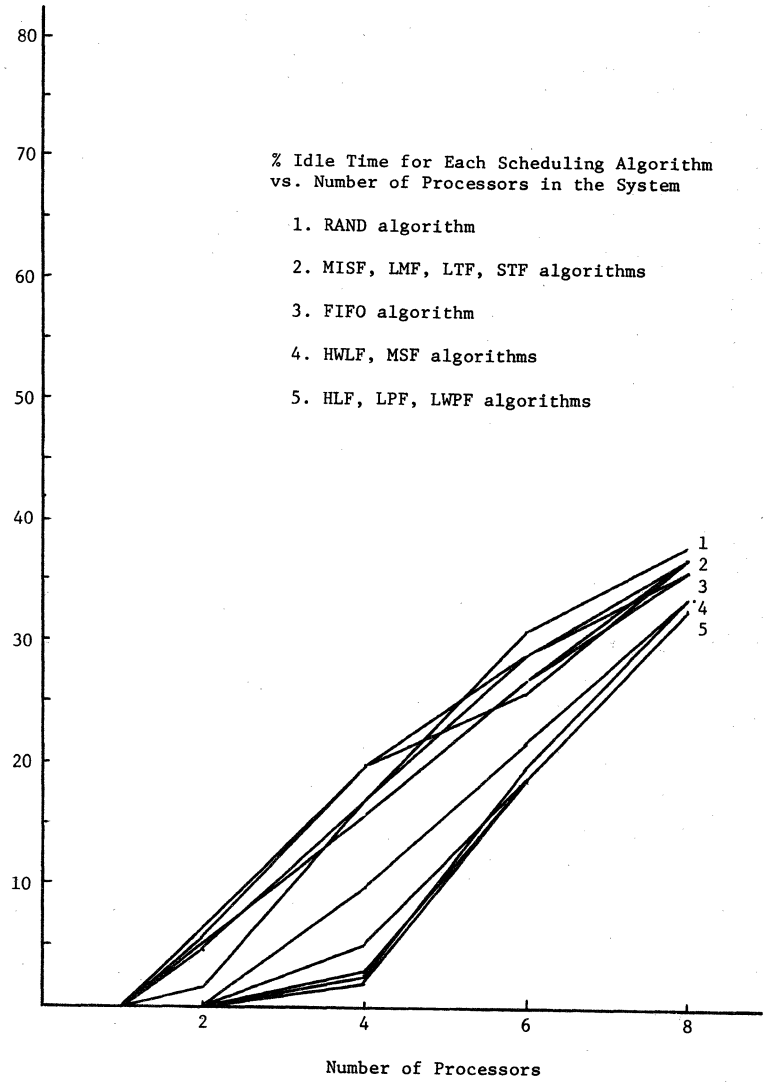


Figure 3 - A Comparison of Several Scheduling Algorithms



One goal in this research was to find a local strategy that would perform near to the global strategies or to determine which of the global strategies would perform best. It was hoped that when memory becomes a critical resource that the LMF algorithm might out-perform some of the others. For the remaining simulation runs, a limit is placed on the total memory occupancy at a given time in the system, causing the shceduler to pass over some tasks with large memory requirements for smaller ones if they do not fit in the available memory. (Contiguity of memory is ignored here.) It is expected that as memory becomes restricted, all of the non memory-oriented strategies will degrade rapidly in their performance to the point where a simple memory strategy like LMF may succeed.

For this reason an investigation was made into the memory capacity limit as reported in Figure 4. The LMF algorithm is shown for a few processor values over a wide selection of memory limits. Dotted lines in the figure indicate the performance in the case where an infinite memory is assumed (i.e. when the memory parameter is ignored by the loader). The lowest memory limit considered was 410 because the largest simulated task will just fit in that space. The result of this investigation is that memory limits of 410, 600 and 800 should serve as an appropriate test range for systems within the established ideal load range of 4-6 processors.

In the final simulation series, the global strategies were run under varying memory capacities in order to see how they measure against the LMF algorithm as memory becomes a critical resource. The results for HLF, HWLF, LPF and LWPF were nearly identical, so HLF was chosen as being representative for the comparisons with RAND, FIFO and LMF. The effect of the memory constraint on each algorithm's performance is shown in Figure 5 for each of the memory limits under consideration. FIFO appears to be consistently better than RAND, although not much difference is noted. The performance of LMF is considerably disappointing. The path length or global algorithms are again clearly better than the others, although the margin of improvement dwindles rapidly when memory becomes critical.

IV. Conclusions

A number of conclusions may be drawn from the experiments performed here, some positive, some negative. A significant step has been made in the modeling of multiprocessor systems. A model has been successfully constructed which is capable of modeling the complete system from many different points of view and at many levels of detail. We have also found that the simpler acyclic program model is reasonable in many instances, that demand scheduling is highly desirable if the scheduling processor can be spared to perform the task often enough, and that more work is needed in specifying efficient implementation details of the scheduler's duties. In the way of scheduling algorithms, we have found that the local heuristics are of no help at all, that all of the global strategies listed are of equal value (therefore pick the cheapest one to implement), and that the usefulness of the global strategies is limited when the time spent on determining these priorities is a critical concern. We were also disappointed by the failure of the LMF attempt at introducing the memory parameter to scheduling in an inexpensive manner.

Acknowledgment

The author is deeply indebted to "Referee A" for his constructive criticism and assistance in the organization of this paper.

References

[1] Adam, T. L., Chandy, K. M. and Dickson, J. R. "A Comparison of List Schedules for Parallel Processing Systems" Comm. ACM 17:12 (December 1974), pp. 685-690.

[2] Baskett, F. and Smith, A. J. "Interference in Multiprocessor Computer Systems With Interleaved Memory" Comm. ACM 19:6 (June 1976), pp. 327-334.

[3] Chen, Y. E. and Epley, D. L. "Memory Requirements in a Multiprocessing Environment" J. ACM 19:1 (January 1972), pp. 57-69.

[4] Garey, M. R. and Graham, R. L. "Bounds on Scheduling With Limited Resources" Proc. 4th Symposium on Operating Systems Principles, October 1973, pp. 104-111.

[5] Gonzalez, M. J. Jr. and Ramamoorthy, C. V. "Parallel Task Execution in a Decentralized System" IEEE Trans. on Comput. C-21:12 (December 1972), pp. 1310-1322.

[6] Gountanis, R. J. and Viss, N. L. "A Method of Processor Selection for Interrupt Handling in a Multiprocessor System" Proc. IEEE 54:12 (December 1966), pp. 1812-1819.

[7] Graham, R. L. "Bounds on Multiprocessing Anomalies and Related Packing Algorithms" Proc. AFIPS 1972 Spring Joint Computer Conf., pp. 205-217.

[8] Jensen, J. E. Dynamic Task Scheduling in a Shared Resource Multiprocessor, PhD Dissertation, University of Washington, August 1976.

[9] Jensen, J. E. "A Graphical Representation of Tasks for Multiprocessing" to appear in Proc. ACM 1977 Annual Conference, October 1977.

[10] Jensen, J. E. and Baer, J. L. "A Model of Interference in a Shared Resource Multiprocessor" Proc. 3rd Symposium on Computer Architecture, January 1976, pp. 52-57.

[11] Kafura, D. G. and Shen, V. Y. "Scheduling Independent Processors With Different Storage

Figure 4 - Investigation of the Memory Capacity Limits

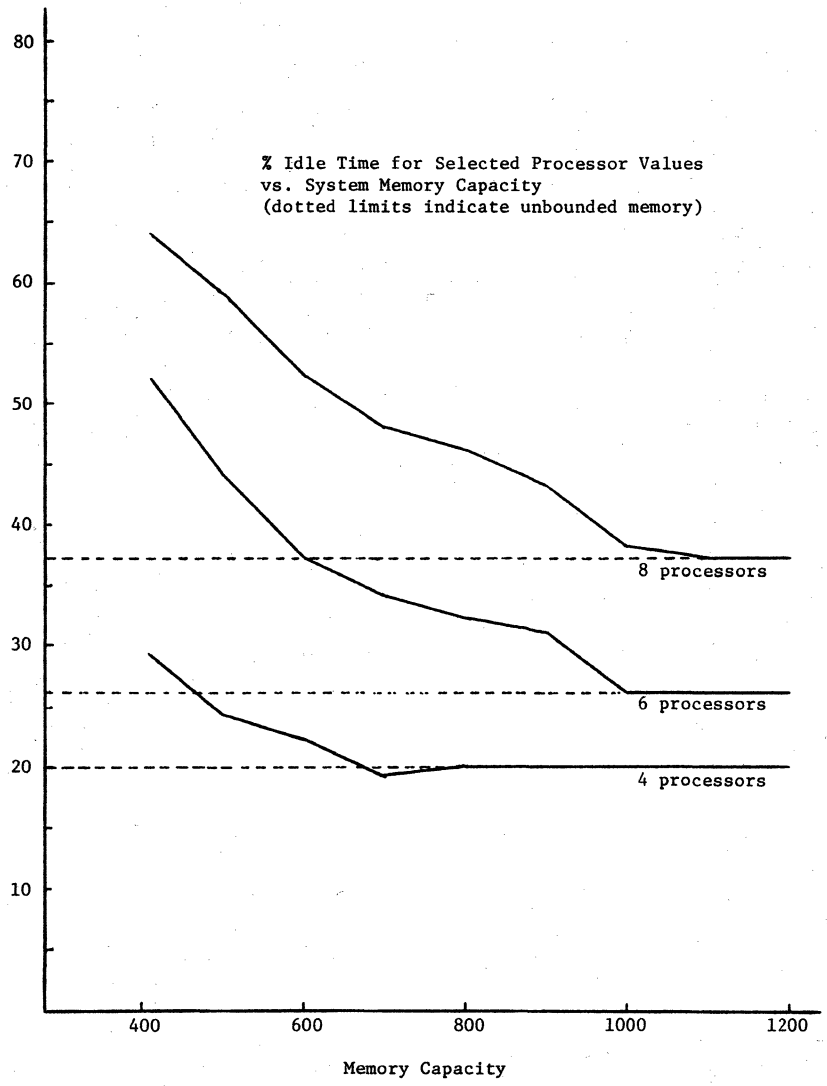
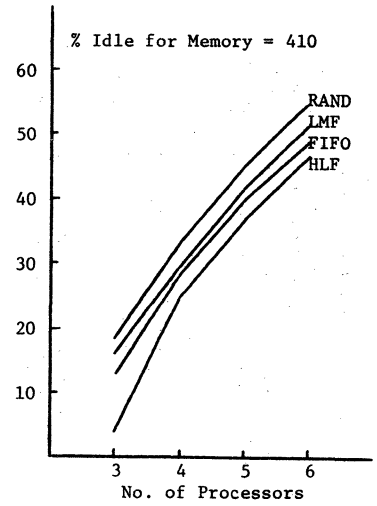
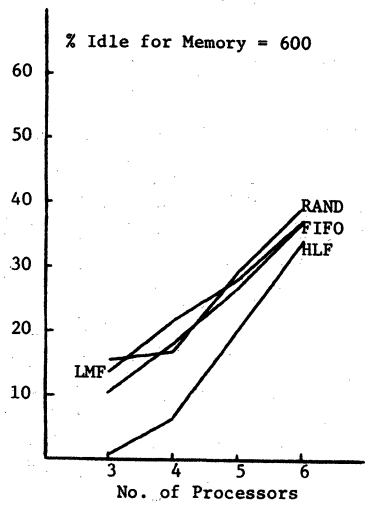
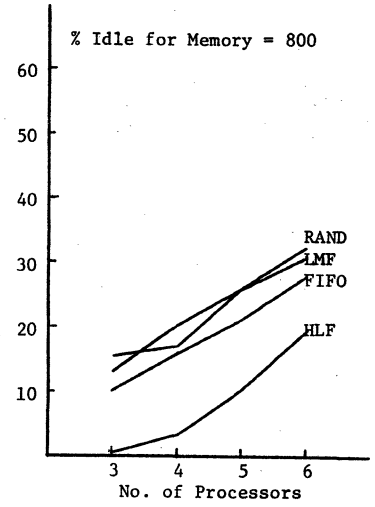
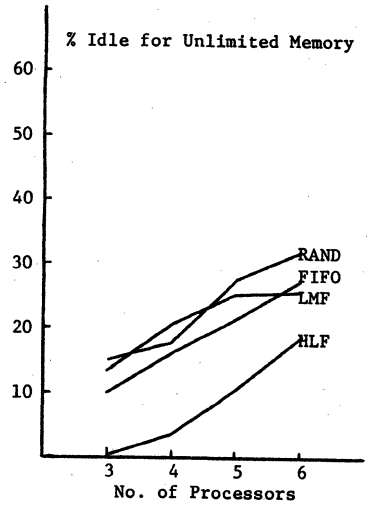


Figure 5 - A Comparison of Algorithms Under Memory Constraints









Capacities" Proc. ACM National Conference, 1974, pp. 161-166.

[12] Krause, K. L., Shen, V. Y. and Schwetman, H. D. "Analysis of Several Task-Scheduling Algorithms for a Model of Multiprogramming Computer Systems" J. ACM 22:4 (October 1975), pp. 522-550.

[13] Martin, D. F. The Automatic Assignment and Sequencing of Computations on Parallel Processor Systems, PhD Dissertation, U.C.L.A., January 1966.

[14] Martin, D. F. and Estrin, G. "Experiments on Models of Computations and Systems" IEEE Trans. on Comput. EC-16:1 (February 1967), pp. 59-69.

[15] Martin, D. F. and Estrin, G. "Path Length Computations on Graph Models of Computation" IEEE Trans. on Comput. C-18:6 (June 1969), pp. 530-536.

[16] Merikallio, R. A. and Holland, F. C. "Simulation Design of a Multiprocessing System" Proc. AFIPS 1968 Fall Joint Computer Conf., pp. 1399-1410.

[17] Mitchell, J. et al. "Multiprocessor Performance Analysis" Proc. AFIPS 1974 National Computer Conf., pp. 399-403.

[18] Texas Instruments A Description of the Advanced Scientific Computer System, Internal Publication M1001P, Texas Instruments, April 1972.

[19] Ullman, J. D. "Polynomial Complete Scheduling Problems" Proc. 4th Symposium on Operating Systems Principles, October 1973, pp. 96-101.

Table 1 - Typical Instruction Mixes[(b)]

Category of Instruction	FORTRAN Floating Point	FORTRAN IO Statements	XPL/S Compiler	SIMTRAN Simulation
Memory Addressed Instructions	.8870	.7685	.7772	.8828
Immediate Addressed Instructions	.1130	.2315	.2228	.1172
Total Memory References	1.6696	1.3932	1.4844	1.4354
Instruction Fetches	1.0000	1.0000	1.0000	1.0000
Operand Addresses	.7244	.3713	.5342	.4930
Operand in Memory	.6360	.3462	.4636	.4085
Register Operands	.0884	.0251	.0706	.0845
Operand Fetches	.5916	.2201	.3980	.3792
Operand Stores	.1328	.1512	.1363	.1138
Indirect Memory References	.0336	.0470	.0208	.0269
Total "P.c" Instructions	.5367	.6727	.7067	.7732
Branch Instructions	.1626	.3972	.2430	.3898
(Successful Branches)	.1501	.2912	.1696	.3281
Load/Store Instructions	.3675	.2726	.4637	.3834
"Execute" Instructions	.0066	.0029	.0000	.0000
Total "D.e" Instructions	.4699	.3939	.3147	.2420
Int. Add, Subtract, Compare	.0714	.3282	.2431	.1428
Floating Add, Subtract	.1668	.0003	.0000	.0197
Multiply Instructions	.1339	.0079	.0001	.0133
Divide Instructions	.0371	.0024	.0003	.0006
Logical Instructions	.0134	.0149	.0313	.0115
Shift Instructions	.0473	.0400	.0471	.0138
Monitor and IO Calls	.0000	.0002	.0003	.0403

[(b)] Values expressed as ratios to the number of instructions counted. Over 600,000 instructions were counted per mix.

PERFORMANCE EVALUATION OF A PARALLEL SYSTEM PROCESSING FAULT-TOLERANT PROGRAMS†

K. H. Kim and M. J. Jenson
Department of Electrical Engineering
and Computer Science
University of Southern California
Los Angeles, California 90007

Abstract. A parallel (multiprocessor) system processing fault-tolerant programs was developed in [4, 5]. The system performance is evaluated in this paper, using an analytic approach based on stochastic models. The analysis confirms the high effectiveness of a parallel system, under all practical circumstances, in reducing the program execution time increase due to run-time validation and system state saving. It also shows how the system performance is affected by various program characteristics.

1. Introduction

A system architecture for parallel execution of fault-tolerant programs (i.e., programs containing redundancy for the tolerance of residual program errors and/or hardware faults [7]) was developed in [4, 5]. The system was designed to execute block-structured fault-tolerant programs developed by Horning et al. [3]. A fault-tolerant block or recovery block is the basic component containing redundancy in these programs and has the following structure: ensure T by O_1 else-by O_2 else-by ... else-by O_n else-error, where T denotes the validation test, O_1 the primary object block, and O_k $(1 < k \leq n)$ the alternate object blocks. All of the object blocks in a fault-tolerant block F compute the same or approximately the same objective function. The validation test T is executed on exit from an object block to confirm that the object block has performed acceptably. The execution of a validation test results in either an acceptance (i.e., confirmation) or a rejection. If accepted, control exits from the fault-tolerant block. If the result produced by an object block is rejected, the next alternate is entered. After the alternate object block finishes its computation, the validation test is repeated. Before an alternate object block is entered, the system state is restored to the state that existed just before entry to the primary object block [1, 2, 3]. To enable this, a state vector that contains the values of all the variables (that may be changed by the object blocks) is saved on entry to a fault-tolerant block.

The goal of the parallel execution is to overlap, as much as possible, execution of object blocks with the validation and system state saving. In this paper, we evaluate the performance of the parallel system. The approach used in this paper for performance evaluation is of an analytic nature and is based on stochastic models for both the parallel system and the sequential system (i.e., one in which the execution of an object block is not overlapped with the execution of a validation test). The evaluation shows the performance gain by parallel execution over sequential execution.

In the next section major characteristics of both an efficient sequential system and a parallel system are compared. Section 3.1 deals with the evaluation of the sequential system. Performance of the parallel system is evaluated in Section 3.2 and compared with the performance of the sequential system in Section 3.3.

2. Distinguishing Characteristics of a Sequential System and a Parallel System

In this section two systems, a sequential system using a memory organization called a recovery cache [1, 3] and a parallel system using a duplex memory [4, 5], are briefly sketched.

The essence of the recovery cache scheme is to save the "original value" of each non-local variable W together with its logical address right before the variable is modified for the first time in a new object block. The original values are thus saved in a compact

† This work was supported in part by the Joint Service Electronics Program under Air Force Contract F44620-76-c-0061.

table structure. For illustration, the fault-tolerant program in Figure 1a is used.

Figure 1b shows a snapshot of the recovery cache taken when primary object block $O_{2.1}$ is in execution. As shown, there is a stack, called the cache stack, used for saving the original values. Similar to the main stack, the cache stack is also divided into regions, one region for each nested fault-tolerant block in the "active" state (i.e., a fault-tolerant block that has been entered but not exited). The top region of the cache stack in Figure 1b contains previous values of non-local variables together with their names (representing logical addresses), i.e., Y2, X1, X2, which have been modified during execution of the current object block $O_{2.1}$. Similarly, the bottom region of the cache stack contains the previous value of non-local variable X1 which had been modified by execution of object block $O_{1.1}$ before $O_{2.1}$ was entered. Figure 1b also shows a flag field in the main stack. The flag attached to a variable indicates whether the original value of the variable has already been saved since the current object block was entered. Thus the flags attached to Y2, X1, X2 in the main stack are currently set.

If the result produced by execution of $O_{2.1}$ fails the validation test V_2, then the top region C_2 of the cache stack can be used to reset the main stack to the state that existed on entry to fault-tolerant block F_2. If it passes the test, execution of F_2 is complete and C_2 is merged into C_1 so that the result will contain previous values of those variables which are non-local to $O_{1.1}$ and have been modified since $O_{1.1}$ was entered. Thus the result will be a single region containing (X1, 9) and (X2, 2). Flags in the main stack are also adjusted such that only flags of X1 and X2 are set. Therefore, the combination of the main and cache stacks usually contains information with which several old state vectors can be reconstructed.

In the case of parallel execution at least two processors are used, a main processor for object block execution and a VR-(validation and recovery) processor or audit processor for execution related to validation and recovery. It is necessary to save a state vector on exit from an object block since the state vector is used by both the main processor and the VR-processor. This is accomplished by simultaneously storing the operand of each WRITE operation into two locations, one in the main stack and the other in the VR-store. When the main processor enters a fault-tolerant block F, a VR-store-segment is created to keep an execution image which consists of records of assignments made by an object block in F. A VR-store-segment consists of two sections, the L-(local variable) section for keeping records of assignments to variables local to the object block in execution and the N-(non-local variable) section for assignment records of non-local variables. A variable local to the object block being entered is allocated one location in the main stack and one location in the L-section of a VR-store-segment. New values assigned to variables that are non-local to the object block in execution are recorded together with the logical addresses (of the variables) in a table structure in the N-section of a VR-store-segment.

For illustration, Figure 1c shows the content of the VR-store at an instant during execution of the program in Figure 1a by a parallel system using a duplex memory. When the main processor entered the program (i.e., the outermost block), VR-store-segment S_0 was created to keep assignment records of local variables X1 and X2. Since there are no variables non-local to the outermost block, S_0 does not contain a N-section. When the main processor entered F_1, VR-store-segment S_1 was created. When non-local variable X1 was assigned the value "8" during execution of object block $O_{1.1}$, a table entry (X1, 8) was made in S_{1N}. Similarly, S_2 was created when the main processor entered F_2 and was filled by execution of object block $O_{2.1}$. The content of the main stack in a duplex memory is that in a recovery cache minus the flag field.

On completion of $O_{2.1}$, the main processor proceeds to the execution of F_3 (which will be imaged in a new VR-store-segment S_3) while the VR-processor starts examining the execution image in S_2 by execution of V_2. If the result produced by execution of $O_{2.1}$ (kept in S_2) fails the validation test V_2, then the non-local variables recorded in S_{2N} (and S_{3N}, if not empty) are those which need to be reset. Segments S_0 and S_1 contain the values of the variables that existed when the main processor entered fault-tolerant block F_2 and their values may be used to reset the main stack. A duplex memory may be implemented such that the previous value can be obtained in a single content-addressable memory (CAM) cycle [4, 5]. If the result of $O_{2.1}$ passes V_2, S_{2L} is discarded and S_{2N} is merged into S_1 so that the result contains the assignment records, of the variables addressable in $O_{1.1}$, made since $O_{1.1}$ was entered. This will result in S_{1L} containing "1","5" and "3" for Y1, Y2, Y3, respectively and S_{1N} containing (X1, 7) and (X2, 8).

Let us now compare the characteristics of the recovery cache scheme for sequential execution with the characteristics of the duplex memory scheme for parallel execution.

1. In both schemes, content-addressable memory modules are needed to obtain an acceptable level of performance in program execution and in the rest of this paper, the use of CAM modules is assumed.

2. The duplex memory takes more space than the recovery cache.

3. The WRITE operation into a non-local variable W involves two steps with the recovery cache, the first step being used for fetching the original value or the flag, while the WRITE operation takes one step (CAM cycle) with the duplex memory. Therefore, the execution of an object block is slower with the recovery cache than with the duplex memory.

4. Overall, it is expected that the recovery cache takes less merging time than the duplex memory. During the execution of a program in which no fault-tolerant block is nested within another fault-tolerant block, there is no merging involved with the recovery cache.

5. The parallel system is slower in recovery because (a) recovery of a variable takes more steps with the duplex memory than with the recovery cache and (b) there are more variables that need to be recovered in the parallel system because while an execution image is being validated, the main processor normally proceeds to the successor block(s).

In summary, the parallel system largely trades recovery time increase for the reduction of total program execution time. There are cases, though highly impractical, where the performance of the parallel system is inferior to the performance of the sequential system. Let α denote the reliability of an object block, i.e., the probability of an average object block producing an accepted execution image. Then there is a lower bound α_L for α such that when $\alpha > \alpha_L$, the parallel system performs more efficiently than the sequential system. This lower bound is one of the values of interest examined in subsequent sections.

3. Performance Evaluation

Given a fault-tolerant program, the average execution time of a fault-tolerant block is defined as the execution time of the program divided by the number of fault-tolerant blocks executed during the program execution. T_s and T_p denote the average execution time of a fault-tolerant block by the sequential system and by the parallel system, respectively. The system throughput is defined as the number of fault-tolerant blocks completed per unit time and is given by the inverse of the average execution time of a fault-tolerant block. We denote the sequential system throughput and the parallel system throughput by THR_s and THR_p, respectively. Throughputs are used in this section as measures of the performance of the sequential system and of the parallel system.

For mathematical tractibility, the following set of global assumptions have been adopted throughout the performance evaluation.

Assumption G

G.1 The programs considered in this analysis are of the type in which no fault-tolerant block is nested within another fault-tolerant block and whose execution becomes a sequential chain of fault-tolerant block executions (Figure 2).

G.2 Primary and alternate object blocks take the same average execution time.

G.3 Each fault-tolerant block contains an unlimited number of alternate object blocks (to eliminate the case of program failure).

In executing a program satisfying assumption G.1, the sequential system does not involve assignment record merging, as mentioned in Section 2. This assumption G.1 is adopted because of the difficulties in (1) dealing with a large spectrum of legitimate program structures, (2) keeping accounts of various execution times during execution of a general program (i.e., a program in which fault-tolerant blocks are nested one within another), etc. However, it is conjectured that results in this paper of performance comparison between two systems for programs satisfying G.1 will not be far different from the results for general programs.

3.1 Throughput Evaluation for the Sequential System

The behavior of the sequential system during execution of a fault-tolerant block is depicted in Figure 3a. The system first enters the "object block execution" state s_o in which the processor executes an object block within the current fault-tolerant block. On completion of an object block, the system enters the "validation" state s_v in which the processor executes the validation test. If the validation results in a rejection, the system enters the "recovery" state s_r, and on completion of the recovery, the system again enters s_o in which the processor executes an alternate object block. If the validation results in an acceptance, the system proceeds to the execution of the successor fault-tolerant block and repeats the above behavior.

During execution of fault-tolerant programs satisfying assumption G, the sequential system continuously repeats the process depicted in Figure 3a. We thus model the system behavior by the following stochastic process for the purpose of evaluating THR_s.

Model S

S.1 There are three states which the sequential system may enter: s_o - object block execution, s_v - validation, and s_r - recovery. (Due to assumption G.1 there is no merging state.)

S.2 The time during which the system is in any state is exponentially distributed.

S.2.1 When the system is in state s_o, the rate gs of generating an execution image (i.e., the probability of the system completing the execution of an object block within an infinitesimal time interval Δt is $gs^*\Delta t$), is $gs = 1/t_{os}$ where t_{os} denotes the mean object block execution time in the sequential system. gs is called the generation rate.

S.2.2 When the system is in state s_v, the rate v of completing the validation, called the validation rate, is $v = 1/t_v$ where t_v denotes the mean validation time.

S.2.3 When the system is in state s_r, the rate rs of completing the recovery, called the recovery rate, is $rs = 1/t_{rs}$ where t_{rs} denotes the mean recovery time in the sequential system.

S.3 The probability of the system entering state s_o after leaving state s_v is α, while the probability of entering state s_r is $\alpha' = 1 - \alpha$.

Figure 3b depicts Model S. Let p_o, p_v, p_r denote the equilibrium probabilities [6] of the system being in s_o, s_v, s_r, respectively. The steady-state behavior of the system is expressed by the following equilibrium equations.

$$p_o \cdot gs = p_r \cdot rs + p_v \cdot v \cdot \alpha$$

$$p_v \cdot v = p_o \cdot gs \qquad (1)$$

$$p_o + p_v + p_r = 1 \text{ (normalizing equation).}$$

Solving Eq. 1, we obtain

$$p_o = rs \cdot v/(gs \cdot v \cdot \alpha' + rs \cdot v + gs \cdot rs)$$

$$p_v = rs \cdot gs/(gs \cdot v \cdot \alpha' + rs \cdot v + gs \cdot rs) \qquad (2)$$

$$p_r = gs \cdot v \cdot \alpha'/(gs \cdot v \cdot \alpha' + rs \cdot v + gs \cdot rs).$$

By definition system throughput is equal to the number of execution images accepted per unit time. Throughput THR_s and its inverse T_s can thus be obtained as follows.

$$THR_s = p_v \cdot v \cdot \alpha$$
$$= rs \cdot gs \cdot v \cdot \alpha/(gs \cdot v \cdot \alpha' + rs \cdot v + gs \cdot rs) \qquad (3a)$$

$$T_s = 1/THR_s$$
$$= (gs \cdot v \cdot \alpha' + rs \cdot v + gs \cdot rs)/(rs \cdot gs \cdot v \cdot \alpha)$$
$$= (1/\alpha) \cdot (t_{os} + t_v) + (\alpha'/\alpha) \cdot t_{rs}. \qquad (3b)$$

3.2 Throughput Evaluation for the Parallel System

In most cases the main processor need not be synchronized with the VR-processor. However, when the next fault-tolerant block to be executed specifies irreversible actions of critical nature, the main processor waits until the VR-processor accepts all the execution images in the queue (i.e., the execution images of the predecessor fault-tolerant blocks) [4, 5]. An execution image generated immediately before a block specifying an irreversible action is entered, is a "synchronizing" execution image (or for short, S-image). The other execution images are "normal" execution images (or N-images).

An abstract representation of the parallel system with unbounded queue is shown in Figure 4. The main processor continuously constructs execution images and puts the completed execution images into the queue of execution images except when (1) the VR-processor stops it on rejection of an execution image and enters the recovery state, or (2) the main processor has generated a synchronizing execution image and put it into the queue. The VR-processor validates execution images in the order of their arrival. When it accepts an execution image, it enters the "merging" state. On completion of merging, it checks if another execution image is waiting in the queue. If an execution image is rejected, the main processor is stopped and recovery is initiated. Recovery involves a sequence of assignment reversals using the assignment records in the execution images and thus can be thought of as a process of "erasing" the execution images in the queue. On completion of the recovery, the queue is empty and the main processor is restarted. The parallel system is thus modeled by the following stochastic process.

Model P

P.1 The state of the system at any instant is characterized by (1) the state of the VR-processor which may be in wait, validation, merging or recovery, and (2) the number and types of execution images in the queue. The state of the main processor is busy or waiting

and is determined by the state of the VR-processor and the state of the queue. Thus each system state is denoted by

$$s_{\text{VR-processor state, queue state}},$$

where (1) VR-processor state = w (wait), v (validation), m (merging), or r (recovery), and (2) queue state = $\emptyset$ (empty), N (one normal execution image), S (one synchronizing execution image), \$ (= N or S), NN, NS, \$N, \$S, NNN, NNS, \$NN, \$NS, ...

Some possible states of the system are shown in Figure 5, where some possible state transitions are also indicated. For example, $s_{v,N}$ is the state where the queue contains one normal execution image which the VR-processor is validating. There are four states which the system may enter from $s_{v,N}$: $s_{v,NN}$ which is entered if the main processor generates another normal execution image; $s_{v,NS}$ which is entered if the main processor generates a synchronizing execution image; $s_{m,\$}$ which is entered if the VR-processor accepts the normal execution image in the queue; and $s_{r,N}$ which is entered if the VR-processor rejects the normal execution image in the queue. In $s_{r,N}$ the system erases the normal execution image in the queue and thereafter enters state $s_{r,\emptyset}$ in which the system erases the partially constructed execution image contained within the main processor. Note that the type of the first image in the queue is not distinguished in some states (e.g., $S_{m,\$N}$). This is because once an execution image is accepted, the system's future behavior is independent of the type of the execution image just accepted.

P.2 The time during which either processor is in a particular state is exponentially distributed.

P.2.1 When the main processor is in a busy state, the generation rate gp is $gp = 1/t_{op}$, where t_{op} represents the mean object block execution time (which is different from t_{os}).

P.2.2 When the VR-processor is in a validation state, the validation rate v is $v = 1/t_v$, where t_v represents the mean validation time.

P.2.3 When the VR-processor is in a merging state, the rate mp of completing the merging, called the merging rate, is $mp = 1/t_{mp}$ where t_{mp} represents the mean merging time.

P.2.4 When the system is in a recovery state other than $s_{r,\emptyset}$, the rate rp of erasing an execution image, called the recovery rate, is $rp = 1/t_{rp}$ where t_{rp} represents the mean time for erasing an execution image.

P.2.5 The size of the partially constructed execution image remaining within the main processor when the system enters a recovery state is assumed to be proportional to the amount of time that the main processor has spent in construction of that execution image. Borrowing a result in the renewal theory, the mean size of the execution image partially constructed (when the system enters a recovery state from a state where the main processor is busy), is the same as the mean size of a completed execution image [6]. Thus when the system is in $s_{r,\emptyset}$, the rate of moving from $s_{r,\emptyset}$ to $s_{w,\emptyset}$ is also rp.

P.3 The probability of a validation resulting in an acceptance is α as before, while the probability of a rejection is $\alpha' = 1 - \alpha$.

P.4 The probability of a newly generated execution image being an N-image is η, while that for being an S-image is $\eta' = 1 - \eta$.

Figure 5 depicts Model P. It also shows the notation for the equilibrium probability of the system being in each state $s_{i,j}$. The probabilities are denoted by I (for $s_{w,\emptyset}$), J (for $s_{m,\$}$), z_k, y_k, x_k, w_k, u_0 (for $s_{r,\emptyset}$), u_k, and q_k, where $k = 1, 2, \ldots$ except that there does not exist y_1 nor x_1. The subscript k indicates the number of execution images present in the queue. The steady-state behavior of the system is then expressed by the following equilibrium equations.

(a) $I \cdot gp = J \cdot mp + u_0 \cdot rp + q_1 \cdot rp$

(b) $J \cdot (gp + mp) = (z_1 + w_1) \cdot v \cdot \alpha$

(c) $z_1 \cdot (gp + v) = I \cdot gp \cdot \eta + y_2 \cdot mp$

(d) $z_k \cdot (gp + v) = z_{k-1} \cdot gp \cdot \eta + y_{k+1} \cdot mp$ for $k = 2, 3, \ldots$

(e) $y_2 \cdot (gp + mp) = J \cdot gp \cdot \eta + z_2 \cdot v \cdot \alpha$

(f) $y_k \cdot (gp + mp) = y_{k-1} \cdot gp \cdot \eta + z_k \cdot v \cdot \alpha$ for $k = 3, 4, \ldots$

(g) $x_2 \cdot mp = J \cdot gp \cdot \eta' + w_2 \cdot v \cdot \alpha$

(h) $x_k \cdot mp = y_{k-1} \cdot gp \cdot \eta' + w_k \cdot v \cdot \alpha$ for $k = 3, 4, \ldots$

(i) $w_1 \cdot v = I \cdot gp \cdot \eta' + x_2 \cdot mp$

(j) $w_k \cdot v = z_{k-1} \cdot gp \cdot \eta' + x_{k+1} \cdot mp$ for $k = 2, 3, \ldots$

(k) $u_0 \cdot rp = u_1 \cdot rp$

(l) $u_k \cdot rp = z_k \cdot v \cdot \alpha' + u_{k+1} \cdot rp$ for $k = 1, 2, \ldots$

(m) $q_k \cdot rp = w_k \cdot v \cdot \alpha' + q_{k+1} \cdot rp$ for $k = 1, 2, \ldots$

(n) $I + J + u_0 + \sum_{k=1}^{\infty} (z_k + y_k + x_k + w_k + u_k + q_k) = 1$ (normalizing equation) (4)

Solving this system of equations, we can obtain the quilibrium probabilities. This system can be solved in closed form, but the solution pro-

cedure is not described here. Since the system throughput THR_p was defined as the number of acceptances made per unit time, THR_p and T_p can be obtained by

$$THR_p = v \cdot \alpha \cdot \left(\sum_{k=1}^{\infty} z_k + \sum_{k=1}^{\infty} w_k \right)$$

$$T_p = 1/\left(v \cdot \alpha \cdot \left(\sum_{k=1}^{\infty} z_k + \sum_{k=1}^{\infty} w_k \right) \right). \qquad (5)$$

Another measure of interest is the expected queue-length E(QL).

$$E(QL) = J + \sum_{k=1}^{\infty} (k \cdot (z_k + y_k + x_k + w_k + u_k + q_k))$$

$$\text{where } y_1 = x_1 = 0 . \qquad (6)$$

Figure 6 depicts the expected queue-length E(QL) for various values of $\alpha, \eta, t_v/t_{op}$, $t_{mp}/t_{op}, t_{rp}/t_{mp}$. In examining Figures 6 and 7 we are mostly interested in the cases where α is greater than 0.9. Since fault-tolerant programs dealt with here are supposed to have undergone a testing phase before being put into operation, one or more erroneous object blocks out of ten seems highly improbable. On the other hand, η is application-dependent and may not be very close to 1. For example, $\eta = 0.999$ implies that only one among 1000 execution images generated is an S-image. In this evaluation, η is set mostly within the range of 0.9 - 0.95 and the most frequently used values are 0.9 for η and 0.95 for α. The following practical constraints were also adopted.

$$t_v < t_{op}$$

$$t_{mp} < t_{op}$$

$$1 < t_{rp}/t_{mp} \leq 1.5 . \qquad (7)$$

As expected, E(QL) becomes larger as α or η increases. Furthermore, comparison of curve 3 in Figure 6a (which is a result of changing α when $\eta = 0.95$) with curve 2' (a result of changing η when $\alpha = 0.95$) indicates that E(QL) is more sensitive to the change of η than to the change of α. This is also shown by a comparison of curve 2 (a result of changing α when $\eta = 0.9$) with curve 1' (a result of changing η when $\alpha = 0.9$). Figure 6b shows that E(QL) increases as mean validation time t_v or mean merging time t_{mp} increases. When $t_v + t_{mp} < t_{op}$, E(QL) is generally smaller than 5. The data obtained but not plotted in Figure 6 indicated that mean recovery time t_{rp} affects E(QL) to a negligible extent. This is because (1) when α is large, the system rarely enters a recovery state and (2) when α is small, the system rarely enters a state where the queue-length is large.

3.3 Performance Comparison Between the Sequential System and the Parallel System

A simple way of assessing the performance of the parallel system is to compare the throughput THR_p with the throughput THR_s of the sequential system. THR_p/THR_s is then the throughput ratio and is a function of α, η, t_v/t_{op}, t_{mp}/t_{op}, t_{rp}/t_{mp}, t_{os}/t_{op}, and t_{rp}/t_{rs}. Here t_{os}/t_{op} represents the object block execution time ratio while t_{rp}/t_{rs} represents the recovery time ratio. These parameters are within the following ranges (cf. Section 2 or [5] for more details).

$$1 < t_{os}/t_{op} \ll 2$$

$$1 < t_{rp}/t_{rs} < 1.5 . \qquad (8)$$

Figure 7 depicts the throughput ratio for various values of parameters subject to the constraints in Eqs. (7) and (8). First, Figure 7a discloses that variation of recovery time ratio t_{rp}/t_{rs} within a practical range has little effect on the throughput ratio. This is again because (1) when α is large, the system rarely gets into a recovery state, and (2) when α is small, E(QL) becomes small and thus a recovery involves mostly a small number of execution images. Figure 7b indicates that the throughput ratio is not much affected by the change of t_{rp}/t_{mp} for α within a practical range, while it is significantly affected by object block execution time ratio t_{os}/t_{op}. Object block execution time ratio t_{os}/t_{op}, recovery time ratio t_{rp}/t_{rs} and t_{rp}/t_{mp} are machine characteristics while other parameters represent program characteristics.

Figure 7c shows that the throughput ratio decreases as merging time t_{mp} (more precisely t_{mp}/t_{op}) increases. The obvious reason is because under assumption G.1 merging is involved only in parallel execution. It also shows that increase of t_v causes a throughput ratio increase approximately until $t_v + t_{mp}$ surpasses t_{op} but further increase of t_v does not change (actually slightly decreases) the throughput ratio. This can be explained as follows. As $t_v + t_{mp}$ becomes larger than t_{op}, E(QL) becomes large and thus, each time a synchronizing execution image is generated, the queue contains a large number of execution images. The validation and merging of these are not overlapped with object block execution. Figure 7d confirms the expectation that as η increases, the throughput ratio increases.

In summary, (1) for a practical α, the performance improvement by parallel execution is most sensitive to object block execution time ratio t_{os}/t_{op} and t_{mp}/t_{op}, less sensitive to t_v/t_{op} and the least sensitive to t_{rp}/t_{mp} and recovery time ratio t_{rp}/t_{rs}, and (2) the throughput ratio ranged over 1.02 - 1.65 (or 2 - 65% gain) for $\alpha = 0.95$ and for the values of other parameters plotted in Figure 7.

Figure 7a also displays the existence of α_L (defined in Section 2 as the lower bound of α to make the performance of the parallel system superior to that of the sequential system). The data obtained but not fully plotted in Figure 7 showed that in all the cases depicted in Figure 7, α_L did not exceed 0.87 and rarely went above 0.6. It can conservatively be said that the practical range of α is far above α_L.

4. Summary

The analysis made in this paper confirmed that parallel execution can reduce the execution time increase inherent in fault-tolerant programs. The analysis demonstrated largely two points. First, under all practical circumstances the parallel system showed good performance. The performance was particularly good when α was above 0.9 or 0.95. It is believed that α would always be in such a range for programs which have undergone a reasonable degree of testing before being put into operation. Second, it showed how the effectiveness of parallel execution was affected by various program characteristics. Although no real statistics on various program characteristics are available, it is believed that our examination covered a broad range of reasonable values for each parameter. Availability of a parallel system may influence the program characteristics to some extent.

In short, the parallel execution approach allows the incorporation of extensive validation and recovery facilities without associated expensive execution time overhead. The price paid is the increased hardware requirement.

Acknowledgements

The authors would like to thank Drs. David L. Russell, C. V. Ramamoorthy, and L. R. Welch for helpful discussions. The authors also wish to acknowledge the help of Messrs. M. Naghibzadeh, M. Olumi, and B. Shah.

References

[1] Anderson, T. and Kerr, R., "Recovery Blocks in Action: A System Supporting High Reliability," Proc. 2nd Int'l Conf. on Software Engineering, 1976, pp. 447-457.

[2] Chandy, K.M., "A Survey of Analytic Models of Rollback and Recovery Strategies," Computer, May 1975, pp. 40-48.

[3] Horning, J.J., Lauer, H.C., Melliar-Smith, P.M. and Randell, B., "A Program Structure for Error Detection and Recovery," Lecture Notes in Computer Science, Vol. 16, Springer-Verlag, 1974, pp. 171-187.

[4] Kim, K.H. and Ramamoorthy, C.V., "Failure-tolerant Parallel Programming and Its Supporting System Architecture," Proc. AFIPS Nat'l Comp. Conf., 1976, pp. 413-423.

[5] Kim, K.H., "A Parallel System Processing Fault-Tolerant Programs - I. System Architecture," submitted for publication. (Also Tech. Memo. PETP-2, Electronics Science Laboratory, University of Southern California.)

[6] Kleinrock, L., Queueing Systems Volume 1: Theory, Wiley-Interscience, 1975.

[7] Computing Surveys, Vol. 8, No. 4, December 1976 (Special Issue on Fault-Tolerant Software edited by R. T. Yeh).

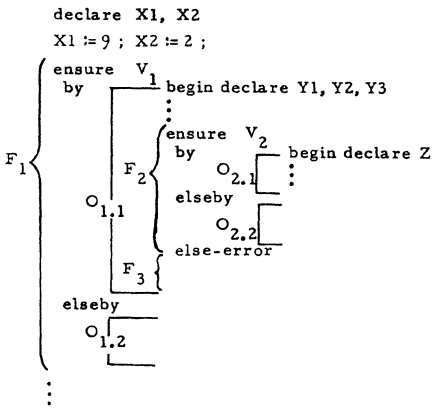


Figure 1a. A block-structured fault-tolerant program.

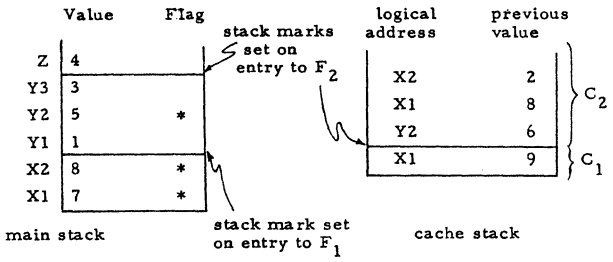


Figure 1b. Recovery cache during execution of 1a.

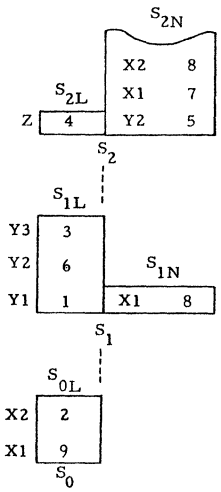
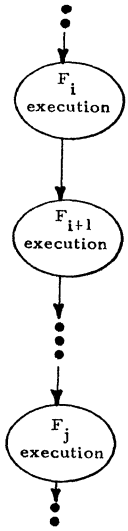


Figure 1c. VR-store during execution of 1a.



F_k: Fault-tolerant block k

Figure 2. Execution of a fault-tolerant program of the type assumed.

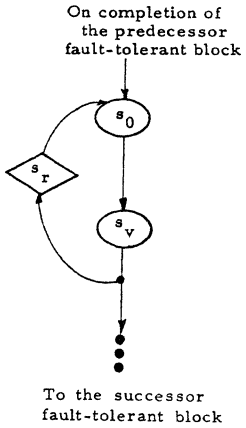


Figure 3a. Behavior of the sequential system during execution.

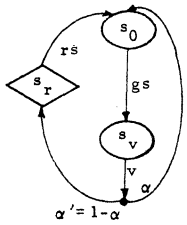


Figure 3b. Model S.

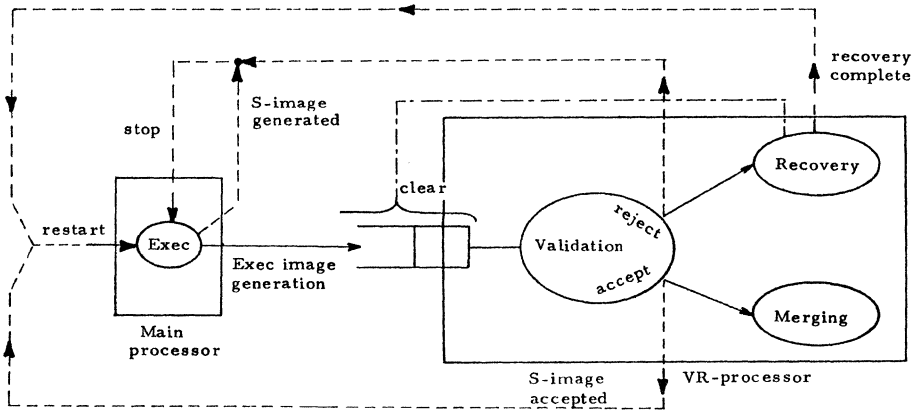


Figure 4. Parallel system modeled in a queueing system.

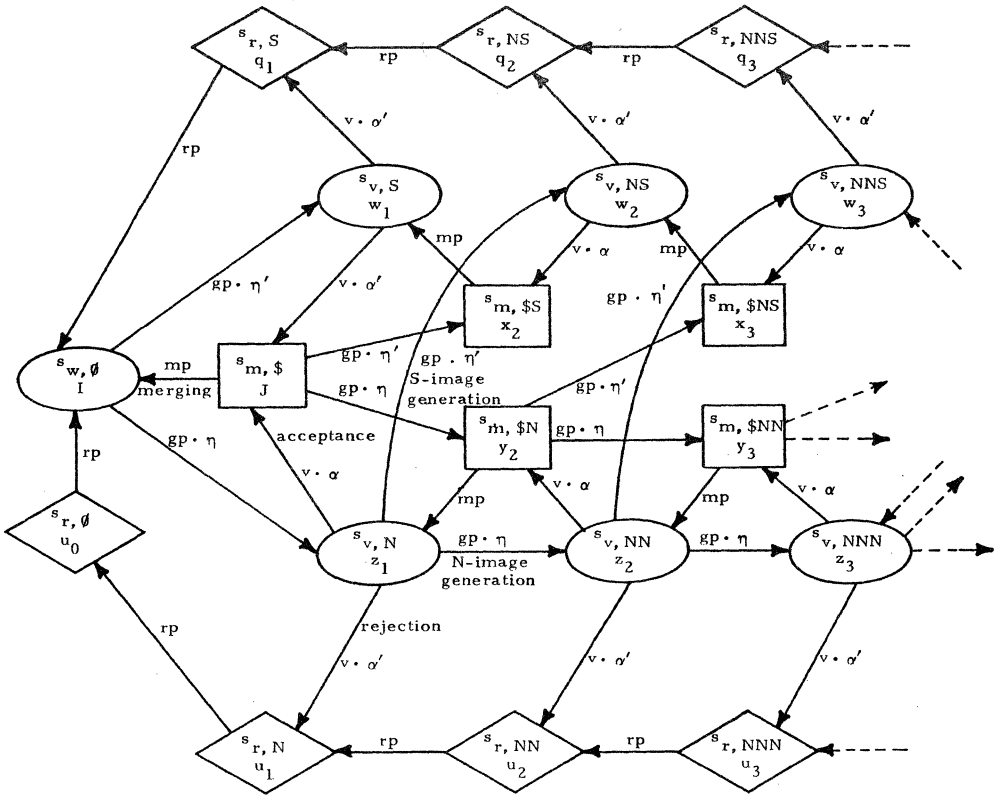


Figure 5. Model P

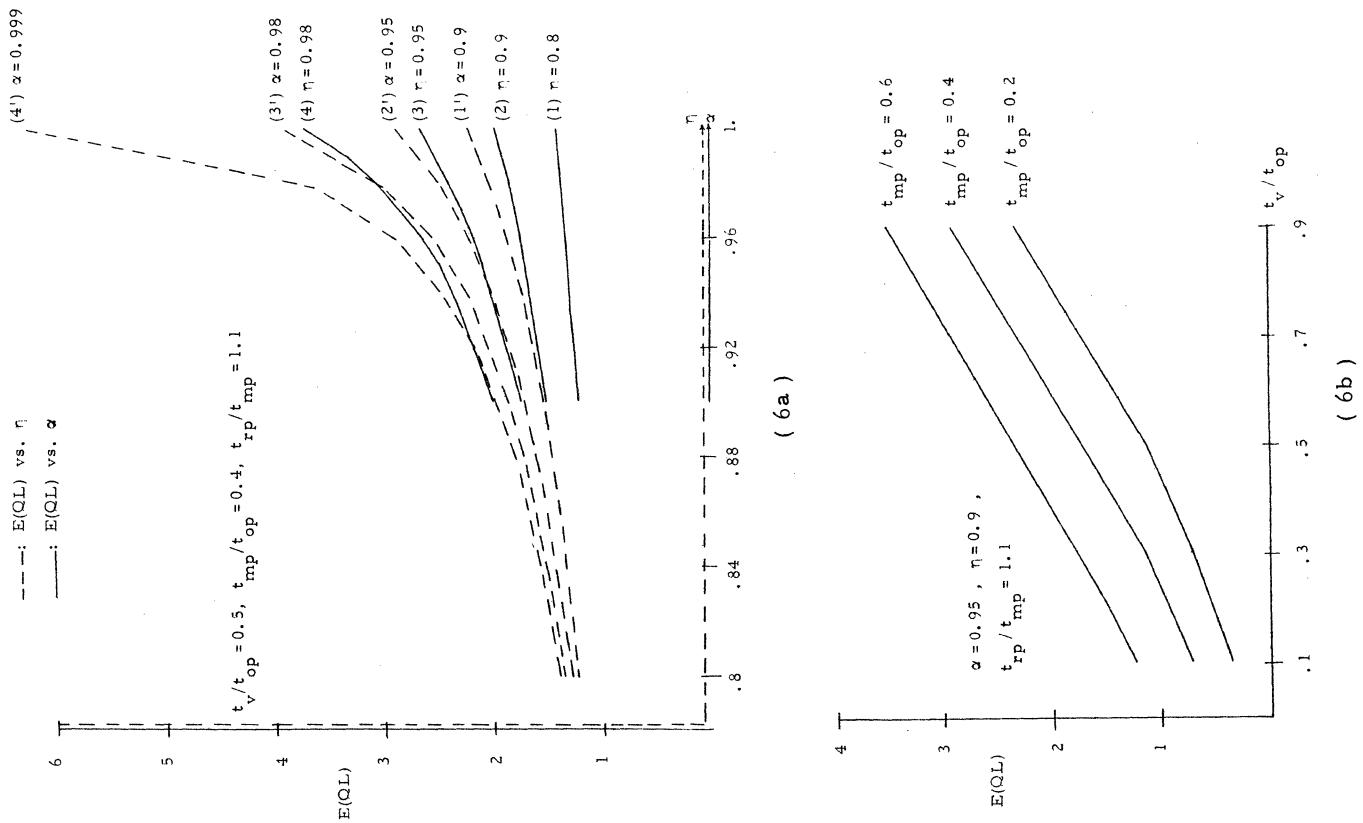
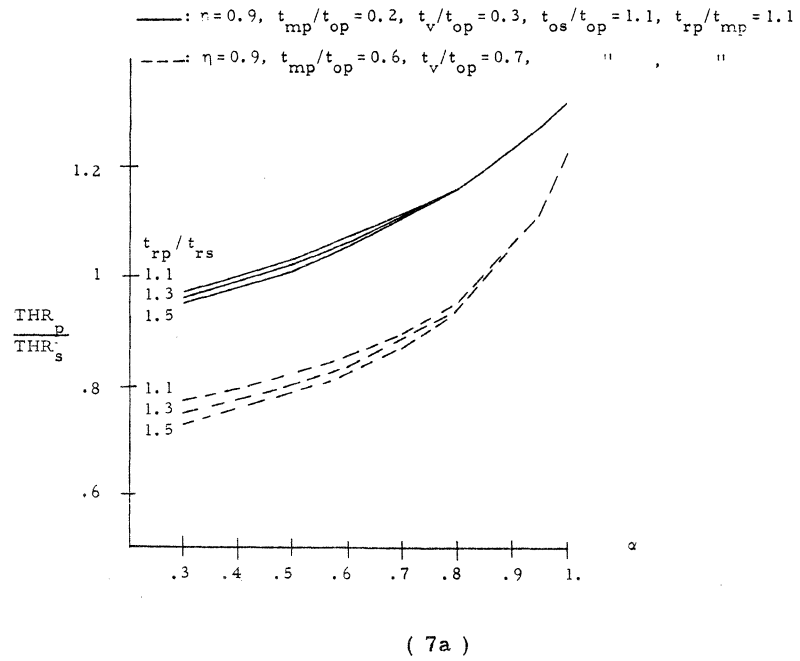
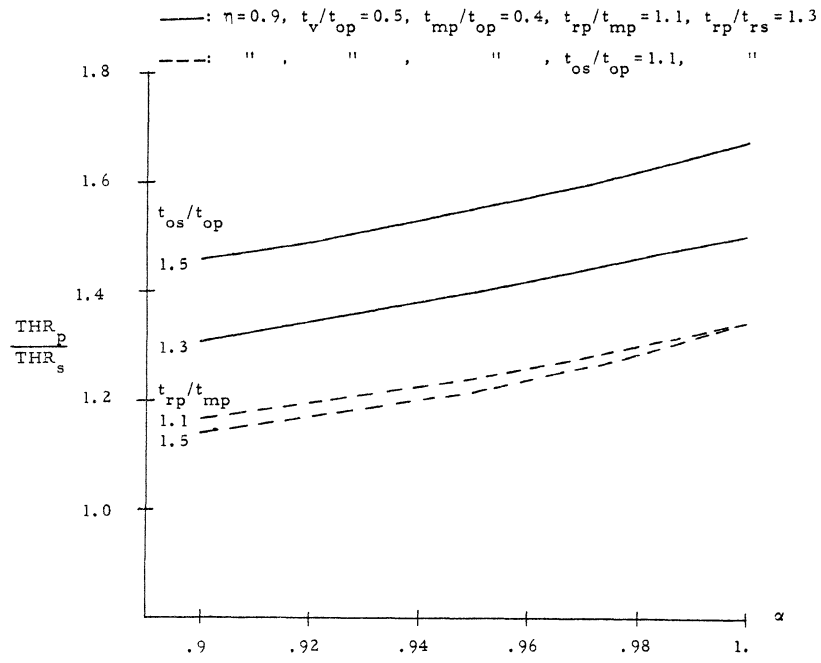


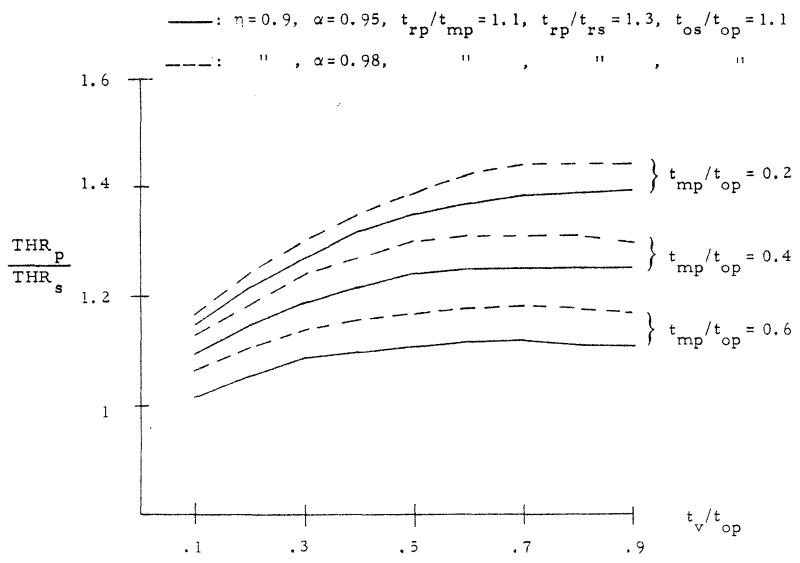
Figure 6. Expected queue-length E(QL).



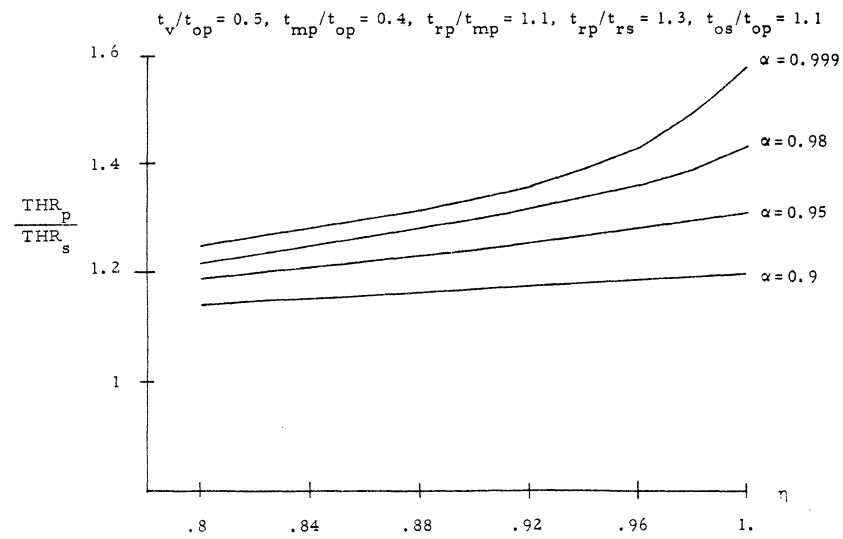
(7a)



(7b)



(7c)



(7d)

Figure 7. Throughput ratio THR_p / THR_s.

Analysis of Asynchronous Multiprocessor Algorithms with Applications to Sorting[a]

John T. Robinson
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract - Efficient algorithms for asynchronous multiprocessor systems must achieve a balance between low process communication and high adaptability to variations in process speed. Algorithms which employ problem decomposition can be classified as static and dynamic. Static and dynamic algorithms are particularly suited for low process communication and high adaptability, respectively. In order to find the "best" method, something about mean execution times must be known. Techniques for the analysis of the mean execution time are developed for each type of algorithm, including applications of order statistics and queueing theory. These techniques are applied in detail to (1) static generalizations of quicksort, (2) static generalizations of merge sort, and (3) a dynamic generalization of quicksort.

1 - Introduction

We consider the design and analysis of k-process algorithms for an asynchronous multiprocessor system, which consists of k or more processors sharing a common memory by means of a switch or connecting network. In addition there is an operating system providing such functions as process creation, scheduling of processes, allocation of memory, synchronization, etc. A real example of such a system is described in [7], and a general discussion of asynchronous parallel algorithms is presented in [5]. A k-process algorithm will be presented by giving the procedure each process executes when assigned a processor. We will assume that a processor is always available for any of the k processes that is runnable.

Given a task we wish to execute on such a system, in order to exploit parallelism we must decompose the task into a set of subtasks. Some subtasks cannot begin until others which they depend upon finish; this establishes a precedence relation between tasks. Inefficiency in an algorithm arises when some process must spend too much time waiting for other processes to complete subtasks, and again towards the end of execution when there are fewer than k subtasks. Attempts to remedy this by "evenly" dividing the original task are hopeless, since task execution time will vary due to variations in the input, the effects of other users, properties of the operating system, processor-memory interference, and many other causes. Any efficient algorithm must adapt to these variations. However, this adaptation is expensive, in that it requires process communication. Thus the trade-off between adaptability and process communication must be considered in the design of multiprocessor algorithms. In the algorithms considered in this paper, process communication takes place by means of global data accessible by all processes. Since in many cases access to this global data must be confined to a critical section, one cause of process communication overhead is the interference between processes seeking access to this global data.

Two methods of decomposition naturally arise: (1) static decomposition, in which the set of subtasks and their precedence relations are known before execution, and (2) dynamic decomposition, in which the set of subtasks changes during execution. Static decomposition algorithms offer the possibility of very low process communication, providing there are not too many tasks; however, their adaptability is limited. Dynamic decomposition algorithms can adapt to variations in task execution time very well, but only at the expense of high process communication.

Given a problem which can be decomposed into subproblems, which method is best? Is the extra expense necessary for fast process communication (thus supporting efficient dynamic algorithms) justified? If a dynamic algorithm is used, how far should decomposition proceed? In order to answer these questions we need techniques for finding mean execution times for these types of algorithms.

(a) This research was supported in part by the National Science Foundation under grant MCS75-222-55 and the Office of Naval Research under contract N00014-76-C-0370, NR044-422.

In section 2 algorithms employing static decomposition are considered. We develop techniques for finding the probability distribution of total execution time in terms of the distributions of individual task execution times, and when these are not known, techniques for finding bounds on the mean execution time. In section 3, the mean execution time for a simple model of a dynamic algorithm is found, assuming exponentially distributed task execution times. In sections 4 and 5 the results of section 2 are applied to static generalizations of quicksort and merge sort. Certain partitioning strategies are shown to be unsuitable for a static decomposition version of quicksort. In addition, a parallel merging algorithm is presented and analyzed. In section 6 a dynamic generalization of quicksort is presented. Using a result of section 3, the mean execution time is found, and an expression for the optimal degree of decomposition is derived. Section 7 contains a summary of the main results.

2 - Static Decomposition Algorithms

Given a set of tasks $T_1,T_2,...T_n$ partially ordered by a precedence relation $<$, we call T_i a predecessor of T_j (T_j a successor of T_i) if $T_i<T_j$. If there is no task U such that $T_i<U<T_j$, T_i is said to be an immediate predecessor of T_j (T_j an immediate successor of T_i). Tasks with no predecessors are called initial, and tasks with no successors are called final. In the execution of the static algorithm, each process does the following:

(1) Select either an initial task or a task all of whose predecessors have been completed, which has not already been selected. Check in the order $T_1,T_2,...T_n$.
(2) If no task can be selected, go to sleep, unless all tasks have already been selected, in which case terminate. When awakened go to (1).
(3) Execute the selected task.
(4) For each immediate successor of the task, record that an immediate predecessor has completed, and wake up a sleeping process if possible.
(5) Repeat from (1).

For the purposes of analysis we assume that steps (1),(2),(4), and (5) take zero time, and that the execution time of task T_i is given by the random variable t_i, with cumulative distribution function (c.d.f.) F_i.

Definition - The task-graph G associated with $T_1,T_2,...T_n$ and $<$ is a directed graph with nodes $T_1,T_2,...T_n$ and arrows from T_i to T_j if T_i is an immediate predecessor of T_j.

Note that there is a one-to-one correspondence between partially ordered sets of tasks and task-graphs.

Definition - G is a chain if the tasks are totally ordered.

The length of a chain is the number of tasks in the chain. If in a chain the initial task is T_i and the final task is T_j we say it is a chain from T_i to T_j. A sub-graph of a task-graph G which is a chain is said to be a chain in G.

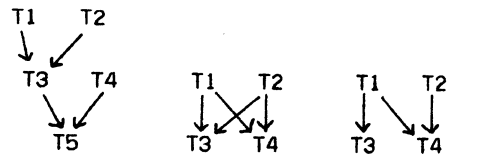
Definition - The level of a task T in a task-graph G is the maximum length of any chain in G from an initial task to T. The depth of G is the maximum level of any task.

Definition - A set of tasks is independent if for any tasks T_i, T_j in the set, neither $T_i<T_j$ nor $T_j<T_i$. The width of a task-graph is the maximum size of any independent subset of tasks.

Given a task-graph G, let t_G be the random variable representing total execution time (the time from when all processes are started until the last process terminates). Assume t_G has c.d.f. F_G. In the following definition a class of task-graphs is defined for which F_G can be expressed simply in terms of the F_i.

Definition - Let $C_1,C_2,...C_m$ be all chains from initial to final tasks in G. For each chain C_i containing tasks $T_{i_1},T_{i_2},...$, let E_i be the expression $(x_{i_1}\cdot x_{i_2}\cdot...)$, where $x_1,x_2,...x_n$ are polynomial variables. Then G is said to be simple if the polynomial $E_1+E_2+...+E_m$ can be factored so that each variable appears exactly once (see figure 2.1).

Figure 2.1



Theorem - If k≥width(G), then t_G can be expressed in terms of the t_i using only + and max. Furthermore, if G is simple and the t_i are independent, then F_G can be expressed in terms of the F_i using only · (multiplication) and * (convolution).

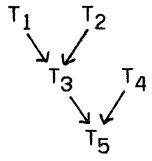
Proof: Note that since k≥width(G) each task begins

immediately after its last predecessor completes. Let $C_1, C_2, \ldots C_m$ be all chains from initial to final tasks. Then

$$t_G = \max_{1 \le i \le m} \left(\sum_{T_j \epsilon C_i} t_j \right) .$$

Next note that + and max are commutative and associative operations, and that + distributes over max (i.e., max(a,b)+c=max(a+c,b+c)). Thus if G is simple the expression for t_G above can be factored in terms of max and + so that each random variable appears only once. Then, if the t_i are independent, the expression for F_G may be found by substituting F_i for t_i, $*$ for +, and $\cdot$ for max in the expression for t_G (see figure 2.2).

Figure 2.2



$t_G = \max(\max(t_1, t_2) + t_3, t_4) + t_5$ $\qquad$ $F_G = (((F_1 F_2) * F_3) F_4) * F_5$.

Thus in the proof of this theorem we have a method for calculating the c.d.f. of total execution time for simple task-graphs with independent task execution times, providing we know the c.d.f. of the execution time of each task. When the c.d.f.s of each task's execution time are not known, the best we can do is derive bounds on mean execution time, such as those of the following theorem. The expected value of a random variable x is denoted by E(x).

Theorem - Given a task-graph G with k≥width(G) and with the t_i independent, let $C_1, C_2, \ldots C_m$ be all chains in G from initial to final tasks. Also let H_i be the set of all tasks of level i, for $1 \le i \le l$ where l=depth(G). Then

$$\max_{1 \le i \le m} \left(\sum_{T_j \epsilon C_i} E(t_j) \right) \le E(t_G) \le \sum_{1 \le i \le l} E\left(\max_{T_j \epsilon H_i} t_j \right) \qquad (2.1)$$

Proof: From above,

$$t_G = \max_{1 \le i \le m} \left(\sum_{T_j \epsilon C_i} t_j \right) .$$

The lower bound then follows from $E(\max\{x_i\}) \ge \max\{E(x_i)\}$ for any random variables x_i. For the upper bound, let $t_0 \equiv 0$ and define f(i,j)=0 if $C_i \cap H_j$ is empty, otherwise f(i,j) is the index of the single task in $C_i \cap H_j$. Then

$$t_G = \max_{1 \le i \le m} \left(\sum_{1 \le j \le l} t_{f(i,j)} \right) \le \sum_{1 \le j \le l} \left(\max_{1 \le i \le m} (t_{f(i,j)}) \right)$$

from which the result follows.

The upper bound in equation 2.1 is useful only if something can be said about $E(\max\{t_j\})$. An applicable result from order statistics (see [2]) is that if the independent random variables $x_1, x_2, \ldots x_m$ are identically distributed with mean u and standard deviation s, then

$$E(\max\{x_i\}) \le u + \frac{m-1}{\sqrt{2m-1}} s \qquad (2.2)$$

Hence the following corollary:

Corollary - If k≥width(G), the t_i are independent, depth(G)=l, and the m_j tasks on level j have identically distributed execution times with mean u_j and standard deviation s_j, then

$$\sum_{1 \le j \le l} u_j \le E(t_G) \le \sum_{1 \le j \le l} \left(u_j + \frac{m_j - 1}{\sqrt{2m_j - 1}} s_j \right) \qquad (2.3)$$

Let w=width(G). When w>k, F_G cannot in general be expressed simply in terms of the F_i, even when G is simple and the t_i are independent. For example, let G consist of T_1, T_2, T_3 with the set $\{T_1, T_2, T_3\}$ independent, and let k=2. Then $t_G = \max(\min(t_1, t_2) + t_3, \max(t_1, t_2))$, and t_G cannot be simplified further.

When w>k, the lower bounds for $E(t_G)$ given above still hold. For an upper bound we take the following approach. It is assumed that w processes are created, and each process has a processor available at least k/w of the time. For example, the bound given in the corollary becomes

$$\sum_{1 \le j \le l} u_j \le E(t_G) \le \frac{w}{k} \sum_{1 \le j \le l} \left(u_j + \frac{m_j - 1}{\sqrt{2m_j - 1}} s_j \right) \qquad (2.4)$$

Finally, when the t_i are dependent, in general special techniques must be used, such as those in the analysis of partitioning strategies (section 4) or parallel merging (section 5) .

3 - A Dynamic Decomposition Algorithm

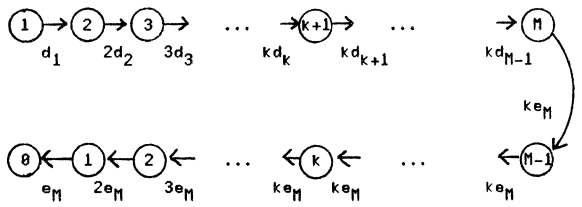
Given a task T and a procedure which decomposes a task into two tasks which may be executed concurrently, we consider the following dynamic algorithm: First, there is a decomposition phase, in which each process repeatedly removes tasks from the task-queue TQ (which initially contains only T), decomposes the task and inserts the two new tasks in TQ, until there is a total of M tasks. Next, there is an execution phase, in which each process repeatedly removes tasks from TQ and executes the task.

We analyze this algorithm under the following assumptions:

(1) In this section the time to access TQ is assumed to be 0.
(2) The time to decompose a task is assumed to be exponentially distributed with mean d_i^{-1}, where i is the current total number of tasks.
(3) The time to execute a task is assumed to be exponentially distributed with mean e_M^{-1}.

We use standard queueing theory techniques in the analysis (see for example [3]). Adopting as a state variable the total number of tasks in TQ or currently being executed or decomposed, the state-transition-rate diagram is given by figure 3.1.

Figure 3.1



The mean execution time is found to be:

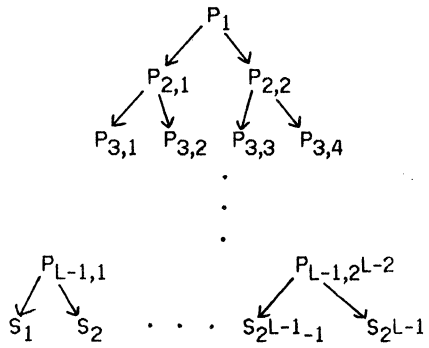
$$T = \frac{1}{e_M}\left(\frac{M-1}{k} + H_k\right) + \sum_{1 \le i \le M-1} \frac{1}{\min(i,k)d_i} \quad (3.1)$$

where $H_k = (1 + 1/2 + 1/3 + \ldots + 1/k)$.

4 - Static Quicksort

We consider a static generalization of quicksort as given by the task-graph of figure 4.1 (see [6] for a complete discussion of sequential quicksort):

Figure 4.1



The tasks may be described as follows:

(1) P_1 is a partition of the file to be sorted.
(2) $P_{i,j}$ (j odd) is a partition of the left subfile produced by $P_{i-1,(j+1)/2}$.
(3) $P_{i,j}$ (j even) is a partition of the right subfile produced by $P_{i-1,j/2}$.
(4) S_j (j odd) is a quicksort of the left subfile produced by $P_{L-1,(j+1)/2}$.
(5) S_j (j even) is a quicksort of the right subfile produced by $P_{L-1,j/2}$.

First consider the simplest case, where k is a power of 2 and L=1+lg(k) (where lg is $\log_2$). In this case the width of the task graph is k. The question arises as to what partitioning strategy to use, that is, how should the partitioning element be selected in the P tasks? First a definition of asysmptotic mean speedup:

Definition - Given an algorithm for k processes, let the mean total execution time be $T_k(N)$, where N is the size of the input. Then the asymptotic mean speedup S_k is defined to be

$$S_k = \lim_{N \to \infty} \frac{T_1(N)}{T_k(N)}$$

We would prefer a partitioning strategy which gives asymptotic mean speedup of k even in the simplest case; strategies which depend on large L for speedup are unsuitable since the number of tasks increases

exponentially with L, and one of the main advantages of static algorithms is low overhead.

It is now necessary to make some assumptions about the execution times of tasks. In the sequential analysis of quicksort it is found that partitioning a file of size N takes O(N) time with standard deviation O(N), and that sorting a file of size N takes O(N lg(N)) time with standard deviation O(N) (see [6]). Thus in analyzing asymptotic mean speedup it is only necessary to consider the sorting task times.

(1) When the partitioning element for a partition of a file of size M is selected at random, it is natural to assume that either subfile size is uniformly distributed between 0 and M. This, together with the fact that the sum of the subfile sizes is M, gives an expected maximum subfile size of 3M/4. Using this, it is easy to show that of the k subfiles to be sorted in the sorting tasks, the expected maximum subfile size is at least $(3/4)^{\lg(k)}N$, which implies $S_k \le k^{\lg(4/3)}$

(2) If the median of three method is used to select the partitioning element, and if it is assumed that the final position of each of the three elements in the subfile is uniformly distributed between 0 and M, then the probability density function for the size of either subfile is:

$$f(x) = \frac{6}{M}\left(1 - \frac{x}{M}\right)\frac{x}{M} \quad .$$

This gives an expected maximum subfile size of 11M/16. As in (1), it can be shown that the expected maximum size of the subfiles to be sorted is larger than $(11/16)^{\lg(k)}N$. It follows $S_k \le k^{\lg(16/11)}$.

(3) If the partitioning elements for all partitioning tasks are found using the method of samplesort (first pick k-1 elements randomly, sort, and use these for the k-1 P tasks), and if the final position of each of the k-1 elements is assumed to be uniformly distributed between 0 and N, then the probability density function for the size of the largest subfile to be sorted is:

$$f(x) = \sum_{1 \le j \le \lfloor N/x \rfloor} (-1)^{j-1} k(k-1)\binom{k-1}{j-1}\left(1 - \frac{jx}{N}\right)^{k-2} .$$

(See the discussion on the random division of an interval in [2]). It follows the expected maximum size of the subfiles to be sorted is:

$$\int_0^N x f(x)\,dx = \frac{N}{k}\sum_{1 \le j \le k} (-1)^{j-1}\binom{k}{j-1}\frac{1}{j} = \frac{H_k}{k} N.$$

Hence $S_k = k/H_k$.

(4) Finally we turn to the partitioning strategy of first finding the median (in O(M) time, where M is the size of the subfile) in each P task, and using the median as the partitioning element. This does give $S_k=k$, but it should be noted that median finding represents a large overhead. Unless process communication is extremely expensive, a dynamic generalization of quicksort (such as the one presented in section 6) is probably better.

If the mean and standard deviation of the time to quicksort a file of size M are $a_qM\lg(M)$ and b_qM, and the mean and standard deviation of the time to find the median of a file of size M and partition the file using the median as partitioning element are a_pM and b_pM, then from equation 2.3 we find that the mean total execution time is less than

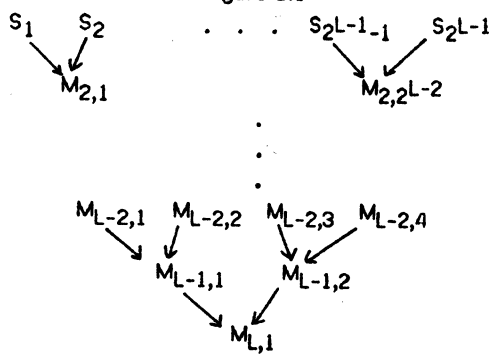
$$a_q\left(\frac{N}{k}\right)\lg\left(\frac{N}{k}\right) + \left[2a_p\left(1 - \frac{1}{k}\right) + \frac{k-1}{k}\frac{b_q}{\sqrt{2k-1}} + \sum_{1 \le j \le \lg(k)-1} \frac{2^{j-1}}{2^j}\frac{b_p}{\sqrt{2^{j+1}-1}}\right] \cdot N \quad .$$

When L is greater than 1+lg(k) a similar result may be found using equation 2.4.

5 - Static Merge Sort

Consider a static generalization of merge sort as given by figure 5.1 (see [4] for a discussion of sequential merge sort):

Figure 5.1



The tasks may be described as follows, assuming the file to be sorted consists of records 1 through N:

(1) S_i is a merge sort of all the records between $(i-1)(N/2^{L-1})$ and $i(N/2^{L-1})+1$.
(2) $M_{2,i}$ is a merge of the two sorted files produced by S_{2i-1} and S_{2i}.
(3) $M_{i,j}$ $(i>2)$ is a merge of the two sorted files produced by $M_{i-1,2j-1}$ and $M_{i-1,2j}$.

When k is a power of 2 and $L=1+\lg(k)$, the width of the task graph is k and equation 2.3 may be applied. Assuming the time to merge sort a file of size N has mean $a_s N \lg(N)$ and standard deviation $b_s N$, and that the time to merge two files of sizes M and N has mean $a_m(M+N)$ and standard deviation b_m (see [4]), we find that the mean total execution time is less than

$$a_s\left(\frac{N}{k}\right)\lg\left(\frac{N}{k}\right)+2a_m\left(1-\frac{1}{k}\right)N+(k-1)\frac{b_s\sqrt{N}}{\sqrt{2k^2-k}}$$

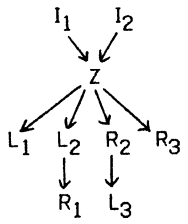
$$+\sum_{1\le j\le \lg(k)-1}(2^j-1)\frac{b_m}{\sqrt{2^{j+1}-1}}$$

When L is larger than $1+\lg(k)$ a similar result holds, using equation 2.4.

In the remainder of this section we consider one possible improvement: replacing the merging tasks with parallel merges. A two task merge of two files is possible by letting each task be an instance of the usual sequential two-way merge (see [4]), except that in one task merging begins with the two smallest items of the two files (a merge from the left), and in the other task merging begins with the two largest items (a merge from the right). In addition the two tasks are interlinked as follows: in sequential two-way merge, the pointers to the files are compared to the ends of the files; in a two task merge, the pointers of one task are compared to the pointers of the other task. Because of this, the two tasks finish together almost exactly, providing one has not already finished before the other starts. We now assume a sequential two-way merge of two files each of size N takes time $2a_mN$. Hence a two process merge using the above method would take time a_mN.

Next consider the following merging algorithm, for k=4:

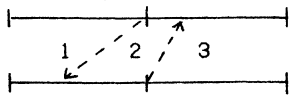
Figure 5.2



Assume the elements to be merged are $x_1<x_2<x_3<\ldots<x_N$ and $y_1<y_2<y_3<\ldots y_N$. The tasks are:

I_1: Insert $x_{\lfloor N/2\rfloor}$ into the y_i's.
I_2: Insert $y_{\lfloor N/2\rfloor}$ into the x_i's.
Z: The results of the insertions determine three pairs of subfiles, as shown below. Z determines the subfile pairs and initializes the L_i and R_i tasks.
L_i: Merge from the left of the i'th subfile pair.
R_i: Merge from the right of the i'th subfile pair.

Figure 5.3



If process 1 executes L_1 and process 2 executes L_2 and then R_1, process 1 finishes before or with process 2. Let the sizes of the subfiles in the second subfile pair be X and Y. The execution time for process 2, starting at the completion of Z, is:

$$a_m\max\left(\frac{X+Y}{2}\ ,\ \frac{X+Y}{2}+\left(N-Y-\frac{X+Y}{2}\right)\Big/2\right)\le a_m\left(\frac{N}{2}+\frac{|X-Y|}{4}\right)$$

since $(X+Y)/2\le N/2$. The same result holds for the process executing R_2 and L_3. In order to find the distribution of $|X-Y|$, it is assumed all elements x_i, y_i are distinct, and that all permutations are equally likely. Then the probability of inserting $x_{\alpha N}$ in position i is:

$$P(y_i<x_{\alpha N}<y_{i+1}) = \frac{\binom{i+\alpha N-1}{\alpha N-1}\binom{N(2-\alpha)-i}{(1-\alpha)N}}{\binom{2N}{N}}$$

$$= \frac{\alpha N \binom{N}{i}\binom{N}{\alpha N}}{(i+\alpha N)\binom{2N}{i+\alpha N}}$$

$$\doteq \frac{2\alpha N}{(i+\alpha N)\sqrt{N\pi}} e^{-(i-\alpha N)^2/N}$$

using the normal approximation to the binomial distribution. This distribution is again approximately normal, with mean αN and standard deviation $\sqrt{N/2}$. Assuming X and Y are actually distributed normally, the mean of |X-Y| can be calculated to be $\sqrt{2N/\pi}$. Hence,

$$E(t_G) \leq a_m\left(\frac{N}{k} + \sqrt{\frac{N}{8\pi}}\right) + O(\lg(N))$$

where the O(lg(N)) term is from the insertion tasks.

Other merging algorithms for k=4 and for higher k can be devised by using various element insertion strategies. Similar techniques may be used in their analysis.

6 - Dynamic Quicksort

We may use the dynamic algorithm of section 3 for sorting, where tasks are considered to be subfiles, the decomposition of a task is a partition of the subfile into two subfiles, and the execution of a task is a sort of the subfile. In analyzing this algorithm we make the following assumptions, where the file to be sorted contains N records:

(1) If M is the total number of subfiles to be produced during the decomposition stage, the total number of task-queue accesses is 3M-2, and each process makes an approximate average of 3M/k such accesses. We therefore assume the overhead due to process communication is linear in M, and is given by w(k)·M.

(2) When there are i subfiles, the mean subfile size is N/i. It is assumed the time needed to partition a subfile is exponentially distributed, and that when there is a total of i subfiles the mean time is aN/i.

(3) During the task execution phase, the average subfile size is N/M. It is assumed the time to sort one of the M subfiles produced by decompositioning is exponentially distributed, with mean b(N/M)ln(N/M).

From equation 3.1, the mean execution time T(M,N,k) is:

$$T(M,N,k) = w(k)M + b\left(\frac{N}{M}\right)\ln\left(\frac{N}{M}\right)\left(\frac{M-k}{k} + H_k\right) + \sum_{1\leq i\leq k-1} a\frac{N}{i^2} + \sum_{k\leq i\leq M-1} a\frac{N}{ki}$$

$$= w(k)M + \frac{N}{k}(b\ln N - aH_{k-1} + aH_{M-1} - b\ln M) + b\left(\frac{N}{M}\right)\ln\left(\frac{N}{M}\right)(H_k - 1) + aNH_{k-1} \quad (2).$$

Given N and k, we seek to find M so as to minimize T(M,N,k). If we approximate H_{M-1} by ln(M), then M must satisfy

$$\frac{\partial T}{\partial M} = w(k) + \frac{N(a-b)}{kM} + bN(H_k-1)\left(\frac{\ln M - \ln N - 1}{M^2}\right) = 0 .$$

Let $A = \frac{w(k)}{bN(H_k - 1)}$ and $B = \frac{(a-b)}{bk(H_k-1)}$,

then the optimal value of M is the solution of

$$M\cdot e^{(AM^2+BM-1)} = N.$$

A short table of the optimal integer value of M for various values of w(k)/b follows, for the case k=4, a=b, $N=10^6$:

w(4)/b	M
10	930
10^2	313
10^3	105
10^4	35
10^5	11

Thus, given a,b,N, and k, the optimal degree of decomposition is determined by w(k), the process communication overhead.

7 - Summary

We have classified asynchronous multiprocessor algorithms which employ problem decomposition as static and dynamic. Static decomposition algorithms require little process communication and would be well-suited for systems where process communication is expensive, e.g., "loosely-coupled" computer networks.

A static decomposition algorithm is described by a task-graph. Simple task-graphs have the property that there is a simple expression for the probability distribution of total execution time in terms of the probability distributions of each task, providing the result of one task does not affect the execution time of another. If the probability distributions of each task's execution time are unknown, it is still possible to bound mean total execution times providing the means and variances of task execution times are known.

Regarding the upper bound given by equation 2.3, the bound is tight in that task-graphs and task execution time probability distributions may be constructed so that equality holds, using distributions derived in [2]. Any improved bound would require either more detailed information about the partial ordering of the tasks in the expression of the bound, or additional assumptions about the probability distributions of task execution times.

When process communication is inexpensive, dynamic decomposition algorithms are suitable. One technique for analyzing these algorithms is by means of a queueing model. Queueing models may be used in analyzing other types of asynchronous parallel algorithms as well (e.g., in [1] a queueing model is used to analyze asynchronous iterative methods).

For some static decomposition algorithms the bounds derived in section 2 may be directly applied, such as static quicksort with median finding and static merge sort. In other cases where task execution times are dependent other techniques must be used. This is the case for static quicksort when median finding is not used and in the parallel merging algorithm presented. These algorithms have dependent task execution times since there are tasks where the input size depends on the result of a previous task.

The assumption that process communication overhead is negligible in static decomposition algorithms is valid only if the total number of tasks is not very large. For this reason we have given bounds on mean execution time only for those algorithms in which the width of the task-graph is k (although a technique for greater width task-graphs has also been presented). These bounds give an indication of the performance that can be expected when process communication overhead is high enough to warrant the use of static decomposition. However, in dynamic decomposition algorithms we may choose the degree of decomposition, which should ideally be chosen so as to balance process communcication overhead and adaptability to variations in the execution times of tasks. For example, by applying a queueing model to a dynamic generalization of quicksort, we have derived an expression relating process communication overhead and the optimal degree of decomposition.

References

[1] Baudet, Gerard (1976) "Numerical Computation on Asynchronous Multiprocessors", Thesis Proposal, Department of Computer Science, Carnegie-Mellon University

[2] David, Herbert A. (1970) Order Statistics, Wiley

[3] Kleinrock, Leonard (1975) Queueing Systems, vol. 1, Wiley-Interscience

[4] Knuth, Donald (1972) The Art of Computer Programming, vol. 3, Addison-Wesley

[5] Kung, H. T. (1976) "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", Algorithms and Complexity - New Directions and Recent Results, ed. J. F. Traub, pp. 153-200, Academic Press

[6] Sedgewick, Robert (1975) Quicksort, Ph.D. Thesis, Computer Science Department, Stanford University

[7] Wulf, W. A., and C.G. Bell (1972) "C.mmp - A Multi-Mini-Processor", Proceedings of the AFIPS 1972 Fall Joint Computer Conference, vol. 41, pp. 765-777

ON THE PERFORMANCE AND COST-EFFECTIVENESS OF SOME MULTIPROCESSOR SYSTEMS

Terry T. Hsu
Digital Image Systems Division
Control Data Corporation
Minneapolis, Minnesota 55440

Summary

The performance (maximum throughput) and cost are analyzed for multiprocessor systems using the global bus, shared memory, full interconnection, and ring configurations for inter-processor communications [1] - [3]. The approach is to identify the major characteristic parameters and express the throughput and cost for all configurations in similar forms for easy comparisons.

Let the throughput of a single processor element (PE) be defined as Tp = 1/tp, where tp is the average processing time of an instruction. In the absence of inter-PE communication, any multiprocessor system of N PE's has a maximum system throughput (T) of NTp. This bound is processor-limited. However, assuming each PE has to do a fraction (q) of its processing in communicating with some other PE's through the bus, the shared memory, or paths between PE's, as the case may be, then the system throughput may be bounded by the transfer rate or bandwidth of the communication path. This bound is said to be bandwidth-limited. By considering the utilization of the path by each PE, the bandwidth-limited upper bounds on system throughputs can be shown as follows:

System	Bound On Throughput	Utilization Factor
Global Bus	Tp/b	$b = qt_b/tp$
Shared Memory	Tp/m	$m = qt_m/(tpM)$
Full Connection	NTp	
Ring	NTp/r	$r = qdt_r/tp$

Where t_b, tp, and t_r are the cycle times to communicate an instruction or data between PE's (adjacent in the case of a ring system) in above systems, respectively. M is the size of the common memory, and d is the distance (number of links) between the communicating PE's.

The bandwidth-limited system throughput becomes minimum when a PE has to wait for all other (N-1) PE's to complete their use of the path first. This leads to the expression for a lower bound for all cases.

$$T(min) = \frac{NTp}{1 + (N-1)\,u}$$

Where u is the utilization factor for each case (b,m, and r above) and is $q(N-2)/(N-1)^2$ for the fully interconnected system.

Of course the achievable maximum system throughput is determined by the smallest of all above bounds - processor or bandwidth limited.

The system cost is assumed to be composed of five major component costs: PE (including local memory), cable, I/O port, switching unit, and extra memory. Let the unit costs be normalized to the cost of one PE (Cpe) and denoted as Kc, Kp, Ks, and Km, respectively. The total system cost can then be expressed in terms of Cpe, the normalized cost coefficients and some coefficient multipliers, which are as follows:

System	Kc	Kp	Ks	Km
Global Bus	N	N	N	-
Shared Memory	N	2N	N^2	1
Full Connection	N(N-1)	N(N-1)	N(N-1)	-
Ring	N	N	2N	-

For example, the cost of a shared memory system with N PE's is expressed as Cpe $(N + NKc + 2NKp + N^2Ks + Km)$.

With above analytic expressions for the four systems, a cost-performance ratio can be obtained readily. Figure 1 compares the four systems on their best performance, assuming b = m = r = .1, and Kc = 0, Kp = .06, Ks = .02, Km = 1. The purpose of the diagram is more illustrative than conclusive. Since many parameters are involved, one should be cautious in interpreting the graphs.

References

[1] G.A. Anderson and E.D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples", Computing Surveys, Vol. 7, No. 4, December 1975, pp. 197-213.

[2] J.L. Baer, "Multiprocessing Systems", IEEE Trans. Comput., Vol. c-25, No. 12, December 1976, pp. 1271-1277.

[3] R.C. Pearce and J.C. Majithia, "Upper Bounds on the Performance of Some Processor-Memory Interconnections", Proceedings 1976 International Conference on Parallel Processing, pp.303.

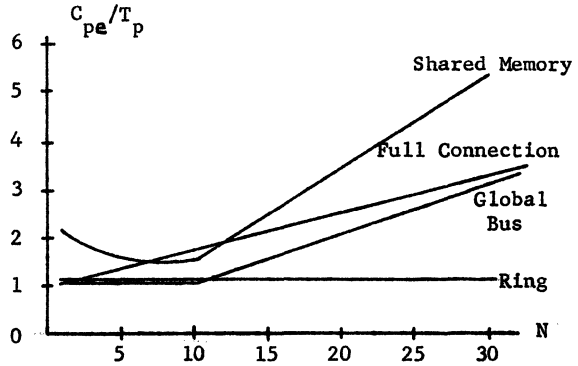


Fig. 1. System Cost/Throughput Comparisons.

A PERFORMANCE STUDY OF DISTRIBUTED CONTROL LOOP NETWORK*

Ming T. Liu, Roberto Pardo, and Gojko Babic
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Summary

This paper presents some analytical results on the performance of three types of distributed-control loop networks, viz., the Newhall loop (single message of variable length), the Pierce loop (multiple messages of fixed length), and the DLCN loop (multiple messages of variable length). The primary goal of the paper is to show, through queueing analysis, that the DLCN loop has superior performance over the other two loops, viz., it has better channel utilization and shorter message delay. A secondary goal is to verify our previous simulation results which made such a claim.

Introduction

The loop network is becoming increasingly popular today for the design of distributed processing systems. A loop network is composed of a high-speed digital communication channel, arranged as a closed loop to which computers, terminals and other devices are attached through loop interfaces. Messages from a sender are multiplexed on to the loop by its interface, then travel around the loop from interface to interface until received by the interface of the addressed receiver. Thus the design of the loop interface and the transmission mechanism it incorporates are very important in the operation of a loop network.

Three transmission mechanisms have been proposed for use in distributed-control loop networks. In the Newhall loop, a round-robin control passing token circulates around the loop and allows only one interface at a time to transmit onto the loop a single message of variable length. In the Pierce loop, communication space on the loop is divided into fixed-size time slots, and it is possible for more than one interface to transmit onto the loop multiple messages of fixed length at a time. In the DLCN loop, the use of a delay buffer in the interface allows simultaneous transmission of multiple messages of variable length.

The superior performance of DLCN transmission mechanism has been verified by an extensive simulation study [1]. In this paper we present some analytical results on the performance of DLCN as compared with Newhall and Pierce loops. The main problem in making the comparison is that, in the analysis of each loop, different characteristics of data sources are assumed.

Analytical Results

In this section formulas for channel utilization and average message delay for the three loops are given. Only symmetric loops (symmetric traffic pattern and identical nodal characteristics) are considered. A glossary of terms to be used in the following is listed in Table 1.

C	Capacity of communication channel (bits/sec)
N	Number of nodes in the loop
λ	Arrival rate of messages from the data source (messages/sec)
$1/\mu$	Average message length (bits)
$\overline{a^2}$	Second moment of message length ($bits^2$)
$1/\mu'$	Average duration of active period of data source (seconds)
$1/\lambda'$	Average duration of idle period of data source (seconds)
b	Bit rate during active period (bits/sec)
u	Utilization of data source
B_p	Number of information bits per packet (bits)
T	Average message delay (including all waiting times and time for multiplexing message onto loop)(second)
W	Average message waiting time ($= T - 1/\mu C$)
U	Channel Utilization
B	Number of bits in the address field (bits)

Table 1. Glossary of Terms

DLCN Loop Analytical Results

Formulas presented below for the DLCN loop are derived in a paper by Liu, Babic and Pardo [2]. Data sources are assumed to be characterized by instantaneous generation of message according to the Poisson process and distribution of message lengths is general.

1. Channel Utilization:

$$U = \lambda N/2\mu C \quad (1)$$

2. Average Message Delay:

$$T = T_1 + 1/\mu C + NB/2C + (N/2 - 1)T_4 \quad (2)$$

where

$$T_1 = N\lambda\overline{a^2}\mu/4C(C\mu - \lambda) \quad (3)$$

$$T_4 = N\lambda\overline{a^2}\mu^2/2(C\mu - \lambda)(2C\mu - N\lambda) \quad (4)$$

Pierce Loop Analytical Results

Formulas presented below for the Pierce loop are derived in a paper by Hayes and Sherman [3]. In their model data sources are characterized by having active and idle periods and both are assumed to be exponentially distributed. During an active period a data source produces bits in constant rate and generates one message.

1. Channel Utilization:

* Research reported herein was supported in part by AFOSR Grant No. 77-3400 and by OSU Resaerch Grant No. 221110. A full-length version of the paper is available from the authors.

$$U = rNB_p/2C \quad (5)$$

where

$$r = \mu' Qu \quad (6)$$

$$Q = (1 - EXP(-B_p\mu'/b))^{-1} \quad (7)$$

$$u = \lambda'/(\lambda' + \mu') \quad (8)$$

2. Average Message Delay:

$$T = \gamma/\mu' + \gamma\theta^*/\mu + \gamma^2\theta(1 + \theta^*)^2/\mu'(1 - \gamma\theta(1 + \theta^*)) + U^*/M^*(1 - \gamma\theta(1 + \theta^*)) \quad (9)$$

where

$$\gamma = \mu' QB_p/C \quad (10)$$

$$\theta = \lambda'/\mu' \quad (11)$$

$$\theta^* = R^*B_p/(C - R^*B_p) \quad (12)$$

$$R^* = r(N/2 - 1) \quad (13)$$

$$U^* = R^*B_p/C \quad (14)$$

$$M^* = \mu'/\gamma(1 + \theta^*) \quad (15)$$

Newhall Loop Analytical Results

Formulas presented below for the Newhall loop are derived in a paper by Kaye [4]. Data sources have the following characteristics: After a data source generates one message it will be inactive until that message is multiplexed onto the loop. Afterward the data source behaves as a Poisson process with parameter λ, but only until the next message is produced. Then the data source is again inactive, and so on. This process generates messages with effective average interarrival time $\lambda_{eff} = \lambda(1 - P_L)$ (P_L is given below).

1. Channel Utilization:

$$U = (1 - P_L)\lambda/2\mu C \quad (16)$$

where P_L is the portion of lost messages given by

$$P_L = \lambda T/(1 + \lambda T) \quad (17)$$

2. Average Message Delay:

$$T = 1/\mu C - 1/\lambda + (1/\bar{n}) \sum_{n=0}^{N-1} \tau_n e^{\lambda\tau_n} n(N - n)p_n \quad (18)$$

where

$$\bar{n} = \sum_{n=0}^{N} np_n \quad (19)$$

$$\tau_n = N/C + n/\mu C \quad (20)$$

$$p_n = K\binom{N}{n} \prod_{j=0}^{n-1} (e^{\lambda N/C} e^{\lambda j/\mu C} - 1) \quad (21)$$

for $n = 1, 2, \ldots, N$, and $p_0 = K$ for $n = 0$, where K is determined by $\sum_{n=0}^{N} p_n = 1$.

Performance Comparison

Because of different characteristics of data source assumed in the analysis of each loop, we have decided to compare the DLCN loop against the Pierce loop and the Newhall loop separately. Figures 1 and 2 show average message delay of the DLCN loop against the Pierce and the Newhall loops, respectively, and clearly verify that the DLCN loop has shorter message delay. Better channel utilization can also be easily verified by comparing Equations (1), (5) and (16).

Reference

1. C. C. Reames and M. T. Liu, "Design and Simulation of the Distributed Loop Computer Network (DLCN)," Proc. 3rd Annual Symp. on Computer Arch., (Jan. 1976), pp. 124-129.
2. M. T. Liu, G. Babic, and R. Pardo, "Traffic Analysis of the Distributed Loop Computer Network (DLCN)," to appear in Proc. 1977 National Telecomm. Conf., (Dec. 1977), Los Angeles, CA.
3. J. P. Hayes and D. N. Sherman, "Traffic Analysis of a Ring-Switched Data Transmission System," BSTJ, Vol. 50 (Nov. 1971), pp. 2947-78.
4. A. R. Kaye, "Analysis of a Distributed Control Loop for Data Transmission," Proc. Symp. Compu. Comm. Networks and Teletraffic, (April 1972), pp. 47-58.

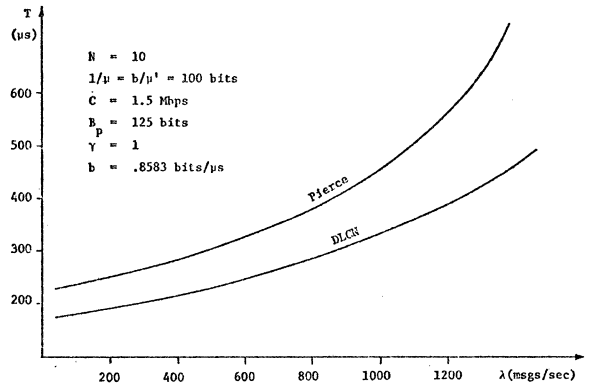


Fig. 1. Average Message Delay (DLCN vs. Pierce)

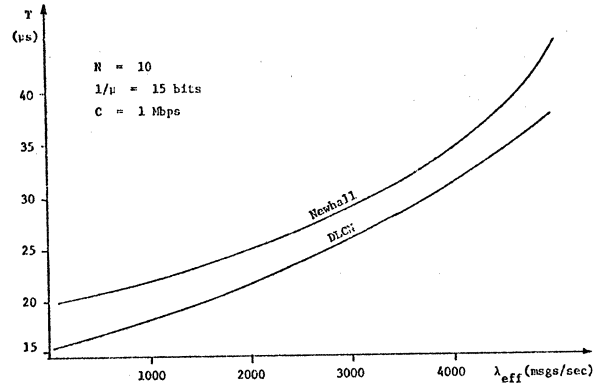


Fig. 2. Average Message Delay (DLCN vs. Newhall)

PERFORMANCE ASPECTS OF MULTIPROCESSING IN A TIME SHARING ENVIRONMENT

Major Thomas C. Darr
Rome Air Development Center
Griffiss AFB
Rome, NY, 13441

Summary

One of the goals of the computer architecture program at RADC is to investigate the cost/performance benefits and trade-offs in the application of current and future technology to Air Force and DOD information processing systems. It has been recognized [1] that software complexity is responsible for a major percentage of the life cycle cost of most systems. It is also recognized that the availability of very low cost microprocessors and other LSI devices presents the potential for the design of highly reliable, low hardware cost systems for a number of applications. Opportunities also present themselves for the reduction of software complexity by judicious use of such low cost hardware and firmware.

This paper investigates the performance of a modern demand paged, virtual memory, multiprogrammed time sharing system (the MULTICS system at RADC). The average system response time for interactive users (editing, debugging, small compilations) is the basic performance yardstick for such systems. It is shown that the system response time is governed and limited by the memory management mechanism. A queueing network model of the system is used to aid the analysis [2].

A model of a multiprocessing system is analyzed [3] with particular attention to comparison of system response time, cost, complexity, and reliability with that of the single processor system. It is shown that performance is still limited by the memory management mechanisms.

It is conjectured that, within the capabilities of forecasted technology in LSI, microprocessors, and computer communications, cost/effective organizations of such systems will eliminate the interactive processing function from the centralized resources. These functions will be placed in highly intelligent terminals, leaving the secondary storage media with user and system data and programs, and a sophisticated data base processor, at the central location. It is shown how this organization simplifies required operating system software, allows for nearly unlimited growth in the user population, and increases system reliability. Other benefits, such as ease in processing secure data, are discussed.

References

[1] D.A. Fisher, Automatic Data Processing Costs in the Defense Department, Institute for Defense Analysis, Paper P-1046, (Oct, 1974).

[2] J.P. Buzen, Queueing Network Models of Multiprogramming, Ph.D Dissertation, Div. Eng. Appl. Phys., Harvard Univ., Cambridge, MA, (1971).

[3] Y. Chow and W.H. Kohler, "Performance of Several Queueing Models for Multiprocessor Multiprogramming Systems," Proc. COMPCON76, (Aug, 1976), pp. 66-71.

STARAN SERIES E

Kenneth E. Batcher
Digital Technology
Goodyear Aerospace Corporation
Akron, Ohio 44315

Abstract. STARAN[(a)] Series E is an enhanced version of the STARAN parallel array processor. Multi-dimensional-access data storage and high speed control storage have been increased several-fold. Faster IC's and new processing algorithms improve the processing rates significantly. A new I/O unit allows faster and more flexible input and output of data. The cost impact of these changes has been minimized by using standard parts which were not available when the first STARAN was built. In this paper we discuss the enhancements and the reasons they were incorporated.

Introduction

As explained in [1] all logic and memory devices in STARAN are standard high-volume integrated circuits. The semi-conductor industry is continually improving these circuits, e.g., more bits on a chip and higher speeds. Thus, it is relatively easy to enhance STARAN by using the newer circuits as they become available. The goal in the design of the Series E systems was to enhance the capabilities of STARAN using newer circuits and better processing algorithms while preserving software compatibility with the original design.

The major enhancement in the Series E systems is a much larger multi-dimensional-access (MDA) memory in each array module [2]. Other enhancements are a larger high-speed control store, faster processing rates and a new I/O unit.

Larger MDA Memory

Need for Larger Memory

The MDA memory in the original design used 256-bit random-access-memory (RAM) devices. These were the largest bipolar RAM devices that were generally available at the time (1971-1972). The size of these devices governed the choice of 256-bit words with a simple PE per word.

After several applications were programmed on the system it became evident that larger words would be better. In most applications the size of the machine (number of array modules) is dictated by memory requirements instead of processing speed requirements. In some applications some data is off-loaded into the control store to make room in the MDA memories. In some other applications two or more words are used per item and some of the PE's are wasted.

Another indication that a larger MDA memory is desirable appears when the ratio of storage bits to processing speed (MIPS) is examined for several large computer systems. STARAN has a low ratio compared to other systems (either we have too many MIPS or too few bits). Increasing the storage by a factor of 20 would bring us in line with other systems.

MDA Memory in Series E

In the original STARAN design the number of bits per word was fixed at 256. Each array module contained 8K bytes in a 256 x 256 bit array and 256 PE's. To increase the storage capacity of a machine one added array modules and added more PE's as well. In the Series E machines one can size the processing power (number of PE's) and memory capacity independently. This is accomplished by multiplying the MDA memory address space by a large factor (256) and allowing the space to be partially populated.

Bipolar 1,024-bit RAM devices are generally available today. These devices have ECL - compatible interfaces so that they match the logic levels of STARAN better than the TTL - compatible devices of the original design and have faster access and cycle times. Thus, these devices are a natural choice for Series E.

MOS 4,096-bit RAM's are also generally available. To keep the slow MOS speed from severely affecting the processing rate of the machine, part of the storage should be bipolar and algorithms modified to move most of the memory accesses into the high-speed bipolar storage. The section on faster processing rates discusses these modifications.

A mixture of high-speed bipolar and low-speed MOS devices is allowed in the MDA storage of a Series E array module (Figure 1). The MOS MDA memory is an array of 256 mK bits where K= 1024 and m is a multiple of 4. It uses 4,096-bit MOS RAM's. The band width of the MOS memory is 256 bits in or out every 420 nanoseconds (76 megabytes/second). The bipolar MDA memory is an array of 256 by nK bits where n is an integer. It uses 1,024-bit bipolar RAM's. A read cycle in the bipolar memory requires 120 nanoseconds (267 megabytes/second) and a write cycle takes 160 nanoseconds (200 megabytes/second). For comparison, the MDA memory of the original STARAN had a read cycle of 120 nanoseconds and a write cycle of 300 nanoseconds.

The maximum MDA storage in each array module, limited by the address space, is 256 x 64K bits (2,097,152 bytes). The physical size of the array module grows with storage capacity and depends on

(a) T.M. Goodyear Aerospace Corporation, Akron, Ohio 44315

the mix of bipolar and MOS storage. In the first Series E machine, m=8 and n=1, for a capacity of 262,144 MOS bytes and 32,768 bipolar bytes per array module. Two array modules are packaged in one STARAN cabinet (array modules with greater storage capacity are packaged one per cabinet). For a cost increase of less than 50% these array modules have 36 times the storage of the original STARAN modules.

Accessing MDA Memory

In the original design each MDA memory access (fetch or store) required two parameters: an 8-bit access mode to select one of 256 stencil shapes and an 8-bit address to position the selected stencil at one of 256 positions [2].

The larger MDA address space in Series E requires a 16-bit address. Some thought was given to also increasing the access mode parameter and allowing some stencils to access every ith bit slice of a word (i=2,4,8,etc.). This idea was rejected because such stencils do not appear to be generally useful and they are hard to implement in a memory which is partially populated with both fast and slow memory devices. Because of the way data is scrambled in memory it is important that all 256 memory bits accessed at one time actually exist and either be all "fast" bits or all "slow" bits. The basic memory increment is 1,024 bit-slices so the maximum allowable access mode parameter is 10 bits--the increase from 8 bits to 10 bits does not add any significant MDA capability so the access mode parameter was left at 8 bits.

In Series E, one may view the MDA memory of an array module as a number of 256 x 256-bit planes (Figure 2). The leftmost 8 bits of the 16-bit address select one of the planes. The 8-bit access mode selects a stencil shape and the rightmost 8 bits of the address positions the stencil within the selected plane. All 256 bits covered by the stencil are fetched or stored in one memory cycle. The shapes of the 256 possible stencils are discussed in (2).

Bipolar MDA memory occupies the first 4n planes (n=1,2,3,...) and MOS MDA memory the next 4m planes (m=0,4,8,12,...).

Base Registers

In the original design the 8-bit MDA memory address came from one of five sources: an address field in the instruction, the resolver through the link pointer or one of three field pointers. The instruction address field is usually used to reference flag bits in fixed locations. The resolver and link pointer are used to reference particular words in the MDA memory, e.g., a word satisfying an associative search operation. The three field pointers are used to step through the bit-slices of fields in arithmetic and search operation; e.g., to add field A to field B with the result put in field C the three field pointers reference corresponding bit-slices of fields A, B and C.

In the original design the 8-bit MDA access mode came from one of two access mode registers (AMR0 and AMR1) depending on the state of a mode bit in the instructions referencing MDA memory.

In Series E, five base registers are included in the control unit; one for each of the five sources of MDA memory addresses. The final 16-bit MDA memory address is formed by adding the 8-bit source to a 16-bit base address in the associated base register. The 8-bit access mode comes from a field in the base register. This arrangement allows addressing of: 1. flags in a flag-bit region, 2. words satisfying a search, and 3. fields in scattered memory regions without modifying the base registers.

The speed of arithmetic instructions with long sequences of micro-steps (multiply, divide, square root and floating-point) would be severely affected if all micro-steps addressed fields in the MOS MDA memory. To speed up these instructions, a sixth base register is used as a pointer to the base of a vector stack in the fast bipolar MDA memory. The long arithmetic instructions move vector operands from MOS memory to the bipolar stack, operate on the stacked vectors and then return the results to the MOS memory. Guard bits are added to the vectors when moved onto the stack and rounded-off when results are unstacked.

The mode bit of the instruction (used in the original design to select AMR0 or AMR1) is used in Series E to select vector stack addressing or the five-base-register addressing. In vector stack addressing, the 16-bit stack base address in the sixth base register is added to the 8-bit source regardless of its source so system micro-programs operating on stacked vectors can use all address sources.

A set of sixteen 32-bit registers is added to the control unit in Series E. Six of the registers are the MDA base registers just discussed. Another eight registers are the return-jump registers (R0 - R7) of the original design. The other two registers can be used as general-purpose registers. The Series E instruction set is augmented with instructions to manipulate these registers.

Control Memory

The control memory of the original design had an address space of 65,536 32-bit words populated with three high-speed 512-word page memories, one high-speed 512-word data buffer and a 16,384-word magnetic core memory. The remainder of the address space could be used to address the memory of a host computer if such an interface exists or to double the capacity of the pages, the high-speed data buffer and/or the core memory.

The page memories hold the micro-program instructions of the system subroutines and user-generated micro code. For some applications page memory space was tight and measures such as executing some micro-code in core memory or swapping

micro-code in the pages were necessary. With the larger memory devices available now it is easy to expand the page memory capacity without increasing their physical size. In Series E, each of the three page memories holds 4,096 words and can be doubled to 8,192 words if necessary. The memory devices are faster so 100-nanosecond instruction fetch rates can be supported (compared to 120 nanoseconds in the original design).

Magnetic-core memory space was also tight in some applications of the original design. In these applications a significant amount of core memory space was used to unload the small MDA memories and/or buffer data on the I/O channels. In Series E the MDA memories are much larger and the I/O channels communicate with the MDA memories directly so the core memory is relieved of this burden. The core memory capacity in Series E is the same as the original design (32,768 words).

Faster Processing Rates

There are about two MDA memory read steps, and one array register transfer for every MDA memory write step in the typical application program. The following table uses this ratio and the read and write cycle times of the original MDA memory and the MDA memories of Series E to show the effect of MDA memory times on the processing rate.

	Original MDA Memory	Series E Bipolar Memory	Series E MOS Memory
Array Register Move Time(nsec)	120	100	100
Read Time (nsec)	120	120	420
Write Time (nsec)	300	160	420
1 Reg.Move + 2 read + 1 write (nsec)	660	500	1360
Relative Processing Rate	1	1.32	0.49

With no changes in the system micro routines, the processing rate of Series E would be close to that of the original design.

The small MDA memory in the original design limited the arithmetic micro-routines to little or no temporary space for their calculations. This had a severe impact on the execution times of the multiply, divide, square root and floating-point operations. With the much larger MDA memory of Series E some of the memory space can now be given to these operations for temporary storage. The micro-routines for these operations were rewritten to use the vector stack in bipolar MDA memory for temporary storage. Some examples of the speed improvement are shown in the following table.

Operation	Series E speed/ original speed
32-bit floating-point add	3.0
32-bit floating-point multiply	4.0
32-bit floating-point divide	2.0
16-bit fixed-point multiply	1.8
16-bit fixed-point divide	1.5

Since the floating-point micro-routines had to be recoded, it was decided to allow other precisions besides single-and double-length. The precision of these operations can then be tailored to match the precision of an attached host computer or to match problem requirements. A maximum precision of 100 bits was selected -- this is large enough to cover most applications and small enough to be handled conveniently in the MDA memory vector stack. No special problems arise if the precision is two or more bits so a minimum precision of 2 bits was selected. Users can adjust the precisions of floating-point operands anywhere in this large range. Operands with different precisions can be combined and results stored with another precision in the four basic floating-point operations: add, subtract, multiply and divide. The execution time and vector stack space used depends on the operand precisions.

To accomodate host computers with different exponent lengths, floating-point operands can have a base-2 exponent with 7 to 11-bits. In the basic operations all operand exponent lengths must agree.

The format of floating-poing numbers in Series E was selected to maximize performance. The format is one sign bit followed by 7 to 11 base-2 exponent bits followed by 2 to 100 mantissa bits. Exponents are biased by 64, 128, 256, 512, or 1024 depending on exponent length. Non-zero numbers have normalized mantissas (the most-significant mantissa bit is always 1). All bits of a floating-point-zero are 0.

The format of fixed-point numbers is the same as in the original design--a two's-complement representation with three or more bits, and any scale factor.

Input-Output

The array modules of the original design had two ways of inputting and outputting data: through the 32-bit common register or through an optional parallel input-output (PIO) unit. The PIO unit had wide ports (256 bits) into each array module and allowed transfer of data at 80 megabyte/second rates. Each array port had 1024 wires (256 twisted-pair inputs and 256 twisted-pair outputs) so the PIO unit was relatively expensive. In some applications the common register path was too slow, inconvenient to use, or required large buffer space in the control memory.

In Series E the array I/O was redesigned. We found that data can be reliably transferred over a 32-bit-wide path at 80 megabytes/second so the high I/O rates supported by the original PIO unit can be accomplished with busses only 1/8 as

wide. The 8-to-1 reduction of bus width through the I/O unit reduces its cost dramatically.

Each array module has a multiplexer-demultiplexer (MPX/DEMPX) to pack and unpack data between the 256-bit-wide internal busses and the 32-bit-wide I/O busses (see Figure 1). A control unit associated with the MPX/DEMPX steals an MDA memory cycle to fetch or store I/O data -- both the access mode and the address come from registers in the MPX/DEMPX control unit.

The I/O busses of the array modules are coupled to a cross-bar to permit data transfers between array modules and I/O to external devices.

[1] J. D. Feldman and L. C. Fulmer, RADCAP - An operational parallel processing facility, AFIPS Conf. Proc. Vol. 43, pp.7-15 (1974 Nat'l. Computer Conference).

[2] K. E. Batcher, The Multidimensional Access Memory in STARAN, IEEE Trans. on Computers, Vol. C-26, no.2, pp. 174-177, Feb. 1977.

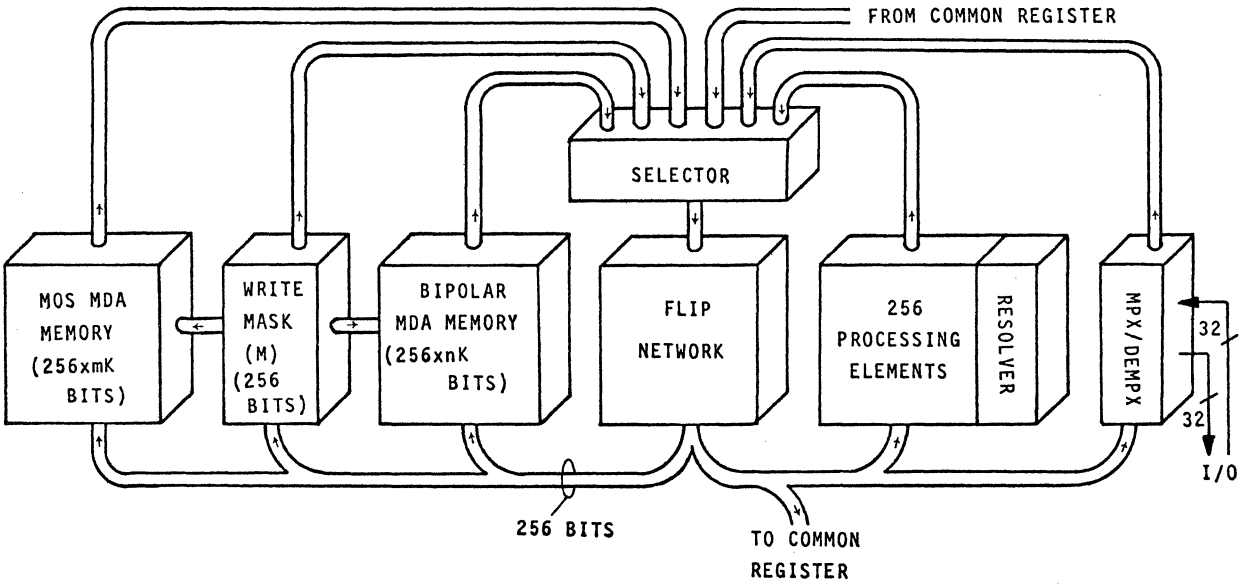


Figure 1 - Series E Array Module

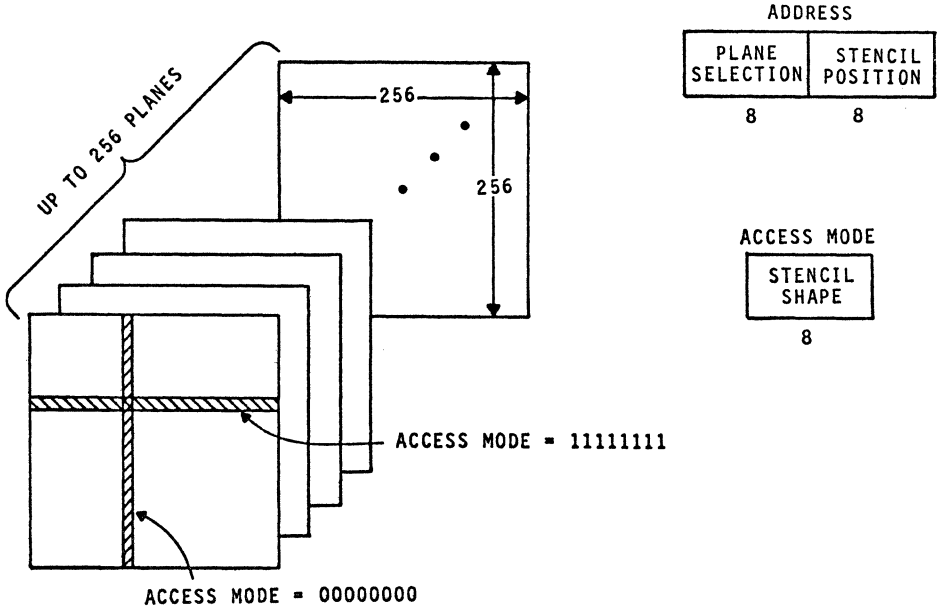


Figure 2 - Accessing Series E MDA Memory

STARAN E PERFORMANCE AND LACIE ALGORITHMS

Roger L. Boulis and Rudolf O. Faiss
Application Engineering
Goodyear Aerospace Corporation
Akron, Ohio 44315

Abstract. The features of the Goodyear Aerospace Corporation's new STARAN(a) E model computer are described and its architecture is discussed. The implications of the new architecture on array storage, input/output (I/O), arithmetic, program design, and software generation capabilities are related to corresponding features of the earlier STARAN B model. Through examination of the use of the STARAN E for performing LACIE(b) tasks now done by the STARAN model B machine, the utility of the new features is shown.

Introduction

The purpose of this paper is to introduce the reader to the improvements that can be afforded to a STARAN E model user over the earlier B model system.

Throughout the paper, potential users of parallel processors are acquainted with the features of the Goodyear Aerospace Corporation STARAN B and E model computers. A critique of the STARAN B model as currently applied to the LACIE investigation [1] provides the basis upon which the STARAN E model establishes its merit.

This paper is presented in three basic sections. The first section provides a short summary of the STARAN computer and its usage experience at the various STARAN installations. This is accompanied by a more detailed description of the functions the STARAN B is required to perform on LACIE algorithms at Johnson Space Center (JSC). The second section describes the highlights of the STARAN E model, making reference to the STARAN B model where applicable. The last section discusses the advantages the STARAN E model can demonstrate over the STARAN B model when the LACIE algorithms are adapted to the new STARAN E architecture.

Background

STARAN Architecture Summary

The STARAN is a modularly constructed computer in which many identical operations may be executed simultaneously; that is, it is a "single instruction stream, multiple data stream" processor.

The basic STARAN building block module is called an array. It consists of an array memory section, 256 bit-oriented processing elements, and a routing network/address structure that allows for multidimensional access of the array memory by the processing elements. Local array control allows selective enabling of data streams. A single STARAN control unit broadcasts instructions to all enabled array modules.

STARAN's modular construction allows for the incremental increase of not only working storage, but also, memory-to-processing element bandwidth and processing elements. For example, in a one module STARAN, an "add" operation can be executed simultaneously for 256 pairs of numbers (or 256 data streams). The parallel execution of an operation for many data pairs is made possible by employing many processing elements (256) per module. In a two module STARAN, twice as many adds can be performed in the same time interval because twice the resources are provided; 512 data streams may be treated simultaneously.

The high processing and throughput speeds of STARAN are a direct result of its parallel processing architecture.

STARAN Usage Experience

Since its introduction to the commercial marketplace in the spring of 1972, STARAN B has been used to solve a variety of applications problems. Independent of the applications, the tasks that have been implemented in STARAN for problem solving can be categorized into two major types, namely, bit and bit-group manipulation type tasks.

The former type tasks generally require access to specific individual bits of both input and intermediate task data items. Bit manipulation capabilities are commonly employed for solving problems that require automatic decisions, e.g., problems that arise out of data base management, text searching, command and control, and air traffic control applications. They are also used for problems that are dominated by one bit data items. A number of problems that arise out of cartography, graphics/drafting, weapons sensor processing simulation, and attribute-to-boundary correlation applications are of this nature.

The second task category, the bit-group manipulation type task, allows access to data items by bit-groups. An N-bit data item may be treated N bits at a time (as in the multiplication of two such items). Such tasks are generally of an arithmetic nature and most often are add or multiply tasks. Applications that employ bit-group processing are far ranging, and those applications that are most likely to demand that bit-group tasks be executed at high rates are those that deal with vectors or arrays of N-bit data items. Examples of applications that treat data of this type at high rates include

(a) TM, Goodyear Aerospace Corporation, Akron, Ohio 44315

(b) Large Area Crop Inventory Experiment at NASA's Johnson Space Center, Houston, Texas

image processing, signal processing, weather forecasting, reactor design, and fluid dynamics applications.

STARAN's processing elements were designed for the manipulation of individual data bits of data items in a great number of data streams, i.e., it is a bit manipulator. At present, STARAN is optimized for performing bit manipulation tasks.

Bit-group manipulation tasks are accomplished in STARAN by executing a regular sequence of operations on the individual bits of bit-group operands. As a result of the manner in which bit-group tasks are accomplished by STARAN, arbitrarily sized N-bit group data items are treated with equal ease. Thus, a 5-bit x 51-bit multiply task is accomplished with no more programming difficulty than a 16 x 16-bit multiply task; both multiply tasks demand about the same processing time.

Three STARAN B installations other than those at GAC - the Rome Air Development Center (RADC), the Engineering Topographic Laboratory (ETL), and the Johnson Space Center (JSC) installations are now in regular use. The RADC and ETL STARAN's are being used primarily for applications that require STARAN to perform bit manipulation tasks, whereas the STARAN of the most recent installation at JSC is being used primarily for accomplishing bit-group manipulation tasks; more specifically, for vector arithmetic processing tasks.

While the usage experience at all three installations has influenced the choice of STARAN enhancements implemented in GAC's upgraded STARAN E machine, the LACIE application task set executed by the present JSC STARAN B will be used as the primary vehicle to demonstrate the improvement of the new STARAN E over the older STARAN B machine.

This task set has been chosen as a demonstration vehicle deliberately for the following reasons: (1) the STARAN B installation is being used in a semi-production fashion, and so good installation usage statistics are available; (2) the LACIE application requires the repeated execution of bit-group tasks; and (3) the cost statistics for the development of the LACIE STARAN software are well known.

Rationale for STARAN in LACIE

Prior to the LACIE program, NASA had developed an enhanced Earth Resources Interactive Processing System that was structured to utilize one of the five IBM 360/75 computers in the real time computer complex in the Mission Control Center of JSC. It was determined to be desirable to use this same software/hardware system for implementing the LACIE program. Yet, the complete implementation of the LACIE software set within the computers of the Mission Control Center would have severely drained the complex of its compute power. On the basis of a competitive assessment, NASA ultimately chose a two array STARAN B to off-load the computationally demanding LACIE data processing tasks [1]. In particular, STARAN B was required to perform: (1) Statistics, (2) Iterative Clustering, (3) Adaptive Clustering, (4) Maximum Likelihood Classification, and (5) Mixture Density tasks.

This paper will show why the STARAN E would simplify the design and coding of software for achieving the above LACIE tasks. It will further show that STARAN E could have performed the tasks more rapidly and potentially could have reduced the number of arrays required for processing from 2 to 1. It will show that STARAN E would have been a more desirable off-loading vector processor than the STARAN B.

STARAN E Highlights

General

Since its introduction, the STARAN B model computer has satisfied and even exceeded system performance predictions for a variety of applications. The extensive and varied use has pinpointed certain STARAN B limitations. At times, users have found the STARAN B word length too short, the data transfers between array memory and program memory too slow, and the size of the page memories insufficient. To remove these limitations, the STARAN B model architecture was modified in three major areas: (1) multidimensional access (MDA) array memory size/speed, (2) control memory size/speed, and (3) MDA array memory I/O.

Significant Hardware Features

Figure 1 is a block diagram of the resulting STARAN E with the modified hardware outlined. The new hardware will be discussed below.

MDA Array Memory Size/Speed. The heart of the STARAN E model is the MDA array memory, a two dimensional matrix of bits. There are three basic components of each array module in a STARAN system, whether it be an E or B model (see Figure 2). A set of processing elements (PE's) is connected through a permutation network to a high bandwidth MDA memory. Rows, columns, or other subsets of data can be read in parallel from the memory, permuted in various ways as they pass through the permutation network, and then can be combined with other data in the processing elements. The processed results can be again permuted and stored into memory in various ways. The arrays in the STARAN E model are still the basic modules from which STARAN E systems of varying size and power are constructed. The maximum number of STARAN E arrays in a given system is eight. The size of the STARAN E arrays has been increased from 256 bits per word, as in the STARAN B model, to an allowable maximum of 65,536 bits per word. Each array still contains 256 words. Any of the 256 PE's within an array still has access to data in any array in the system. A maximum STARAN E model configuration is shown in Figure 3.

Two types of memory may be employed in each memory module - fast bipolar 1,024-bit random access memory (RAM) and slower metal-oxide semi-

conductor (MOS) 4,096-bit RAM. The former has emitter coupled logic ECL electrical characteristics that provide a read time of 120 nsecs and a write time of 160 nsecs. The latter exhibits standard MOS electrical characteristics, yielding 420 nsecond read and write times. The allowed mix and amount of these two types of memory is somewhat arbitrary, but has been chosen at 1K fast RAM memory and 8K of slower MOS memory for the first STARAN E built.

Each array of the first STARAN E model is organized as a matrix of 256 words by 9216 bits, as shown in Figure 4. This matrix is further broken down into 36 256-word x 256-bit square segments. As in the STARAN B model, within these segments, it is possible to read or write all bits of one word, or one bit of all words, or a few bits of many words, or many bits of a few words - all in one memory operation. The M, X, and Y processing element (PE) registers (Figure 4) are still 256 bits wide each and may be used as temporary storage for data moved to or from the array.

Addressing the STARAN E model arrays is almost identical to the scheme used by the STARAN B model. Due to the extended length of each array word, a base register philosophy was incorporated in the STARAN E model and is depicted in Figure 5. A set of six, 32-bit registers provides the capability of addressing the 36 segments of 256 words by 256 bits of the first STARAN E model. A maximum capability of addressing 256 of these segments (corresponding to the maximum array size) is provided.

Control Memory Size/Speed. The main function of the STARAN B control memory is to contain assembled application programs. It has also proven very necessary as a data buffer used by STARAN control, the MDA arrays and the I/O channel of a host computer connection. The maximum address space of the STARAN B model is 65,536 32-bit words. A maximum of 32,768 32-bit words are reserved for magnetic core memory; 1024 words for the high speed data buffer memory. These figures have not changed in the STARAN E model. The high speed page memory system has been enlarged considerably though. Up to 8,192 32-bit words can be stored in each of the three page memories. Instruction execution speed has been decreased almost 20% to 100 nanoseconds. The remaining memory address space can be utilized for addressing the memory of a host computer if so connected. The first STARAN E model built has 16,384 words of core memory, 4,096 words in each of 3 page memories, and 512 words of high speed data buffer memory.

MDA Array Memory I/O. Interarray communication and I/O between the arrays and external devices can occur in two different ways for the STARAN B model. The most usual method is by way of the 32-bit wide common register - a path that can achieve a 12 to 15 megabit per second data transfer rate. The optional parallel I/O (PIO) path, with a channel width of 256 bits to each array in the system, provides the larger bandwidth in the STARAN B model of 640 megabits per second, but at the expense of a great deal of circuitry and cabling.

One of the main features of the STARAN E model is direct access to data within the MDA arrays from an external device. The array access is made by stealing a machine cycle from STARAN control. In this way, STARAN does not have to expend processing time assisting in I/O, but instead may devote all its time to array processing.

Three I/O ports are provided at each MDA array as shown in Figure 6. The first, a 256-bit PIO path, is capable of transfer rates from 512 megabits per second up to 2560 megabits per second. The second is a 32-bit wide multiplexed I/O (MIO) path with the same basic bandwidth as the STARAN B model's PIO - from 80 to 640 megabits per second. The third is the standard common register path to STARAN control that currently exists in the STARAN B model. Its transfer rate varies from 12 to 15 megabits per second. Of these three ports, only the latter two are utilized in the first STARAN E model built.

Implementation of the array I/O is made possible by four hardware units:(1) the array access resolver, (2) a multiplexed I/O controller, (3) the multi-port crossbar switch, and (4) a STARAN command channel (SCC). These are shown in Figure 6 for a two array STARAN E model configuration. The resolver block was shown separately in this figure for clarity. Normally, its function is assumed by the MIO block for diagrammatic purposes.

- Access Resolver. The STARAN E array can be controlled by any one of three units - STARAN control, MIO control, or PIO control. Conflicts are controlled by the access resolver. A "snapshot" scanning resolver is employed to decide which device is allowed access to the array. There are four types of requests that the resolver looks for. In order of priority, they are: (1) STARAN control, (2) PIO, (3) MIO read, and (4) MIO write. A snapshot is taken only when a PIO, MIO read, or MIO write request is made and the requested array is available. STARAN control is the unit that in effect grants the resolver's request to service another unit. The array(s) will only be available if STARAN control is not currently utilizing that array. The highest priority request within the snapshot is then honored for one MDA array memory cycle. If other requests are pending, each of them gets one memory cycle on a priority basis. This procedure continues until all devices eventually drop their requests.

STARAN's exclusive utilization of the arrays is not interrupted unless an array tests busy. At that point, basic array access efficiency remains high but STARAN control efficiency drops slightly.

- MIO. As shown in Figure 6, each array in a STARAN E model system employs a MIO that allows

array I/O to occur on a cycle stealing basis. The array I/O can be initiated by the execution of an I/O instruction of an external I/O processor (IOP) or by the execution of an associative instruction of the STARAN E proper during an interarray transfer. These data transfers take place over the 32-bit wide array I/O busses as coupled together by the crossbar circuitry. The same bandwidth as the STARAN B's PIO is maintained by "burst" transmitting the data with a clock rate of 50 nanoseconds per 32-bit word. Another 32-bit port connected to the MIO is the common register data path from the STARAN E model mainframe. The same data transfer rate as the STARAN B's common register data path is maintained (15-20 megabits per second).

The smallest data item passed through the MIO from the crossbar unit is 256 bits. A clocked multiplex/demultiplex scheme is employed in the MIO that breaks a 256 bit item from the array into eight 32-bit words during transmission to the crossbar. Likewise, when eight 32 bit items are received by the MIO from the crossbar, they are packed into a 256-bit item for parallel transfer to the MDA array. The MIO, as shown in Figure 7 is designed such that data transfers may occur in both directions simultaneously, but not necessarily at the same rate. Data input to the MIO from the crossbar touches both the demultiplex buffer and the control register input. MIO data output to the crossbar comes from the multiplexed output buffer or the control registers. Control registers may be transferred between each other over an internal path. The MIO circuitry is capable of several I/O functions that are useful to internal STARAN E model data manipulation as well as external I/O. In most cases, the MIO is activated by loading the internal control registers shown in Figure 7. The functions that the MIO is designed to perform include: (1) continuous transmit, (3) block transmit, (3) receive, and (4) exchange. These are pictorially represented in Figure 8.

Continous Transmit. An IOP may initiate this mode when it is required to read MDA array data. It establishes a hardware connection to the specific array MIO through the crossbar switch, writes the "continous transmit" register (CR2) and then breaks the connection. The MIO is now activated and will establish the connection to the IOP and begin transmitting array data. The transmission continues until: (1) IOP control terminates the transfer, or (2) CR2 is loaded again by another IOP. The continuous transmit mode allows several IOP's to obtain data from the same array during a given time interval.

The continous transmit mode requires that the intelligence controlling the transfer be on the receiving (IOP) side of the transfer. Data transmission from the MIO can cease at any time and it would be up to the IOP hardware to restart it.

Block Transmit. An IOP may also read array data a block at a time. A connection is made as before and registers CR2, CR3, and CR4 are written into the source array MIO with source and destination addresses, block size, and other pertinent data. The connection is broken as before and the MIO is activated. The contents of register CR3 are placed on the output and loaded into the IOP. The corresponding array words follow. Once a block transfer is initiated, only an "exchange" operation can intervene. Each time an exchange does occur, the MIO will reinitiate the connection and continue the transmission where it left off. A "receive" can also occur at the MIO and allow data to be written in the array on a cycle stealing basis using the "block transmit" capability.

The block transmit function can also be initiated by STARAN control writing CR2, CR3, and CR4 in the source array MIO and allows data to be sent to one or more arrays from the source array. Register CR3 in the source array is loaded into CR1 of the destination MIO(s) and the array words follow. The source array MIO in a block transmit mode will not accept another continuous transmit or block transmit request until the current one is complete. However, the source array MIO will allow "exchange" and "receive" operations to occur as explained above.

Receive. An IOP may write data to an array by first connecting to the MIO. Next, CR1, the "receive control register" is loaded. Data is then transmitted into the MIO. The receive operation will allow any other operation to intervene on a cycle stealing basis. However, the IOP or source array must have the capability to reinitiate the "receive" operation.

Exchange. Another type of interarray communication that can be performed by the STARAN E model is the "exchange" function. This function uses the data path afforded by the MIO and the crossbar switch to perform a synchronous data transfer. Array address and control information are provided by an associative instruction executed within STARAN control that triggers the exchange function. Array connectivity is controlled by the connection registers within the crossbar switch. All data manipulations performed by the associative exchange instruction are identical to their non-exchange counterparts. Its basic purpose, however, is to provide interarray data communication. The contents of one memory location may be swapped synchronously with the contents of another's memory location. Or, data from one array may replace data in all the other arrays in the system if so desired - all within the same period of time that it takes to read the data from the source array. A variety of other exchange options are available to the STARAN user.

- Crossbar Data Switch. The crossbar data switch of the STARAN E model provides part of the path that allows MDA array data to be accessed by external devices (and other arrays) without direct STARAN participation in the transfer. The crossbar is an eight port data switch that has one of its ports connected to each array in the system. Each port contains 32-bit wide data paths and operates at 80 to 640 megabits

per second. Remaining ports may be connected to IOP's for direct MDA array transfers. A port is defined as having simultaneous input and output capabilities.

Large systems may have up to eight arrays and have several IOP's attached. In order to accommodate situations like this, the crossbar design allows a second crossbar to be added to the first, thus providing 14 ports. Up to four crossbars may be ganged together like this to provide 20 free data ports.

All STARAN E systems may also use the crossbar data switch to transfer information from one array to another. The transfer may be synchronous under control of a STARAN associative instruction, or asynchronous under MIO control.

The STARAN E model instruction set was expanded to incorporate several new instructions that control the MIO, crossbar and SCC. Those required for the crossbar proper allow four basic functions to be performed: (1) reset, (2) read registers, (3) write registers, and (4) strobe status. These instructions are issued over the SCC to the crossbar and MIO devices. The control registers in the crossbar as well as the MIO are accessible by any device attached to the SCC.

- STARAN Command Channel (SCC). The SCC is the vehicle by which command and control information is passed from STARAN control to the IOP's and MDA array MIO's that are attached to the crossbar data switch. The SCC is driven by I/O instructions executed by STARAN. In addition to command transfer, the SCC also provides a data output and input path 36 bits wide, including 4 bits of parity. IOP interrupts are also supported by the channel. The SCC, crossbar data switch, and MIO are all involved during interarray operations and are initiated by STARAN I/O instruction execution. The SCC originates at STARAN control and can be daisy-chained from its first connection at the crossbar control logic to any IOP's within the system.

STARAN E vs. STARAN B in LACIE

STARAN Program Storage

It was indicated earlier that the LACIE processing at NASA is handled by IBM 360/75 host computer(s) that off-load five computationally demanding LACIE pattern recognition tasks to a STARAN B computer. The particular 2-array STARAN B used is equipped with a 136K byte (34K - 32-bit word) control memory. The STARAN B is connected to the selected host computer via a custom-built channel-to-channel interface unit that connects to STARAN's buffered I/O (BIO) port. Data that transfer through this port are read from or are written to the STARAN B model's program memory (i.e., program memory is used for data as well as instruction storage).

The maximum intermachine data transfer rate is restricted to less than one Megabyte per second as a result of the need for very long cable lengths. (Since fundamental data transfer rates that can be supplied by the host during the execution of a STARAN LACIE task lie substantially below this rate, the one Megabyte per second intermachine transfer rate limit imposed by the channel causes no apparent impact on LACIE processing.) To support block data transfers between machines, receive and send I/O buffers are provided by both machines. The size of these buffers is related to the maximum data block size allowed in a single one-way data transfer. To minimize the number of I/O interrupts the host machine is required to process, it is desirable to use large data block transfers. Furthermore, to support simultaneous STARAN processing, host processing, and intermachine data transfers, more than one I/O buffer is required in each machine. As a result, a considerable portion of STARAN B program memory is allocated for use as I/O buffers. At present, data block sizes for one way intermachine transfers are restricted to 20K or less bytes. For reasons imposed by intermachine I/O protocol, on the order of 60K bytes of buffer space must be made available in STARAN to achieve a double buffering capability.

Thus, a large percentage of STARAN control memory is allocated for use by transient data. Yet other data, namely input parameter data, require large allocations of program memory storage space for classification tasks when pixel measurement vectors have many components. In particular, when 20-component measurement vectors are associated with the pixels to be classified, when the reference statistics for 60 crops are to be used in classifying pixels, and when the Mixture Density Classification task is used to accomplish classification, the input parameter data (crop reference statistics) require on the order of 56K bytes of program memory. Another 4K bytes of program memory is required for intermediate data storage when the above task is executing. As a result, only on the order of 16K bytes of program memory is available for the LACIE executive, STARAN I/O handlers and LACIE task application modules. The actual system software requires about 8K bytes of the remaining 16K bytes, leaving 8K bytes of program memory for all five LACIE applications tasks. Since the amount of code for the LACIE tasks exceeds 40K bytes of instructions, insufficient program memory exists in the JSC STARAN to allow the various STARAN LACIE TASK modules to co-reside within the STARAN-B program memory. In fact, the complete instruction set for only one LACIE application task exists in the program memory at any one time. The remaining programs are stored on a disc associated with the STARAN system. Thus, when the host calls upon STARAN to execute a task, STARAN must first determine whether or not the desired task program already exists in program memory. If the program is not resident in the memory, it must be called in from disc storage. Such an operation results in substantial delays and becomes particularly noticeable when task calls treat relatively few pixel measurement vectors.

In order to alleviate the delay problem, the applications programs were separated into initialization and processing segments; the ini-

tialization segments for all five tasks were combined and are kept in program memory. Thus, if a task needs to be loaded from disc, STARAN may proceed with the execution of initialization actions as the processing segment of the task program is being loaded.

The need for such "cleverness" in managing the LACIE application programs disappears when using the STARAN E machine in place of the B machine. The program memory of the E machine is supplemented by 288K bytes of array memory. If an E machine were used for the LACIE program, it would be tied to the host via a STARAN E crossbar port. Pixel measurement data would pass directly into STARAN array storage rather than into program storage. Intermediate task results would remain in array memory. And the bulk of the program memory would be available for program storage. As a result, all five STARAN LACIE applications programs would co-reside in the three 16K byte page memories. Delays in task execution disappear with the elimination of the requirement to load programs from disc. Even short tasks would be executed efficiently since all tasks would execute out of high speed page memory. The systems software and the applications program design would be reduced substantially. All software now would fit comfortably within the 48K byte page memory system, and program segmenting would be eliminated. The availability of large I/O buffers would permit simpler I/O handlers.

STARAN Array I/O

When the JSC STARAN B I/O buffer contains pixel data and the task program requires it to be moved to array storage, the data must be moved to array storage by way of the common register. When the transfer is made most efficiently, the full bandwidth of this path is only on the order of 2 Megabytes per second (as compared to the 512 Megabyte per second bandwidth for the processing elements-from-arrays path). During such transfers, all STARAN processing stops.

Because data transfers between the STARAN program and array memories occur at a slow rate, and because such transfers degrade the processing power of the machine, much effort was spent to attain a software design for each task that would eliminate the use of the program memory for storing intermediate task data. Only the software design for the Maximum Likelihood Classification task was able to eliminate such transfers; all other tasks required the movement of at least some intermediate data through the common path.

The impact of moving intermediate data was most severe on the timing of tasks that are used to establish crop reference statistics; namely, the Statistics, Iterative Clustering, and Adaptive Clustering tasks. The ratio of the amount of time spent for making common path data transfers compared to the time required for arithmetic processing is dependent on both the task and task setup parameters. Ratios of 5:1 for Iterative Clustering and 1:5 for classification are representative of those likely to be observed while performing commonly encountered LACIE jobs.

The STARAN E would eliminate the requirement to move input, intermediate, or output data through the common path since such data would remain in the array memory throughout a task. The execution speed of all tasks would increase significantly; for the clustering tasks, dramatic execution time improvements in excess of 2:1 could be expected. The construction of code to implement intermediate data storage would be simplified since the temporary storage region would be in the same array memory that produced the intermediate data.

STARAN Algorithm Development

The STARAN B model array word is 256 bits long; the two array JSC STARAN B model provides 512 such words. Regardless of the LACIE task called, 512 pixel measurement vectors are treated at a time; one vector is assigned to each array word. To satisfy LACIE requirements, vectors with up to 20 8-bit components have to be accepted as input. To provide worst-case capability, 160 bits of a STARAN array word are required for storage of the input pixel data. The remaining 96 bits are available for storing intermediate pixel related data and for performing required arithmetic/logic operations. The 96-bit wide space is inadequate for storing intermediate results for all but one task (Maximum Likelihood Classification) and, as was described earlier, forces the use of program memory for storage.

Because of the undesirability of using program memory for the storage of intermediate results, the software design effort sought to (1) minimize the number and types of intermediate data items and (2) reduce the precision of all such items to the bare minimum required to fulfill LACIE accuracy requirements. The design guidelines resulted in the intensive examination of the basic arithmetic descriptions of the tasks. This allowed arithmetically equivalent forms to evolve that could be computed fast, operate in minimal field space, and generate intermediate data with well defined statistics. As examples, two major subroutines that allow the Maximum Likelihood Classification task to hold intermediate data in array storage are the common multiply and accumulate routine and the rounded square routine. The former routine conserves array field space by eliminating the process of first generating a product field and then adding it to the accumulation of previous products. Instead, the product field is added directly into the accumulation field as the product is formed. The latter rounded squares

Clearly, the array space of the STARAN B model forced the software designer in this case to conserve as much field space for intermediate storage as possible. A STARAN E model applied to the LACIE application has no requirements for special field conserving subroutines. Furthermore, the new variable floating point arithmetic routines that are provided by STARAN E's software language are at least as efficient as those developed specifically for the LACIE tasks.

routine squares the contents of a particular input field and simultaneously rounds the squared field to the length of the final output field.

Summary/Conclusions

The commercially available STARAN E model hardware/software enhancements include:

- faster MDA arrays that provide a minimum of 36 times the storage capacity of present STARAN B model arrays,
- new crossbar hardware that allows interarray data transfers from 8 to 64 times faster (at composite rates up to 640 megabytes per second per crossbar) and will allow host-to-STARAN array moves (at rates up to 80 megabytes per second per array),
- new page memories that provide 8 times the storage and allow up to 65% faster array instruction execution times, and
- a set of floating point arithmetic modules that allow the STARAN programmer to arbitrarily specify the mantissa and exponent lengths.

The impact of using a STARAN E model rather than a STARAN B model, for the LACIE program, is summarized below.

- All five LACIE applications software modules could be pre-loaded into the page section of STARAN E's control memory. No calls to the STARAN disc would be required after a task was requested by the host computer. As a result, even tasks involving few data items would be able to be performed efficiently in the STARAN E.

- No intermediate data generated during the course of task execution would need to be stored in STARAN program memory. Intermediate data would be stored within the confines of the arrays. The time required to store or access such data would be reduced by a factor of at least 200:1. By eliminating the requirement to store data in program memory, all tasks would be able to execute in less time even if the fast array memory bandwidths had not been improved.

- Intermachine I/O would move data directly between STARAN arrays and the host. By using this path for moving data, no STARAN program memory data exchanges are required. As a result, the degrading effect on processing power, that data transfers along this path cause, is eliminated.

- Software design and layout costs would be reduced dramatically as a direct result of having ample control and array memory resources. A one-array 9K bits/word STARAN E model provides about 3 times the total storage of the two-array JSC STARAN B. The effort that was necessary to fit both programs and data into a limited storage facility would not be required. The new standard STARAN E model arithmetic modules provide execution times that rival those of the specialized modules developed in LACIE. Thus, software debug time would be limited largely to main routines.

Based on the results of the study, a single array STARAN E model machine can perform the overall LACIE tasks as well as a two-array STARAN B model. The one-array E model would execute clustering tasks faster than the two-array B model machine, but would execute classification tasks somewhat slower than on the B model machine. It would outperform the B model when executing tasks involving only a small number of pixels. Both the STARAN application and systems software for the E machine would have been simpler to design, required less code, less time to code, less time to debug, less time to document, and less time to maintain. The overall software costs would likely have been halved.

Acknowledgment

The authors wish to express their gratitude for the technical support provided by John P. Rasmussen, one of the major design engineers responsible for the new array I/O architecture.

References

[1] R. Faiss, J. Lyon, M. Quinn, S. Ruben, "Application of a Parallel Processing Computer in LACIE," 1976 International Conference on Parallel Processing, pp. 24-32.

[2] Goodyear Aerospace Corporation, "The STARAN E System - An Overview," GAC Document Number AP-123226, 29 September 1976.

[3] K. E. Batcher, "STARAN Series E," 1977 International Conference on Parallel Processing.

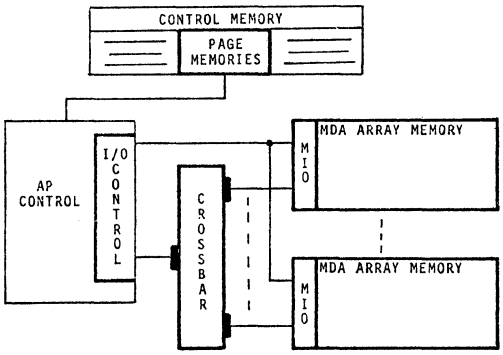


Figure 1. STARAN E Block Diagram

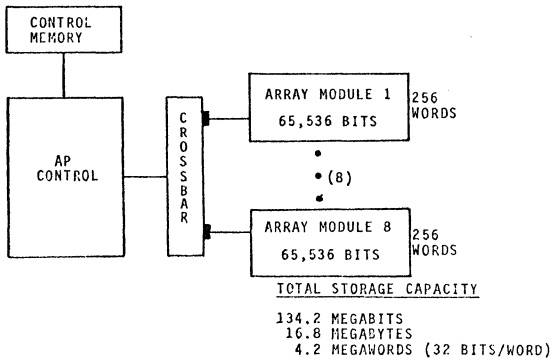


Figure 3. A Maximum STARAN E Configuration

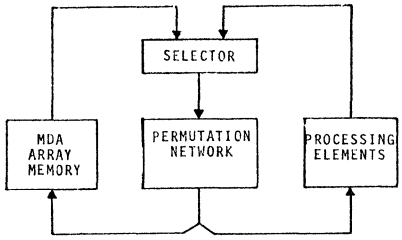


Figure 2. MDA Array Memory Module

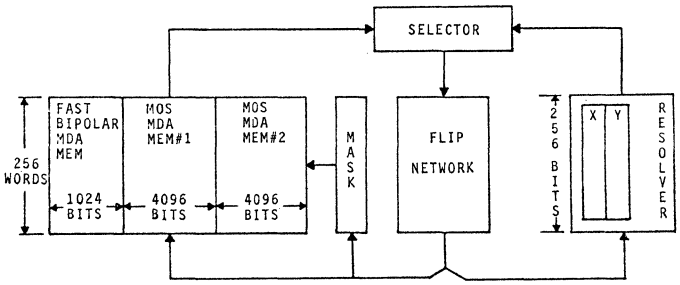


Figure 4. Typical MDA Array - First STARAN E System

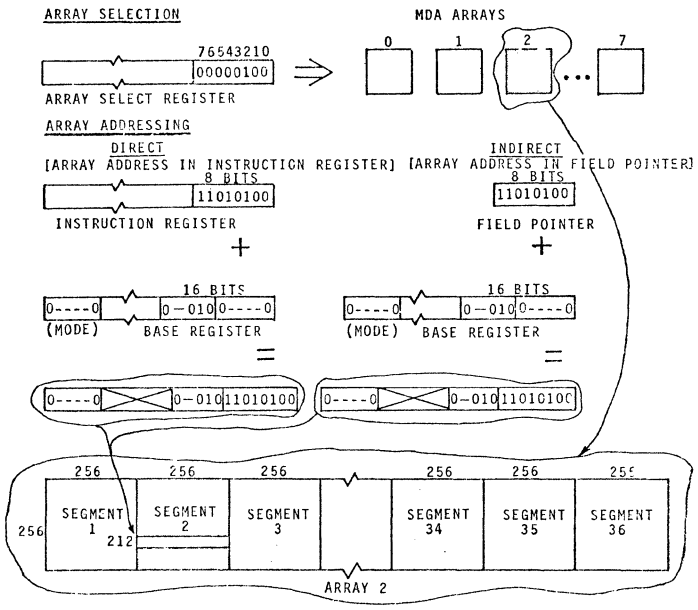


Figure 5. Addressing Example - First STARAN E System

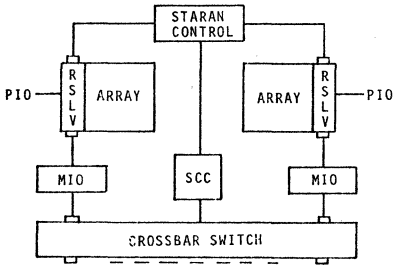


Figure 6. STARAN E Array I/O

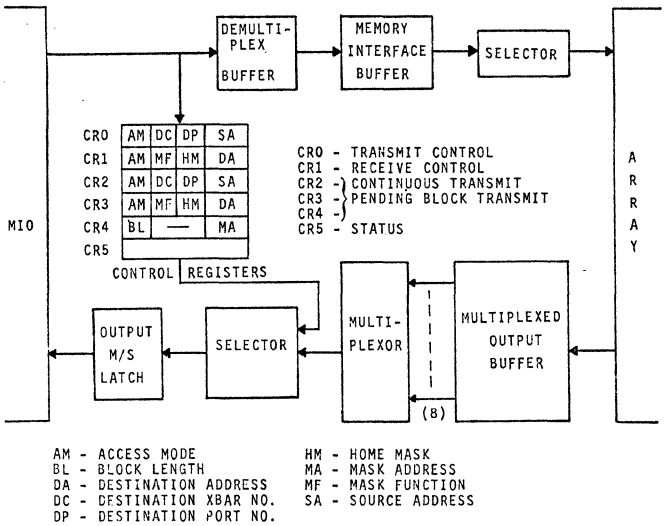


Figure 7. MIO Unit - STARAN E System

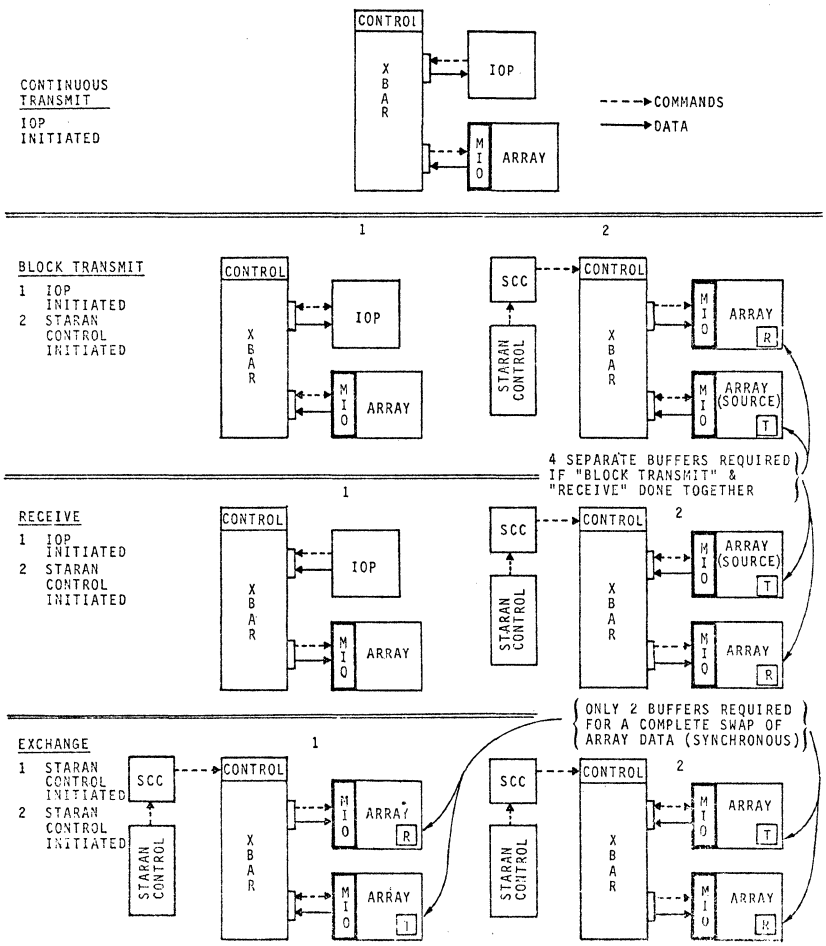


Figure 8. MIO Data Transfer Functions

A MODIFIED ALAP CELL FOR PARALLEL TEXT SEARCHING

Hubert H. Love, Jr.
Hughes Aircraft Company
Culver City, California 90230

Summary

The Associative Linear Array Processor (ALAP) and several of its applications are described in (1) and (2). An ALAP memory consists of a set of cells organized around four bit-serial data channels. Three of these are common buses connected to external registers. One of the common channels permits both arithmetic and match operations to be performed between the data in selected cells and an external operand. The selection of the operation is global. The other two common channels are for input and output. All three common channels can operate simultaneously.

The fourth channel, the "chaining channel", permits each cell, under a combination of local and global control, to transfer data and control states to its nearest neighbor in one direction only. Each cell, under local program control, can either accept the data from its chaining channel input, or else relay the data to its neighboring cell.

The major components of the ALAP cell are:

1. A 64-bit shift register, the "data register", which holds the cell's data.
2. An arithmetic unit which performs arithmetic and match operations between the contents of the data register and the data from the chaining channel or one of the common channels. The results can either be retained in the cell or put on the chaining channel to the next cell.
3. A six-bit "flag register". The states of the bits determine the operation of the cell's routing logic, and also determine whether or not the cell is to participate in input, output, data transfer or arithmetic operations during command execution. The flag bit states can be reordered and logically combined, and can also be transferred from cell to cell via the chainring channel.

There are three basic ALAP commands. One of these is used in conjunction with fault-detection software, and can effectively remove a malfunctioning cell from the array. The second command is the "flag shift" command which alters and reorders the flag register contents. The third command is the "word cycle" command. This command causes the data register contents of a selected subset of cells to be shifted simultaneously while data transfer, match or arithmetic operations take place as specified by the flag registers and the states of global control and data lines.

The chaining channel and the cell's arithmetic and control logic gives the ALAP memory the capability for performing arithmetic, match and data-transfer operations among many sets of cells simultaneously.

The basic text-processing task discussed here is that of locating all occurrences of a specified set of "key" words appearing, in any order, in a large data base of raw text, such that the words lie within a specified range of character positions. The sentences which encompass each occurrence of the set are output to the user together with the associated document identifiers.

For this application, several modifications to the basic ALAP cell are required. The structure of the modified cell is shown in Figure 1.

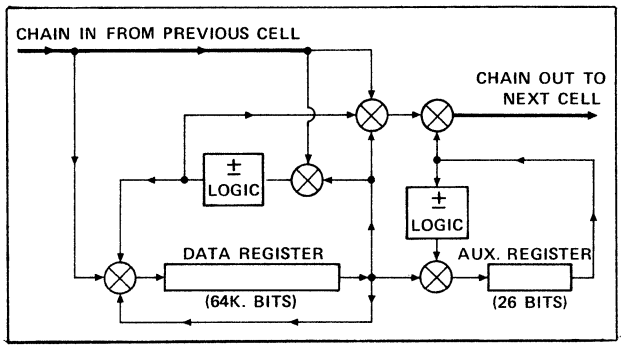


Figure 1. Modified ALAP Cell

The principal modifications are the following:

1. Instead of 64 bits, the modified cell has a data register of 64,000 bits for storing the text. It is hoped that the use of a very long data register will reduce the cost-per-bit of the modified ALAP memory to the order of that for a medium priced disk. The data registers are fabricated on the same multi-cell wafer with the rest of the cell logic.
2. The modified cell contains an additional register, the "auxiliary register", together with some associated data routing and arithmetic logic. This 24-bit register can recirculate through a small arithmetic unit which can subtract or add a globally-specified operand to the register contents. In addition, the auxiliary register interfaces with the chaining channel in the same fashion as the data register.
3. Unlike the original ALAP cell, the arithmetic logic for the modified cell includes no step-multiply, step-divide or step-square-root capability.

For the text-processing task, the entire text resides in a succession of eleven-bit "character fields" in the data registers, each field containing one character plus three flag bits. Sentences and even character fields may lie across cell boundaries. Since the chaining logic permits all cells to shift their contents simultaneously on the

chaining channel, the entire ALAP memory can act as one large shift register, and the cell boundaries can usually be ignored during processing.

The search procedure is conducted in three phases. The first of these is the operation of locating all occurrences of each key word in the specified set, and of tagging them in the flag bits of their most-significant characters. The second phase is the process of locating and tagging all occurrences of the set of key words which lie within the specified range. The third phase consists of outputting to the user the sentence or sequence of sentences which contain each matching set of words, together with the document identifiers. All three phases of the process are performed in parallel for all cells, and are independent in execution time of the size of the data base. However, the execution time for the third phase is dependent on the number of matching sets, since the matches are output sequentially.

In Phase 1, a number of parallel searches equal to the number of characters in the word are performed so that all occurrences of the word may be found, regardless of their orientation. Figure 2 illustrates this process. The contents of three (abbreviated) cells are shown with the chaining channel interconnections indicated by the arrows. They key word being sought is "TRUTHS". Below are shown four of the six comparands, corresponding to four of the six orientations of the key word, as they are fed into the ALAP memory from an external register. The third of these is seen to match successfully. Phase 1 requires n complete word cycle operations, where n is the total number of characters in all of the key words.

Phase 2 of the operation requires a separate search for each possible permutation of the set of key words. In searching for each permutation, one (12-bit) field of the auxiliary register is used to count the characters between the first key word and the last, according to the range specification. A second (11-bit) field contains each single character from the text in turn, shifted in from the data register. This permits each text character to be compared against several characters without requiring complete word cycles for each comparison. Several comparisons are required because the various cells may be in different states with regard to the search (for example,

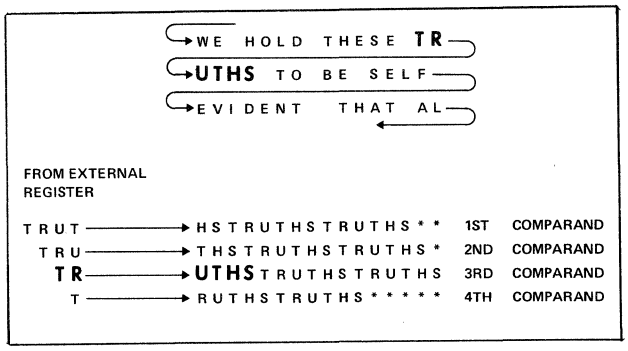


Figure 2. Phase 1 Search Operation

different key words may be the object of the current search, depending on the cell). The state of the operation within the cell is denoted by the setting of a 3-bit field of the auxiliary register.

For each permutation of a set of k key words, Phase 2 requires 2k complete data register shifts. Each shift is about k-1 times slower than those in Phase 1 because of the aforementioned repeated comparisons. The total Phase 2 execution time is therefore equivalent to 2k(k-1)k! word cycle operations. (In practice, k will rarely be greater than 4.)

Phase 3 requires two data register shifts (i.e., word cycle operations) for each matching key word set. At a clock rate of 5 mHz, a word cycle requires 13.1 msec. If the ALAP clock frequency is 5 mHz, and if key words are six characters long, total search times for sets of two, three and four key words are 0.6 sec, 1.7 sec, and 9.7 sec respectively, including a 20 percent overhead for flag shifting operations.

References

(1) C.A. Finnila and H.H. Love, Jr., "The Associative Linear Array Processor", IEEE Transactions on Computers, Vol. 0-26, No. 2, (February, 1977), pp. 112-125.

(2) Hubert H. Love, "Radar Data Processing on the ALAP", Proceedings of the 1976 International Conference on Parallel Processing, IEEE Computer Society, (August, 1976), pp. 161-167.

PERFORMING SUMMATION AND PRODUCT IN AN ASSOCIATIVE PROCESSOR

I-Ngo Chen
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada

Summary

It is well known that successive operation on a set of n numbers (e.g. the summation or the product of n numbers) required $\log_2 n$ steps for parallel processings no matter how many processors are assumed to be available. For an associative processor employing bit-sequential-word-parallel operation [1], the number of steps required for the summation of n numbers each of length m is

$$(\tfrac{1}{2}\log_2 n + m)\cdot\log_2 n \quad \ldots(1)$$

For the product, the number of steps required will be

$$\tfrac{1}{3} m^2(n^2-1) \quad \ldots\ldots\ldots(2)$$

if the general shift-and-add multiplication is employed.

In this short paper, we present two procedures which would reduce the bounds of Eqs. (1) and (2), if the associative memory is sufficiently large. We shall thus assume:

1. that the associative memory has at least n words and that the word length is sufficiently long;
2. that there is a data-manipulator [2] which can perform some simple data manipulating functions [3] like
   shift
   En(X) - take all even elements of the vector X
   Od(X) - take all odd elements of the vector X;
3. that for simplicity, all numbers are positive integers;
4. that there is a bit-slice full adder (be it hardware or routine like in STARAN [4]) which takes a bit-slice of augend Ai, a bit-slice of addend B_i, and a bit-slice of previous carry C_{i-1} and gives as outputs, a bit-slice of carry C_i and a bit-slice of sum bit S_i. i.e. We shall have F_1 and F_2 such that

$$\left.\begin{aligned} C_i &= F_1(A_i, B_i, C_{i-1}) \\ S_i &= F_2(A_i, B_i, C_{i-1}) \end{aligned}\right\} \quad \ldots\ldots\ldots(3)$$

To perform summation of n numbers each of length m, first we divide the numbers into 2 equal parts A^1 and B^1 as shown in Fig. 1. Next we read a bit-slice of A^1_j and a bit-slice of B^1_j to the full adder and write the odd and the even elements of the sum S^1_j respectively into A^2_i and B^2_i. i.e.

$$\left.\begin{aligned} A^2_i &= Od(S^1_i) \\ B^2_i &= En(S^1_i) \end{aligned}\right\}, \text{ for } i = 1,2,\ldots,m.$$

In general

$$\left.\begin{aligned} A^{j+1}_i &= Od(S^j_i) \\ B^{j+1}_i &= En(S^j_i) \end{aligned}\right.$$

for j = 1,2,...,k, where k is the least integer greater than or equal to $\log_2 n$.

As depicted in Fig. 1, the number of steps required is

$$m + 2\log_2 n \quad \ldots\ldots\ldots\ldots\ldots(4)$$

while the number of read and writes is

$$2 \times 2(m + 2\log_2 \tfrac{n}{2})$$

compared favorably to

$$\log_2 n\ (2m + \log_2 n - 1) \text{ reads}$$

and

$$\log_2 n\left(m + \frac{\log_2 n + 1}{2}\right) \text{ writes}$$

as required by ordinary tree sum addition.

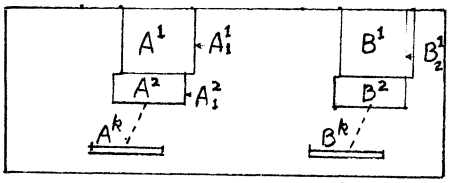


Fig. 1 Lay-out of operands in memory

Multiplication of two numbers $Q=q_m q_{m-1}\cdots q_1$ and $R=r_m r_{m-1}\cdots r_1$ can be performed by the following summation

$$T=\sum_{i=1}^{m}(2^{i-1}\cdot q_i\cdot(r_i r_{i-1}\cdots r_1)+2^{i-1}\cdot r_i(q_{i-1}\cdots q_1)) \quad \ldots\ldots(5)$$

Let

$$\left.\begin{aligned} Q_i &= q_i\cdot(r_i r_{i-1}\cdots r_1) \\ R_i &= r_i\cdot(q_{i-1}\cdots q_1) \\ P_i &= \sum_{j=1}^{i} 2^{j-1}\cdot(Q_j+R_j) \end{aligned}\right.$$

Then Eq. (5) can be rewritten as

$$T=P_m.$$

But $P_i=2^{i-1}\cdot(Q_i+R_i)+P_{i-1}=G_1(Q_i,R_i,P_{i-1})\ldots(6)$

and T_i, the i^{th} bit of T, can be obtained at the i^{th} iteration of Eq.(6) for i from 1 up to m. In fact, T_i is the ith bit of P_i. So we may write

$$T_i=G_2(Q_i,R_i,P_{i-1}) \quad \ldots\ldots\ldots(7)$$

The similarity of Eqs. (6) and (7) to that of Eq.(3) allows us to perform successive multipli-

cation the same way as performing successive addition. For product of n number each of length m, the total number of steps required will be $m \cdot m \cdot \log_2 n$.

References

[1] Feldman, J.D. and Reiman, O.A. "RADCAP; An Operational Parallel Processing Facility", Proceedings of the Sagamore Conference on Parallel Processing, August 1973.

[2] Feng, Tse-Yun, "A Versatile Data Manipulator", Proceedings of 1973 Sagamore Computer Conference on Parallel Processing.

[3] Feng, Tse-Yun, "Data Manipulating Functions in Parallel Processors and Their Implementations", IEEE TC Vol. C-23, No. 3, March 1974.

[4] Batcher, K.E., "STARAN/RADCAP Hardware Architecture", Proceedings of 1973 Sagamore Computer Conference on Parallel Processing.

THE NODE KERNEL: RESOURCE MANAGEMENT IN A SELF ORGANIZING PARALLEL PROCESSOR

Herbert Sullivan
Sullivan Associates
200 West 79th Street
New York, New York

T.R. Bashkow, D. Klappholz
Dept. of Electrical Engineering and Computer Science
L. Cohn
Center For Computing Activities
Columbia University
New York, New York

INTRODUCTION

CHoPP

This paper describes certain aspects of software support for CHoPP (Columbia Homogeneous Parallel Processor), a large scale MIMD machine which has been under study by our group for almost two years [1,2,3,4]. CHoPP is intended to speed up digital computing by use of parallelism; its applications are not restricted to any special class of algorithms, nor limited to numerical analysis, or to non-numeric computation. The goal of speed up is common to most proposals for parallel computers. However the idea of a single machine capable of achieving such speed up for practically all of the mainstream algorithms of digital computing is not usually to be found in previous and contemporaneous proposals and implementations. CHoPP's approach consists of supporting vast amounts of parallelism at much lower levels of hardware and software than has heretofore been attempted in MIMD machines. At this low level, parallelism is to be found in virtually every program, and this parallelism may be exploited to speed up computation.

There exists a large and growing literature on the complexity of parallel algorithms, [5,6] which shows that significant parallel speed up may be obtained for the great majority of algorithms of computer science. In some cases, the speed up available is proportional to the number of processors employed. These theoretical results have been confirmed [7] and extended [8] by analysis of typical FORTRAN programs, and by measurements, using trace techniques, of the run-time parallelism available in such typical programs. The speed up calculations in the previous work do not usually take into account any of the "overhead" functions which contribute to the execution time in a practical multiprocessor. Specifically, no allowance is made for time required to assign a processor to a task, or for time required to communicate results from one task to another. Moreover, it is assumed that tasks are scheduled, in accordance with the constraints of the parallel algorithms presented, without any delay. The parallelism* identified in this work is characterized by very short tasks, which execute for a few instructions, and then terminate, transmitting results to some other task. Implicit in these analyses is a model of a parallel processor as an MIMD machine in which negligible time is required for assignment and reassignment of processors (task switching) and negligible time is required for communication of results.

*This form of parallelism is often called a "tightly coupled process". However the term has been used in various ways – for example C.MMP has been termed [9] a tightly coupled processor, as compared to the ARPA network of processors. The sense in which we would like to use the term is very different from such usage. We have therefore avoided "tightly coupled" perhaps at the expense of some circumlocution.

Most of the previous designs [10,11,12] consist of an assemblage of conventional computers, minicomputers, or microcomputers interconnected to form some kind of cooperating system. Although a variety of interconnection schemes and memory structures have been tried [14], and sophisticated operating systems devised [15], the existing and proposed systems fall far short of supporting the kind of parallelism described in the last paragraph.

As a consequence, a central theme of multiprocessor research has been [12,13] the effort to discover parallel programs whose peculiar properties permit them to be executed with low task interaction, and infrequent task switching. These endeavors lead away from general purpose computing, to the identification of special applications. This search has thus far produced no large or interesting collection of suitable algorithms. Some special algorithms have indeed been uncovered [22], but it has yet to be shown that these have any bearing on the mainstream problems of digital computing.

CHoPP is intended to support the form of parallelism generally available in computer programs. As such, it is designed to provide a good approximation to the idealized model of parallel multiprocessing described above. This necessitates (among other things) that task switching and intertask communication time both be reduced by several orders of magnitude, as compared with existing practice. To approach this level of performance, we have found it necessary to revise current notions of almost every aspect of multiprocessor architecture, in the processor design, in the memory to processor interconnection network, and in the software support structure. The resulting architecture bears little relationship to that of conventional multiprocessor designs. Some of its structures are more reminiscent of high speed sequential machines and vector processors, in that they rely on extensive pipelining and high bandwidth interleaved memory.

The Virtual Hardware Machine

The hardware implementation of CHoPP has been described elsewhere [1,3,16]; for the purpose of this paper, it is sufficient to explain the virtual hardware machine which this implementation provides for use by the system programmer. This machine runs the operating system and executes the language support software. The facilities available on this machine determine the ease with which systems' programs may be written, and together with the timing of operations, determine the overall speed of execution of system programs. The specification of the virtual hardware machine includes all facilities and timing available to the system programmer, but excludes structures which are implemented in firmware and hardware. Thus the hardware virtual machine is just that machine which is usually described in a computer users manual, but not the one described in the maintenance manual.

Figure 1 shows a block diagram of CHoPP at the virtual hardware level. There are three main elements: a number (N) of identical processors, a shared memory, and an interrupt system which permits one kind of interprocessor communication. The processors are general purpose computers. In preliminary designs a word length of 32 bits appears appropriate; however, such details are beyond the scope of this paper. The memory for these computers is a single, shared entity, organized into a single address space, and accessable, in toto, by every processor on an equal basis. Using conventional semiconductor technology, the speed of each processor will be approximately 200,000 instructions per second, and it is realistic to implement a system with approximately 1000 processors, such a machine would operate at 200 MIPS. There is nothing in the hardware design, nor, as we shall see, in the software structure, to limit the number of processors implemented.

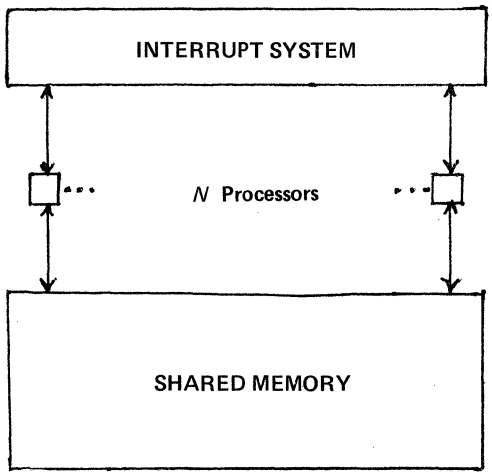


Fig. 1. Virtual Hardware Machine

The key element in the system shown in Fig. 1. is the common shared memory. All processors access the memory independently and concurrently. Conflicting references are resolved by the underlying hardware without loss of efficiency and effectively without any loss of time. Consequently there is no need to provide copies even when code or data is shared by a large number of processors. The most important aspect of this shared memory is that communication of code and data from one processor to the other is accomplished by passing an address pointer. This can be done in one instruction time, independent of the length of the material being transmitted. The common, shared, conflict free memory is thus the mechanism by which the intertask communication requirements (described in the previous section) are met. Ultimately, the shared memory is also the basis for achieving the rapid context switching required for the support of parallel processes in the general purpose environment. Whenever a new task is to be initiated, it can be sent to any desired processor by the execution of a single instruction, which contains an appropriate pointer to a block of shared memory. Of course, additional mechanisms are required to schedule and synchronize parallel tasks. But for all of these, a key factor is intertask communication, which makes possible the transmission of arbitrarily long messages, with negligible overhead. The implementation of the memory processor structures which accomplish the performance just described, is the principle hardware innovation of CHoPP. This implementation is efficient and economical for practically unlimited numbers of processors.

The Problem of Control and Organization

The high performance hardware structure of CHoPP does not, by itself, provide any assurance of high system performance. Efficient techniques for organization and control of parallel tasks are required. The problem may be stated thus: some mechanism is needed which assures that, whenever tasks to be run are waiting, they will be assigned to available processors without delay. In conventional multiprocessors, this organization is provided by the operating system; in CHoPP it is the function of analogous software called the nonde kernel, whose operation is supported by hardware primatives. The importance of this kind of software has been pointed out [14] by Enslow. In the conventional multiprocessor, it might be hoped that system performance improves materially with the addition of a processor, provided the number of processors is sufficiently small. This is not the case; Enslow reports an example where throughput increased by a factor of 1.8 for a two processor system, and by a factor of only 2.1 for a three processor system. He attributes this non-linearity, in part, to the operating system.

Amdahl, as quoted [17] in Computer World, goes further. "The need for control and coordination software . . .[results] in a situation where by the time you got to four processors you actually had less performance than with three."

The problem of control of parallel processes has two sides. From the standpoint of the user, there are instructions which invoke parallelism and provide for intertask communication. From the standpoint of the system; there are structures which interpret the user instructions and carry out the intended parallel execution. The relative roles of the user language and of the system support differ in various approaches to parallelism. In CHoPP, the node kernel does actual assignment of processors, and (with extensive hardware support) mediates communication between tasks. These activities are performed in response to the user program, which contains instructions invoking parallel processing. To understand the node kernel, it is first necessary to briefly consider the form of language constructs.

In CHoPP the application program contains the calls for parallel execution of code. When the user determines that some sequence of instructions can be executed concureently with the rest of the program, he identifies this sequence as a task. When the program is run, the node kernel will schedule the task on one of the processors. The user view of CHoPP is thus similar to that of some multitasking systems (for example UNIX) which permit user invocation of parallel processes, and to that presented by operating system languages such as Concurrent Pascal.

In order to control the sequence of tasks, and insure synchronization, the user must indicate, where appropriate, that a task should wait until results generated by another task are ready. The user need not concern himself with the time at which the result will be ready. Scheduling of processors is the responsibility of the system, and if a task is suspended, waiting for results, the user can expect that the processor on which it has been running will be reassigned to another runnable task until the results are ready.

We may think of the user as programming a virtual user machine. The code for this user machine is the high level language implemented at the installation. For definiteness, we may think of an ALGOL-like language. The user machine has an unlimited number of processors. Whenever a new task is called, a processor starts its execution; when a task stops, the processor goes back into the infinite resevoir of processors. In the high level language which represents the user machine, the only constructs required to support parallelism are CALL *P* TASK, and a pair of synchronizing constructs such as WAIT *result* and SIGNAL *result*. Here *result* is the unique name for the data which has been generated by the task. When a task needs a *result*, the programmer uses a WAIT statement in the code of that task, at the point where the *result* is required. This will cause the task to suspend until the *result* is ready. Whenever a task generates a *result* needed by another task, the programmer uses SIGNAL *result* which trahsmits the pointer *result* and restarts the waiting task. The CALL *P* TASK statement (which we have borrowed from PL 1) simply indicates that the code *P* should be run in parallel with the rest of the program. As seen by the user, this causes a processor to start executing *P* immediately.

In the user machine, the constructs CALL TASK, WAIT and SIGNAL, cause tasks to run and to stop at various times. Between execution of one of these statements, the tasks run completely asynchrononously. However, no task starts until it is called, and every task waits for the specific results (or synchronizing signal) which it requires. Thus the execution of tasks is rigidly controlled by the statements in the language which support parallelism. Taken together, these statements, as they appear in any particular program, constitute a schedule for segments of executable code. Another way of expressing the same idea is that the parallelism statements in any program construct a precedence graph, which governs the order of execution of code segments. This schedule, or precedence graph arises naturally from the algorithm; and provides the (partial) order for the execution of tasks in CHoPP.

In CHoPP, the operations of WAIT and SIGNAL, as well as some other synchronizing monitors, are implemented in the hardware and firmware of the processors. The execution time of these primitive then becomes that of a typical single instruction, and need not concern us here. However, since we are interested in implementing languages which support recursion, and in systems which incorporate virtual memory, the translation of symbolic addresses, such as result must be accomplished at run time. This translation, which is a function of the operating system and language support software in sequential machines, is a function of the node kernel in CHoPP, and will be described further on.

In normal programming for CHoPP, many more tasks are invoked than there are real processors to execute them. The user therefore knows that in the real machine the execution of a task or task activation will normally be deferred for some time, until a processor is available to execute it. But for CHoPP he can expect that 1) All but a negligible number of processors will be busy, whenever tasks to execute are available 2) The overhead associated with a task call will be about the same as that of a subroutine call in a sequential machine 3) No time will be lost in moving programs, data or results from one processor to another.

We shall describe the three functions of the node kernel which make possible this kind of performance. The first of these is processor allocation. When a task running in one of the processors executes a CALL TASK statement, a processor must be assigned to the new task. Because many tasks are running in parallel, many CALL TASK statements may be executed concurrently. The key to efficient execution is that the activities required to generate the new tasks and assign the processors be carried out fully in parallel, without central programs or central tables. The second function is that of **memory management**. New tasks will, in general, require memory space in which to run. This space must be assigned from a general pool. When space is released, it must be returned to the pool. But, as before, the management of memory space must be carried out fully in parallel. Otherwise, creation and deletion of tasks would depend on sequential mechanism. Such a mechanism will have of course, some limited capability. And when many tasks request its service, the resulting bottleneck may stop the whole machine. The third function is that of **reference resolution.** We have described the transmission of information between tasks, using synchronizing constructs (e.g. WAIT and SIGNAL) which are hardware supported. The data or results which are transmitted by these statements are referenced symbolically by this programmer in simple languages, like FORTRAN. The symbolic references are translated to machine addresses at compile time. But in the general case which the CHoPP ooperating system supports this is not possible. Tasks are created at run time, often as a result of computation. Moreover, the language which CHoPP supports will permit recursion, as the most natural and efficient way of creating new tasks. Under these circumstances (as is well known) the symbolic references in the program must be translated at run time, into machine addresses. Again, this activity must be performed in parallel, by many processors, without reference to central tables.

THE NODE KERNEL

The node kernel is software that performs the operating system functions, such as allocating tasks to processes, allocating memory to tasks, and mediating intertask communication. In a parallel computer, these functions cannot be accomplished by a single processor as this would require all other processors to queue up to obtain services, leaving them idle almost all the time. Each operating system function must be distributed among all the processors, so each processor must have its own kernel. In CHoPP, the kernels that run at each processor are identical. Each kernel behaves as an autonomous program and no processor kernel assumes a master or control role. All node kernels may be run in parallel and there is no hierarchical structural relationship between them. We will show how some of the operating system functions can be carried out by a large number of node kernels running in parallel.

Task Allocation

The first function that we will consider is the allocation of tasks to processors. In CHoPP we run any task at any node since the applications programmer cannot know ahead of time what processors will be available. Further, in a machine designed to use programming languages that support recursion, the number of tasks to be performed is, in general, data dependent, hence the applications programmer does not know that number of tasks to be assigned, let alone to which processor to assign them. This function must be carried out by the node kernels at run time. A task may be created by either of two methods, as a job that has been entered into the system at a node, or spawned by an existing task already at a node. In either case the node where the task is created is not necessarily the node where the task will eventually be run.

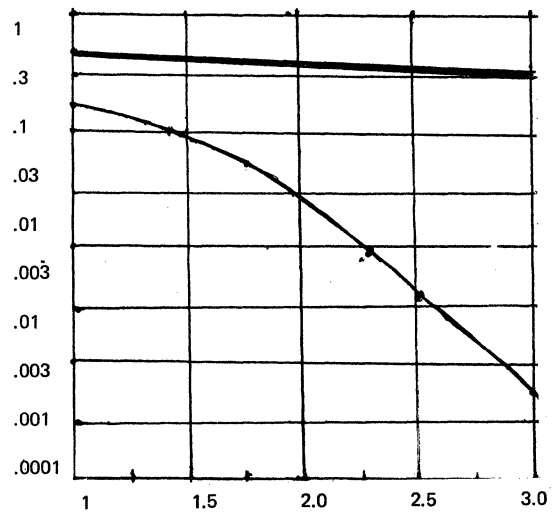
In sequential machines that support multi-tasking, the operating system determines the assignment of processors to tasks in the following way (this description closely follows Denning [18]. A task manager, part of the operating system, maniputlates two queues and a task list. .The task list is merely the collection of all activation records (called "state-words" by Denning). The activation record contains the task's unique index number, the location in memory associated with the task, the instruction pointer, and registers. It thus consists of all information necessary to initiate the execution of a task. The two queues maintained by the task manager are the queue of ready tasks and the queue of tasks waiting for some event to occur. Entries in each queue consist of pointers to appropriate activation records. Whenever the running task terminates, a new task is taken from the ready queue by the task manager, which initiates its running. When a running task is blocked, waiting for an event to occur, the task manager removes it from the processor, places it in the queue of tasks that are blocked, and initiates execution of a task from its ready queue. When an event takes place which unblocks a blocked task, the task manager puts this task in the ready queue.

This is a description of a multitask, single processor system. Each node of CHoPP will be run in this manner. When a CALL TASK statement is executed by a program running at, a node, a trap to the node kernel is generated in the hardware. The response of the node kernel is to construct an activation record and place the pointer to this activation record in its ready queue. For high utilization of CHoPP, a mechanism is needed to assure that no processor is ever idle. The only circumstance under which a processor may be idle is if its ready queue is empty when the task it is running terminates or suspends. To eliminate the possibility of any node having an empty ready queue while other nodes have ready queues with more than one task waiting, the node kernel must equalize the length of its ready queue with the ready queues of the other nodes in the machine. We call this mechanism queue balancing, which is accomplished as follows:

1. When a processor creates a task, it selects at random another processor and assigns this task to it. The receiving node kernel places the task in its ready queue.

2. Whenever the number of runnable tasks in the queue at a node either increases as a result of step 1 above, or decreases as a result of a task leaving the queue to be run, the node kernel selects another node at random to perform balancing. That is, enough runnable tasks are sent from the node with the longer queue to other node to equalize their lengths.

The performance of the algorithm just described is measured by the expected number of processors whose ready queues are empty. These processors may not be idle since they may still be running a task; therefore this measurement provides a lower bound on the efficiency of the algorithm. A computer simulation was carried out to determine, for a 256 node CHoPP the expected number of empty ready queues. The results of this simulation are shown in Fig 2., where the expected fraction of empty ready queues is plotted against mean queue lengths, s, The upper curve shows application of step 1 only of the algorithm. The lower curves show the application of step 2. These results indicate that the algorithm is entirely adequate whenever the number of tasks exceeds the number of processors by a reasonable factor.



As CHoPP is presently conceived, no further improvement on the algorithm is necessary, from the standpoint of efficiency. For the record, we observe that the following modification (which we were considering before the results of Fig. 2 were available) will further decrease the number of idle processors. Assume an otherwise idle node will periodically select another node at random and balance its queue with it. The otherwise idle node will continue this procedure until it obtains one or more tasks to run. This will have the effect of further reducing the mean queue length for 100% utilization.

It should be pointed out that the activation record itself is kept in the memory. When a runnable task is passed from one node kernel's queue to another, only a pointer to the activation record is passed, not the record itself.

Memory Management

The conventional structure for memory management in system design has been to maintain a central list of page frames for available storage. It is not feasible in CHoPP, (where memory is managed by the kernel in each processor), to have multiple memory managers updating a central table simultaneously because sequential access produces unacceptable bottlenecks. Just as in the case of task assignment it is necessary to devise techniques to distribute not only the management of the list but the list, itself, equally over all nodes.

The conventional structure for memory management in system design has been to maintain a central list of page frames for available storage. It is not feasible in CHoPP, where memory managers updating a central table simultaneously because sequential access produces unacceptable bottlenecks. Just as in the case of task assignment it is necessary to devise techniques to distribute not only the management of the list but the list, itself, equally over all nodes.

When a task is created, it will require memory space assignment. This will be controlled by the memory manager at the node kernel where the execution of the task begins. Recall that memory in the CHoPP system is a single address space memory equally accessable by every node. All nodes can concurrently access every word of memory. The CHoPP memory is organized in page frames. The size of the page frame will be considerably smaller than that for an IBM system. The reason for this is that the expected segment size is small, perhaps 32 words, as opposed to 16K in IBM systems. This kind of segment size has been experienced in Burroughs Machines and Multics [19]. For a discussion of the relation between segment size and page frame size, see Brinch Hansen [20].

Pointers to page frames are transferred from processor to processor during memory assignment. Initially, all nodes have the same number of page frame pointers. When a task is initiated, the memory it needs will be obtained from the list of available page frames at the initiating node. It is desirable to maintain equal numbers of page frames at every node as long as memory is available in the machine. If any node discovers it has too little memory to meet the demands of its running tasks, then it must invoke some overflow protection mechanism or virtual memory. However, these mechanisms should not be triggered unless there is, in fact, a global lack of memory.

Hence, the two functions of the memory manager are the maintenance of even length lists of available page frames, and the detection of global memory overflow. The principle on which the memory manager controlls its list of page frames is analogous to the methods used for queue balancing by the task manager. When a task releases page frames, these are added to the local list of available page frames at the node. When a task requests page frames, the request is filled, if possible, from the local node's list of available page frames. If insufficient page frames are available to satisfy any request, the memory manager selects another node at random, and acquires page frames, repeating this process as often as necessary to fill the requirement.
In order to achieve the goal of maintaining about the same number of available page frames at each node, a balancing operation is used. The memory manager in each node maintains a quantity, the page frame estimator, (PFE) which, at all times, is an estimate of the average number of page frames in all nodes. Whenever the number of available page frames at a node changes, for any reason, the memory manager compares the number of available page frames, with the PFE. If the difference between these two quantities exceeds a preset *initiation limit* (which may depend on the PFE), a balancing operation takes place. This balancing operation consists of selecting, at random, another node, and equalizing the number of page frames in the two nodes. The balancing operation is repeated until the difference between the number of pages in the node and the PFE is within a preset acceptance limit (which may

also depend on the PFE) Note that there are two preset values involved, namely the initiation limit, which determines the point at which the balancing operation is initiated and the acceptance limit, which causes the balancing operation to stop. By adjusting the relative size of these limits, the performance of the system may be modified.

Each time that a memory manager acquires page frames from another node, or balances with it, the length of the list of available page frames in the other node is noted. The list lengths thus accumulated constitute a random sample of the average list length in the whole machine. Therefore, the average of these measurements constitutes a statistic which is an estimator of this average list length. The average of the measurements of the length of lists in other nodes is the PFE. (Of course, each memory manager maintains its own PFE). By adjusting the initiation limit and the acceptance limit, the designer can assure that the lists throughout the machine are equalized, to within any previously specified tolerance. The mechanism just described for determining the PFE is a sequential sampling technique for estimating the mean of a distribution, and as such it requires a minimum number of balancing operations to achieve a desired tolerance, with a fixed (predetermined) confidence limit.

In any case, the local PFE provides at each node a reliable estimator for the amount of available memory in every other node, and therefore in the whole machine. Note that this estimate has been derived without any central structures which might produce a bottleneck, and essentially as a bi-product of the basic memory allocation structure. When the total amount of memory remaining in the machine has decreased below some preset level, the mechanism described above will produce an excessive number of search operations whenever a node requires memory. At this level, the memory is effectively full. When the PFE is dropped below this predetermined lower limit, at any node, the overflow mechanisms of the machine should be invoked. In this way the second function of the memory manager has been implemented.

Reference Resolution: Distribution of Central Tables

As we have seen previously the central tables required by an operating system must be handled differently in CHoPP. The two techniques we have seen for task and memory allocation are not, however sufficient for all such tables. We now give a third technique which is useful for a variety of purposes. To describe this structure and show how it is used, let us use the example of the table containing buffer information used for intertask communication. In a sequential processor, when a task needs information about a buffer, it sends its request to the operating system, which searches for the buffer name in a central table. Upon locating the name, the operating system retrieves the requested information and sends it to the task.

In CHoPP, the buffer name for each buffer is hashed. The resulting hash is interpreted as the address of a node. Each node of CHoPP will maintain the portion of the table that is hashed to it. Thus, instead of one large central table, CHoPP will have one small table at each node and each buffer name will occur in exactly one of these small tables. When a task needs information on a given buffer, it sends its request to its local node kernel. This node kernel hashes the buffer name to find the address of the node where the requested information will be found. A message is sent to the kernel at the node to obtain the information. Upon receiving that message, the information is obtained and sent back to the task's local node kernel, which then relays it to the task.

Any table can be distributed among the nodes of CHoPP by hashing the search key. The number of accesses to any table at any time is never greater than the number of nodes. Hence, if the hash gives a reasonably uniform distribution of node addresses from a table's search key, then the queue of requests at any node will never be unacceptably long.

Discussion and Conclusions

The hardware basis of CHoPP provides a new memory/processor relationship in which large numbers of processors can concurrently access a large common high speed memory, without incurring penalties due to conflicting references. This permits an unprecedented level (for an MIMD machine) of intertask communication, and facilitates rapid context switching.

In this paper, three important aspects of the ChoPP software support have been discussed. The theme that runs through the operating system design is decentralization of control functions and of tables. In each of the examples presented, the tables and control functions have been distributed in such a way that every node processor shares equally in the operating system task. Moreover all functions are performed autonomously yet cooperation is achieved. By analogy with the tessalated automata of Von Neumann, we call this system "self-organizing".

Thus the focus of our research on operating system techniques is decentralization of function. But, at the same time, it becomes clear that a sophisticated operating system (the node kernel) is required, and that it will bring with it a number of "overhead" functions. We would maintain that these functions are an inevitable concomitant of the control of parallel tasks. Our goal is not to eliminate these functions, for we believe that such an attempt would be futile but rather to find the means of providing them in such a way that they do not interfere with processing. This is accomplished when there are no shared control mechanisms, whose limited capacity can cause resources to be idle, while waiting for some service function. We expect that, within tolerable limits, every processor in CHoPP will always be kept busy. We also expect that some large fraction of the time of each processor (perhaps 30-50%, as in some existing sequential machines) will be spent in overhead functions.

This philosophy also conditions our CHoPP hardware design. In a parallel processor, hardware support (at the instruction level) of task generation and synchronizing primatives are just as important as branch instructions in a sequential machine.

The achievement of conflict free parallel access memory seems to require complex and elaborate interconnection circuits. All of these factors increase the cost of the machine, so that a CHoPP configuration of N processors, will cost more (we now estimate by about 50%) than N separate processors of the same general capability. This is the price paid for the speed up execution of algorithms in a general purpose context.

This increased hardware complexity, dedicated to the support of parallelism, provides a benefit in reducing the running time of the operating system functions described in this paper. The principle factor in node kernel overhead is the execution, at each node, of a conventional multitasking system on behalf of the tasks managed by the local node kernel Primatives specifically oriented toward improving the efficiency of execution of parallelism will make the node kernel functions, including multitasking, significantly more efficient.

We end by a mention of the origins of some of the concepts in this paper. In one or another form, these concepts incorporate techniques for distributing central tables. The germ of this idea goes back at least as far as the original paper on C.MMP [13], where it is pointed out that dividing global tables into smaller tables will improve the efficiency of parallel processors. The node kernel programs maintain tables, and assure synchronization by permitting only a single request to be active at any time. Such programs are *monitors* as described by Hoare [21] and Brinch Hansen [20].

It will be noted that some aspects of the node kernel which were described in a previous paper differs from the present description, which we consider an improvement.

REFERENCES

1. H. Sullivan, T.R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I" 4th Annual Symposium on Computer Architecture Proceedings, March 23-25, 1977.

2. H. Sullivan, T. R. Bashkow, D. Klappholz, "A Large Scale, Homogeneous Fully Distributed Parallel Machine, II" 4th Annual Symposium on Computer Architecture Proceedings, March 23-25, 1977.

3. H. Sullivan, T. R. Bashkow, "Parameters of CHoPP", These Proceedings.

4. H. Sullivan, T.R. Bashkow, D. Klappholz, "High Level Lanugage Constructs in a Self-Organizing Processor", These Proceedings.

5. J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing", Computer Surveys, Vol. 5, No. 1, March 1973, p. 31.

6. D. Heller, "A Survey of Algorithms In Numerical Linear Algebra", Technical Report, Department of Computer Science, Carnegie Mellon University, Feb., 1976.

7. D. Kuck, *et. al.* "Measurements of Parallelism in Ordinary Fortran Programs" Proceedings of the 1973 Sagamore Computer Conference on Parallel Processing, Aug. 22-24.

8. D. Kuck. "Parallel Processing Architecture, A Survey", Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, August 19-22.

9. P. R. Reddy, *et. al.*, "Hearsay-I Speech Understanding System" An Example of the Recognition Process", IEEE Transactions on Computers, Vol. C-25, No. 4, April, 1976, p. 422.

10. P. Enslow, Editor, "Multiprocessors and Parallel Processing", Wiley, 1974.

11. L.C. Widdoes, "The Minerva Multi-Microprocessor", Proceedings 3rd Annual Conference on Computer Architecture, Jan. 19-21, 1976.

12. R. J. Swan, S. H. Fuller, D. P. Siewiorek, "The Structure and Architecture of Cm*: A Modular, Multi-microprocessor" Computer Science Research Review, 1975-76, Dept. of Computer Science, Carnegie Mellon University, p. 25.

13. W. A. Wulf, C. G. Bell "C. mmp - A Multi-Mini Processor", Fall Joint Computer Conference, 1972, p. 765.

14. P. Enslow, "Multi-processor Architecture, A Survey", Proceedings 1975 Sagamore Computer Conference On Parallel Processing, Aug. 19-22.

15. E. Cohen, *et. al.* "HYDRA, Basic Kernel Reference Manual", Dept. of Computer Science, Carnegie-Mellon University, No. 4, 1976.

16. H. Sullivan, T. R. Bashkow, D. Klappholz, L. Cohn " CHoPP, An Overview. 1n Preparation.

17. "Amdahl Warns Hopes For DDP For Beyond What's Available", Computerworld, Vol. XI, No. 25, June 20, 1977.

18. P. Denning, "Fault Tolerant Operating Systems," Computer Survey, Vol. 8, No. 4, Dec. 1976, p. 359.

19. A. Tanenbaum, "Structured Computer Organization", Prentice-Hall, 1976.

20. P. Brinch-Hansen, "Operating System Principles", Prentice-Hall, 1973.

21. C.A.R. Hoare "Monitors, An Operating System Concept", CACM, Vol. 17, No. 10, Oct. 1974.

22. R. McGill, J. Steinhoff, "A Multiprocessor Approach To Numerical Analysis, An Application To Gaming Problems", 3rd Annual Symp. On Computer Architecture Proceedings, 1976.

HIGH LEVEL LANGUAGE CONSTRUCTS IN A SELF-ORGANIZING PARALLEL PROCESSOR

Herbert Sullivan, Sullivan Associates, 200 West 79th Street
T.R. Bashkow, D. Klappholz
Dept. of Electrical Engineering and Computer Science
Columbia, University, New York, New York

SUMMARY

CHoPP [1,2] is an architecture for MIMD parallel processors intended to support programs which consist of many short tasks. User written programs on hundreds to thousands of processors will typically run for less than one hundred instructions, and then be suspended, awaiting some results generated by another task or be deleted and replaced. Although the vast majority of parallel algorithms described in the literature [3] operate in this manner, CHoPP is the first MIMD architecture oriented toward such a high degree of task switching and task interaction.

The basic framework for an appropriate language is generally understood. Consider an ALGOL-like language with all the constructs necessary for sequential programming. To this we add CALL *P* TASK (borrowed from PL/I) which is used to initiate the parallel execution of a procedure *P*. Since tasks run asynchronously in CHoPP (as in any multitask machine), we need constructs for coordinating their activities. Suppose a task *X* needs a *result* generated by a task Y. Then, at the point where the *result* is needed, the code for *X* will contain the statement WAIT *result*. At the point just after the *result* has been generated, the code for Y will contain the statement SIGNAL *result*.

The WAIT statement causes *X* to suspend execution until Y executes the SIGNAL statement; thus the activities of X and Y are synchronized. These three constructs CALL TASK, WAIT, and SIGNAL, when used in conjunction with the control statements in the language, are adequate to express parallelism. (Of course, in a practical language, a richer vocabulary would be provided). Adequate rules for the control of access to variables are these: private variables may be used freely inside a task; shared variables must always be guarded by a semaphore, or, more generally, accessed through a monitor [4,5].

A program for creating large numbers of tasks may be written by placing CALL (*P* I) TASK inside a DO loop where I is the index of the loop. This will cause a single processor to sequentially create the tasks. In CHoPP since task creation is a major activity, many processors must concurrently create tasks; otherwise, many processors would be idle much of the time waiting for a new task, and the purpose of parallelism would be vitiated. An algorithm in which a task creates two or more tasks, each of which in turn create two or more tasks, etc., will implement the parallel creation of tasks, and create N tasks in order log N time. The natural program for this algorithm uses recursion. It consists simply of a procedure *P* which contains two or more CALL *P* TASK statements. Note that *P* is calling itself as a parallel task; of course, each successive activation of *P* will have different parameters. This algorithm generates a tree of tasks, which we call the "spawning tree". Note the unexpected role of recursion in the parallel program. Instead of being an attractive but inefficient feature, as in sequential languages, it is the natural way of achieving efficient task generation. The convenience of recursion comes as an added benefit.

Every program in CHoPP starts as a single task in one processor and spreads through the machine using the mechanism just described. This does away with structures found in many previous multiprocessors, namely, hardware for broadcasting tasks, a "host" machine which generates the broadcasts, and the sequential scheduling programs that run in host machines.

Run time creation of tasks is not only required for efficiency, as just explained, but also for the execution of many kinds of algorithms in which it is not know, prior to execution, which parallel tasks will be needed or, often, even how many tasks will be generated. Transmission of data from one task to another does not, necessarily follow the spawning tree; in other words, communication takes place between tasks which are not necessarily related as parent/child by task generation. How is this communication to be expressed, in the parallel language?

The natural way to express communication between tasks is by using a common name for each shared variable. This is consistant with usual language conventions for communicating data between various parts of sequential programs. To support recursion in the sequential case, it is necessary to resolve name references, at run time, for each activation of a subroutine. This is implemented by consulting a stack. To support the more general type of communication between parallel task activations, we suggest that it is necessary to permit run time computation of names. When the same name for some variable is computed by two tasks, that variable value may then be transmitted between them. (Synchronization, as described above, must also be established.) The functions necessary to match computed names, and supply the required pointers, are similar to those implemented in virtual memory systems for resolving page references. For CHoPP, a mechanization of such functions has been devised which employs no central tables or central control mechanisms. [7]

REFERENCES

1. H. Sullivan, T.R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I" 4th Annual Symposium on Computer Architecture Proceedings, March 23-25, 1977.
2. H. Sullivan, T.R. Bashkow, D. Klappholz, "A large Scale, Homogeneous Fully Distributed Parallel Machine, II" 4th Annual Symposium on Computer Architecture Proceedings, March 23-25, 1977.
3. D. Kuck, "Parallel Processing Architecture, A Survey", Proceedings of 1975 Sagamore Computer Conference on Parallel Processing, August 19-22.
4. C.A.R. Hoare, "Monitors, An Operating System Concept", CACM, Vol.17, no. 10, Oct 1974.
5. P. Brinch-Hansen, "Operating System Principles", Prentice Hall, 1973.
6. Bobrow, D., Wegbreit, B., "A Model and Stack Implementation of Multiple Environments" CACM, Vol. 16, No. 10, pages 951-602. 1973.
7. H. Sullivan, T.R. Bashkow, D. Klappholz, L. Cohn, "The Node Kernel: Resource Management in a Self Organizing Parallel Processor", These Proceedings.

PARAMETERS OF CHoPP

Herbert Sullivan, 200 West 79th Street
T.R. Bashkow, Dept. of Electrical Engineering and Computer Science, Columbia University
New York, New York

SUMMARY

The CHoPP architecture [1] is a radical departure from previous MIMD machines, in that it is intended to support parallel tasks of extremely short duration. CHoPP is therefore designed to switch tasks, reassign processors, etc. in a few microseconds. Functions which heretofore have been regarded as fundamentally global in nature, such as processor scheduling and memory management, are accomplished without central control mechanisms or central tables [2]. To support this kind of activity the architecture of CHoPP employs many techniques reminiscent of SIMD architectures. Like conventional modern large scale computers, and like vector processors, it employs extensive pipelining techniques and a very high bandwith, shared, interleaved memory.

CHoPP hardware implementation embodies three basic concepts:

1) CHoPP consists physically of N identical nodes, each of which contains a processor, a memory bank, a switch element.
2) CHoPP nodes are connected by a high capacity multistage network in the form of a binary k cube. Each node is at the corner of this k cube and there must be $2^k = N$ nodes. Each node therefore has a unique address k bits long.
3) All memory banks are interleaved to form a single main memory which can be accessed by any node. The maximum distance from any processor to any memory bank is k, so that for a 64K processor, for example, this maximum distance is 16.

In this machine a memory access may be initiated by each processor every machine cycle. Memory accesses are accomplished in the following way. The processor assembles a packet giving its own address (source), the d esired memory address (destination node memory bank plus displacement within the bank.) and an operation (fetch instruction, read data, write data, etc.). The message is then transmitted by relaying from node-to-node until the memory is reached. The memory access is made and a return packet is sent to the requesting processor. Thus in the worst case there is a delay of 2k communication steps before the initiating processor completes the memory reference operation. In addition, there is the actual memory access time, plus possible queueing delays at intermediate nodes. This total delay may be called the latency time.

In order to ameliorate this effect, each node processor is, in fact, a multi-tasking machine. It sends out packets not for a single task, but for as many tasks as it needs to, in order to do some instruction processing at each machine cycle. We define a machine cycle as the time required to:

- inspect any returning packet
- accomplish the operation (add, subtract, etc.) required by this packet.
- initiate a new packet.

How many tasks must it actively be running in order to keep busy in this fashion? It must be running as many, on the *average*, as the *average* latency time requires. One can think of this as a pipelining technique in which the depth of the pipe is not fixed but which grows to just exactly the size required to satisfy the requirement that each processor is doing some instruction processing at each machine cycle.

The architecture, as just described, has the remarkable property that the CHoPP machine executes a fixed number of instructions per unit time *regardless* of this latency time as will be shown below.

We will now discuss the parameters controlling this performance. Clearly, if I know the machine cycle time and the latency time (average) I know the number of tasks (average) which must be active. However, this assumes that:

- there are always enough new tasks available to be initiated as running tasks.
- the network bandwidth and the memory bandwidth are adequate to support this level of packet transmission.

To see that the instruction execution time is fixed, consider the following simplified examples. Suppose that each task requires 100 machine cycles. (Each instruction requires some small number of machine cycles). After the packet for the first task is initiated, a packet for a second task is initiated, then a third, etc., until a response is received for the 1st task. We can then expect a steady state condition in which a packet is received and initiated on each machine cycle. Suppose the latency time is L microseconds, and the machine cycle is L/10 microseconds. Then we will initiate L/(L/10) = 10 tasks which will run to completion in about 100 L microseconds.

If we somehow reduce the latency time to L/2 microseconds, we will then initiate (L/2)/L/10 = 5 tasks which will take 100 (L/2) = 50 L microseconds to run to completion. Therefore we will initiate 5 more tasks which will complete in the next 50 L microseconds. Again we have run 10 tasks in 100 L microseconds. As indicated earlier the speed of the machine is fixed regardless of the latency time.

To see what this means, we will make some estimates in order to see what performance can be expected from CHoPP. To continue with our simplified discussion we assume an old fashioned 2 memory cycles per instruction machine, thus 2 of the machine cycles are required for each instruction. With present day circuitry we can conservatively allow 500 nanoseconds per machine cycle or 1 microsecond per instruction, giving us a 1 MIP machine per node. The latency time, L, consists of 2 components, the actual memory cycle time Tm and the mean communication time delay in the network Tn.

Tm we can reasonably estimate as 500 nanoseconds. Tn is the product of the mean number of nodes transversed, which will be k/2 going toward a memory plus k/2 returning for a total of k, and the mean waiting time at a node $\overline{w}$. Queueing analysis shows that $\overline{w}$ is of the order of one processor machine cycle. Thus we expect the mean latency, L to be 500 nanoseconds (1 + k) and the mean number of active tasks, $\overline{n}$, to be:

$$\frac{500 \text{ nanoseconds } (1+k)}{500 \text{ nanoseconds.}} = (1+k) \text{ tasks}$$

Table 1 shows these parameters for various sizes of machine. It can be seen that both latency and task queue length are reasonable even for the 64K machine because, of course, these values only increase logarithmically with N. Moreover, the introduction of a local cache memory, which reduces memory accesses across the network, or in effect reduces L, can be used to reduce these numbers substantially.

TABLE 1

k	4	7	10	13	16
N	16	128	1024	8K	64K
L	2.5	4	5.5	7	8.5
$\overline{n}$	5	8.	11	14	17

REFERENCES

1) H. Sullivan, T.R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, 1", 4th Annual Symposium on Computer Architecture Proceedings, March 23-25, 1977.
2) H. Sullivan, T.R. Bashkow, D. Klappholz, L. Cohn "The Node Kernel; Resource Management in a Self-Organizing Parallel Processor", These Proceedings.
3) R. N. Noyce, "From Relays to MPU's, "Computer, V 9, December 1976, p. 26.

A RECONFIGURABLE VARISTRUCTURE ARRAY PROCESSOR

G. Jack Lipovski, Anand Tripathi
Department of Electrical Engineering
University of Texas-Austin
Austin, Texas 78712

ABSTRACT

This paper describes a multiprocessing system using conventional microprocessors which are dynamically restructured to get the desired word width and dynamically reconfigured to obtain the desired memory height. The system is space shared so that several tasks concurrently execute in different blocks of the partitioned resources, and communication is provided between and within task blocks. The dynamic reconfiguration and restructuring of the different processors and memory modules and interprocessor and I/O communications are achieved using an interconnection network called an SW-banyan, the cost of which is proportional to n Ln n, where n is the number of modules to be interconnected. A great deal of flexibility and power is available by means of an inexpensive SW-banyan switching network. The user has unprecedented capabilities to configure and structure the machine to fit his problem. When the problem allows it, very high memory word width and thus bandwidth is obtainable, so that memory is more effectively used; but when word width is unwanted, as in string manipulation, it can be efficiently limited. This architecture therefore, seems very well suited to "scientific" processing requirements of the future.

1. Introduction

The cost effectiveness of the microprocessors opens the most exciting and challenging research area of using an assemblage of microprocessors to achieve the capabilities of large machines. Some rather well researched techniques for this include the reconfigurable multiprocessor approach and SIMD or vector approach. The variable structure approach and the techniques for communications between the cooperating processors such as data flow techniques are also under investigation. These will be described in the following paragraphs.

The reconfigurable architecture uses a cross-point switch to connect resources, like memory modules and I/O devices to the processors. These include the RW-400 [13] and the C·mmp [20]. We will use the term "configure" in this sense: to connect resources to the processor to use it more effectively. The SIMD or vector architecture uses several data operators or ALUs commanded by the same controller to execute the same operation on each data contained in each ALU. Early SIMD machines were associative memories because the simplest data operator is the comparator of an associative memory. More recently, the n-bit-sliced microprocessors like the Intel 3000 [8] and AM 2901 appear to make more cost-effective data operators.

A very similar concept to SIMD, but not so well researched, is the varistructure concept. Such an architecture offers the prospect of dynamically coupling the n-bit-sliced microprocessors and their associated memory to get the desired word-width as well as structure the system for vector or array processing applications. We will use the term "structure" in this sense: to modify the apparent structure of the memory to utilize it more effectively. The first reference the varistructure concept appears in a paper by Estrin [7] which suggests a fixed part of the processing unit and a variable part consisting of some registers and functional units. Estrin suggested that the variable part of the machine could be used to expand the word-width. However, he did not suggest any cost-effective way to do this.

More recently other investigators have suggested varistructure architectures in terms of dynamically coupling the n-bit-sliced microprocessors in the system to fit the desired data structures and word-widths [3,9,12,26]. The first attempt to couple such microprocessors, by Lipovski [9], used a fast tree structure, which was difficult to schedule. In the architecture proposed by Okada, Tajima and Mori [12] the processing units are connected to their left and right neighbors. Each processor i has data paths to processors $i\pm1$, $i\pm2$, ... $i\pm2^n$. However this architecture puts the restriction that the processors coupled to make the desired word-width processing unit be physically adjacent to each other. Another varistructure architecture was proposed by Lipovski [10] in which processors communicate with their left and right neighbors but the communication is actually done by a carry-look-ahead-tree-like structure to achieve speed and fail-soft capabilities. Again the processors coupled to work on bytes of a word must be physically adjacent to each other. These architectures are suitable for only a small number of processing elements, or for large numbers of elements if the operation is CPU bound, because their I/O capabilities significantly degrade as the number of processing elements in the array is made large.

In any computing system where a number of independent processors concurrently and separately execute their own tasks (i.e. when the resources are space-shared) the possibility for them to cooperate depends on interprocessor communication. We will use the term "communicate" in this sense: to provide means between

possibly independent processors to coordinate tasks and pass data between them. Generally, such "processors" can be extended to include input/output devices. Communication is fundamentally different from structuring or configuring. Structuring and configuring relate intimately to the fetch-execute cycle where as communication relates indirectly to it (e.g. like input/output operation). Either data or control or both are communicated. One of the early key papers by Conway [5] showed how 'Fork' and 'Join' can specify the way independent processors can be utilized, when available, to solve a problem. Control communication is necessary to coordinate this type of operation. More recently, Dennis' data flow techniques [6] show how data implicitly carries control, so that data communication can be used to coordinate independent processors.

The design of a flexible cost-effective interconnection network for configuring, structuring, and providing communication between processing modules forms the core of an efficient reconfigurable varistructure system of multiple microprocessors. We are encouraged by the discovery that all the three functions can be obtained by the same inexpensive interconnection network. The interconnection network used in the proposed machine is called banyan network [15] and has cost-function proportional to n Ln n as compared to n^2 of a cross-point switch, where n is the number of processors and memory modules to be interconnected. The cost advantage of n Ln n is obtained at the expense of reduced total interconnection capacity. Preliminary measurements indicate that in the order of 10% of the possible interconnection structures are not connectable (i.e. they are blocked) [16]. Therefore these networks are blocking networks [4]. The objective of this paper is to show an architecture that effectively utilizes an SW-banyan for structuring, configuring and providing communication between processing modules.

The next section shows how the resources are interconnected in the SW-banyan. Its objective is to show how simple and modular the hardware is. The following section shows how the fetch execute cycle and interprocessor communication function. It shows how simple is the control of the processors and switch. Software is briefly discussed next. The final section summarizes our results and points to further work.

2. Static Structure of the Architecture

The SW-banyan interconnection structure is defined and its control is explained in the first section. The internal organization of resources--processors, memory and I/O is discussed next. Finally, interconnections to affect structuring, configuring and communication are delineated.

2.1 Interconnection Structure

The interconnection network used in this architecture belongs to the class of banyan networks which were proposed by Goke [16] for partitioning multiprocessor systems. For the sake of completeness some of the important properties of these networks, relevant to the proposed architecture are briefly reviewed here.

A banyan can be defined as a particular type of directed graph. A base in the banyan is defined as a vertex having no arcs incident into it and an apex is any vertex having no arcs incident out from it and all the other vertices are called intermediates. The graphical representation of a banyan is a Hasse diagram of a partial ordering (i.e. an irreflexive, asymetric, intransitive graph) such that there is one and only one path from any base to any apex. Larger banyans can be synthesized from the smaller banyan modules such that the resultant network holds this property. A regular banyan is one in which the number of arcs incident into each vertex and incident out from each vertex are constants called fan-out (F) and spread (S) respectively. Our proposed machine uses the SW-banyan [15] which can be obtained by recursively expanding a cross bar structure. It can also be obtained by replicating a tree. Draw an L-level tree with fan-out F and with root at the top. Then replicate the top (root-ward) branches S times. Then replicate all the structure above the next level of nodes S times. Continue replicating at each lower level of nodes until the bottom node is reached. Note that the resulting SW-banyan is quite regular in three dimensions. See Figure 1(a).

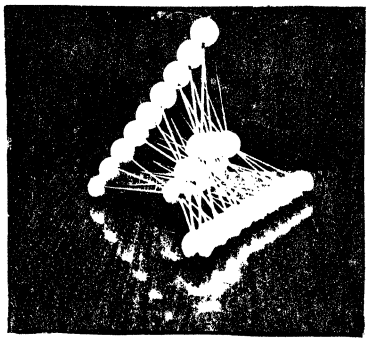


Figure 1(a) - Three dimensional view of an SW-banyan F=S=3, L=2

Interconnection of devices by means of a banyan generally follows this strategy. Devices to be explicitly connected are attached to base nodes only. (In this paper, some devices are attached to apex nodes, but these are implicitly connected by the strategy given below.) The set of the devices is partitioned, and a bus-like interconnection for each block of the partition

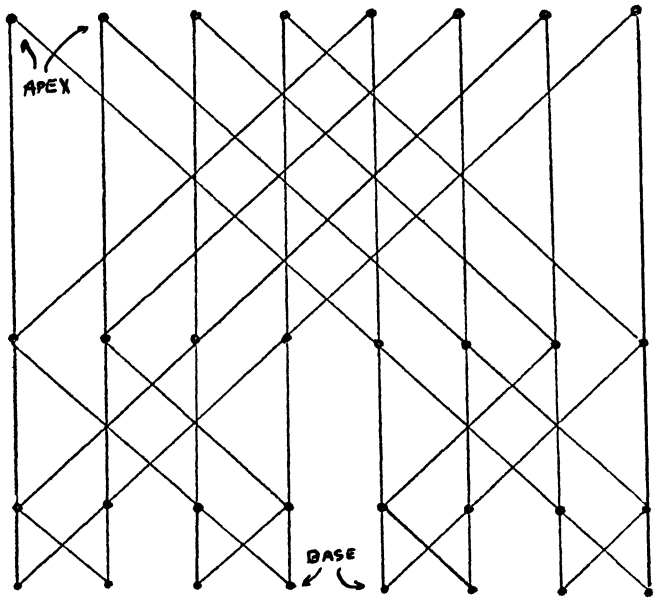


Figure 1(b) - An SW-banyan L=3, S=F=2

is to be set up in the switch. This bus-like interconnection is actually graphically a tree of bidirectional amplifiers such that any device on the leaf of the tree can broadcast data to all the leaves of the tree "instantaneously". Several trees may be set up in an SW-banyan for each block to provide independent bus-like interconnections for them. Setting up a tree having n leaves can be done in one step using four separate control lines, however it is easier to understand using n+2 steps and two control lines. Assume that other trees already utilize some arcs and nodes, and some arcs or nodes are known to be faulty. In the first i steps, i = 1, 2, ... n the ith selected leaf node broadcasts a signal towards the apex, which is blocked if it encounters a used or faulty node or arc. Each node records the resulting signal it gets. An apex node is a potential tree root if it gets signal from each selected leaf of the tree. In step n+1, exactly one potential root is selected by a priority circuit. In step n+2 the selected root broadcasts downwards as selected leaves broadcast upwards; any arc getting a signal from the selected root and any one of the selected leaves becomes connected to form the tree. This algorithm is converted by DeMorgan's law into a one step algorithm.

We choose the SW-banyan because it is capable of implementing all the three types of interconnection strategies -- structuring, configuring and communication -- as we will soon show. The shared bus and shared memory technique characterized by Anderson and Jensen [1] would certainly experience the contention problems and excessive delays which would limit the size of the system. Store and forward switching like Pierce loop [14] and "perfect shuffle" [18] are useful for packet switching but not for bus like operations, so they would not be suitable for structuring and configuring by means of the fetch execute cycle operations in a varistructure processor. The only flexible interconnection structure known before the banyan was the crosspoint switch which is too expensive for large systems. A banyan is flexible as a crosspoint switch and is easier to control in hardware. It makes the best interconnection structure for structuring as it can make bus like interconnections and carry-look-ahead links whose delay is proportional to the log (number of processors). Two banyan structures, the SW and the CC banyans, are known [16]. The SW-banyan switch is chosen for interconnection network because it is easy to expand such a switch into larger banyan switch without rewiring the inner structure.

In this varistructure architecture, n-bit-sliced microprocessors are connected to the apexes, and the memory modules or I/O devices are connected to the bases of the banyan network as shown in Figure 4(a). Each link of the banyan network contains, for instance, 8 data lines, 16 address lines, 4 control lines for search and set-up and three lines for carry-look-ahead functions. Each node has a carry-look-ahead logic circuit fabricated from standard TTL gates as shown in Figure 2 a. The data lines and address lines in a link will be used to connect a tree, which will be used like a data bus or address bus between processor and memory. They can be fabricated from standard integrated circuits like DM7833 quad transceivers [21] or CD4066 [22] analog switches. However a specially designed digital bidirectional amplifier, shown in Figure 2b, [19,25] would be preferred for this purpose for two reasons. It has higher noise immunity as compared to CD4066 analog switch. It does not require control signal to direct data through the tree as compared to DM7833, which requires additional logic to enable the tristate gate in the direction of data flow through the switch.

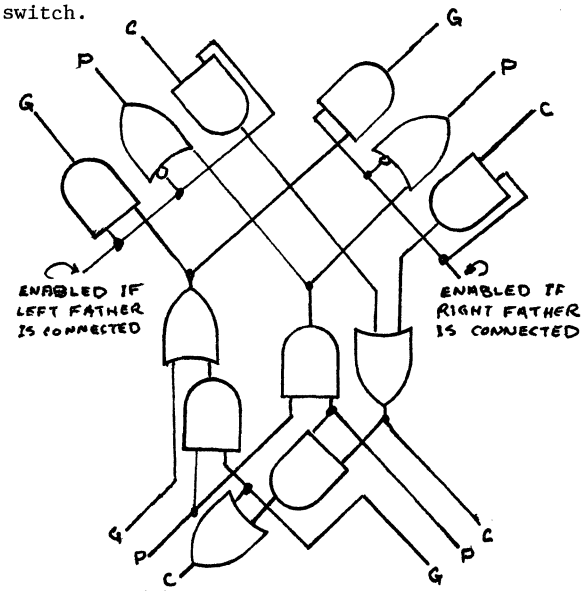
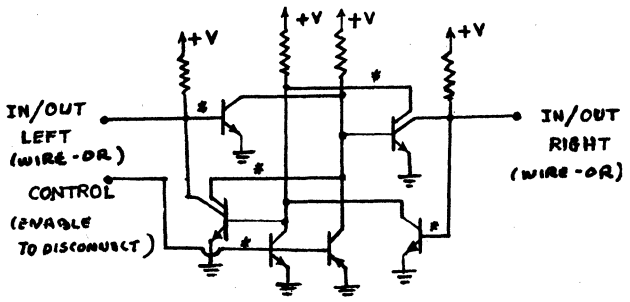


Figure 2(a)Carry-Look-Ahead tree at a node



* Circuitry to prevent current hogging and to match critical delay paths is not shown in this figure. But has to be added where * appears

Figure 2(b)A Bidirectional Amplifier

2.2 Processor, Memory and I/O Modules

We specify the structure of the processors, memory and I/O modules to show what the SW-banyan will connect together. The processor is, by design, a rather conventional microprocessor. Special memory is required so that a physical page of memory can be assigned different page numbers, in order that a collection of memory modules can be freely assigned to store pages of data required by a task. The logic is distributed, so that each physical page has just the required logic to associate different page numbers to it. This technique is essential to both configuring and structuring memory efficiently. As a bonus, the logic is so similar to that required for virtual memory that we find it quite simple to incorporate it. Input/output connections are provided to page data into and out of memory modules at high data rates, and a secondary memory is implemented to store pages efficiently. Since a directory creates attendant problems in a reconfigurable, vari-structure machine and since the memory utilization may well be sparse (i.e. a task may use pages 0 to 5, 20 to 27 and 47) the secondary memory should be intelligent enough to store these efficiently without any external directory. We discuss the processor first, then we describe the memory and secondary memory modules in more detail.

All the processors are connected at the apexes of the network. These processors are conventional byte slice microprocessors as shown in Figure 3(a). It has a ROM for storing the microinstructions, an ALU, a program counter and an instruction register. As in most microprocessors it sends 16 bit addresses to the memory modules and I/O modules, sends or receives 8 bit data bytes to or from them. However, it also provides external accesses to its carry-look-ahead logic (generate, propagate and carry) and a memory cycle state signal (indicating instruction fetch, data recall or data memorize). All the processors in the system are identical and should fit in an LSI chip.

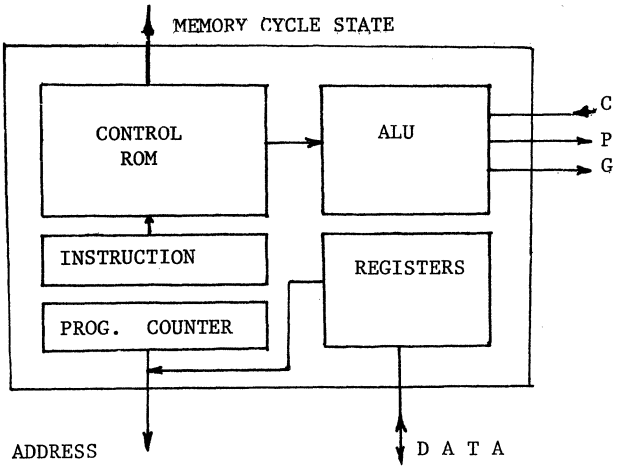


Figure 3(a) - Processor Module

The memory module consists of a RAM, for example (1k x 8 bits) and support logic to implement a virtual memory system as described in [2]. The module contains a six bit page-number register. The higher order six bits of the address are compared with this page register. If the match is successful in a memory module then the lower order ten bits are used to address the RAM in that module. Memory modules are attached to the base nodes of the banyan. A number of memory modules will be connected to a processor by means of a tree formed within the banyan, and each memory module will have a different value in its page register. When a processor sends an address to all the memory modules through the tree, only one of the memory modules connected to this processor has page number register matching with the higher order six bits of the address. See figure 3b.

This memory module can easily be extended to support virtual memory. If none of the modules match the page number then one of the pages has to be swapped out. For this purpose support hardware (such as age-use-counter and dirty bit) can be provided on the virtual memory (VM) module.

Execution of a task requires a set of secondary memory modules. For back up storage a self managing secondary memory SMSM [11] fabricated from charge-coupled devices or magnetic bubble memories is attached to an I/O port. This memory is capable of storing variable length records along with a label, and the record can

be accessed using this label. More significantly, however, the hardware associated with SMSM can search for the record by its label and can delete, input or output data from that record. Sparce pages can be efficiently stored using the label as a page number. However, data described by capabilities and data in stacks can also be stored on SMSM without maintaining a directory for the devices. Some other I/O ports are connected to peripheral devices, such as teletypes, to communicate with the external world, or the controller of the banyan network.

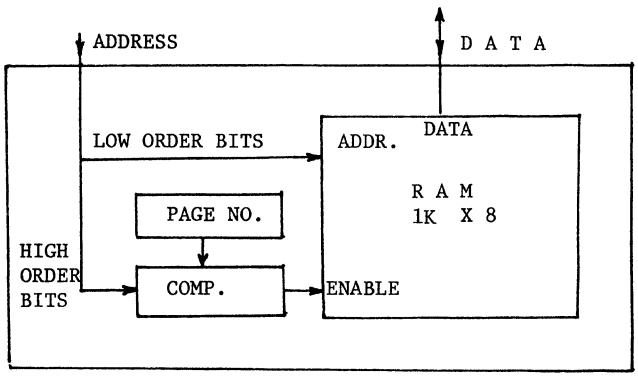


Figure 3(b) - Memory Module

2.3 Instruction and Data Trees

A task execution will generally require some memory modules and I/O devices to be connected to some available processors. A tree structure, called the data tree, is created in the SW banyan network with these memory modules and I/O ports as the leaves and the processor as the root. This is shown in Figure 4(a). This tree is set-up using the four control lines in the manner described in section 2.1 for search and set-up. The processors are indistinguishable; therefore we first choose the memory and I/O ports to be connected and then connect them to any available processor. This way the probability of encountering the blocked paths is reduced.

A programmer may decide to use more than one byte of precision (say p bytes) and possibly he may decide to operate on vector of n elements in SIMD mode. Execution of such a task would require p x n processors to be connected to their respective memory modules using separate data trees. Each separate data tree is set up as in the previous paragraph. In the same banyan now one inverted tree is formed as pointed out by Goke [17], which connects the processors which were the roots first chosen for the data trees. In this inverted tree, the processors are leaves

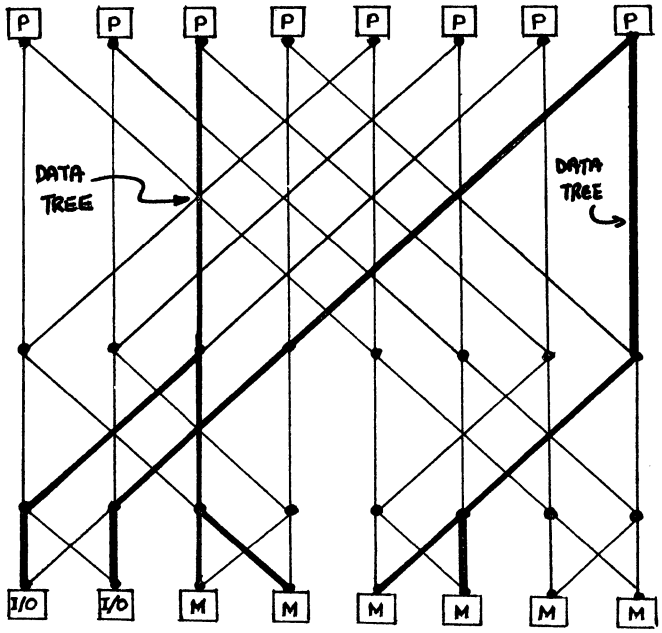


Figure 4(a) - Data trees in the SW-banyan

and an unused memory or I/O module is the root. This tree will serve for instruction transmission and linking the carries as discussed in section 3. The carries are broken by the processors, by forcing propagate and generate to zero in the carry-look-ahead circuits, after every p processors. The carry circuit can be connected so as to either propagate to the left or to the right for left shift/right shift operations or propagate carries for addition. Thus n processing units are formed each of which operates on word of length p x n bytes, and all of them execute the same instruction as transmitted on the instruction tree. The instruction tree is shown in Figure 4(b) and the unfolded instruction and data trees are shown in Figure 4(c). In the dynamic structure, i.e. during every fetch-execute cycle, connection of the instruction trees will be made only during the fetch cycle to get the instruction to each processor.

2.4 Memory Sharing Trees

The data and instruction trees discussed in section 2.3 are used essentially as busses to connect memory and I/O to processor and to connect processors together. That is, the amplifiers in the links of these trees are essentially permanently on. A different tree is now introduced to share memory. Oriented like an instruction tree, the leaves of the memory-sharing tree are processors that are selected to share a page of memory and its root is a memory module. An active chain, from the memory module at the root to a processor at one of the leaves is established from time to time. The amplifier links in the active chain are turned on, and the

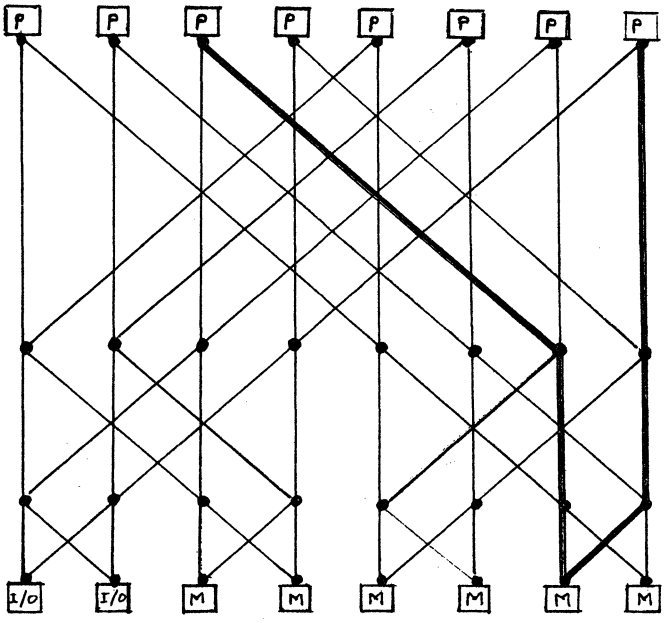


Figure 4(b) - Instruction trees in the SW-banyan

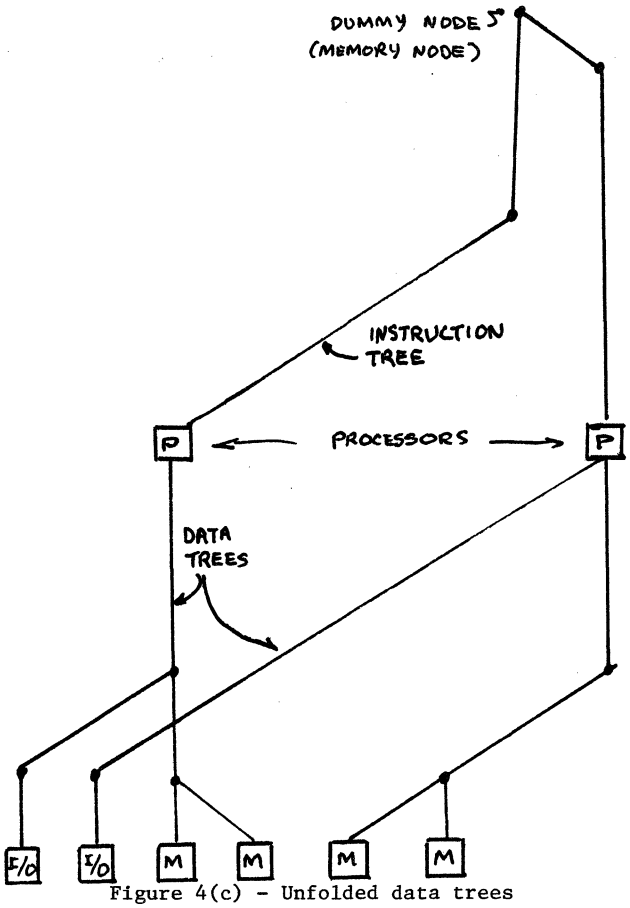


Figure 4(c) - Unfolded data trees and Instruction trees

active chain is effectively appended to the data tree of the processor and the end of the chain. The other links of the memory-sharing tree are not used for data paths, but are available to form active chains when another is to be established. Note that the data trees, even while augmented by the active chains, are mutually exclusive. The fetch-execute cycle will be defined using these mutually exclusive data trees. The machinery to extablish the active chains, an arbiter, is identical to the carry-look-ahead logic used in instruction trees. When an instruction tree is created the machinery acts as a carry-look-ahead generator, and when a memory-sharing tree is formed it acts as an arbiter.

3. Dynamic Aspects of the Architecture

Having presented the basic elements of the architectures we now discuss how they work. Specifically, the scheduler's actions, fetch-execute cycle, and memory-sharing mechanisms are considered.

3.1 Scheduler Mechanisms

The user specifies the desired precision and vector size by means of dimension declarations or it is determined by default. The scheduler will try to set up the process when the required resources are available. Data trees are created one at a time; then an instruction tree is created from the processors selected by the data trees as discussed in section 2.1. Failure to connect the data trees or the instruction trees would abort the process. A single carry linkage is provided in the instruction tree. Some of the processors are flagged to always break the carry linkage by setting generate and propagate to zero and the page numbers are inserted into the virtual memory modules so that each data tree has necessary pages for each byte-slice of data. Program memory is inserted anywhere in the data trees so that each page of the program appears just once in a data tree that is connected to the instruction tree. Finally the memory is loaded from the secondary memory and the process is started.

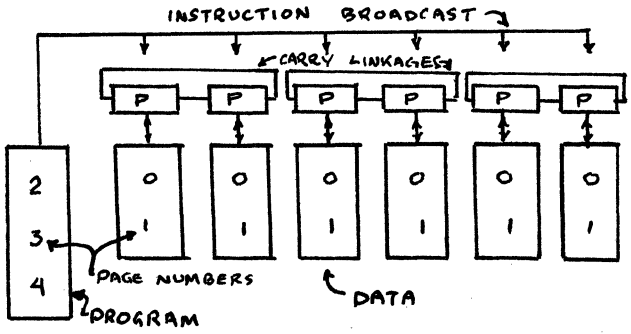


Figure 5(a) - Programmer's View

This is elaborated further with the help of an example. Here the user requires two pages of double precision numbers in a three element vector. The programmer's view of the machine is given in Figure 5(a). Execution of this task requires six processors and fifteen memory modules three of which are used for storing programs. In the machine these modules are connected as shown in Figure 5(b). This example will be continued in the next section.

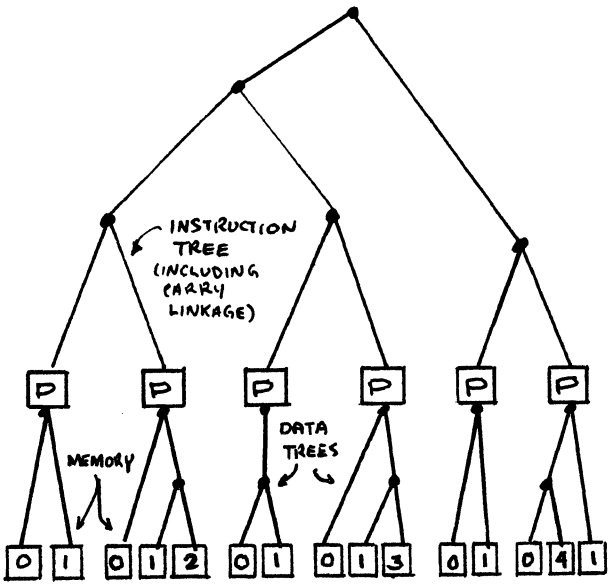


Figure 5(b) - Configuration of switch

3.2 Fetch-Execute Cycle

The fetch-execute cycle operates in terms of data and the instruction trees set up by the scheduler. Consider execution of instruction ADD 7 which is stored in page number 3. The programmer views this machine in terms of Figure 5(a) where the instruction from page 3 is sent to all the processors. We now show how the instruction is fetched from page 3 to all the processors and how data in each processor is independently and concurrently recalled.

In the machine, at the beginning of the fetch cycle, all the processors present the same address to their memory modules, as the program counters in all the processors have identical values. Only one memory module matches the address and pulls out the instructions from its memory which is sent to all the processors through the data and instruction trees. These load the words into their instruction registers, decode and execute the instructions.

In this example the instruction requires word 7, which is on page 0, to be recalled. The programmer views in terms of Figure 5(a) where the multiprecision vector word in page 0 is recalled to the corresponding processors. The address 7 is simultaneously generated by each of the processors and transmitted on their respective data trees. One of the virtual memory modules in each data tree, with page number 0, matches this address and sends back the word on the data tree. These six bytes of data are recalled into six processors via the six disjoint data trees assigned to this task. The instruction tree is disconnected for this cycle.

The processors appear to add the word recalled from memory into their accumulators, as suggested by Figure 5(a). The instruction is executed and the carries propagate through the carry-look-ahead circuit created in the instruction tree. At the word boundaries, identified by flags set, the carries are inhibited by forcing propagation and genrate to zero, e.g. in the processors second, fourth and sixth from the right.

3.3 Shared Memory

When several processor request the shared-memory module in the same cycle then the carry-look-ahead circuit is used as an arbiter to grant only one of the processors access to the shared memory. The priorities can be assigned in a fixed way or can be moved on a round-robin fashion, as we now show.

All the processors except one have propagate equal to 1 and any processor requesting shared memory makes generate equal to 1. The carry output is connected to the carry input to effect "end-around carry".

By moving this propagate equal to zero position dynamically among the processors sharing the memory module, round-robin priority discipline is implemented. If at any processor carry-in = 1 is detected then it means that another processor which is to its right is requesting the shared memory; therefore its request is inhibited. The processor requesting the shared memory and having carry-in = 0 is granted access to the shared memory. This processor sends an address to the shared memory and if the page number matches then the active chain is established between the processor and memory. If a page is unavailable then the processor would wait for a few cycles and try again. If a processor's request is just inhibited because of carry-in = 1 then it will try again in the next cycle. Once access is granted, by means of the active chain, the shared memory appears to be in the data tree of the processor that was granted access. It can address this shared memory in a fetch-execute cycle as defined in section 3.2. When the processor no longer needs to access the shared memory, it releases it so that another processor requesting it can be granted access. It is expected that once a processor is granted access to a shared memory page, it will use it for several tens of cycles before releasing it.

Shared memory provides a mechanism for indivisible operations like test and set, because a processor is expected to have complete control of a shared memory module for several cycles. We propose that control communication use only this mechanism. Processors expecting a control signal will continually check a shared memory module. Unexpected control information might be communicated by writing all software so that it periodically checks a shared memory module. The round-robin arbiter mechanism for accessing shared memory should give each processor its opportunity to access its control signals. However, some form of interrupt may be necessary, to alert a processor to look at its control signals. This problem is now being studied.

4. Preliminary Remarks on Software

Although possibilities abound for using complex, powerful software, some very simple software techniques should be easily implemented to take considerable advantage of this machine. One technique might be to schedule independent four byte wide tasks for all users. Space sharing this machine in this simple way is superior to time sharing a conventional large machine, since the scheduler need not find consecutively numbered pages to store a program (the switch can allocate any collection memory modules to a data tree) and since the operating system need not bother to keep processors busy through swapping programs in and out (the individual partitions can be allowed to be idle since they form only a small fraction of the system resources). Moreover, better utilization of memory and of I/O resources should be feasible, since, resources can be carefully assigned to the processors that need them. Finally, fail-soft operation and memory protection can be easily obtained.

Next, it should be easy to write compilers where the instruction set is fixed but the object processor word width is selectable at compile time. If a program is heavily character string oriented, the entire program can be compiled for a one byte wide processor. A program that uses a lot of 64 bit numbers could be compiled for an eight byte wide processor. Although multi-precision do-loops are not completely eliminated (the program compiled for a one byte wide processor may have to operate on occasional 3 byte numbers) their frequency may be reduced. Standard programming languages can be used. New languages that have vector capabilities (like APL) can also be written to use the SIMD features of this machine. Herein, the scheduler gets relatively fixed requests for processors and other resources, which it can effectively handle. Multiple precision Do-loops and vector Do-loops should be substantially reduced to reduce the overhead in processing large numeric programs in particular.

It should be possible to set up pipelines of otherwise independent partitions of the resources to increase throughput. For instance, a compiler can be partitioned into a lexical analyser, a parser, etc., and these can be put in different partitions of the machine. Memory sharing can be used to pass data and control between these partitions. More generally, some simple forms of data flow programming at the "subroutine level" might be easy to implement. This simple utilization of multiprocessing should be quite effective in this flexible machine.

The techniques above seem to be within the reach of current software technology. More exciting possibilities exist in this machine, however. As microprogrammers control multiple registers and busses in parallel through horizontal microprogramming techniques, they may control complete computing partitions of this machine, some of which can be vector (SIMD) processors, in tightly coupled programs. Partitions might be joined together and broken apart to execute subtasks of a program. For instance, Mori et al [12,24] are considering implementing floating point operations in separate partitions of their machine-one for the exponent and one for the fraction. This may work well in this machine. We are studying the use of a separate partition to analyse descriptors for a partition operating as a vector machine [23]. One advantage of this concept is that the width of the vector machine can be assigned at run time, and that width can be different from what the user requested, yet the resources will be efficiently used. High level languages could be interpreted by similar partitions.

This new kind of programming appears very exciting. By analogy, a composer writes a theme, an orchestrator divides the music for different instruments, and the conductor coordinates the music. We believe that exciting challenges lay ahead for the orchestration of partitions - the translation of algorithms already designed for conventional machines into parallel forms to be executed in different partitions, and the conducting of partitions - the operating system and hardware that keep the partitions synchronized. In addition to loosely coupled multitasking using fork and join, or their equivalent, horizontal microprogramming techniques will be used to tightly couple the partitions. Orchestration may become a new discipline in programming technology.

It is unlikely that all programmers will be aware of orchestration. Rather, carefully orchestrated subroutines will be available to the average programmer. The programmer will simply write programs to call up these subroutines. The machine will be restructured and reconfigured as directed by the subroutines at run time. This kind of complex, powerful software may be able to take full advantage of the machine.

It is not clear which level of advancement can justify the cost of this machine. Since simple identical processors and memory modules are used, this machine offers the possibility at the outset of being more cost-effective than large contemporary machines because it can take advantage of LSI technology. However, the switch, though

inexpensive in comparison to other switches, is still costly. It is our contention that by simplifying scheduling, reducing software overhead, providing pipelining and data flow control, we can obtain significant advantages over current designs. Moreover, it is our hope that orchestration will provide unprecedented power to this machine.

5. Conclusions

A computer for scientific applications should parallel inexpensive microprocessors to cost-effectively achieve the throughput necessary to solve massive problems such as weather prediction, yet a number of small problems should be able to cost effectively space-share the assemblage of microcomputers. Reconfigurability, variable structure and memory sharing are emminently suited to this task. Although a switch, even a banyan with cost n.ln n is a rather expensive item and should not be used indiscriminantly, we have shown that the same banyan can provide reconfigurability, variable structure and memory sharing, with little effort. The user has the unprecedented freedom in specifying the apparent width and height of this memory, and can effectively manipulate one byte wide character strings or n element vectors whose elements have precision p. Moreover, the scheduler has the capability to interconnect any available processors, memories and I/O devices without restrictions, such as using contiguous cells in a chain, and very high input-output bandwidth is attainable by connecting an I/O device to each data tree. We confidently brandish our claim that this architecture is the best yet described for scientific applications.

6. Acknowledgements

The authors gratefully acknowledge extended discussions with Jim Browne, Lyndon Taylor, and Charlie Hoch, since this paper was extracted from a proposal to build such a machine, and we have all contributed some ideas to the proposal. The authors are also deeply indebted to Rodney Goke and Ryoichi Mori for continued dialogue on parallel machines.

REFERENCES

1. Anderson, G.A., Jensen, E.D., "Computer Interconnection Structures: Taxonomy, Characteristics and Examples," Computing Surveys, Vol. 7, No. 4, December 1975, pp. 197-213.

2. Anderson, J.A., Lipovski, G.J., "A Virtual Memory for Microprocessors," The Second Symposium on Computer Architecture, 1975, pp. 80-84.

3. Arnold, R.G., Page, E.W., "A Hierarchical Restructurable Multi-Microprocessor Architecture," The 3rd Annual Symposium on Computer Architecture, 1976, pp. 40-45.

4. Benes, V.E., "Optimal Rearrangeable Multistage Connecting Networks," Bell Systems Technical Journal, July 1964, pp. 1641-1656.

5. Conway, Melvin E., "A Multiprocessor System Design," Proceedings FJCC, 1963, pp. 140-146.

6. Dennis, J.B., Misunas, D.P., "A Preliminary Architecture for a Basic Data Flow Processor," Proceedings of the Second Annual Symposium on Computer Architecture, pp. 126-132.

7. Estrin, G., "Organization of Computer System--The Fixed-Plus-Variable Structure Computer," Proceedings of WJCC, 1960, pp. 33-40.

8. Intel Application Notes, Intel 3000, Intel Corporation.

9. Lipovski, G.J., "A Varistructure Failsoft Cellular Computer," Proceedings of the First Annual Symposium on Computer Architecture, 1973, pp. 161-170.

10. Lipovski, G.J., "On a Varistructured Array of Microprocessors," IEEE Trans. on Computers, February 1977, pp. 125-138.

11. De Martinis, Manlio, Lipovski, G.J., Su, S.Y.W., Watson, J.K., "Self Managing Secondary Memory System," The 3rd Annual Symposium on Computer Architecture, 1976, pp. 186-194.

12. Okada, Y., Tajima, M., Mori, R., "A Novel Multiprocessor Array," Second Symposium on Microarchitecture, Euromicro, 1976, North Holland Publishing Company, pp. 83-90.

13. Porter, R.E., "The RW-400--A New Polymorphic Data System," Datamation, Vol. 6, 1960, pp. 8-14.

14. Pierce, J.R., "How Far Can Data Loops Go," IEEE Trans. on Communications, June 1972, pp. 527-530.

15. Goke, R., Lipovski, G.J., "Banyan Networks for Partitioning on Multiprocessor Systems," Proceedings of the First Annual Symposium on Computer Architecture, 1973, pp. 21-30.

16. Goke, L.R., "Connecting Networks for Partitioning Polymorphic Systems," Doctoral dissertation, Dept. of Electrical Engineering, University of Florida, 1976.

17. Goke, L.R., personal communication.

18. Stone, H.S., "Parallel Processing with Perfect Shuffle," IEEE Trans. on Computers, Vol. C-20, No. 2, February 1971, pp. 153-161.

19. Vice, W.E., Lipovski, G.J., Brodersen, A.J., "On the Integrated Circuit Bidirectional Amplifiers," IEEE Journal of Solid State Circuits, Vol. SC-8, No. 5, October 1973, pp. 381-388.

20. Wulf, W.A., Bell, C.G., "C.mmp-A Multi-Miniprocessor," AFIPS Proceedings, Vol. 41, FJCC, 1972, pp. 122-131.

21. National Digital Integrated Circuits, January 1974, pp. 9-8.

22. Fairchild Semiconductors MOS/CCD Data Book, 1974, pp. 4-124.

23. Lipovski, G.J. and Hoch, C.G., "A Vari-structured Stack for Microcomputers," to appear in Proc. Euromicro Symposium, Amsterdam, October 1977.

24. Mori, R., private communications.

25. Vice, W.E., Broderson, A.J., and Lipovski, G.J., Patent No. 3,882,274.

26. Lipovski, G.J., Patent No. 4,016,545, April 5, 1977.

PARALLEL PROCESSING RESEARCH IN COMPUTER SCIENCE: RELEVANCE TO THE DESIGN OF A NAVIER-STOKES COMPUTER

Carl F. R. Weiman
Math and Computing Sciences
Old Dominion University, Norfolk, Virginia 23508

and

Chester E. Grosch
Institute of Oceanography
Old Dominion University, Norfolk, Virginia 23508

Abstract -- The design of a special purpose computer for the numerical solution of problems in fluid mechanics was discussed in meetings at RAND Corporation in 1976-77 [9]. The number of computations required puts many important problems beyond the reach of the most advanced computers available today. Speedups attainable by technological advances and software optimization do not help enough; large scale parallelism was deemed necessary. Finite difference methods break the continuous space of physical problems into discrete compartments iterated over a region. Equations relate physical quantities in a small neighborhood of compartments. Since these equations are identical over large regions, a design was proposed which consisted of an array of identical microprocessor chips communicating with nearest neighbors. Each chip carries out the computations expressed in the finite difference equations corresponding to a single compartment and all chips are simultaneously active. This design can be viewed as a digital simulation of the physical system. Reasonable numerical parameters suggest arrays of 100^2 chips, possibly in layers or internally organized to model three dimensional physical problems. The differences between this design and other parallel machines is described. Novel organization is made economically feasible by recent advances in Large Scale Integration (LSI) technology.

A Cellular Array Computer for Fluid Dynamics

There are many practical problems which require the accurate and rapid calculation of fluid-dynamic forces and flow phenomena. These include aircraft and ship design, weather forecasting, and biological modelling. Progress in providing advances in the quantitative analysis of these problems is paced to a great extent by the computing power available in the simulation of the flow phenomena of interest. The major computation in all cases involves solution of the Navier-Stokes equations. Hence the solution of a very general class of problems would be greatly facilitated by the development of a special purpose Navier-Stokes computer.

Increasing memory size permits the use of a finer spatial resolution for a fixed volume of fluid and/or the calculation of flows in larger volumes. The number of operations per time step increases at a slightly faster rate than the number of mesh points (or modes, for spectral methods). In fact, it is easily shown that if N^3 mesh points are used in a calculation the number of operations per time step is proportional to $N^3 \ell n_2 N$ for the best methods.

While there have been some improvements in algorithms, progress has been tied to the development of ever faster general-purpose computers. The most recent "super-computers" incorporate substantial amounts of pipelining and parallelism in their CPUs in addition to using faster processing elements. In short, high-speed computation has been sought via faster and more complex CPUs. It appears that all of today's "super-computers" were designed to handle many different classes of scientific computing problems. The cost of this versatility appears to be that these architectures are far from optimum for any class of problems. In fact there are many important problems for which the fastest existing computers are hopelessly slow.

Recently, Dr. I. E. Sutherland [9] has suggested an architecture which appears to be much closer to optimum, for computational fluid dynamics, than that of existing "super-computers." This architecture is a two-dimensional array of L·M cells, (see Figure 1).

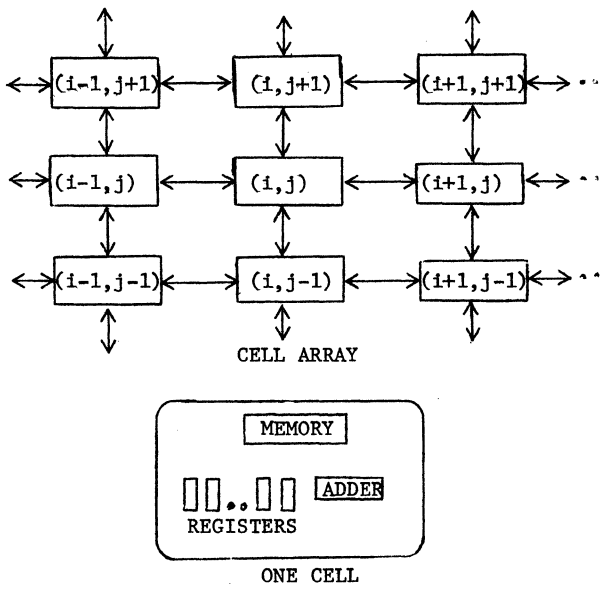


Figure 1: Schematic of the cell computer

Each cell can communicate directly with its nearest neighbors "above" and "below" and to the "left" and "right ". A single cell is assumed to contain some memory, an adder and some registers. The data is stored in memory. The registers are used as working memory to store intermediate results and the adder is used to perform binary addition and multiplication by shift-and-add. It is clear that local communication is relatively cheap and long range communication may be prohibitively expensive because of cell-to-cell propagation.

The potential advantages of this architecture are threefold: first, it appears possible to build such a computer using existing technology (each cell is only a few microprocessor chips, perhaps a single chip) at fairly modest cost, particularly if the intercellular connections are minimized; second, technological developments in the semiconductor industry, i.e., advances in Large Scale Integration (LSI) fabrication, are leading toward increasing complexity and density per chip and lower cost per chip; and third, if the array of cells can be mapped onto the fluid domain such that N^2 operations can be performed in parallel on N^2 chips, the number of sequential operations per time step can be reduced from $O(N^3 \ell n_2 N)$ to $O(N \ell n_2 N)$.

This architecture appears so promising that it is worthwhile to examine a test case. The test case considers the incompressible flow of a fluid in a boundary layer adjacent to a rigid, impermeable, no-slip wall.

The objectives of this exercise are to determine: how well a standard algorithm fits a cell computer; what modifications to the algorithm are required; which part, if any, of the algorithm dominates the calculation; which operation dominates the calculation; how many operations are required per time step; what is the memory requirement per cell; and what is the computation time per time step, using conservative estimates of the transfer, multiply, and add times. The computational region is $0 \le x \le x_o$, $0 \le y \le \infty$, $0 \le z \le z_o$. The equations of motion are, of course, the Navier-Stokes equations for an incompressible fluid

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \nabla p + \frac{1}{R}\nabla^2 \vec{u} \qquad (1)$$

$$\nabla \cdot \vec{u} = 0. \qquad (2)$$

Here $R = U_o \delta/\nu$ is the Reynolds number, where U_o is a characteristic free stream speed, δ is the boundary layer thickness, and ν is the kinematic viscosity. The velocity, $\vec{u} = iu + jv + kw$, has components (u,v,w) in the x, y, and z directions (i, j, and k are unit vectors), and p is the pressure per unit density. A Poisson equation for the pressure can be obtained by taking the divergence of Eq. (1).

Appropriate boundary conditions for the velocity are also specified such as: the inflow velocity field on the plane x = 0 is given; the velocity $\vec{u} = 0$ on the plane y = 0, etc. Neumann boundary conditions for the pressure field can be obtained by evaluating equation (1) on the boundaries. In order to represent the infinite physical domain $0 \le y < \infty$ in the finite computational domain a mapping is used to transform the infinite domain $(0 \le y < \infty)$ onto the finite domain $(0 \le \xi < 1)$. It has been shown [10] that this mapping yields highly accurate results with relatively few grid points in those cases, as in this problem, where the flow field at infinity is a simple laminar flow. The only cost is that the metric coefficients must be stored in each cell.

The physical space, $0 \le x \le x_o$, $0 \le \xi \le 1$, $0 \le z \le z_o$ is divided into $L \cdot M \cdot N$ cells centered on the points $(i\Delta x, j\Delta\xi, k\Delta z)$ where i = 1, 2, ..., L; j = 1, 2, ..., M; and k = 1, 2, ..., N. The x component of the velocity is defined at the center of the front and back faces of the cell, i.e., $u_{i-\frac{1}{2},j,k} \equiv u((i - \frac{1}{2})\Delta x, j\Delta\xi, k\Delta z, t)$. Similarly the ξ and z components of the velocity, v and w are defined at the centers of top and bottom and side faces of the cell. The pressure is defined at point (i,j,k) in the center of the cell.

A column is defined as all those fluid cells at constant x; a row as all those fluid cells at constant ξ; and a rod as all those fluid cells at constant z. It will be assumed that one rod of data, the (i,j) rod, is stored in cell (i,j). This of course implies that cell (i,j) has some multiple of N words of memory plus sufficient memory for constants such as 1/R, the metric coefficients, etc.

The spatial differencing scheme is centered second order. It can be shown that this approximation, when applied to Eqs. (1) and (2) with appropriate boundary conditions is conservative in the sense that mass is conserved for any R and, in the limit $R \to \infty$, momentum, energy, and enstrophy are also conserved.

The time differencing is the Adams-Bashforth type. Let us define $F^n_{i-\frac{1}{2},j,k}$ as the righthand side of Eq. (1) defined at (subscripts) the point $(i\Delta x, j\Delta\xi, k\Delta z)$ in space and at (superscript) time $n\Delta t$. $F_{i,j-\frac{1}{2},k}$ and $F_{i,j,k-\frac{1}{2}}$ are defined in a completely analogous way. Then the Adams-Bashforth approximation to the time derivative is

$$u^{n+1}_{i-\frac{1}{2},j,k} = u^n_{i-\frac{1}{2},j,k} + \frac{\Delta t}{2}\left(3F^n_{i-\frac{1}{2},j,k} - F^{n-1}_{i-\frac{1}{2},j,k}\right) \qquad (3)$$

In order to advance $\vec{u}^n$ and p^n by one time step it is necessary to calculate $\vec{F}^n = (F^n_{i-\frac{1}{2},j,k}, F^n_{i,j-\frac{1}{2},k}, F^n_{i,j,k-\frac{1}{2}})$ and solve the Poisson equation for p^n. As one might expect the most difficult part of the calculation is the solution of the Poisson equation with Neumann boundary conditions. This problem is elliptic, i.e., non-local. A direct solution, say by using an FFT in one direction and a tridiagonal equation solver in the other may be prohibitive because of the transfer time cost. More importantly, these methods are restricted to flows with very regular geometry; general geometries require the use of relaxation methods.

We have assumed that the relaxation method

will be Red-Black SOR which requires only K parallel iterations to converge if SOR requires K - 1 sequential iterations. This method has been simulated using a model problem. The simulation confirms the estimate.

It should be noted that a "standard" relaxation method has been assumed. We have not yet examined other methods such as cell by cell divergence-pressure method [12] or the use of block or multigrid relaxation methods [5], which could increase the convergence rate.

The operation count for one time step can be obtained by writing the finite difference equations, finding where the required data are stored, counting the number of transfers needed to get these data into the registers in cell (i,j) and the number and type of arithmetic operations required. As an example, figure 2 shows cell (i,j) and its neighbors. All the data, and only the data, required to calculate $F^n_{i-\frac{1}{2},j,k}$, $F^n_{i,j-\frac{1}{2},k}$, and $F_{i,j,k-\frac{1}{2}}$ are shown in this figure in the cells where they are stored. It can be seen that twenty words of data must be transferred to cell (i,j); of these twenty only two, $u_{i+\frac{1}{2},\ j-1,\ k}$ and $v_{i-1,\ j+\frac{1}{2},k}$, require a two step transfer.

The number of in-cell transfers, cell-to-cell transfers, additions, and multiplications necessary to advance $\vec{u}$ and p by one time step are:

IN CELL TRANSFERS = 48 N
CELL TO CELL TRANSFERS = (63 + 6fK) N
ADDITIONS = (208 + 8fK + 2 $\ln_2$N) N
MULTIPLICATIONS = (93 + 3fK + 4 $\ln_2$N) N
Number of words of memory = 18 + 12 N
Number of registers = 30

Here N is the number of mesh points in the transverse flow direction (z direction) and the product fK is the number of iterations for relaxation of the Poisson equation.

From an examination of this table, assuming fK = 50 and N = 64, say, several facts become apparent. If we take the in-cell transfer count as unity, the cell-to-cell transfer count is about 7, the addition count 14, and the multiplication count is about 5. Because it is generally true that the in-cell transfer time and the addition time are considerably smaller than the cell-to-cell transfer time and the multiplication time, it is clear that the total calculation time is controlled by the cell-to-cell transfer and multiplication counts and times. The ratio of the

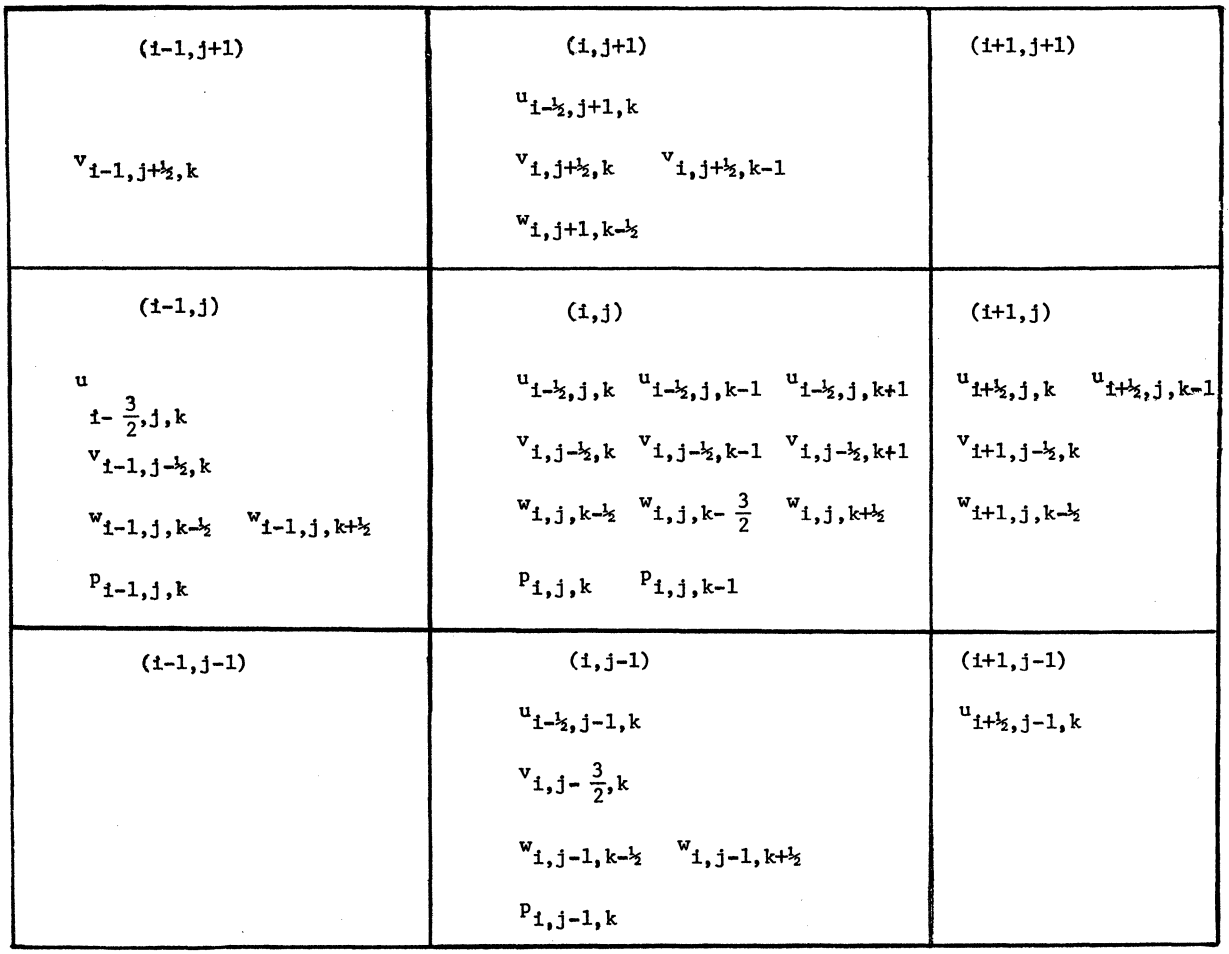


Figure 2: Data required to compute $\vec{F}^n_{i,j,k}$

TIME AND MEMORY ESTIMATES

Problem	Calculation Time for Interior Points per Time Step (sec)	Calculation Time for Boundary Points per Time Step (sec)	Calculation Time for the Pressure Field per Time Step (sec)	Calculation Time for the Velocity Field per Time Step (sec)	Total Calculation Time per Time Step (sec)	Memory per Cell (words/bits)	Total Calculation Time for One Time Step on a Fast Conventional Computer (sec)
#1 1/3 million point problem	0.060	0.027	0.076	0.011	0.087	432/~14K*	20.0
#2 1 million point problem	0.242	0.112	0.309	0.045	0.354	1584/~50K	81.9
#3 10 million point problem	0.242	0.112	0.309	0.045	0.354	1587/~50K	839

*1K = 1024

TABLE 1

multiplication count to cell-to-cell transfer count is about one, so reducing either cell-to-cell transfer time or multiplication time to zero would result in a speedup of only a factor of two.

In order to gain some insight into the performance capabilities of this type of array computer, three sample problems will be considered: (1) A (logical) array of 50 x 200 cells with N = 32; 50 points in the direction normal to the boundary, 200 in the downstream direction and 32 in the cross-stream direction. It is believed that this "one-third of a million point" problem is the absolutely minimum problem of interest in fluid dynamics. (2) A (logical) array of 50 x 200 cells with N = 128; 50 points normal to the boundary, 200 in the downstream direction and 128 across the stream. The "million point" problem is quite interesting. (3) A (logical) array of 100 x 1000 cells with N = 128; 100 points normal to the boundary, 1000 points in the downstream direction and 128 points across the stream. This "ten million point" problem is very interesting because it allows study of important fluid flow phenomena hitherto beyond our reach.

In order to calculate computation time per time step it is necessary to make assumptions about the in-cell transfer, cell-to-cell transfer, addition, and multiplication times as well as assume values for f and K. It will be assumed that: In-cell transfer time is 100 nsec; Cell-to-cell transfer time is 3 μsec; Addition time is 500 nsec; Multiplication time is 5 μsec; f is ½; and K is 100. It will also be assumed that the memory words are 32 bits and the registers are 64 bits long.

In order to compare the performance of this cell computer to a conventional computer, operation counts and time estimates for the conventional computer are needed. The operation counts for a three-dimensional finite-difference technique for a conventional computer are in [6]. In this technique, the spatial differencing is the same as used in the analysis of the cell computer performance, second-order central differences; the time differencing is Leap-Frog; and a Fast Poisson Solver is used. The downstream direction has L points, the cross-stream direction N points and the direction normal to the boundary has M points. It will be assumed that, for the conventional computer: Addition time is 100 nsec; Multiplication time is 200 nsec; and Memory transfer time is 1 μsec.

The time estimates for one time step are given in Table 1 for each of the three problems. These estimates are broken down in several ways (all times are in seconds). Column one lists the problems. In the second column the time to advance the interior points one time step is given, while the third column is the time required to advance the boundary points one time step. The sum of these is the total time required to advance all the grid points one time step, and this is given in column six. This total time is also the sum of the time required to compute the pressure field, given in column four; and the time required to compute the velocity field, column five. The total number of bits of memory required for each problem is listed in column seven. Finally, column eight is the total time required to calculate one time step on a conventional computer.

From an examination of Table 1 it is clear that most of the computation time of the array computer is spent computing the pressure field for the interior points of the grid; the ratio of the calculation time for the interior to the calculation time for the boundary is about 2, while the ratio of the calculation time for the pressure field to that of the calculation time for the velocity field is about 7. This holds true for all three problems. In short, the

dominant part of the calculation is the relaxation solution of the pressure field on the interior of the grid.

Although it is not shown in this table, precisely the opposite holds true for the calculation on the fast conventional computer using a fast Poisson solver, wherein the ratio of the velocity field calculation time to the pressure field calculation time is about 3.

For problems #1 and #2, the ratio of total calculation time on the conventional computer to that of the array computer is ~230. Increasing the number of grid points in the cross-stream direction from 32 to 128 changes this ratio by less than 1 percent. However, increasing the number of cells from 10^4 to 10^5 (problem #3) increases the ratio to about 2370. As was expected, the ten-fold increase in the number of cells is fully reflected in the speed ratio. Note that for problems #1 and #2 the speedup due to parallelism reduces a week of computation time on a sequential computer to under one hour. For problem #3, a week is reduced to about five minutes. Thus calculations whose times and costs were prohibitive on existing computers could be used routinely as standard tools by researchers in fluid dynamics and designers of vehicles such as aircraft and ships.

The problem with 32 grid points in the cross-stream, z, direction requires 432 words of memory, about 14K bits per cell. Increasing the number of grid points in the z direction to 128 increases the size of the required cell memory to 1584 words or about 50K bits. Problem #1 fits nicely into a 16K bit memory and problems #2 and #3 are good fits to a 64K bit memory per cell.

Overview: Relevance to Parallel Processing

Far fewer parallel computers have been built than sequential computers because of organizational complexity and economic barriers in the past. Processors were expensive, and iteration of components implied iterated costs, plus control overhead. Recent LSI advances have removed the economic barrier for component iteration; once design and set-up are paid for, chip reproduction cost is vanishingly small. There has been a corresponding scarcity of parallel processing research in the computer science literature, though recently such publications are increasing. Let us now briefly survey the parallel processing literature, relating important results to the proposed Navier-Stokes computer design. The short bibliography here points to major sources ranging over a broad spectrum of research areas, leading to most of the important work which has been carried out since the sixties. Research in parallel processing will be divided into the following areas: construction of parallel computers, programming languages for parallel computers, parallel evaluation of ordinary arithmetic expressions, parallel numerical algorithms and parallel grammars in formal language theory. These are discussed in order below.

The cellular design proposed here shares some characteristics with several existing parallel machines but differs in ways which suggest far greater computing power for the intended application. The existing machines, compared below with out design, are described in detail [26] and [27].

ILLIAC IV can be viewed as an 8 x 8 array of cells (PE's) arranged in a grid with communication with nearest neighbors and control and I/O busses. Each PE is a rather sophisticated general purpose computer with seven working registers and 2K 64-bit words of memory. Our cells, in contrast, contain far less memory, smaller words, and more limited processing capability, retaining only enough computation power to evaluate some finite difference expressions and communicate with nearest neighbors. This permits LSI construction of each PE on a single microprocessor chip. This has important consequences which make the design potentially thousands of times as effective as ILLIAC IV. Reduced cost of components, accellerated by chip replication, permits a design with as many cells as mesh points in the physical problem. Instead of a potential 64-fold speedup, ratios in the range of several thousand to one, relative to sequential processing, are quite feasible. More important than the magnification of scale is the fact that an 8 x 8 mesh is insufficient for most PDE problems, therefore much of the 64-fold potential speedup of ILLIAC IV is lost in overhead consisting of rearranging data to fit in the small grid. This overhead is non-existent for our design since the computation grid corresponds exactly to the problem mesh. By restricting as much of the communication as possible to nearest neighbors, i.e., avoiding long-distance high-bandwidth data transfers, effective execution rates of 5,000 to one million MIPs (millions of instructions per second) are attainable with present technology.

At the opposite end of the PE quantity-complexity tradeoffs are associative processors such as PEPE and the Goodyear STARAN. Far more cells, but each with lesser capability, are used. STARAN, for example, contains 32 256 word (256 bits each) memory arrays. Though the communications paths are not comparable with ILLIAC IV, if the 8k words are regarded as cells, each is capable of a few simple logic operations between its contents and those of a word to be matched. Arithmetic must be synthesized from these operations via finite state logic in order to be carried out in parallel. Parallelism is achieved by having cells respond according to their contents (associatively) rather than by address. The individual words do not have enough processing power to carry out the computations required for mesh cells in reasonable PDE problems.

Taxonomies for parallel (or other sequential) computers simplify comparisons between designs by abstracting essential features. Choices of what constitute essential features are moot and can cause Procrustian classifications, particularly when a new design such as the one proposed is involved. A popular taxonomy is the division of instructions (I) and data (D) into single (S) or multiple (M) streams. For example, a pipeline machine such as the CDC STAR-100 or Texas Instruments ASC is classified as MISD since one data stream is passing though an "assembly line" of

processors. Many parallel and vector machines, including ILLIAC IV, are classified as SIMD because individual PE's contain different data, but all carry out the same instruction sent by a controller. While our design is clearly MD, it could be regarded as either SI or MI, depending on how closely one looks at program control architecture. One alternative is to allow each chip to store a few simple programs, such as the sequences of instructions for boundary condition computations, interior computations, and the null computation of cells outside the region. This view yields an MIMD classification though the number of distinguishable instruction streams is small if cells are viewed as types (i.e., boundary vs. interior) and large if viewed as individuals. All cells ought to be synchronized to the same clock to preserve integrity of the physical time modelled by the computation system. If the clock is regarded as the controller, the machine should be classified SIMD. Incidentally, we specifically avoid the machine organization of hanging multiple processors on the same bus as embodied in Carnegie-Mellon's C.mmp [3]. That is, though the clock bus may have a wide enough bandwidth (say 8 bits) to issue general orders to all cells, it is never to be used for addressing individual cells nor intercell communication. This eliminates address conflict and dataflow bottlenecks (which rapidly worsen as the number of cells increases) but raises important questions about I/O. Loading programs and data into the cells could be done by feeding contents sequentially into one end of the array and using the nearest neighbor data paths to treat the entire array as a shift register. While far slower than the computation this machine is designed for, most of this machine's activity in PDE applications is very heavily compute-bound rather than I/O bound and the inconvenience is a small price to pay at the beginning and end of very long sequences of computations. The output problem is more important than input when sequences of glimpses at the physical system are desired, for example in displaying details of the transition from laminar flow to turbulence. Technological solutions such as LED output or false color video interpretation of register contents may be possible.

Despite the difference between our design and existing machines, there are many hardware implementation details in the latter that are the product of decades of experience and expertise; these are worthy of thorough study for application in our design.

Hardware organization has a large impact on software; a clear programming language is essential if this machine is to be used effectively. High level languages for array machines [13, 8] have had to deal with the problem of arranging vectors or matrices so that FORTRAN-like arithmetic expressions can be simultaneously evaluated by the parallel processors of a particular system. Looping and indexing required sophisticated analysis (see SIGPLAN conference proceedings at GISS in bibliography) compounded with the problems of compiling a high level language such as FORTRAN. Since no rearrangement of cells occurs in our design, and communication is only between nearest neighbors, indexing can be avoided entirely. The bookkeeping for sequencing through an array has been done during the fabrication of the array of chips. The subscripts in the finite difference equations are replaced by the labels of a few nearest neighbor cells. The simplicity of chip operations and stereotypy of the calculations means that we can get by with a simple assembly type language for describing the behavior (program) of a cell. The special purpose mission of this machine obviates the need for a general purpose high level language, thus saving considerable time and manpower in software development.

An area of parallel processing research which does not appear very applicable to our design is the parallel evaluation of ordinary arithmetic expressions [16, 29]. Basically, operands are grouped by pairs (since operators are binary) for simultaneous processing, pairing the results until a single number is left. Nesting and operator hierarchies introduce complications, but the overall idea is that the binary joining yields a tree structure whose depth (number of time steps) is $\log_2$ of the number of operands (m) in the original expression. For the relatively short expressions (small number of operands) in the finite difference equations, the $\log_2$ (m) speedup does not appear to be great enough to pay for the overhead of finding the appropriate tree and arranging the data in it. However, cell programs should embody efficient arrangement of calculations.

Research in parallel numerical algorithms has been intensively pursued, even prior to the likelihood of implementation [17]. Many publications concern matrix methods for the solution of simultaneous finite difference equations on grids [28]. Most of the results are not applicable to our proposed design because the row-column grids of the matrices in those methods constitute a very different representation of the problem mesh than our array of cells. That is, such matrices really correspond closely to a graph connectivity matrix of the system, where mesh points and communication paths correspond to nodes and arcs respectively. Local interaction corresponds to a sparse matrix whose number of rows (columns) is the number of mesh points. Thus, for an n x n mesh, the matrix is n^2 x n^2 and topology is totally different from that of the mesh, depending strongly on the labelling of rows and columns. For our 100 x 100 mesh, the matrix is 10,000 x 10,000; it is inconceivable that such a matrix be represented by one chip per element. Even if it were, sparseness implies wasted hardware and the matrix simplification methods in the literature [4] would require data transfers over large distances by pivoting, transpose and the like. Of course, the algorithms in the literature do not involve such inefficient data structures. Sparseness is exploited to yield a few vectors of length n^2 which get rearranged to improve independence of operations and hence parallelism. In sharp contrast, our design is a relaxation machine and data rearrangement is avoided as much as possible. Nevertheless, the ingenious effort which has been expended in direct methods warrants intensive study for applications here.

Stone's observations [25] on parallel versus

serial numerical algorithms are important enough to mention here before concluding. He notes that efficient serial algorithms may be totally unlike efficient parallel algorithms, that many apparently inherently serial algorithms in fact are not, that numerical convergence rates differ in serial versus parallel algorithms, and that arrangement of data structures is much more important in parallel processing than serial. The last point results from the fact that random access in serial processing makes all locations equally accessible; random access between all processors in parallel is ruled out because of access conflicts or interconnection structures whose complexity grows as the square of the number of elements. Hence communication is restricted to, for example, nearest neighbors. These observations suggest that much of what we know about serial processing is not applicable to parallel processing. This idea is confirmed on a fundemental level by recent developments in the theory of formal languages [11]. Study of formal systems yields principles whose value in applications includes determining algorithm design, programming language principles, hardware complexity tradeoffs, and establishing ultimate limitations on computation power [24]. Recent results in parallel grammars indicate that many problems which are high on the complexity scale when algorithms are stated sequentially are much simpler when parallel operations are permitted. Detailed investigation of a broad spectrum of formal languages find this to be the general case rather than an exception [19]. Immediate applications include the design of parallel programming languages. The powerful, unexpected new results characteristic of this field suggest long range applications in the design of future parallel computers.

References

[1] Baer, J. L., "A Survey of Some Theoretical Aspects of Multiprocessing", ACM Computing Surveys, Vol. 5 (March, 1973) pp. 31-80.

[2] Batcher, K. E., "The Flip Network in STARAN", in [7], pp. 65-72.

[3] Bell, C. G. and W. A. Wulf, "C.mmp - A Multi-Mini-Processor", AFIPS Conference Proceedings, Vol. 41, 1972, pp. 765-778.

[4] Birkhoff, G. and A. George, "Elimination by Nested Dissection", in [28], pp. 221-269.

[5] Brandt, A., "Multi-Level Adaptive Solutions to Boundary-Value Problems," Institute for Computer Applications in Science and Engineering Report 76-27, Institute for Computer Applications in Science and Engineering, NASA Langley, Hampton, Va., 1976.

[6] Case, K. M., A. M. Despain, E. A. Frieman, F. W. Perkins, and J. F. Vesecky, "Problems in Laminar Flow-Turbulent Flow," Stanford Research Institute Technical Report JSR-74-2, 1975.

[7] Enslow, P. H., (ed.) Proceedings of the 1976 International Conference on Parallel Processing, IEEE, ACM, Wayne State University.

[8] Erickson, D. B., Array Processing on an Array Processor, in [21], pp. 17-24.

[9] Gritton, E. C., W. S. King, I. Sutherland, R. S. Gaines, C. Gazley, Jr., C. E. Grosch, M. Juncosa, H. Peterson. Feasibility of a Special-Purpose Computer to Solve the Navier-Stokes Equations, RAND Corporation, R-2183-RC, (June, 1976), 74 pp.

[10] Grosch, C. E., and S. A. Orszag, "Numerical Solutions of Problems in Unbounded Domains: Coordinate Transformations," Journal of Computational Physics, (in press), 1977.

[11] Herman, G. T. and G. Rozenberg, Developmental Systems and Languages, North-Holland, 1975.

[12] Hirt, C. W., and J. L. Cook, "Calculating Three-Dimensional Flows Around Structures and Over Rough Terrain," Journal of Computational Physics, Vol. 10, p. 324, 1972.

[13] Lamport, L., On Programming Parallel Computers, in [21], pp. 25-33.

[14] Lang, T. and H. S. Stone, "A Shuffle-Exchange Network with Simplified Control", IEEE Trans. on Computers, Vol. C-25, (January 1976), pp. 55-65.

[15] Kohler, W. H., "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems" IEEE Trans. on Computers, Vol. C-24 (Dec. 1975), pp. 1235-1238.

[16] Kuck, D. J., Multioperation Machine Computational Complexity in 25), pp. 49-82.

[17] Miranker, W. L., "A Survey of Parallelism in Numerical Analysis", SIAM Review, Vol. 13, No. 4, October 1971, pp. 524-547.

[18] Morris, D. and P. C. Treleaven, A Stream Processing Network, in [21], pp. 107-112.

[19] Moshell, J. M. and J. Rothstein, "Bus Automata and Parallel Computation", Proc. 1976 Southeastern Symposium on System Theory, University of Tennessee, Knoxville, pp. 246-252.

[20] Moshell, J. M. and J. Rothstein "Parallel Recognition of Patterns: Insights from Formal Language Theory", in [7], pp. 222-229.

[21] Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines, March 18-19, 1975, Goddard Inst. for Space Studies, N.Y., N.Y. ACM SIGPLAN notices Vol. 10, No. 3, Sponsored by ACM, SIGPLAN, NASA, GISS.

[22] Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, Syracuse University.

[23] Rose, D. J., "A Graph-Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations", pp. 183-217 in Read, Graph Theory and Computing, Academic Press, 1972.

[24] Rothstein, J., "On the Ultimate Limitations of Parallel Processing", (Best Paper Award) in [7], pp. 206-212.

[25] Stone, H. S., "Problems of Parallel Computation", in [28], pp. 1-17.

[26] Thurber, K. J. Large Scale Computer Architecture: Parallel and Associative Processors Hayden, Rochelle Park, N.J., 1976.

[27] Thurber, K. J. and L. D. Wald, "Associative and Parallel Processors", ACM Computing Surveys, (Dec. 1975), pp. 215-255.

[28] Traub, J. F. (Ed.), Complexity of Sequential and Parallel Numerical Algorithms, (Conf. Proc.) Academic Press, 1973.

[29] Winograd, S., On the Parallel Evaluation of Certain Arithmetic Expressions. JACM, Vol. 22, No. 4, pp. 477-492, 1975.

CONTROLLING THE ACTIVE/INACTIVE STATUS OF SIMD MACHINE PROCESSORS

Howard Jay Siegel
School of Electrical Engineering
Purdue University, West Lafayette, IN 47907

Masking schemes are used to control the active/inactive status of each of the processing elements (PE's) in an SIMD machine. Three schemes - data conditional masks, general masks, and PE address masks - are analyzed in terms of hardware and software implementations, time and space requirements, ability to activate arbitrary sets of PE's, and ease of programmer use.

The PE address masking scheme uses an m-position mask to specify which of $N=2^m$ PE's are to be activated. Each position of the mask will contain either a 0, 1, or X ("don't care") and the only PE's that will be active are those whose address matches the mask: 0 matches 0, 1 matches 1, and either 0 or 1 matches X. Superscripts are repetition factors; square brackets denote a mask. The structure of these masks allows them to perform tasks such as activating the following sets of PE's: even numbered PE's - $[X^{m-1}0]$; the 2^i PE's beginning with J, where $J > 2^i$ and $J = j_{m-1}\cdots j_{i+1}j_i 0^i$ - $[j_{m-1}\cdots j_{i+1}j_i X^i]$; every 2^ith PE beginning with J, where $J < 2^i$ and $J = j_{i-1}\cdots j_1 j_0$ - $[X^{m-i}j_{i-1}\cdots j_1 j_0]$.

A mask may accompany each instruction, or may be executed whenever a change in the active status of the PE's is required. Consider the task of activating an arbitrary set of PE's with a set of these masks. For example, if N=8, and only PE 0 and PE 7 are to execute instruction A, A must be executed with [000] and then with [111]. The set of masks is required to be of minimum size and each PE to execute a given instruction must execute it exactly once.

Theorem: The lower and upper bounds on the size of the set of masks necessary to activate an arbitrary set of PE's are N/2.

Proof: Lower bound: Let J be the set of PE addresses whose binary representations each contain an odd number of ones. If a mask has an X in its ith position, then it will activate two PE's whose addresses are identical except for their ith bit position, i.e., one address will have an odd number of ones, the other even. Thus, for each address in J, whose size is N/2, a separate mask will be required. Upper bound: Consider the arbitrary pair of PE's $y_{m-1}\cdots y_2 y_1 0$ and $y_{m-1}\cdots y_2 y_1 1$. If both are to be activated use $[y_{m-1}\cdots y_2 y_1 X]$, if only the former use $[y_{m-1}\cdots y_2 y_1 0]$, if only the latter use $[y_{m-1}\cdots y_2 y_1 1]$, and if neither do nothing. ○

It is most likely that sets of PE's such as J above will be anomalies, but this is highly user dependent. By using a modified Karnaugh map procedure the smallest set of masks necessary to activate a given set of PE's can be computed.

A negative PE address mask is the same as a regular PE address mask, except it activates all those PE's which do not match the mask. Negative masks are prefixed with a minus sign. For example, $[-1^m]$ activates all PE's except 1^m, a task which would require m regular PE address masks. In most, but not all, cases the combination of negative and regular masks is better than regular masks alone.

Theorem: The lower and upper bounds on the size of the set of masks, consisting of both types of masks, necessary to activate an arbitrary set of PE's are N/2.

Proof: Similar to previous theorem. ○

To quantify the additional power negative masks contribute consider the number of distinct masks formable from these two schemes, where two masks are considered to be distinct if they activate different sets of PE's.

Theorem: The number of distinct masks formable from the set of all possible regular and negative PE address masks is $2(3^m-m)$.

Proof: The regular PE address masks form 3^m distinct masks. The mask $[-X^m]$ does not activate any PE's. A negative mask with m-1 X's has an equivalent regular mask, e.g. $[X^{m-1}1] = [X^{m-1}0]$. Each negative mask with fewer than m-1 X's is distinct from any regular mask. There are $3^m-(2m+1)$ such masks. ○

Let n be the number of X's in a mask. A regular mask activates 2^n PE's and a negative mask activates $N-2^n$ PE's.

If two bits are used to represent each mask position, then a fourth symbol, in addition to 0, 1, and X, could be used. For example, "S" could mean "same as the bit to the right," so that [1SX] would activate PE's 4 and 7, or "D" could mean "different from the bit to the right," so that [DDX] would activate PE's 2 and 5.

The PE address masking scheme and its variations present a clear and concise notation for masking. If we assume each PE knows its own address then, using data conditional statements for software decoding, the notation of PE address masks could be implemented without additional hardware costs.

The analyses and comparisons presented in the full paper aid the machine designer in choosing a masking system and a method to implement it.

This is a summary of Purdue TR-EE 77-25, supported by NSF Grant DCR 74-21939 at the Princeton University Electrical Engineering and Computer Science Department and by the Purdue University School of Electrical Engineering.

ARCHITECTURAL DESIGN CONSIDERATIONS FOR A FAULT-TOLERANT ARRAY PROCESSING SYSTEM [a]

Alexander Thomasian [b] and Algirdas Avižienis
Computer Science Department
University of California, Los Angeles
Los Angeles, California 90024

Summary

A large number of architectural issues should be resolved in designing parallel processing systems for large scale numerical computations. We discuss here the approach adopted in a particular case, the design study of an array processing system called the Shared Computing Resource - SCR [1], [2].

Realization of high computational capacity for array processing. High computational capacity is achieved in the SCR system by means of two arrays of homogeneous units. An array of multifunctional, high-speed arithmetic processors is provided to perform arithmetic operations on large arrays of data. An array of address generators handles the fetching and storing of array operands residing in a large, high-bandwidth memory.

Realization of high performance for array processing. The evaluation of expressions involving array operands is speeded up in the SCR system by allocating several arithmetic processors and address generators to the computation. For example, the evaluation of the vector expression: A←B×C+D, requires two arithmetic processors and four address generators. The evaluation of vector expressions can be speeded up in the SCR system by applying segmentation, which consists of breaking down long computations into segments which are then distributed among the units. Because of the additional setup time overhead incurred when this approach is applied, task segmentation is performed considering the length of the vector operands involved in a computation and the availability of other tasks to be executed in the system. This leads to the space-sharing approach, where several computations can execute concurrently in the SCR system.

Scheduling of computations. A centralized scheduling unit performs the scheduling of computations in the SCR system. The scheduler keeps track of the status of the SCR units and initiates a computation when the resources required for the execution of a computation become available. When a computation is scheduled for execution, a control unit sets up the assigned arithmetic processors and address generators, as well as the data transmission paths among them. The assigned units then proceed autonomously with the computation and a very simple intercommunication scheme is required to coordinate their operation. The scheduling and setup overhead is acceptable in the SCR system, since it is prorated over a large number of array elements.

Continuous availability. Continuous availability is achieved in the SCR system by applying the pooling approach, such that computations are assigned dynamically to the SCR units. It is assumed that the arithmetic processors and address generators have builtin checking capability to detect hardware failures. When a unit fails, the computation whose execution was suspended is reenqueued for execution. A prerequisite of this scheme is that each computation can only request a subset of the SCR units.

Memory bandwidth utilization. Due to the high computational capacity associated with multiple functional units, the performance of the system might be constrained by the bandwidth of the memory holding the array operands. The performance of the SCR system can hence be increased by providing an interconnection network among the arithmetic processors. The data-flow graphs of array computations are then mapped into this interconnection structure by setting up the configuration required by the computation.

Conditional processing of vector operands. At a small additional cost in hardware complexity and making use of the multiplicity of functional units in the SCR system, efficient processing of conditional expressions with vector operands can be realized. More generally, the SCR system can be shown to be suitable for the direct implementation of most array processing operators in APL.

References

[1] A. Thomasian, and A. Avižienis, "A Design Study of a Shared-Resource Computing System," Proceedings of the Third Annual Symposium on Computer Architecture, January 1976, pp. 105-112.

[2] A. Thomasian, "A Design Study of a Shared-Resource Array Processing System," Technical Report UCLA-ENG-7702, Computer Science Department, University of California, Los Angeles, April 1977.

(a) This research was sponsored by the National Science Foundation, Grant No. MCS72-03633 A04.

(b) Dr. Thomasian is currently with the Computer Engineering and Information Sciences Department at Case Western Reserve University, Cleaveland, Ohio.

PEPE HARDWARE AND SYSTEM OVERVIEW

Alf John Evensen
System Development Corporation
Huntsville, Alabama 35805

Summary

The PEPE (Parallel Element Processing Ensemble) hardware has been produced and is currently installed at the Ballistic Missile Defense Research Center. The Experimental facility includes a partial PEPE machine and the software system needed for coding, evaluating, and demonstrating experimental BMD processes on the machine. The PEPE operates in conjunction with a Burroughs B1700 Test and Maintenance computer and a CDC 6400/7600 host computer. This paper presents current details relative to the existing physical and performance characteristics which are shown in Tables 1 and 2. PEPE is configured as shown in Figure 1.

Reference

[1] Evensen, A. J. and Troy, J. L., "Introduction to the Architecture of a 288 Element PEPE," Proceedings of the 1973 Sagamore Computer Conference on Parallel Processing.

Table 1. PEPE Element Bay Characteristics

Partitioning	4 rows with 9 elements/row (36 elements/bay)
System Capability	8 bays (288) elements
Bay Profile	4 rows of element boards and each row provided with individual power supply (9 elements)
Element	6 boards/element
Board Dimensions	16" X 18" X .1"
Multilayer Configuration	8 copper PC layers: (S_1, S_2, GND, V_{CC}, V_{EE}, GND, S_3, S_4)
Dual In-Line Packages	300 DIPs/board (maximum) attached by means of sockets
Power Dissipation/Board	130 W (average)
Power Dissipation/Element	20 kW
Power Supply Dimensions	18" X 17.75" X 26"
5.2 Volt Supply Load	560 A/supply
2.0 Volt Supply Load	544 A/supply
Dimensions	84" Height X 82" Width X 30" Deep
Cooling	Chilled-water heat exchanger, with forced-air dual blower

Table 2. Control Console Characteristics

PC Boards	3 Rows of 54 Boards/row
Board Partitioning	10 ACU 11 CCU 11 AOCU 10 ACU Memory 24 Input/Output Units 1 EMC-ODC 11 ICL 6 MCDU 78 Spare Positions 2 Door Mounted Signal Distribution System Boards
Board Configuration	6 Layers (S_1, GND, V_{CC}, V_{EE}, GND, S_2) Plus Wire Wrap
Power Dissipation	7 kW
Dimensions	84" Height X 82" Width X 30" Deep
Cooling	Chilled Water Heat Exchanger With Forced-air Dual Blower

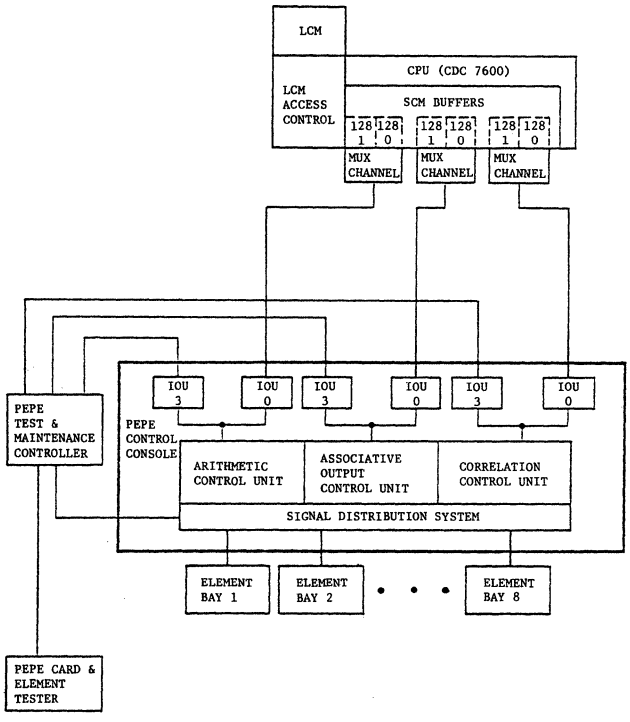


Figure 1. PEPE Functional Configuration

NUMERICAL WEATHER PREDICTION IN THE PEPE PARALLEL PROCESSOR

Howard O. Welch
System Development Corporation
Huntsville, Alabama 35805

Abstract -- The mapping of a generic numerical weather prediction model onto the PEPE parallel associative array processor is described in this paper. The case study demonstrates that the PEPE array processor can apply significantly greater computational power to the problem than can be achieved with available conventional computer architectures. The paper describes PEPE architecture, mapping of the finite difference approximations onto the array, conversion from Fortran to Parallel Fortran, run time measurements, and comparative results.

The study shows that PEPE architecture is well matched to finite difference approximations for the solution of partial differential equations even though there is no hardware provision for parallel interelement data transfer, provided the finite difference mesh is fairly large.

Introduction

The PEPE processor was designed expressly to handle enormous real time data processing loads of the types encountered in ballistic missile defense (BMD) applications. PEPE was designed as an augmentation of a commercial serial computer to assume that part of the BMD data processing load having the following characteristics:

1) Correlation of input data with the existing data base by one or more attributes.

2) Repetitive, highly arithmetic processing on a large number of independent data sets.

3) Multi-leveled ordering and search of a large, complicated data base.

The PEPE computer architecture, a parallel array with three independent processors per array element and associatively addressed operand memory, is well suited to this data processing problem [1]. The BMD requirements for radar return correlation, digital filtering, tracking and radar resource allocation are solved in the three independent PEPE processing units described in the following section.

PEPE, while designed specifically for the BMD problem, has a general purpose instruction set and the PEPE parallel Fortran language (PFOR) does not limit the user to any specific application [2]. There is therefore a reasonable expectation that the potentially enormous data processing power inherent in parallel associative architecture might be applied to other problems which currently severely stress or are beyond the capability of the most powerful commercially available computers.

The second order partial differential equations of fluid flow do not, in general, have analytic solutions and hence are solved only with numerical methods which require great computational power. Of these equations, those describing hydrodynamic flow, global weather models, and magnetohydrodynamic models are of current interest. [3][4][5]

Parallel computer architectures expressly designed for solution of partial differential equations have featured provision for interelement data transfer to solve the finite difference representations of the equations. This paper shows that PEPE associatively addressed array elements suffice for interelement data transfer given a large finite difference lattice; and furthermore that the PEPE unstructured array architecture provides certain advantages over a machine with direct hardware interelement communications.

SDC has selected the global weather modeling problem as representative of the general class of partial differential equations and has coded and executed a benchmark supplied to SDC by the Geophysical Fluid Dynamics Laboratory, Princeton University. This paper describes the implementation and results of the benchmark effort.

PEPE Architecture

PEPE (Figure 1) consists of an ensemble of independent digital processing elements, indefinite in number, which operate in parallel under global control. The current 288 element PEPE configuration [1] has three modules, each consisting of an independent global control driving an associated processing unit in each PEPE processing element. The three processors in the element share a common element data memory for parallel operand storage. The three modules are optimized for correlative data base search and data input, floating point arithmetic and associative data base search and output respectively, reflecting a design response to the BMD data processing problem.

Each control unit has a data memory and a program memory independent of the other global control units. The global control units have a limited data processing capability, the instruction repertory being limited to logical test, branch, index register and input/output control, plus sufficient integer arithmetic to compute indexes and addresses. This instruction set serves primarily to control the parallel instruc-

tion sequence rather than to execute code directly related to the solution. Parallel and global control unit instructions are stored in the program memory associated with the global control unit.

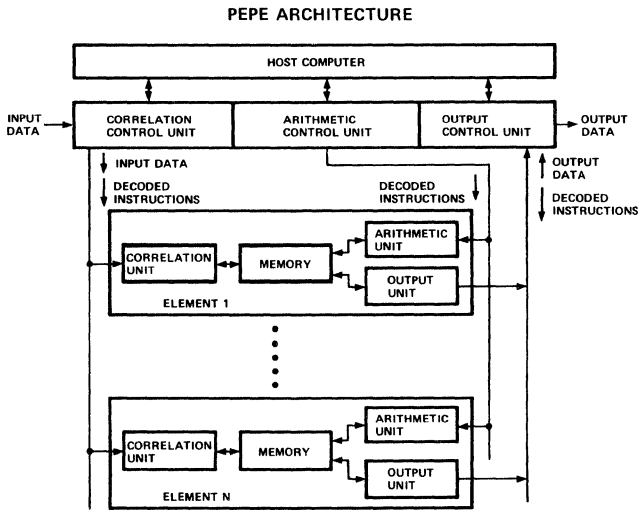


Figure 1. PEPE Architecture

Parallel instructions are decoded in the global control units to cause a microinstruction sequence of control pulses to be moved from a wide bandwidth, read-only memory to the parallel instruction bus for transmission and execution in the processing elements. The control pulses cause simultaneous and identical operations in each of the active processing elements.

Each processing element has three units, corresponding to and independently controlled by the three global control units by means of three instruction/data busses. Each processing element has 2048 words of memory shared by the three element processing units for parallel operand storage.

Each of the three element processing units has an associated activity indicator which can be programmed active or inactive according to the contents of the unit's accumulator or other condition register. The parallel instruction stream from a control unit is routed only to active element processing units. A full set of parallel instructions, analogous to logical test and branch instructions, allow the programmer to set activity and hence to control the set or subset of elements which participate in processing. The subset can be selected by data attribute rather than data location address, hence, the claim for associative memory in PEPE. A hardware extremum search algorithm is included in the parallel instruction repertory [6]. This instruction allows selection of the processing element with the largest (or smallest) accumulator value, providing a data ordering capability in the PEPE hardware.

Consider the aggregate of element memory as a two-dimensional MxN memory array with any column formed by the M words of a single element memory, and rows formed by the N PEPE processing elements. Rows are addressed conventionally by the parallel instruction operand address; each row consists of a vector whose elements are defined by the set of active processing elements, where activity is specified by the activity selection instructions described above. Arithmetic and logical operations are performed simultaneously on all vector elements so that data access and manipulation in the row dimension, i.e., across the processing elements, uses the same processing time for n vector elements as for one element in the array or for none. The PEPE instruction format has no direct hardware provision for addressing any specific physical element and there is no provision for specifying any set size (except for isolation of a single element).

Direct communication between two physically adjacent PEPE elements is not provided, reflecting the BMD origins of PEPE architecture. The only mechanism available for interelement data transfer is to move data from a given element to the global control unit and then back to other elements. This process is serial in nature and hence slower than purely parallel operations in PEPE.

GFDL Benchmark

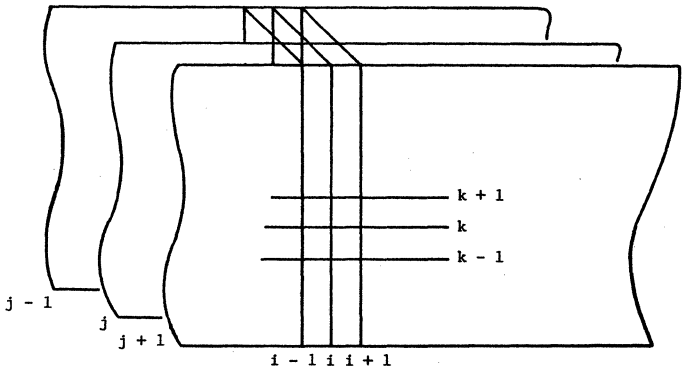
The Geophysical Fluid Dynamics Laboratory (GFDL) benchmark is a program for evaluating large computer performance. The benchmark resembles currently used atmospheric simulation models but does not contain a complete set of consistent atmospheric processes. Its purpose is the exercise of the computer rather than a physical simulation.

Physical processes modeled in the benchmark include horizontal and vertical advection, earth's rotational effects (Coriolis Force), horizontal pressure gradient, non-linear horizontal viscosity, and diffusion, and heat convection. Excluded physical processes include vertical diffusion, a hydrologic cycle, radiation and surface effects. See Appendix A for the differential equations governing the model.

The prediction domain is the Northern and Southern hemispheres, each projected on a polar stereographic plane tangent at the pole. Provision is made to confine flow within a hemisphere. The vertical domain is divided into nine spatially unequal layers defined in a normalized pressure (sigma) coordinate system. See Figure 2.

At each time step, the tendencies of the prediction variables are evaluated in a vertical slice of nine rows which form a plane of the finite difference lattice. The total length of each row is 161 points, although computations are carried out at only a limit number of points, the number being a function of latitude

and averaging 125 over a single time step.



i - Longitudinal Index
j - Latitudinal Index
k - Altitudinal Index

Figure 2. Finite Difference Lattice

PEPE Implementation

General Characteristics

Time is advanced in the model by the equation

$$\alpha^{t+1} = \alpha^{t-1} + 2\Delta t \frac{\partial \alpha^{t}}{\partial t} \quad \text{for all } t > 0$$

where α represents any prediction variable and the superscript refers to time.

The finite difference representations of the equations take the form

$$\alpha^{t+1}_{ijk} = f(\alpha^{t-1}_{ijk}, \alpha^{t}_{ijk}, \alpha^{t}_{i \pm 1\, j \pm 1\, k \pm 1})$$

where the subscripts refer to spatial position in the finite difference lattice.

Data for the finite difference lattice are stored in columns, 9 points deep. The prediction variables require 39 single precision computer words of storage, hence the lattice of 161 x 125 columns requires about 785000 words of storage. The integration algorithm requires space for three complete sets of these data for a total of 2.35 million computer words. This storage requirement exceeds the capacity of PEPE element memory so that mass storage, accessed through the CDC 7600 host, is used. The time update is performed one lattice plane at a time. A plane represents all points at a given latitude and may vary from 25 to 161 points. Each column of a plane is evaluated in a PEPE element so that a complete plane is processed as vectors of the prediction variables.

The PEPE correlation unit is used as the input device. Data are read from mass storage by the host, formatted into messages and transmitted to PEPE. Two messages per lattice column are required, one for the time step t - 1 values and one for the time step t values. Since time t data from the adjacent lattice columns are required for update of any column, the correlation unit stores data from a given column into the two logically adjacent elements. It is not necessary that data from adjacent lattice columns be stored in physically adjacent PEPE elements since the associative addressing of elements suffices to find logical adjacencies.

Adjacencies in latitude and altitude are accessible directly in element memory and hence do not require the redundant data storage described above for the longitudinal adjacencies.

Programs

The PEPE implementation of the GFDL benchmark was designed according to the following criteria:

1) Provide maximum vector lengths for the predictions for maximum execution time reduction over the serial implementation.

2) Provide maximum overlap of correlation unit and associative output unit operations with arithmetic unit operations.

3) No changes to the algorithm are made, i.e., the data are processed in the same order and with the same arithmetic operations as in the serial implementation.

The code conversion for PEPE implementation falls into 6 categories:

1) FORTRAN subroutines essentially unchanged.
2) FORTRAN subroutines subtantially modified.
3) Subroutines rewritten into PFOR, either retaining their distinct identity or incorporated into other subroutines.
4) New FORTRAN code.
5) New PFOR code.
6) Deleted subroutines.

The serial implementation consists of 33 code segments (subroutines plus the main program) with 631 lines of FORTRAN code excluding data descriptors and nonoperable statements. The PEPE implementation consists of 23 code segments with 854 lines of code. Of this code, 229 lines are FORTRAN, executed in the host and 625 lines are PFOR executed in PEPE.

Of the host code, three subroutines are directly taken from the serial implementation. Two of them, DUMDAT and CONST are essentially unchanged while the third, GFDL, is approximately 50% rewritten and radically restructured to interface with the PEPE/7600 Real Time Executive.

The PEPE or PFOR code consists of 17 code segments. Twelve of these segments, totaling 519 lines of code, are direct translations to PFOR of 30 FORTRAN subroutines totaling 481 statements. Many small subroutines were incorporated into the calling code segment, either because the subroutine was specific to the data structure of the serial implementation or because the number of formal call parameters appeared difficult to handle in the parallel implementation.

Five new PFOR routines totaling 106 statements were written. Four of these provide the message handling code for input and output of data and constants while the fifth is a segment of the control program GFDL moved to PEPE.

The primary reasons for the 35% increase in the number of lines of code appear to be from four causes. First, calculation of some intermediate variables is done three times; for the i - 1, i, and i + 1 indexes, in order to obviate any requirement for interelement data transfer. Second, 18 FORTRAN subroutines were incorporated into in-line code, in some cases several times. Third, the new subroutines to interface the host and PEPE represented new and unique requirements. Last, certain routines were highly logical and branched in the FORTRAN version and did not efficiently convert to PFOR.

Of the 625 lines of PFOR code, approximately 451 lines or 72% was directly transferred from the FORTRAN with only a change in the index specification of the data items.

Data Base Conversion. The major design task in converting a program from serial to parallel implementation is the conversion of the data base. PEPE has a complicated data memory structure in that it has element memory, three global data memories and, in the case of the CDC 7600, a large core, small core and mass storage.

This conversion placed a major emphasis on the PEPE configuration, hence little effort was expended to simplify or reduce host data storage requirements even in cases where redundant or nonrequired data space was used.

The mass storage interface was essentially left unchanged, except for the initial mass storage reads necessary to initialize PEPE at the start of each time step.

Data is read from disc to the CDC 7600 Small Core Memory (SCM) and then to Large Core Memory (LCM), one lattice plane at a time. The plane of colatitudinal lattice points 9 rows deep is stored in PEPE as columns with longitudinal index i distributed across the PEPE elements, i.e., each i entry is assigned to a PEPE element. The immediately neighboring columns, indexes i - 1, and i + 1 are also stored in the i element so that there will be no requirement for interelement data access during the update. Three lattice planes, indexes i - 1, j and j + 1 are also necessary to evaluate the finite difference equations. The time update equation requires the variables for point ijk at the previous time step.

Finally, the overlapped I/O in PEPE requires that space be allocated for plane j + 2 and for the updated point at plane j - 1. This totals 14 lattice columns stored in each PEPE element memory. Figure 3 illustrates the memory configuration.

α^{T}_{i-1j+2}	α^{T}_{ij+2}	α^{T}_{i+1j+2}
α^{T}_{i-1j+1}	α^{T}_{ij+1}	α^{T}_{i+1j+1}
α^{T}_{i-1j}	α^{T}_{ij}	α^{T}_{i+1j}
α^{T-1}_{ij+1}	α^{T}_{ij-1}	α^{T-1}_{ij+2}
α^{T+1}_{ij}	α^{T+1}_{ij-1}	

Figure 3. PEPE Element Memory Map

Performance

Measurement Methods

The PEPE real time clock, counting at a 5×10^6 Hz rate, is used for all execution time measurements. The clock provides measurements to a 200 ns granularity. There are 8 additional counters/timers in the PEPE arithmetic control unit, under program control, which can measure 127 different PEPE hardware events or time intervals associated with those events. Events include control unit instruction execution, parallel instruction execution, element memory conflicts, etc. Timing in the eight counter/timers takes place to a granularity of 100 ns. PEPE instructions control the reading and starting of the clock and counters.

Measurement Limitations

All timing information was collected on the Advanced Research Center PEPE hardware which has 11 processing elements. Execution of a problem which requires 161 elements maximum, 125 elements average, may be timed with confidence on the 11 element PEPE due to the unstructured nature of the associative array architecture. The parallel instruction set execution time is not, in general, sensitive to the number of active elements in the ensemble. Parallel code executes in exactly the same time with 11 elements, 125 elements or 161 active elements. The only exceptions to this in the GFDL benchmark is in the mixing ratio adjustment subroutine MRADJ where interelement transfer of the mixing ratio variable is required. The execution time of this subroutine is strongly dependent on the calculated value of mixing ratio at each pressure altitude point in the updated lattice plane. Since no valid values of mixing ratio are calculated, execution time for MRADJ benchmark represents an absolute minimum.

Execution Time

Execution time is measured over a single lattice plane and then extrapolated to a 161 plane time step. The lattice plane was selected to have 125 columns, the average number of columns per time step. A 125 column lattice plane requires execution of 72254 instructions. Of these, 42565 instructions are executed in the parallel elements and 29689 are executed in the control unit, with the control unit instructions substantially time overlapped with the parallel instructions. The average length in the 161 planes is 125, hence PEPE executes 125 x 42565 + 29689 = 5350314 instructions per plane. Measured time per plane is .024703 seconds giving an effective average instruction execution rate on the GFDL problem of 216.5 million instructions per second. A complete time step of 161 lattice planes requires 3.977183 seconds to complete. This is a minimum time due to the mixing ratio adjustment calculation.

Table I describes performance of various machines executing 257 time steps on the GFDL benchmark and the extrapolated time for PEPE to execute the same problem. The estimated PEPE time is a minimum time and would be larger if the full 161 element PEPE were available to allow MRADJ to run on correct data.

Table I. Comparison Execution Times

MACHINE	TIME (MIN)
IBM 360/91	245
CDC 7600*	205
IBM 360/195	120
TI ASC (FOUR PIPE)	20
PEPE/CDC 7600*	17

*EXTRAPOLATED TIME

Table II provides a breakdown of executed instructions and execution times of the subroutines in the benchmark. Totals differ slightly from above due to slight differences in measurement technique and overhead.

Table II. Subroutine Execution Statistics

SUBROUTINE	CONTROL UNIT INST	PARALLEL INST	EXECUTION μS
NEXTRW	1544	2366	1336.8
HRDIF1	1566	1556	1067.4
INNER1	25765	36483	21282.0
UTAUP	195	440	217.1
PSTAR	154	210	124.4
MNT	508	349	293.0
KPHI	1144	1212	805.5
TTAUP	288	599	303.3
RTAUP	189	385	196.2
VTAUP	167	205	127.2
MRADJ	111	92	69.4

Appendix A

GFDL Benchmark Differential Equations

$$\frac{\partial u}{\partial t} = -m\left[\frac{\partial}{\partial x}\left(u\frac{mu}{P_*}\right) + \frac{\partial}{\partial y}\left(\frac{mu}{P_*}\right)\right] - \frac{\partial}{\partial Q}(\bar{\omega}u) + \left[f - M\left(x\frac{v}{P_*} - y\frac{u}{P_*}\right)\right]v - P_*\left(\frac{\partial \phi}{\partial x}\right)_P + F_x \quad (1)$$

$$\frac{\partial v}{\partial t} = -m\left[\frac{\partial}{\partial x}\left(v\frac{mu}{P_*}\right) + \frac{\partial}{\partial y}\left(u\frac{mv}{P_*}\right)\right] - \frac{\partial}{\partial Q}(\bar{\omega}v) - \left[f - m\left(x\frac{v}{P_*} - y\frac{u}{P_*}\right)\right]u - P_*\left(\frac{\partial \phi}{\partial y}\right)_P + F_y \quad (2)$$

$$\frac{\partial}{\partial t}(P_*T) = -m^2\left[\frac{\partial}{\partial x}(Tu) + \frac{\partial}{\partial y}(Tv)\right] - \frac{\partial}{\partial Q}[P_*\bar{\omega})T] + \left(\frac{RT}{f_P}\right)\left(\frac{\omega}{Q}\right) + F_T + \Delta(P_*T)_C \quad (3)$$

$$\frac{\partial}{\partial t}(P*r) = -m^2[\frac{\partial}{\partial x}(ru) + \frac{\partial}{\partial y}(rv)] - \frac{\partial}{\partial Q}[P*\bar{\omega})r] + F_r + \Delta(P*r)_N \quad (4)$$

$$\frac{\partial}{\partial t}P* = -m^2 \int_0^1 (\frac{\partial v}{\partial x}) + \frac{\partial v}{\partial y})dQ \quad (5)$$

$$P*\bar{\omega} = -Q\frac{\partial P*}{\partial t} - m^2 \int_o^Q (\frac{\partial}{\partial x} + \frac{\partial}{\partial y})dQ \quad (6)$$

$$\frac{\omega}{Q} = -Q\frac{\partial}{\partial Q}(\frac{P*\bar{\omega}}{Q}) - m^2P*[\frac{\partial}{\partial x}(\frac{u}{P*}) + \frac{\partial}{\partial y}(\frac{u}{P*})] \quad (7)$$

$$F_x = m^2[\frac{\partial}{\partial x}(P*K\frac{D_T}{m^2}) + \frac{\partial}{\partial y}(P*K\frac{D_S}{m^2})] \quad (8)$$

$$F_y = m^2[\frac{\partial}{\partial x}(P*K\frac{D_S}{m^2}) - \frac{\partial}{\partial y}(P*K\frac{D_T}{m^2})] \quad (9)$$

$$F_T = m^2[\frac{\partial}{\partial x}(P*K\frac{\partial T}{\partial x}) + \frac{\partial}{\partial y}(P*K\frac{\partial T}{\partial y})] \quad (10)$$

$$F_r = m^2[\frac{\partial}{\partial x}(P*K\frac{\partial r}{\partial x}) + \frac{\partial}{\partial y}(P*K\frac{\partial r}{\partial y})] \quad (11)$$

$$D_T = \frac{\partial}{\partial x}(\frac{m^2u}{P*}) - \frac{\partial}{\partial y}(\frac{m^2v}{P*}) \quad (12)$$

$$D_s = \frac{\partial}{\partial x}(\frac{m^2v}{P*}) + \frac{\partial}{\partial y}(\frac{m^2u}{P*}) \quad (13)$$

$$K = \frac{.1}{m^2}\sqrt{D_T^2 + D_s^2} \quad (14)$$

$$u = \frac{P*}{m^2}\frac{dx}{dt}, \quad v = \frac{P*}{m^2}\frac{dy}{dt} \quad (15)$$

$$m = \frac{80^2 + x^2 + y^2}{80^2}, \quad M = \frac{2m}{80^2} \quad (16)$$

$$f = 14.584 \times 10^{-5}(\frac{z-m}{m}) \quad (17)$$

$$\frac{\partial\phi}{\partial Q} = \frac{RT_v}{Q}, \quad \nabla\phi = -RT_v\Delta(\log Q) \quad (18)$$

$$T_v = (\frac{.622+r}{1+r})(\frac{T}{.622}) \quad (19)$$

$\Delta(P*T)_c \equiv$ heat convection where the lapse rate exceeds moist adiabatic (20)

$(P*r)_n \equiv$ moisture adjustment to avoid negative mixing ratio (21)

$$Q = P/P_x \quad (22)$$

u, v : GRID ORIENTED, PRESSURE WEIGHTED HORIZONTAL WIND COMPONENTS

T : TEMPERATURE

R : MIXING RATIO (MASS OF WATER VAPOR/ MASS OF DRY AIR)

P* : PRESSURE AT SEA LEVEL

TIME ADVANCE

$$\alpha_1 = \alpha_0 + \Delta t(\frac{\partial\alpha}{\partial t})_0 \quad t = 0$$

$$\alpha_{t+1} = \alpha_{t-1} + 2\Delta t(\frac{\partial\alpha}{\partial t})_t \text{ for all } t > 0$$

References

[1] Evensen, A. J. and Troy, J. L., "Introduction to the Architecture of a 288 Element PEPE," Proceedings of the 1973 Sagamore Computer Conference on Parallel Processing.

[2] Dingeldine, J. R., Martin, H. G. and Patterson, Wm., "Operating System and Support Software for PEPE," 1973 Sagamore Conference on Parallel Processing.

[3] Gates, W. L., BaHan, E. S., Kahle, A. B., and Nelson, A. B., "A Documentation of the Mintz-Arikawa Two Level General Circulation Model," APRA Document R-897-ARPA, December 1971.

[4] Daley, J. and Underwood, B. D., "Short Term Weather Prediction on the Illiac IV," 1975 Sagamore Computer Conference on Parallel Processing.

[5] Chapman, D. R., Mark, H. and Pirtle, M. W., "Computers Vs. Wind Tunnels for Aerodynamic Flow Simulations," Astronautics and Aeronautics, April 1975.

[6]. DiVecchio, M. C., "Design and Implementation of a High/Low Magnitude Search Instruction on PEPE," 1975 Sagamore Conference on Parallel Processing.

PEPE APPLICATION TO BMD SYSTEMS

Charles E. Blakely
System Development Corporation
Huntsville, Alabama 35805

Abstract -- The PEPE Development Program, for the past 4-1/2 years, has been concerned primarily with the design and development of an experimental hardware and software facility for conducting research on parallel and associative data processing techniques as applied to Ballistic Missile Defense (BMD) service. Preliminary investigative work on PEPE applications started in January 1974 employing functional simulation tools.

Concurrent with the functional simulations, an analytic benchmarking effort on the hardware was conducted. The benchmarks were primarily BMD routines to collect data on the hardware functions.

Some results of the PEPE benchmarking study for a Kalman Filter are presented and discussed in the paper. The PEPE results are compared with the results for several other computers benchmarked with the same filter.

The results of the BMD benchmark studies have provided data for comparing the performance of PEPE, CDC 7600, CDC 7700, TI-ASC and CRAY-1.

Curves are presented for the combined radar scheduling and object tracking function for the PEPE, CDC 7700, and the CRAY-1. These curves indicate that the PEPE can out perform the other computers for systems as small as 108 elements.

Introduction

The PEPE (Parallel Element Processing Ensemble) Development Program, for the past 4-1/2 years, has been concerned primarily with the design and development of an experimental hardware and software facility for conducting research on parallel and associative data processing techniques as applied to Ballistic Missile Defense (BMD) service. The experimental facility includes a partial PEPE machine and the support software system needed for coding, evaluating, and demonstrating experimental tactical processes operating on the machine in a simulated BMD environment.

Preliminary investigative work on PEPE applications work involved an analysis of the utility of PEPE as an adjunct to the CDC 7700 data processor. The study objective was to find out if it were possible to secure a moderate improvement in the data processing performance with no hardware modification other than the addition of PEPE to standard CDC 7700 interfaces, and with minimal change to existing software. The study successfully achieved its objective with a rather simple implementation of PEPE in a fast response "offloading" configuration, and led to another study to develop a more sophisticated, higher performance implementation, but still with the constraint that changes to existing software be minimal. Results of this study were demonstrated using functional simulations. This study successfully achieved its objectives in July 1975.

The above demonstration was followed by a study in which the minimum software breakage constraints were relaxed. The results of this study, which permitted a much higher performance implementation, were demonstrated by functional simulations.

Concurrent with the functional simulations, an analytic benchmarking effort on the hardware was conducted. The benchmarks were primarily BMD routines to collect data on the hardware functions. The results of the BMD benchmark studies have provided data for comparing the performance of PEPE and the CDC 7600, CDC 7700, TI-ASC and CRAY-1.

Benchmarks have also been run which permit a comparison of PEPE with the STAR-100, IBM 370/195, AMDAHL 470/V6 and the TI-ASC 4 pipe.

Curves have been derived for the combined radar scheduling and object tracking functions on the CDC 7700, PEPE and the CRAY-1. These data indicate that the PEPE can outperform the other systems with PEPE limited to 108 parallel elements.

Offload Studies and Results

A study was made to identify the heavy resource users in a BMD system, and to investigate their behavior throughout a threat. The six highest resource users were identified as:

Radar Interface Processing
Object Tracking Processing
Interceptor Control Processing
Track Initiate Returns Processing
Track Initiate Designation and Beam Pointing Processing
Passive Object Discrimination Processing

The three tasks with the most potential for implementing on PEPE were Radar Interface Processing, Object Tracking and Passive Discrimination. The main considerations in choosing these tasks were the CPU resources required by the tasks, commonality of data base, and inherent parallelism. The average CPU resource requirements for the three tasks were 28.32%, 21.39% and 9.4%, respectively, for Radar Interface Processing, Object Tracking and Passive Discrimination.

A design, implementing these three functions on PEPE, was generated and implemented in an existing functional simulator. The Object

Tracking and Passive Discrimination tasks were implemented in PFOR [1] (Parallel FORTRAN) directly on the parallel elements controlled by the Arithmetic Control Unit (ACU); however, the Radar Interface Processing (radar scheduling) had to be redesigned for implementation on PEPE. The redesigned task consisted of two parts: (1) a radar returns sorting task (SORT) and (2) a radar pulse schedular (SCHED). SORT was designed to be implemented on the Correlation Control Unit (CCU) while SCHED would operate on the Associative Output Control Unit (AOCU).

A PEPE/7700 system was designed with the radar data processing subsystem interface connected to the PEPE as shown in Figure 1. The Radar Interface, Object Track task and the Passive Discrimination function were implemented on the PEPE as described above. The incoming data were sorted in the PEPE CCU and those data not needed in PEPE were transmitted directly to the 7700.

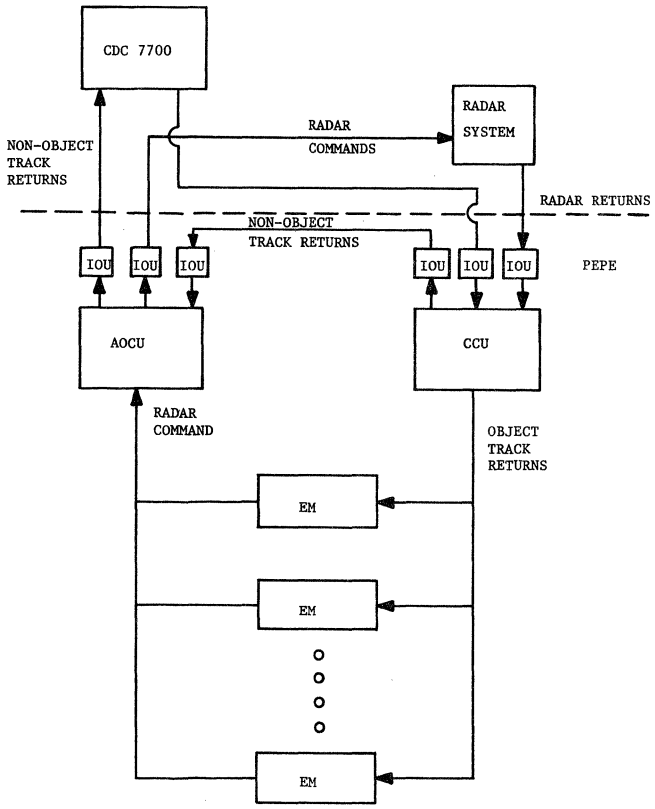


Figure 1. PEPE/7700 Data Flows

A simulation of the above system, employing the SDC PEPSIE executive, was implemented and driven by a simple threat generator. The results of the simulation runs indicate that the average 7700 CPU loading was reduced by 40%, queue build-ups were almost eliminated, 7700 CPU1 peak loading never exceeded 65% utilization, 7700 CPU2 peak loading never exceeded 80% and polling loops delays were significantly reduced for all polling loops. Additional statistics obtained from the simulator are shown in the table below.

	HOST ONLY	7700-PEPE	7600-PEPE
AVERAGE CPU 1 USAGE (%)	95.1	36.6	86.7
AVERAGE CPU 2 USAGE (%)	82.5	61.7	---
AVERAGE CPU USAGE	88.8	49.2	86.7
LCM USAGE (%)	17.4	13.2	12.6
LOOP 1 WAIT (MSEC)	1.419	1.257	1.747
LOOP 2 WAIT (MSEC)	1.882	.172	1.742
LOOP 3 WAIT (MSEC)	1.725	.268	3.473
LOOP 4 WAIT (MSEC)	3.011	.446	7.680
LOOP 6 WAIT (MSEC)	8.317	1.044	12.973
LOOP 7 WAIT (MSEC)	13.164	.499	20.527

All of the data in the table are average percent utilizations and wait times in milliseconds. Data with respect to maximum, minimum and standard deviation (σ) for CPU usage are presented in the following table.

	7700		PEPE/7700			
	CPU 1	CPU 2	CPU 1	CPU 2	7600	PEPE/7600
MAXIMUM	100%	100%	64.7%	81.5%	100%	100%
MINIMUM	65.2%	48.8%	16.4%	39.9%	100%	57%
AVERAGE	95.2%	82.8%	36.6%	61.7%	100%	86.7%
STD DEVIATION	8.72%	10.95%	9.89%	9.03%	0	11.1%

Continuous curves of CPU1 and CPU2 loading are shown in Figures 2 and 3, respectively. The unbalance between CPU1 and CPU2 loading was caused by dedicated tasks left on CPU2.

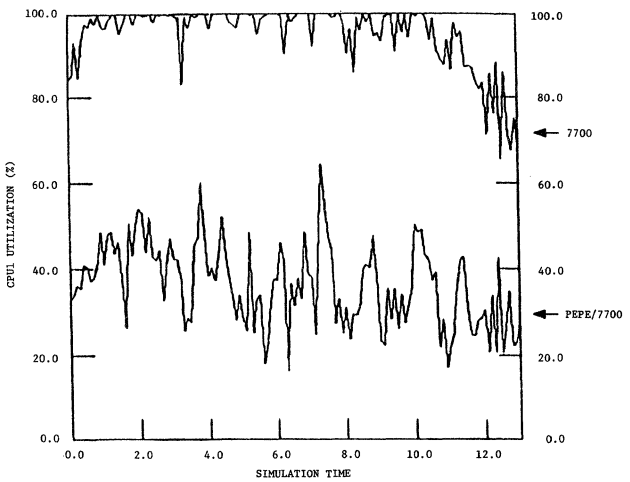


Figure 2. CPU1 Utilization Versus Time

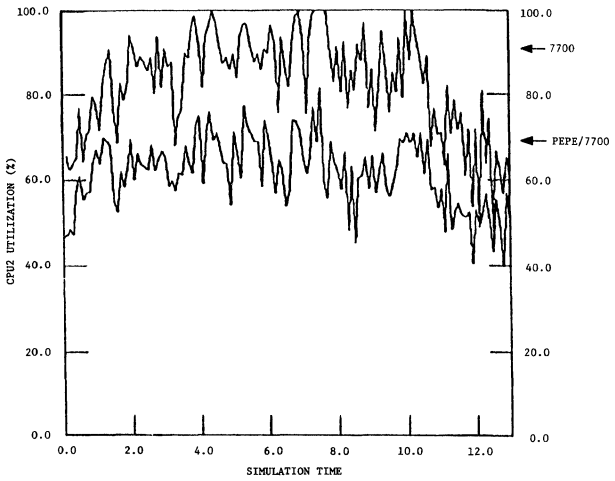


Figure 3. CPU2 Utilization Versus Time

Subsequent to the simulation study, the most important resource users implemented on PEPE have been benchmarked on the hardware. The results of these studies are presented in the next section.

BMD Benchmarks

The results of a PEPE benchmarking study for a Kalman Filter object tracking routine are presented in this section. The particular filter implementation employed in this study was a seven state fully coupled version developed by Teledyne Brown Engineering several years ago for a Computer Comparison study. The filter is referred to as the TKPRC subroutine throughout this paper.

The TKPRC subroutine was run on the TI-ASC and compared with the CDC 7600. The code was then extensively revised to adapt it to the ASC as a result of this Comparison [2]. The recoded version for the ASC showed a very substantial reduction in execution time, which was attributed to the increase in code vectorization. The subroutine executed in virtually a pure vector mode at this point.

The initial code as run on the ASC was then executed on the CDC 7600. The one exception was that the ASC FORTRAN extensions were replaced by standard FORTRAN statements for the CDC 7600. It is interesting to note that the execution time on the 7600 was approximately 20% less due to the vectorizing done for the ASC.

The filter implemented on the CRAY-1 was the version run on the CDC 7600. Thus the subroutine TKPRC was initially optimized for the TI-ASC when run on the CDC 7600 and the CRAY-1. The filter initially implemented on PEPE was the original unoptimized version of TKPRC. It was subsequently optimized for matrix operations as described in the next section.

Implementation On PEPE

The original version of the TKPRC subroutine was translated directly from the FORTRAN code to PFOR. The time required to plan the conversion, prepare the coding sheets, and obtain the first debug run was 8 hours.

When this version was run and timed on PEPE, the execution time was 6.98 milliseconds. An examination of the code revealed that the covariance matrix prediction process was extremely inefficient on PEPE. The covariance matrix prediction process was recoded in PFOR (not assembler) and new timing runs made. Subsequently, all the matrix operations were recoded. The recoding consisted of removing a total of 20 lines of FORTRAN code and adding 12 lines of new code. The results of this change was to reduce the run time to 4.10 milliseconds.

Results

Table I contains the results of running and timing the subroutine TKPRC on the PEPE hardware. Several interesting results are contained in Table I. The most impressive result is the large reduction in the number of sequential instructions executed when the matrix multiply was optimized (reduced to 4920 from 20,964). This largely accounted for the reduction from 6.8588 to 4.1446 milliseconds run time. Another interesting result is the 2.57 MIPS effective speed for the ACU. This result is due to the overlapping of the parallel and sequential instruction executions.

Table I. Data for One Cycle of Filter for N Objects

	ORIGINAL CODE	MATRIX MULTIPLY OPTIMIZED	BEST SO FAR
NO. OF PARALLEL INSTRUCTIONS EXECUTED	6349	5744	5656
NO. OF SEQUENTIAL INSTRUCTIONS EXECUTED	20964	4920	5684
RUN TIME	6.8588 MILLISEC	4.1446 MILLISEC	4.1036
PARALLEL INST. TIME	4.2435 MILLISEC	3.9870 MILLISEC	3.9699
SEQUENTIAL INST. TIME	2.1066 MILLISEC	0.5040 MILLISEC	0.4804
MIPS	3.98	2.57	2.52

Figure 4 shows a comparison of the run times for the CDC 7600, TI-ASC and the PEPE. The data for the CDC 7600 and the TI-ASC were taken from [2]. Figure 4 shows a crossover at 7 targets for the PEPE versus the CDC 7600. This means that the PEPE is more efficient for object tracking alone when the number of targets exceed 7. Stated another way, the full PEPE is 41 times as powerful as the CDC 7600 while tracking 288 targets. When compared with the TI-ASC, the crossover is at 8 targets or 36 times more powerful for a full target load.

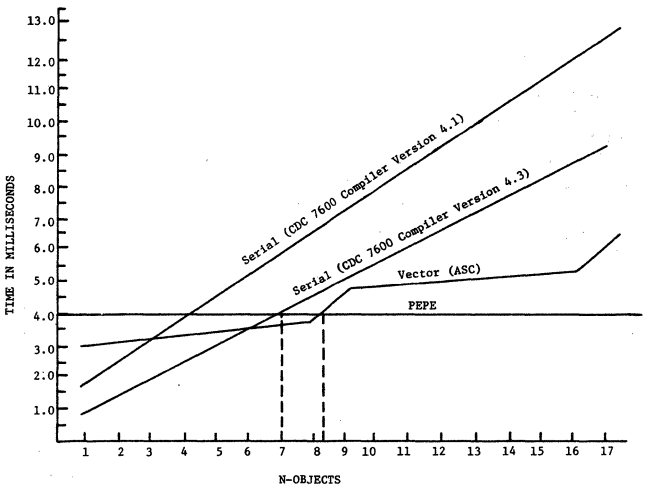


Figure 4. TRACK Benchmark - Subroutine TKPRC, Serial/Vector Crossover

The results for the TKPRC benchmark on the CRAY-1 are for the CDC 7600 optimized version of the filter [3]. That is, the filter on the 7600 would be approximately 15% better than the results from [2]. The filter was hand coded in assembler for the CRAY-1 since a FORTRAN Compiler did not exist at that time. Employing the results from [3] of $2.17*10^{-3}$ seconds for 64 objects, the PEPE is 2.4 times more powerful than the CRAY-1 for object track. The crossover is at 121 targets. These results are considered to be an optimistic upper bound for the CRAY-1 due to the fact that a Fortran compiler was not used in the tests.

Multiple Task Performance

The preceeding results were derived for a single function, object tracking, operating on each computer. The true performance of a computer can only be assessed when multiple tasks are present. In this section the preceeding results are extrapolated to the combined task of object tracking and radar scheduling. Since TKPRC track filter benchmark has been run on all the computers discussed in this paper, run time equations can be derived. The method for deriving timing estimates for the combined functions is discussed below.

Figure 5 contains the plotted data for TKPRC on the CRAY-1 computer. The data have been reduced to run times for one iteration versus N objects. The best fit to the data appeared to be a straight line of the form

$$y = m x + b.$$

The results for the data in Figure 5 are

$$y(\text{CRAY TKPRC}) = .2939 + .02923\ N$$

where y is the run time in milliseconds. The data in Figure 4 can also be fitted to derive a similar curve for the CDC 7600 [4]. The results were

$$y(\text{7600 TKPRC}) = .5 + .5N \text{ milliseconds.}$$

Run time estimates for the RADAR SCHEDULER function (operating on the 7600) were derived from the PEPE applications simulator results. The time required to consider M objects for scheduling by the Radar Interface Processing function (in milliseconds) was found to be

$$y(\text{7600 SCHED}) = .35889 + .1298\ M.^{(a)}$$

Run times for Radar Scheduler operating on the CRAY-1 were derived in the following manner. Assume that Radar Scheduler would be implemented in the sequential unit of the CRAY-1 since it does not appear to lend itself to vectorizing. The cycle time of the CRAY-1 is 12 nanoseconds which is 2.2 times as fast as the 27.5 nanoseconds for the 7600. Using these assumptions, the equation for the CRAY-1 Radar Scheduler run time in milliseconds is

$$y(\text{CRAY SCHED}) = .163 + .059\ M.$$

The above equations do not contain any allowances for differences in the number of machine cycles required to execute an instruction, overhead, interrupts, etc. Since the sequential and vector operations are mutually exclusive in the CRAY-1, the combined run time for Object Tracking and Radar Scheduling is given by

$$y(\text{SCHED + TKPRC}) = .02923\ N + .059\ M + .457.$$

The PEPE run time for the TKPRC track filter is a constant of 4.1 milliseconds for up to 288 objects or the maximum number of elements in the system. Therefore, the PEPE TKPRC equation, in milliseconds, is

$$y(\text{PEPE TKPRC}) = 4.1.$$

The PEPE run time for the Radar Scheduler function taken from the Simulator mentioned above is given by

$$y(\text{PEPE SCHED}) = .4 + .0432\ M.$$

(a) Subsequent studies have shown that the coefficient of M may be as large as .25.

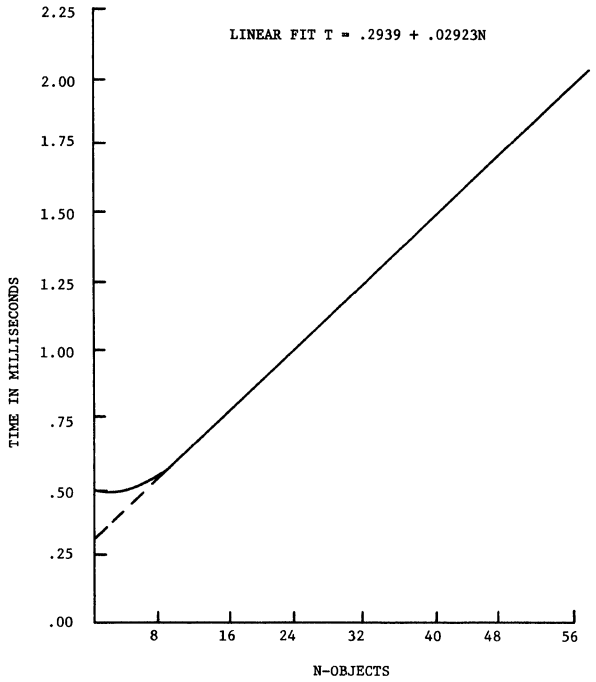


Figure 5. CRAY-1 Timing Data for the TKPRC Benchmark

The above equations have been plotted on Figure 6 as a function of the number of objects in track plus the number of instances inputted for radar scheduling [5]. The results indicate that a PEPE can outperform the other systems for systems as small as 3 cabinets (or 108-elements) of parallel elements. Larger systems permit large reserves for growth.

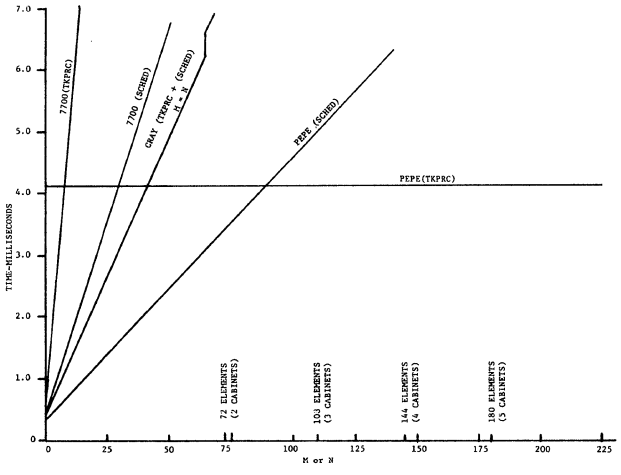


Figure 6. Combined TKPRC and TRIP Run Times

Matrix Factorization

A matrix factorization benchmark, employed in a benchmarking effort at the Systems Engineering Laboratory University of Michigan [6] was programmed and run on the PEPE. This benchmark is an example of the application of a problem, for which the machine was not designed, to the PEPE. The matrix factorization process requires the transfer of data between the elements which is an essentially serial process in the present design. Figure 7 shows the PEPE results superimposed on the University of Michigan results. It is evident from these data that PEPE is competitive on this problem for very large matrices. The fitted equation for these data is

$$T(\text{nanoseconds}) = 6544.28 + 5705.0N + 4417.86\ N^2$$

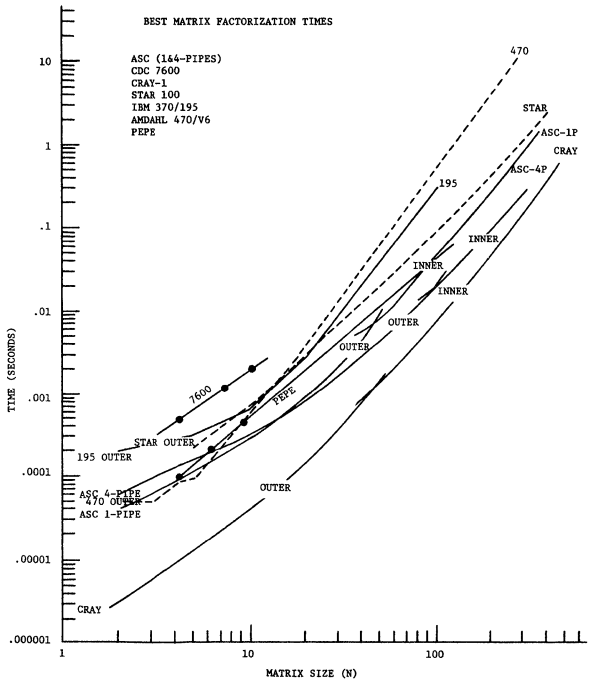


Figure 7. Best Matrix Factorization Times

Summary and Conclusions

It has been demonstrated through simulations and benchmarking that the PEPE can be introduced, as designed, into existing BMD systems and assume approximately one-half of the CPU load. If PEPE were designed into the system at the beginning, it is estimated that it could assume up to 75% of the CPU loading. The PEPE system provides for considerable growth by adding cabinets of parallel elements up to 288.

Benchmarking efforts are continuing to demonstrate the application to other areas of the BMD problem, such as the processing of optical data.

Note

The data and conclusions presented in this paper are the results of a preliminary evaluation effort. These conclusions do not represent an

official BMDSCOM position since further detailed studies are now underway utilizing a different design configuration.

References

[1] Dingeldine, J. R., "Parallel Fortran (PFOR), PEPE Assembly Language (PAL) User's Manual," Contract No. DAHC60-73-C-0060, System Development Corporation, TM-HU-046/400/01, 1 August 1976.

[2] Bakkegard, I. G., Eris, D., "ASC Timing Study," System Development Corporation, Contract No. DAHC-60-73-C-0094, TM-(L)-HU-180/000/00, 20 December 1974.

[3] Pessoney, M. D., "CRAY-1 Vector Code Benchmark Results," Analysis International Corporation, Project No. PF2174, Document No. HSV/76-001, 1 March 1976.

[4] Moore, P., "CRAY-1 Benchmark Report," System Development Corporation, Contract No. DASG6075-C-0047, Report No. TM-HU-215/000/00, 5 December 1975.

[5] Blakely, C. E., "PEPE Benchmark Studies: Object Tracking and Radar Scheduling," System Development Corporation, TM-HU-059/000/00, Contract Number DAHC60-73-C-0060, 16 May 1977.

[6] Calahan, D. A., Joy, W. N., Orbits, D. A., "Preliminary Report on Results of Matrix Benchmarks on Vector Processors," Dept. of Electrical and Computer Engineering, Systems Engineering Laboratory, The University of Michigan, SEL Report No. 94, 24 May 1976.

Bibliography

[1] SDC Tech Memo, "Preliminary Hardwite Demonstration (PHSD) PEPE Implementation Specifications," 1 April 1971, TM-WP-04/025/00.

[2] SDC Demonstration, Functional Simulation of PHSD on the MSI PEPE.

[3] TRW Systems Group, "PEPE Utilization Study," 1 July 1974, TSD-K4055, 74.6914.06-053.

[4] SDC, "PEPE Applications to Site Defense Preliminary Study Results, Phase I," 16 December 1974, TM-HU-049/000/00.

[5] TRW Systems Group, "PEPE Utilization Study Interim Report, Parts I, II & III," 1 April 1975, TSD-K5037*/PEPE 75.6914.06-039.

[6] SDC, Phase II PEPE-Site Defense Utilization Study Final Report, 30 July 1975, TM-HU-049/007/00.

[7] SDC, Phase II PEPE-Site Defense Utilization Study Interim Report, Parts I, II & III, 15 April 1975, TM-HU-049/005/00.

[8] TRW Systems Group, "Phase II PEPE Functional Simulator User's Manual," 1 August 1975, TSD-K5072*/PEPE, 75.6914.06-073.

[9] TRW Systems Group, "Phase II, PEPE Utilization Study Final Report," 1 August 1975, TSD-K5071*/PEPE, 75.6914.06-073.

[10] SDC TM, "PEPE Installation Requirements for Site Defense (SD) System Development Center (SDC)," 3 February 1975, TM-HU-049/001/00.

[11] SDC TM, "PEPE Installation Requirements for Site Defense (SD) KMR-Defense Unit (DU)," 26 February 1975, TM-HU-049/002/00.

[12] TRW Systems Group, "Phase II PEPE Utilization Study Extension Final Report," 1 December 1975, TSD-5109*/PEPE, 75.6914.06-111.

[13] SDC, "Parallel Element Processing Ensemble (PEPE) Program Plan," 1 May 1976, TM-HU-052/000/00.

[14] TRW System Group, "Phase III PEPE Functional Users Manual," 1 September 1976, No. 75-412.

[15] TRW Systems Group, "Phase II PEPE Utilization Study Extension Final Report," 29 November 1976, No. 75-412.

[16] SDC, "PEPE Large Host Applications," 1 December 1976, CDRL. No. B005, TM-HU-049/008/00.

[17] SDC, "PEPE Hardware Reference Manual," J. L. Troy, 1 February 1977, TM-HU-051/001/00.

[18] SDC, "Parallel Fortran (PFOR), PEPE Assembly Language (PAL) User's Manual," J. R. Dingeldine, 1 August 1976, TM-HU-046/400/01.

[19] SDC, "Version One Real-Time Operating System User's Manual - Preliminary," H. G. Martin, 2 March 1976, N-HU-00016/400/01.

[20] SDC, "System Functional Design Specification Volume II," TM-HU-048/001/03, 2 February 1977.

[21] SDC, "System Functional Design Specification Volume I," TM-HU-048/000/01, 13 April 1973.

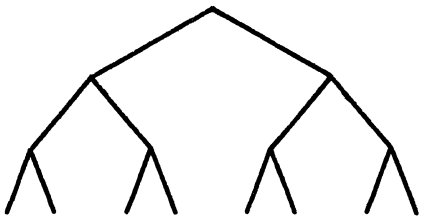
A PARALLEL PROCESSOR APPROACH FOR SEARCHING DECISION TREES

Duane David Marshall
System Development Corporation 205-837-7610
Huntsville, Alabama 35805

Abstract -- The use of a decision tree to represent a decision making problem is well-known. Current methods for examining the entire decision tree are too time consuming. One way to overcome this difficulty is to use a parallel processor computer. A brief description of the capabilities of a parallel associative processor is given and performance results for a tree search algorithm are included. Results indicate that the parallel approach examines the tree much faster than the previously used sequential algorithms.

Introduction

Many decision processes may be expressed as a decision tree as shown:



This tree is composed of a set of nodes and branches where each node represents the selection of an alternative and the branch leading from a node represents the decision made. Here, each decision is assumed to have only two possible alternatives. Since a tree with N decisions has a total of 2^N different combinations, a problem with more than 50 decisions is impractical to approach directly in this format.

In general terms, a problem has some objective or goal which the decision-maker sets. His difficulty lies in the fact that he is attempting to reach his goal while not breaking a set of predetermined restrictions. This problem is usually described as

$$\begin{array}{ll} \text{Maximize} & F(X) \\ \text{Subject to:} & \bar{G}(X) \leq \bar{B} \end{array}$$

where the problem is specified in terms of the decisions X.

A sequential algorithm for the solution of this problem was developed by Balas in 1965 [1] and uses the decision tree approach. Essentially, the algorithm searches the entire solution tree one node at a time to identify those decisions which allow the decision-maker to maximize his objective while remaining within the restrictions. As the algorithm progresses in its search, information collected about the problem is used to discard portions of the solution tree before they are examined in detail. Hence the name "implicit" enumeration. Studies have shown that implicit enumeration is practical only for problems with less than 100 to 150 decisions. Even for small problems with 50 variables or less, the solution time for the sequential algorithm may be several minutes.

The Problem

An example of a decision tree structure is to be found in the resouce allocation problem of assigning interceptors to targets or assigning returns to known targets. Leal [4] has described the use of a decision tree structure to the problem of interceptor allocation. He proposes that the decision tree formulation be used in combination with artificial intelligence techniques to "teach" a program to respond quickly during a BMD attack. This prior "learning" by the defense system monitor would allow many poor alternatives to be discarded and enable the system to perform at a high level of effectiveness. In order to develop this artificial intelligence program, the learning program must investigate the entire solution tree many times. However, since the current method used to search the decision tree is a sequential search over the tree where only one decision is examined at a time, solution times for a problem with many alternatives are excessive. Thus, any model of sufficient complexity to be useful will have a decision tree too large to be processed in a reasonable amount of time using a sequential computer.

A Parallel Processing Solution

Morefield [6,7,8] has proposed that one potential means to overcome this problem lies in the use of parallel processors to search the decision tree. In order to study the tree search methods using a parallel associative processor, an algorithm has been developed which examines many nodes of the solution tree simultaneously [5]. This algorithm is based on the sequential search method which has been studied extensively in the literature [1,2,3]. Essentially, the algorithm uses a modification of the sequential algorithm to generate many candidate nodes simultaneously. Each candidate is assigned to a processing element. All processors perform exactly the same arithmetic operations in examining their own candidates. Once a "good" solution is found, that information is shared among all the processing elements to efficiently search the tree. In general, the solution tree tends to grow exponentially as the algorithm progresses. A mechanism is included in the algorithm to recognize a limited number of

processing elements and to function within that limitation.

In order to study the parallel algorithm, a program has been written which simulates the gross functional characteristics of a parallel associative processor using the parallel algorithm. The basic criterion used for comparison of the sequential and parallel algorithms is the number of algorithm (sequential or parallel) iterations required to find and verify the existence of an optimal solution. Preliminary studies indicate that a parallel associative processor can solve these decision problems in a fraction of the time required by a sequential processor. Representative results with the parallel algorithm on a test problem of Petersen's [9] are given in Figure 1.

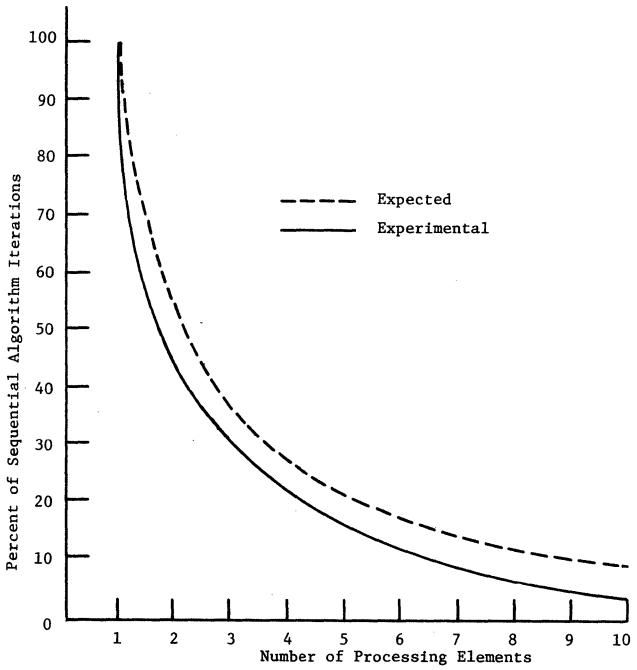


Figure 1. Performance of the Parallel Algorithm with a Variable Number of Processor Elements

Results

Table 1 shows the experimental estimates obtained for the solution of several test problems.

Table 1. Experimental Results

Problem	Number of Decisions	Sequential Algorithm	Parallel Algorithm (Number of PEs) 2	5	10
1	6	.002	.002	.001	.001
2	10	.006	.004	.003	.003
3	15	.037	.018	.010	.007
4	20	.160	.053	.020	.014
5	28	.672	.191	.057	.029
6	39	1.369	.600	.192	.099
7	50	13.052	5.742	2.129	.688

Estimated solution times were obtained by multiplying the sequential algorithm solution time by the ratio of the number of parallel algorithm iterations to the number of sequential algorithm iterations. Times were obtained using a CDC 7600 computer.

In general, the parallel solution method becomes more attractive as the number of processing elements increases. However, tests show that each problem has a certain limit beyond which the addition of processing elements has little effect on the algorithm performance. If the sequential algorithm required M iterations, a parallel associative processor with N processing elements would solve the problem in less than M/N iterations. In fact, as the number of decisions increases, the solution rate becomes much less than the M/N ratio. The reason for this lies in the way in which the decision tree is built as shown in Figure 2. The sequential algorithm essentially builds a "tall" tree in that one branch is examined in depth, whereas the parallel algorithm builds a "wide" tree in that many branches are examined simultaneously. The process of building a "wide" tree enables the parallel algorithm to discard "unfavorable" alternatives faster than the sequential algorithm. The performance of the sequential algorithm would be competitive with the parallel algorithm only if a "good" solution is found in the extreme upper left side of the decision tree, i.e., the sequential algorithm builds the decision tree top to bottom and left to right. The parallel algorithm builds the tree top to bottom.

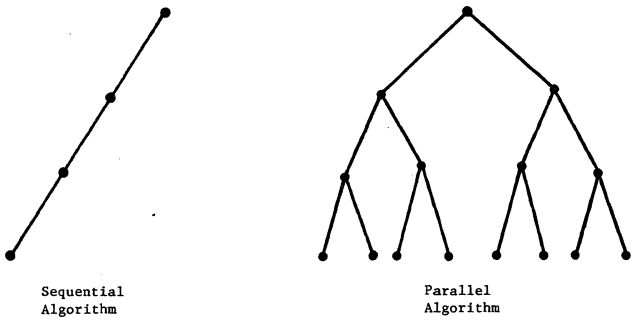
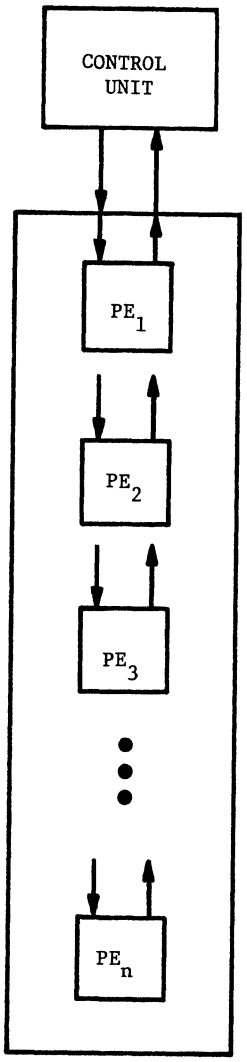


Figure 2. Comparison of Tree Construction After Only Three Algorithm Iterations

Machine Architecture

The architecture of a parallel processor to use the parallel algorithm is extremely simple. Essentially, the machine would consist of a control unit and an ensemble of processing elements as shown in Figure 3. The control unit would require a small memory and would need to block transfer a large number of words to all elements simultaneously. The processing elements would require a parallel indexing capability in order to implement the parallel algorithm efficiently. Element memory would be on the order

of one thousand (decimal) words. The ensemble should contain at least as many processing elements as the number of decisions in the original problem.



- Figure 3. Parallel Processor Architecture

Additional Questions

Additional research is required to describe the best way to use the parallel algorithm. Since each processing element is working its own independent problem, is there some way to share information to make the search more efficient? Can some internal measure of how efficiently the parallel algorithm is working be developed? When a "good" solution is found by one processor, how may the other processors make the most efficient use of this information? These are only some of many additional questions which remain to be considered concerning the parallel algorithm.

Conclusions

A parallel algorithm for use on a parallel associative processor has been developed to search through a decision tree. Preliminary experiments indicate that the solution times using this method are much better than those of the sequential method. This results enables large decision problems to be solved exactly rather than relying on some search heuristic which may or may not lead to an acceptable solution.

References

[]] Balas, E., "An Additive Algorithm for Solving Linear Programs with Zero-One Variables," Operations Research, July, 1965.

[2] Geoffrion, A. M., "Integer Programming by Implicit Enumeration and Balas' Method," SIAM Review, September, 1967.

[3] _____, "An Improved Implicit Enumeration Approach for Integer Programming," Operations Research, 1969.

[4] Leal, Antonio, "Adaptive Decisions in Ballistic Missile Defense," IEEE Transations on Systems, Man and Cybernetics, May, 1977.

[5] Marshall, D. D., "The Solution of 0-1 Programming Problems: A Parallel Processing Approach," Ph.D. Dissertation, The University of Alabama in Huntsville, Huntsville, Alabama, 1977.

[6] Morefield, C. L., "Solution of Multiple Choice Estimation Problems via 0-1 Integer Programming," IEEE Conference on Decision and Control, November, 1974.

[7] _____, "Applications of 0-1 Integer Programming to a Track Assembly Problem," IEEE Conference on Decision and Control, December, 1975.

[8] _____, "Application of Bayesian Decision Theory to Multitarget Surveillance Problems," Proceedings of the IEEE 1976 National Aerospace and Electronics Conference.

[9] Petersen, C. C., "Computational Experience with Variants of the Balas Algorithm Applied to the Selection of R&D Projects," Management Science, May, 1967.

PARALLELISM IN SORTING

Franco P. Preparata
Coordinated Science Laboratory*
University of Illinois
Urbana, Illinois 61801

Abstract

In this paper we describe a family of parallel sorting algorithms for a multiprocessors system. These algorithms are enumeration sorts, i.e., they are based on subdividing the keys into subsets and determining for each key the number of smaller keys (count) in every subset. The novelty is that parallel merging is used to implement the acquisition of the counts. By using Valiant's merging scheme, n keys can be sorted in parallel using $n\log_2 n$ processors in time $C\log_2 n$; if memory fetch conflicts are not allowed, then for $0 < \alpha \leq 1$ sorting on $n^{1+\alpha}$ processors runs in time $(C'/\alpha)\log_2 n + o(\log_2 n)$.

1. Introduction

The efficient implementation of comparison problems, such as merging, sorting, and selection, by means of multiprocessor computing systems has attracted considerable attention in recent years. One of the earliest fundamental results is due to K. E. Batcher [1], who proposed a sorting network consisting of comparators and based on the principle of iterated merging; as is well-known, such scheme sorts n keys with $O(n(\log n)^2)$ comparators in time $O((\log n)^2)$. Batcher's network is readily interpreted, in a more general framework, as a system of n/2 processors with access to a common data memory of n cells: obviously, the network structure induces a nonadaptive schedule of memory accesses. After the appearance of Batcher's paper, substantial work was aimed at filling the gap between the upper-bound $O((\log n)^2)$ on the number of steps which is achievable by a network of comparators and the lower-bound $O(\log n)$; the lack of success, however, convinced several workers to look for more flexible forms of parallelism.

The first scheme shown to sort n keys in time $O(\log n)$ is due to D. E. Muller and F. P. Preparata [2], but it requires a discouraging number of $O(n^2)$ processors. Subsequently, new results were obtained on parallel merging by F. Gavril [3]. L. G. Valiant [4] must be credited with addressing the fundamental question of the intrinsic parallelism of some comparison problems and with the development of faster algorithms than were previously known. In particular, in [4] he described an algorithm for merging with $\sqrt{nm}$ processors two sorted sequences of n and m keys, respectively, $(n \leq m)$, in $2\log\log n + O(1)$ [1] comparison steps; this algorithm can then be applied to sort n keys with n processors in $2\log n \cdot \log\log n + O(\log n)$ steps. His method assumes a computational model in which there is no penalty for memory-processor alignment and the overhead, corresponding to the reassignment of sets of processors to subsequences to be merged, is ignored.

A new family of sorting algorithms has been recently discovered by D. Hirschberg [5]. Assuming as a computation model a parallel processing system of the SIMD type (single-instruction stream, multiple-data stream) with random access capabilities to a common memory, Hirschberg shows that n keys can be sorted in time $O(k \log n)$ with $n^{1+1/k}$ processors, where k is an arbitrary integer ≥ 2. These schemes are not free of memory fetch conflicts (simultaneous reading of the same location by more than one processor) and Hirschberg poses as an open question the possibility of achieving analogous performances without memory fetch conflicts.

In this paper we shall present two results. The first, discussed in Section 2, is an algorithm for sorting n keys in time C log n (where C is a constant) with nlogn processors: this algorithm combines a number of known techniques, and makes crucial use of Valiant's merging algorithm. The second result (Section 3) is a family of very simple sorting algorithms, which have the same running time as Hirschberg's, but use basically different techniques and are entirely free of memory fetch conflicts. As our computation model we adopt a system of several identical processors, each capable of random-accessing a common memory with no alignment penalty. Store, fetch, and arithmetic operations have unit costs, and fetch conflicts are disallowed when appropriate.

All of the algorithms described in this paper - as well as Hirschberg's [5] - are instances of enumeration sorting, in Knuth's terminology ([6], p. 73). In these methods each key is compared with all the others and the number of smaller keys determines the given key's final

---

*Also, Departments of Electrical Engineering and Computer Science.

This work was supported in part by the National Science Foundation under Grant MCS76-17321 and in part by the Joint Services Electronics Program under Contract DAAB-07-72-C-0259.

[1] Throughout this paper "log" means "logarithm to the base 2".

position. Specifically, three distinct tasks are clearly identifiable in enumeration sorting algorithms:

(i) count acquisition. The set of keys is partitioned into subsets and for each key we determine the number of smaller keys in each subset (this informal description momentarily assumes that all keys are distinct);

(ii) rank computation. For each key the sum of the counts obtained in (i) gives the final position (rank) of that key in the sorted sequence;

(iii) data rearrangement. Each key is placed in its final position according to its rank.

Less informally, an enumeration sorting scheme has the following format, where we assume for simplicity that, for some given integer r, n=kr. Data structures to be used are arrays of keys. By A[i:j] we denote a sequence A[i]A[i+1]...A[j].

Input: A[0:n-1], the array of the keys to be sorted, integer r

Output: A[0:n-1], the array of the sorted keys.

1. begin Define $A_i[0:r-1] \leftarrow A[ir:(i+1)r-1]$, for i=0,...,k-1.

2. $$C_\ell^{(ij)} \leftarrow \begin{cases} |\{A_j(h) \mid A_j[h] \le A_i[\ell]\}| & \text{for } j < i \\ |\{A_j(h) \mid A_j[h] < A_i[\ell]\}| & \text{for } j > i \end{cases}$$

$$C_\ell^{(ii)} \leftarrow |\{A_i[h] \mid A_i[h] \le A_i[\ell], h < \ell\} \cup \{A_i[h] \mid A_i[h] < A_i[\ell], h > \ell\}|$$

3. $$\text{rank}(A_i[\ell]) \leftarrow \sum_{j=0}^{k-1} C_\ell^{(ij)}$$

4. $$A[\text{rank}(A_i[\ell])] \leftarrow A_i[\ell]$$

end

Note that count acquisition, rank computation, and data rearrangement are performed, respectively, in steps 2, 3, and 4. Also, the algorithm must insure that all ranks be distinct, which is a crucial condition for the data rearrangement task (otherwise memory store conflicts would occur). This clearly poses no problem when the keys are all distinct. In the opposite case, some convention must be adopted for the ordering of sets of identical keys. One such convention is that sorting be stable (see [6], p. 4), that is, the initial order of identical keys is preserved in the sorted array. Thus, all of our sorting schemes will be stable. This is reflected in the rules for the computation of the parameters $C_\ell^{(ij)}$ in Step 2 of the above algorithm.

The simple algorithm proposed by Muller and Preparata in [2] is a crude example of enumeration sorting, in which the sets A_i are chosen to be singletons. With this choice, each key is compared with every other key, thereby using $O(n^2)$ processors; similarly, rank computation uses $O(n^2)$ processors, since O(n) processors are assigned to each key. The time bound O(logn) is due to Step 3 (counting in parallel the number of 1's in a set of n binary digits), whereas Steps 2 and 4 run in constant time in our present model.

In the more complex procedures to be later described, the operations of rank computation and data rearrangement are essentially carried out as in the basic scheme described above. The main difference occurs with regard to count acquisition. In the Muller-Preparata method the counts are acquired by comparing each key with every other. The comparison of two keys A[i] and A[j] could be viewed as merging A[i] and A[j]. Suppose now that, rather than dealing with single keys, we deal with sorted sequences of keys $A_j[0:r-1]$ and $A_i[0:r-1]$, where r > 1 and, say j < i. We easily realize that the number of keys in $A_j[0:r-1]$ which are no greater than $A_i[\ell]$ (ℓ=0,...,r-1), as well as the number of keys in $A_i[0:r-1]$ which are less than $A_j[h]$ (h=0,...,r-1), can be obtained by merging the two sequences $A_i[0:r-1]$ and $A_j[0:r-1]$. In fact, let $B[0:2r-1]$ be the array obtained by merging the two sorted arrays $A_j[0:r-1]$ and $A_i[0:r-1]$, with the ordering convention $A_k[s] \le [s+1]$ (k=i,j) and $B[s] \le B[s+1]$. Suppose also that the merging be stable, that is, the order of identical keys in the concatenated array $A_j[0:r-1]A_i[0:r-1]$ is preserved in $B[0:2r-1]$. If $B[q] = A_i[\ell]$, then there are $(q-\ell)$ entries of $A_j[0:r-1]$ in $B[0:q-1]$ which are no greater than $A_i[\ell]$; similarly if $B[q] = A_j[h]$, then there are (q-h) entries of $A_i[0:r-1]$ in $B[0:q-1]$ which strictly less than $A_j[h]$. This is central idea of the algorithms to be described.

2. A Fast Parallel Sorting Algorithm

In this section we assume that in our computational model memory fetch conflicts are permitted. To provide the feature required by Valiant's merging algorithm, that a key be simultaneously compared with several other keys, we may assume that the processors have broadcast capabilities. The only overhead we shall neglect is the reassignment of processors to the operation of merging pairs of subsequences, as occurs in Valiant's method [4]. Notice that this model of parallel computation coincides with that required by Valiant's merging algorithm.

We assume inductively that the following algorithm, SORT1, for p < n requires at most $\lfloor p \log p \rfloor$ processors to sort p keys. Since SORT1 is recursive, the following presentation constitutes a constructive extension of the inductive step to the integer n. The induction can be started with n ≥ 4.

Algorithm SORT1

begin

1. $k \leftarrow \lceil \log n \rceil$, $r \leftarrow \lfloor n/\lceil \log n \rceil \rfloor$

2. Define arrays $S[0:k;0:k;0:2r-1]$ and $R[0:k;0:k;0:r-1]$ (three-dimensional arrays) and $A_i[0:r-1] \leftarrow A[ir:(i+1)r-1]$ $(i=0,\ldots,k-1)$, $A_k[0:n-kr-1] \leftarrow A[kr:n-1]$.

   Comment: When $n=kr$, array A_k is obviously vacuous. Array S is defined for simplicity as having $2r(k+1)^2$ cells, although the algorithm will only make use of the cells $S[i;j:q]$ for which $i < j$.

3. $A_i[0:r-1] \leftarrow SORT(A_i[0:r-1])$ $(i=0,\ldots,k-1)$
   $A_k[0:n-kr-1] \leftarrow SORT(A_k[0:n-kr-1])$.

   Comment: This step is a parallel recursive call of SORT1 and it involves sorting in parallel k sets of r keys each and, possibly, one set of (n-kr) keys. By the inductive hypothesis it uses at most $k\lfloor r\log r \rfloor + \lfloor (n-kr)\log(n-kr) \rfloor$ processors. Since $n-kr < \lceil \log n \rceil$, the number of processors used is less than $\lceil \log n \rceil \cdot \lfloor n/\lceil \log n \rceil \rfloor \cdot \log \lfloor n/\lceil \log n \rceil \rfloor \rfloor + \lfloor \lceil \log n \rceil \log \lceil \log n \rceil \rfloor$
   $\leq n\log(n/\lceil \log n \rceil) + \lceil \log n \rceil \log \lceil \log n \rceil$
   $= n\log n - \log \lceil \log n \rceil (n - \lceil \log n \rceil) \leq n\log n - 1$
   $\leq \lfloor n\log n \rfloor$, for $n \geq 3$. For the sake of uniformity, array A_k is now extended to size r, where each cell of $A_k[n-kr:r-1]$ is filled with a dummy sentinel larger than any key.

4. $S[i;j;0:r-1] \leftarrow A_i[0:r-1]$ $(i=0,\ldots,k-1;$ $j=i+1,\ldots,k)$
   $S[i;j;r:2r-1] \leftarrow A_j[0:r-1]$ $(i=0,\ldots,j-1;$ $j=1,\ldots,k)$
   Comment: This is a copying operation whose objective is to obtain $S[i;j;0:2r-1]=A_i[0:r-1]A_j[0:r-1]$ for all pairs (i,j) with $i < j$. In our model, this operation could be done with maximal parallelism. However, using only $\binom{k+1}{2}r$ processors, the $\binom{k+1}{2}2r$ elementary copying operations are completed in two time units. For later convenience we assume that the record associated with key $A_i[\ell]$ contains a LABEL consisting of the pair of integers (i,ℓ).

5. $S[i;j;0:2r-1] \leftarrow$ MERGE $(S[i;j;0:r-1], S[i;j;r:2r-1])$ $(i=0,\ldots,k-1;j=i+1,\ldots,k)$
   Comment: This step uses Valiant's merging algorithm and runs in time $C_1 \log\log r$, for some constant C_1, using $\binom{k+1}{2}r$ processors. The original version of Valiant's merging algorithm can be readily modified, so that, whenever two keys are identical the indices of their respective subarrays are compared.

6. Let $(x,\ell) \leftarrow$ LABEL $S[i;j;q]$
   If $x=i$ then $R[i;j;\ell] \leftarrow q-\ell$ else $R[j;i;\ell] \leftarrow q-\ell$
   $(i=0,\ldots,k-1;$ $j=i+1,\ldots,k;$ $q=0,\ldots,2r-1)$

7. $R[i;i;\ell] \leftarrow \ell$ $(i=0,\ldots,k;\ell=0,\ldots,r-1)$.
   Comment: Steps 6 and 7 complete the count acquisition task. In fact after Step 7 the content of $R[i;j;\ell]$ is $C_\ell^{(ij)}$, in the terminology of Section 1. Step 6 can be executed in two time units using $\binom{k+1}{2}r$ processors, whereas Step 7 uses $(k+1)r$ processors and runs in one time unit.

8. $rank(A_i[\ell]) \leftarrow \sum_{j=0}^{k-1} R[i;j;\ell]$ $(i=0,\ldots,k;$ $\ell=0,\ldots,r-1)$
   Comment: This step implements the rank computation. For any pair (i,ℓ) the sum can be computed with $\lfloor (k+1)/2 \rfloor$ processors in time $\lceil \log(k+1) \rceil \simeq \log\log n$. The total number of processors used is therefore $n\lfloor (k+1)/2 \rfloor$.

9. $A[rank(A_i[\ell])] \leftarrow A_i[\ell]$ $(i=0,\ldots,k;$ $\ell=0,\ldots,r-1)$

end

To complete the analysis of the algorithm, we observe that none of Steps 4-7 uses more than $\binom{k+1}{2}r$ processors, but

$$r\frac{k(k+1)}{2} = \lfloor n/\lceil \log n \rceil \rfloor \lceil \log n \rceil \left(\frac{\lceil \log n \rceil + 1}{2}\right) \leq n\frac{\lceil \log n + 1 \rceil}{2}$$

Also, Step 8 uses $n\lfloor (k+1)/2 \rfloor \leq n(\lceil \log n \rceil + 1)/2$

Since for all $n \geq 4$ $(n\lceil \log n \rceil + 1)/2 < \lfloor n\log n \rfloor$, the inductive hypothesis on the number of processors is extended.

Finally, let T(n) denote the running time of the algorithm for n keys. Since $r \simeq n/\log n$ we obtain

$$T(n) = T\left(\frac{n}{\lg n}\right) + C_2 \log\log n + C_3$$

for some constants C_2 and C_3. It is easily verified that a function of the form $C_2(\log n)+o(\log n)$ is a solution of the above recurrence.

3. Parallel Sorting Algorithms with no Memory Fetch Conflicts

We shall now consider a family of algorithms for sorting n numbers in parallel with $n^{1+\alpha}$ processors $(0 < \alpha \leq 1)$ in time $(C'/\alpha)\log n + o(\log n)$, for some constant C'. Each of these algorithms has the same performance as the corresponding algorithm by Hirschberg [5], although no memory fetch conflict occurs in this case. Again, we make the inductive hypothesis that for $p < n$, Algorithm SORT2 requires $p^{1+\alpha}$ processors to sort p keys. The format of SORT2 closely parallels that of SORT1, with a few crucial differences to be noted.

Algorithm SORT2

begin

1. $k \leftarrow \lceil n^{\alpha} \rceil$, $r \leftarrow \lfloor n/\lceil n^{\alpha} \rceil \rfloor$

2. Define arrays $S[0:k;\ 0:k;\ 0:2r-1]$, $R[0:k;0:k;0:r-1]$ and $A_i[0:r-1] \leftarrow A[ir:(i+1)r-1]$ $(i=0,\ldots,k-1)$, $A_k[0:n-kr-1] \leftarrow A[kr:n-1]$.

3. $A_i[0:r-1] \leftarrow$ SORT2$(A_i[0:r-1])$ $(i=0,\ldots,k-1)$

   Comment: This parallel recursive call of SORT2 sorts k sets of r keys each and, possibly, one set of $n-kr < k$ keys. By the inductive hypothesis, at most $kr^{1+\alpha} + (n-kr)^{1+\alpha} \triangleq N$ processors are used. Since $n-kr<k$, then $N<kr^{1+\alpha}+(n-kr)\cdot k^{\alpha} = kr(r^{\alpha}-k^{\alpha}) + n\cdot k^{\alpha}$. Also $kr=\lceil n^{\alpha} \rceil \cdot \lfloor n/\lceil n^{\alpha} \rceil \rfloor \leq n$, whence $N<n(r^{\alpha}-k^{\alpha}+k^{\alpha}) \simeq n\cdot n^{(1-\alpha)\alpha} = n^{1+\alpha-\alpha^2} < n^{1+\alpha}$, where we have used the approximation $r \simeq n^{1-\alpha}$. Steps 1-3 are analogous to the corresponding ones in SORT1; however, the copying operation implemented by Step 4 of SORT1 must be considerably modified, as shown by the following Steps 4-6, to avoid fetch conflicts. Here again, A_k is extended to size r as in SORT1.

4. $S[i;k;0:r-1] \leftarrow A_i[0:r-1]$ $(i=0,\ldots,k-1)$

   $S[0;j;r:2r-1] \leftarrow A_j[0:r-1]$ $(j=1,\ldots,k)$

5. for $m \leftarrow 0$ step 1 until $\lceil \log(k+1) \rceil - 2$ do

   $S[i;j-2^m;0:r-1] \leftarrow S[i;j;0:r-1]$

   $(j=k-2^m+1,\ldots,k;i=0,\ldots,j-2^m-1)$

   $S[i+2^m;j;r:2r-1] \leftarrow S[i;j;r;2r-1]$

   $(i=0,\ldots,2^m-1;j=i+2^m+1,\ldots,k)$

6. Let $\lceil \log(k+1) \rceil - 1 = v$.

   $S[i;j-2^v;0:r-1] \leftarrow S[i;j;0:r-1]$

   $(j=2^v+1,\ldots,k;i=0,\ldots,j-2^v-1)$

   $S[i+2^v;j;r:2r-1] \leftarrow S[i;j;r:2r-1]$

   $i=0,\ldots,k-2^v-1;j=i+2^v+1,\ldots,k)$

   Comment: Steps 4-6 jointly replicate each $A_i[0:r-1]$ the required number k of times. Step 4 is an initial copy; Step 5 consists of $(\log\lceil k+1 \rceil - 1)$ stages, each of which doubles the ranges of the indices; Step 6 accounts for the fact that k may not be a power of 2 and completes filling the array S. Clearly this copying operation is implemented in $\log\lceil k+1 \rceil + 1 \simeq \alpha\log n + 1$ time units. A straightforward analysis shows that the largest number of processors used in any of these stages is at most 5/16 of the total number $\binom{k+1}{2}2r$ of cells of S to be filled. It is also easily shown that $(5/16)\binom{k+1}{2}2r \simeq (5/16)(n^{\alpha}+1)n^{\alpha}\cdot n^{1-\alpha} < n^{1+\alpha}$ for any $n \geq 1$ and $\alpha > 0$.

7. $S[i;j;0:2r-1] \leftarrow$ MERGE $(S[i;j;0:r-1], S[i;j;r:2r-1])$

   $(i=0,\ldots,k-1;j=i+1,\ldots,k)$.

   Comment: This step uses a stable version of Batcher's merging algorithm [1], which is easily obtained by requiring that whenever two identical keys are encountered their subarray indices be compared. The following facts about Batcher's merging algorithm are well-known: (i) no fetch conflict occurs because at any stage (or, time unit) each key is compared with exactly one key; (ii) $\binom{k+1}{2}r \simeq \lceil (n^{\alpha}+1)n^{\alpha}/2 \rceil \cdot n^{1-\alpha} < n^{1+\alpha}$ processors are used; (iii) merging is completed in $\log r \simeq (1-\alpha)\log n$ time units.

8. Steps 8, 9, 10, and 11 of this algorithm are respectively identical to Steps 6, 7, 8, and 9 of SORT1 and are therefore omitted. The latter are clearly free of memory fetch conflicts. The analysis of SORT1 showed that at most $\max\left(\binom{k+1}{2}r, n\lfloor (k+1)/2 \rfloor\right)$ processors were used in any of those steps. In the present case, we have already shown that $\binom{k+1}{2}r < n^{1+\alpha}$; similarly we conclude $n\lfloor (k+1)/2 \rfloor \leq n(n^{\alpha}+1)/2 < n^{1+\alpha}$.

From the performance viewpoint, all steps of the algorithm require at most $n^{1+\alpha}$ processors, as postulated. This extends the inductive hypothesis on the number of processors used by the algorithm. As to the running time T(n), we note the following: Steps 4-6 jointly require $\alpha\log n+1$ time units; Step 7 requires $(1-\alpha)\log n$ time units; Step 10 requires $\alpha\log n$ time units; Steps 8,9, and 11 run in constant time. Since Step 3 is a recursive call of SORT2 on sets of $r \simeq n^{1-\alpha}$ elements, we obtain for T(n) the recurrence equation

$$T(n) = T(n^{1-\alpha}) + (C_1'\alpha + C_2')\log n + C_3'$$

for some constants C_1', C_2', and C_3'. It is easily verified that a function of the form $[(C_1'\alpha + C_2')/\alpha]\log n + o(\log n)$ is a solution of this equation, whence $T(n) \leq (C'/\alpha)\log n + o(\log n)$.

References

1. K. E. Batcher, "Sorting networks and their applications," Proc. AFIPS Spring Joint Computer Conference, Vol. 32, pp. 307-314, April 1968.

2. D. E. Muller and F. P. Preparata, "Bounds to Complexities of Networks for Sorting and for Switching," Journal of the ACM, Vol. 22, No. 2, pp. 195-201, April 1975.

3. F. Gavril, "Merging with parallel processors," Comm. ACM, Vol. 18, 10, pp. 588-591, October 1975.

4. L. G. Valiant, "Parallelism in Comparison Problems," SIAM Journal of Computing, Vol. 4, 3, pp. 348-355, September 1975.

5. D. S. Hirschberg, "Fast Parallel Sorting Algorithms," Tech. Rep., Department of Electr. Eng., Rice University, Houston, Texas, January 1977.

6. D. E. Knuth, The Art of Computer Programming. Vol. III: Sorting and Searching, Addison-Wesley, Reading, Mass., 1972.

A PARALLEL TRIANGULATION PROCESS FOR SPARSE MATRICES*

Omar Wing and John W. Huang
Department of Electrical Engineering
and Computer Science
Columbia University
New York, New York 10027

Abstract

We consider the problem of triangulating a sparse matrix in a number of steps such that in each step all of the arithmetic operations that can be done in parallel are so executed. Our object is to minimize the number of such steps and at the same time to minimize the number of such operations. These two requirements are not compatible and both depend on the ordering of the matrix. A reordering algorithm which is a compromise is proposed. For a given ordering, an algorithm to sequence the operations in order to complete the triangulation in minimal number of steps is presented and bounds on the number of processors required are given. Experimental results on matrices of order 500 are reported.

Background

The triangulation of an nxn sparse matrix $A = [a_{ij}]$ consists of a series of steps each of which requires one of the following two sets of arithmetic operations:

For $k = 1, 2, \ldots, n-1$ and for each $a_{kj} \neq 0$

$$a_{kj} \leftarrow a_{kj}/a_{kk} \quad , \qquad j > k \tag{1}$$

and for each pair $a_{ik} \cdot a_{kj} \neq 0$

$$a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj} \qquad \begin{matrix} i > k, \\ j > k \end{matrix} \tag{2}$$

A total of $2n - 2$ sequential steps are required to triangulate A. We shall call (1) the divide operation and (2) the "update" operation. In (2) if $a_{ij} = 0$ but $a_{ik}\, a_{kj} \neq 0$, a fill-in is generated. It is obvious that if we have sufficient number of processors, the divide operations for each row k can be done in parallel. Also, for each k, the update operations for all pairs $a_{ij}\, a_{kj} \neq 0$ can be done in parallel. Moreover, if A is very sparse, it is possible that the divide and update operations of several rows be done at the same time. The total number of steps to triangulate A might therefore be less than $2n - 2$.

*Research supported by NSF ENG-76-02870.

For example, in the following matrix

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} x & x & & x \\ & x & x & \\ & x & x & x \\ x & x & x & x \end{bmatrix} \end{array} \tag{3}$$

the divide operations of rows 1 and 2 can be done simultaneously in step 1. In step 2, the update operations of a_{42}, a_{44} and a_{33} can be done at the same time. In step 3, a_{34} is divided by a_{33} and a_{34} is updated. Finally in step 4 a_{44} is updated. Thus the matrix is triangulated in four steps.

In this paper, we propose an algorithm to compute the minimum number of steps for triangulating a sparse matrix once the ordering of A is given. We next give upper and lower bounds on the number of processors required. Lastly, we present an algorithm to find an optimal ordering of A to meet the two requirements of minimization of the number of arithmetic operations and the number of triangulation steps.

Triangulation Graph

In order to expose the parallelism among the divide and update operations, we use a unit-execution-time model [1] to represent the triangulation process. This model is defined as a directed, acyclic graph, G(V,E), with node set V and arc set E defined as follows.

$$V = \{ v_i \mid v_i \text{ represents either a divide or update operation, } i = 1, 2, \ldots, m\}$$

The total number of operations is m and we assume that it takes a processor one time unit to execute an operation, and no preemption is allowed.

$$E = \{ (v_i, v_j) \mid v_i \in V,\ v_j \in V,\ i \neq j, \text{ and the operation represented by } v_j \text{ needs the results of the operation represented by } v_i.\}$$

In the graph, an arc goes from v_i to v_j.

Implicit in the definition of E is a set of precedence relations which are specified by a sequential description of the divide/update operations. Corresponding to each sequential description is a graph G(V,E), which we shall henceforth call a triangulation graph.

As an example, consider the following matrix

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} x & & & & x & & \\ & x & x & & & & \\ & x & x & & & x & \\ & & & x & & x & x \\ x & & & & x & & x \\ & & x & x & & x & \\ & & & x & x & & x \end{bmatrix} \end{array} \qquad (4)$$

One possible sequence of operations to triangulate the matrix is listed in Table 1. Its triangulation graph is shown in Fig. 1.

The parallelism that exists among the operations is now clear from the graph. Operations 1, 3, 8 and 7 can be done in parallel in one step. Operations 2, 12, 4 and 11 can be done in parallel in the next step, and so forth. It is also clear from the graph that the minimum number of time steps to triangulate the matrix is 7.

We shall define the length of a path of G(V,E) as the number of nodes from the starting node to the end node and we let D denote the length of the longest path of G. D then is the minimum number of steps to complete the triangulation according to a sequence of operations implied by E of G(V,E). For the example shown, D = 7. The triangulation in the order given results in two fill-ins.

Now suppose that we reorder the matrix as follows.

$$\begin{array}{c} \\ 1 \\ 5 \\ 7 \\ 4 \\ 6 \\ 3 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 5 & 7 & 4 & 6 & 3 & 2 \end{array} \\ \begin{bmatrix} x & x & & & & & \\ x & x & x & & & & \\ & x & x & x & & & \\ & & x & x & x & & \\ & & & x & x & x & \\ & & & & x & x & x \\ & & & & & x & x \end{bmatrix} \end{array} \qquad (5)$$

No fill-ins are created but now D = 12 if we sequence the operations row by row from top to bottom and column by column from left to right. Thus the minimum number of time steps and the number of fill-ins and hence arithmetic operations depend on the ordering of the matrix.

Rearrangement of Nodes

In Fig. 1, both v_6 and v_9 are update operations on a_{66}. From the expressions:

$$v_6: \quad a_{66} \leftarrow a_{66} - a_{63}\, a_{36} \qquad (6)$$

$$v_9: \quad a_{66} \leftarrow a_{66} - a_{64}\, a_{46} \qquad (7)$$

we see that regardless in which order these two operations are executed, the final a_{66} used in v_{15} will be the same. If we remove v_9 from the longest path and place it between v_7 and v_6, we obtain the same final triangulation but D will be reduced by 1. In the sequential description of the operations, operation 6 of Table 1 is now placed after operation 9. The new graph is shown in Fig. 2. Thus by postponing operation 6 we succeed in reducing the triangulation steps by one.

In general, before a matrix element is subject to a divide operation, it may have to be updated several times, i.e.

$$a_{ij} \leftarrow a_{ij} - \sum_{k=1}^{\ell} a_{ik}\, a_{kj} \qquad (8)$$

Numerically, it does not matter which product in the summation is subtracted from a_{ij} first. Our object is to find a sequence which minimizes D. In the next three sections, we describe how this is done.

Depth of a Node

In G(V,E) we shall call a node without any predecessor an initial node. Nodes v_1, v_8, v_3 and v_7 in Fig. 2 are initial nodes. The operations they represent are available for execution in the first time step.

We define the depth d_r of node v_r as the maximum path length from an initial node to v_r, and it signifies the earliest time at which the operation of v_r can be executed. The depths of the nodes of Fig. 2 are shown in Table 2. Clearly the depth of the termination node of G is D.

Operation Set and Depth Set

We have seen that a matrix element a_{ij} is in general subject to a series of update operations. It is convenient to associate with each a_{ij} an operation set Ω_{ij} and a depth set Δ_{ij} defined as follows.

Ω_{ij}: $v_k \in \Omega_{ij}$ if and only if operation v_k applies to a_{ij}, and we write $\Omega_{ij} = \{ \ldots v_k, \ldots, v_j \ldots \}$ if operation v_k precedes operation v_j.

Δ_{ij}: $d_k \in \Delta_{ij}$ if d_k is the depth of $v_k \in \Omega_{ij}$.

For example, from Table 1 and Fig. 1 we have

$\Omega_{15} = \{ v_1 \}$ $\Delta_{15} = \{ 1 \}$
$\Omega_{33} = \{ v_4 \}$ $\Delta_{33} = \{ 2 \}$
$\Omega_{66} = \{ v_6, v_9 \}$ $\Delta_{66} = \{ 4,5 \}$
$\Omega_{77} = \{ v_{12}, v_{14}, v_{16} \}$ $\Delta_{77} = \{ 2,4,7 \}$

On the other hand, in Fig. 2, we have

$\Omega_{66} = \{ v_9, v_6 \}$ $\Delta_{66} = \{ 2,4 \}$
$\Omega_{77} = \{ v_{12}, v_{14}, v_{16} \}$ $\Delta_{77} = \{ 2,4,6 \}$

Thus, by altering the sequence of operations of Ω_{66} the depth associated with each operation may be changed and in this case the final D is reduced from 7 to 6.

Given an ordering of the rows, the order in which the matrix elements are processed is fixed. However, within the set of operations applied to each matrix element it is possible to vary the sequence of update operations to obtain a different triangulation graph, and hence a different number of steps necessary to triangulate the matrix.

Minimal D

In the following, we give an algorithm by which for a given ordering of A the update operations in each Ω_{ij} are sequenced such that D is minimized. The sets Ω_{ij} and Δ_{ij} are constructed at the same time as the ordering of A is generated. We denote the ordering of the rows by p(1), p(2), ..., p(n).

Algorithm 1

(i) Input: Matrix A. We assume all diagonal elements and pivots are nonzero.

(ii) Output: An ordered sequence of operations in each Ω_{ij}, the depth sets and D.

(iii) Initialization: $\Omega_{ij} = \{ \phi \}$ for all i,j
$\Delta_{ij} = \{ 0 \}$ for all i,j
$r = 0$

(iv) Procedure I

```
begin
  for k ← 1 step 1 until n-1 do
    begin
      call Procedure II (described in Algorithm 2) to determine the k^th
           pivoting row; let it be row q;
      p(k) ← q ;
      for each j such that a_qj ≠ 0 and
      j ∉ {p(1),p(2),...,p(k)}
      do
        begin
          r ← r+1 ;
          v_r ← a label assigned to the divide operation on a_qj ;
          d_r ← Max { Δ_qq ∪ Δ_qj } + 1 ;    (A)
          Δ_qj ← Δ_qj ∪ { d_r } ;
          Ω_qj ← Ω_qj ∪ { v_r }
        end
      for each i such that a_iq ≠ 0 and
        each j such that a_qj ≠ 0 and
        i,j ∉ {p(1),p(2),...,p(k)}, do
        begin
          r ← r+1 ;
          v_r ← a label assigned to an update operation on a_ij ;
          d_r ← Max { Δ_iq ∪ Δ_qj } + 1 ;    (B)
          while ∃ d' ∈ Δ_ij , ∋ d'=d_r do
            begin
              d_r ← d_r + 1                  (C)
            end
          call Insert(d_r,Δ_ij) (Insert d_r
               into Δ_ij so that d ∈ Δ_ij
               are in ascending order.);
          call Insert(v_r,Ω_ij) (Insert r
               into Ω_ij so that v_r ∈ Ω_ij
               are in the same order as
               d ∈ Δ_ij.)
        end
    end
      p(n) ← q such that q ∉ {p(1),p(2),
      ...p(n-1)};
      D ← Max { Δ_p(n),p(n) }
end
```

(v) Comments:

Statement (A) signifies that the divide operation on an element a_{qj} must take place at least one time step later than the latest operation on the pivot or itself.

Statement (B) says that an update operation

on a_{ij} should take place at least one step later than the latest operation on a_{iq} or a_{qj}, irrespective when the previous operation on a_{ij} took place. In this way we are guaranteed that the last operation on a_{ij} is completed at the earliest time step. Now, if in Δ_{ij} there is already an element with value d_r, then d_r is increased by one until it is different from every $d \in \Delta_{ij}$. This step is necessary because no two operations can be applied to the same matrix element at the same time. The while-loop of (C) accomplishes this.

A formal proof that D obtained from Algorithm 1 is minimal is long and would require the introduction of additional new concepts. It will not be given in this paper. A plausible argument that the algorithm does produce a minimal D is that each of the three key steps of the algorithm ensures that every operation on a matrix element is assigned the smallest possible depth.

For the matrix of (4), if $p(k) = k$ as shown, D is found to be 6 by Algorithm 1, and the triangulation graph is that of Fig. 2. From the graph, it is not difficult to determine that at least three parallel processors are required to triangulate the matrix in 6 steps. An optimal schedule using three processors is shown in Fig. 3. In this schedule, all operations having the same depth are not executed at the same time. In general, the scheduling problem is NP-complete [2]. In the following, we consider the bounds on the minimum number of processors.

Bounds on the Number of Processors

We define the level number, u_r, of a node v_r of G(V,E) as:

$$u_r = D + 1 - \underset{i}{\text{Max}} \{ p_i | p_i \text{ is a path length from } v_r \text{ to the termination node.} \}$$

The level numbers of the nodes of Fig. 2 are shown in Table 2. Each level number indicates the last time step by which the operation represented by v_r must be executed, if the triangulation is to be completed in D steps.

Let the number of nodes in G(V,E) which have the same level number, say i, be f_i. Then an upper bound on the number of processors to triangulate the matrix in D steps is clearly

$$B_u = \underset{i}{\text{Max}} \{ f_i | i \in \{1,2,\ldots,D\} \} \qquad (9)$$

A lower bound can be defined as

$$B_f = \text{Max} \left\lceil \frac{\sum_{k=1}^{i} f_k}{i} \right\rceil \quad i = 1,2,\ldots,D \qquad (10)$$

The meaning of B_f as defined is as follows. From the previous comments on level numbers, for each step i, $i = 1,2,\ldots,D$, all $\sum_{k=1}^{i} f_k$ operations must be completed in i steps, and at least $(\sum_{k=1}^{i} f_k)/i$ processors are needed. The maximum of the last quantity is then a lower bound on the number of processors required.

Applying (9) and (10) to the graph of Fig. 2, we get $B_u = 4$ and $B_f = 3$. An optimal schedule using three processors is shown in Fig. 3.

Reordering Algorithm

We noted earlier that the number of triangulation steps D depends on the ordering of the rows of the matrix. For a given ordering, the total number of operations is fixed. It appears therefore that the smaller D is, the more operations can be done in parallel. However, the strategy to minimize D alone would leave the generation of fill-ins and hence the number of arithmetic operations uncontrolled, and it is possible that the number of parallel processors required is unreasonable. Also, as the matrix begins to fill, less and less arithmetic operations of different pivoting rows can be done in parallel and D would increase.

Clearly the best that one can do realistically is to do as much local minimization as possible [3]. It is not easy to discern the relationship between D and reordering, and we propose the following scheme as a reordering algorithm.

Algorithm 2

(i) Input:	Matrix A; the order of A, n; the first k-1 pivoting row numbers, p(1),p(2),...,p(k-1); and the depth sets of a_{ij} that have been constructed up to the time this procedure is called.
(ii) Output:	The next pivoting row number, q.

(iii) Initialization: Relative weight assigned to minimization of arithmetic operations, W_a; and relative weight assigned to minimization of D, W_b.

(iv) Procedure II

```
begin
  for each j such that j ∉ {p(1),p(2),
  ...,p(k-1)} do
    begin
      O(j) ← number of update/divide oper-
             ations if row j is picked
             as the next pivot ;
      for each m ∉ {p(1),p(2),...,p(k-1),
      j} do
      begin
      L(j) ← Σ_m (Max {Δ_jm} + Max {Δ_mj})
             + Max {Δ_jj}                (A)
               or
      L(j) ← Max_m {Δ_jm ∪ Δ_mj ∪ Δ_jj} ; (B)
      C(j) ← W_a·O(j) + W_b·L(j) ;
      find j* such that C(j*) =
      Min_j {C(j)} ;
      q ← j*
      end
    end
end
```

Note that in statements (A) and (B), the depth sets are the current ones that have been constructed up to the time the procedure is called, but this is the best that one can do since we are only interested in local minimization. The quantity L(j) has the significance that intuitively, among the pivot candidates which call for about the same number of update/divide operations, the one which requires the least number of time steps to complete all of the update operations, up to the time the procedure is called, of the elements on the pivoting row and column should be chosen as the next pivot.

By adjusting the weights W_a and W_b, different degrees of importance are assigned to operation count and depth. If $W_b = 0$, then the reordering algorithm becomes Markowitz algorithm [4]. If $W_a = 0$, reordering is based on consideration of reducing D alone.

Experimental Results

The proposed reordering algorithm was applied to a number of sparse matrices of order ranging from 100 to 500. The results of four cases are given in Tables 3 to 6. The matrix of Table 3 has a bandwidth much larger than that of the matrix of Table 4. Tables 3 and 5 refer to the same matrix except that L(j) of option (A) is used in Table 3 and option (B) is used in Table 5. Similarly for Tables 4 and 6. The following remarks can be made.

(1) The depth D in all cases is a small fraction of 2n-2, in fact, much smaller than n. This is to say that maximum utilization of parallelism would significantly reduce the number of sequential steps necessary to triangulate A. For example, in Table 3, column 3, we see that 26895 steps are necessary to triangulate A if only one processor is used. The matrix can be triangulated in 91 steps, but more than 378 processors would be required.

(2) As W_b/W_a increases, thus giving more relative importance to minimization of D, more fill-ins are generated. D does decrease for the case with small bandwidth. If the bandwidth is large, the reduction in D is counterbalanced by increase of fill-ins.

(3) While there is no rational basis to determine what the best ratio of W_b to W_a should be, our experiments indicate that going to the extreme of $W_b/W_a = 10^4$ does not result in significant decrease of D. It seems that a good reordering algorithm is one which uses Markowitz's scheme to find the pivots and uses L(j) to break a tie, if any.

(4) The results of the experiments are not sensitive to whether option (A) or (B) is used.

Conclusion

We have shown that a high degree of parallelism exists among the operations in the triangulation process of a sparse matrix. Recognizing this, we take advantage of as much parallelism as possible in every step of the process. In contrast with array processing of matrices [5], our approach assigns operations on matrix elements on different rows to parallel processors. In this way we are able to reduce the total number of steps to triangulate the matrix, and as we have seen, the reduction is dramatic and the number can be significantly smaller than the order of the matrix.

The important results of this paper are: (1) an algorithm to sequence the operations of a triangulation process so as to minimize the number of time steps required; (2) a lower and upper bound on the number of parallel processors required in order to triangulate a matrix in minimal number of time steps; and (3) an algorithm to reorder the matrix to obtain local minimization of the number of operations and the number of triangulation steps.

References

[1] R. Sethi, "Algorithms for minimal-length schedules," in Computer and Job/Shop Scheduling Theory, E.G. Coffman Jr. (ed.), Wiley-Interscience, N.Y., 1976, pp.51-98.

[2] J.D. Ullman, "NP-complete scheduling problems," J. of Computer and System Sciences, vol. 10, 1975, pp.384-393.

[3] I.S. Duff, "A survey of sparse matrix research," Proceedings of the IEEE, vol. 65, no. 4, April 1977, pp.500-535.

[4] H.M. Markowitz, "The elimination form of the inverse and its applications to linear programming," Management Science, vol. 3, 1957, pp.258-269.

[5] D.A. Calahan, "Parallel solution of sparse simultaneous linear equations," Proc. 11th Allerton Conference on Circuit and Systems Theory, 1973, pp.729-735.

label	operation
1	$a_{15} \leftarrow a_{15} / a_{11}$
2	$a_{55} \leftarrow a_{55} - a_{51} \cdot a_{15}$
3	$a_{23} \leftarrow a_{23} / a_{22}$
4	$a_{33} \leftarrow a_{33} - a_{32} \cdot a_{23}$
5	$a_{36} \leftarrow a_{36} / a_{33}$
6	$a_{66} \leftarrow a_{66} - a_{63} \cdot a_{36}$
7	$a_{46} \leftarrow a_{46} / a_{44}$
8	$a_{47} \leftarrow a_{47} / a_{44}$
9	$a_{66} \leftarrow a_{66} - a_{64} \cdot a_{46}$
10	$a_{67} \leftarrow a_{67} - a_{64} \cdot a_{47}$
11	$a_{76} \leftarrow a_{76} - a_{74} \cdot a_{46}$
12	$a_{77} \leftarrow a_{77} - a_{74} \cdot a_{47}$
13	$a_{57} \leftarrow a_{57} / a_{55}$
14	$a_{77} \leftarrow a_{77} - a_{75} \cdot a_{57}$
15	$a_{67} \leftarrow a_{67} / a_{66}$
16	$a_{77} \leftarrow a_{77} - a_{76} \cdot a_{67}$

Table 1

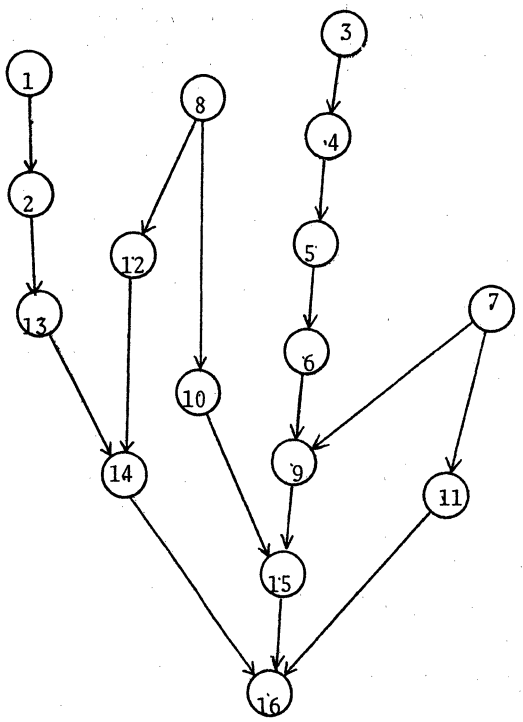


Figure 1

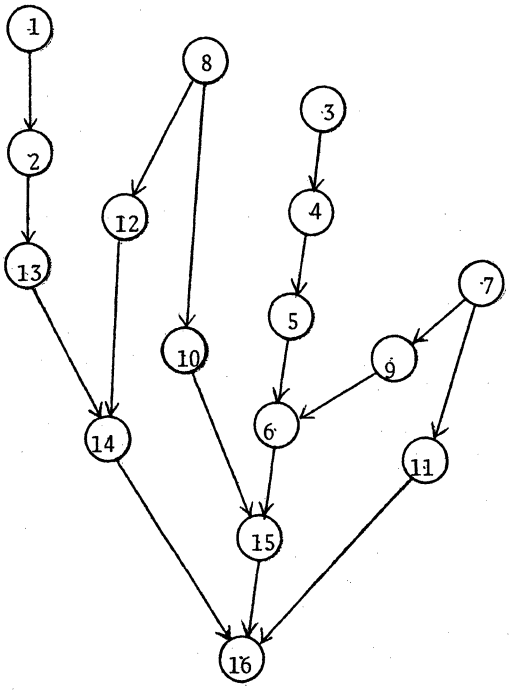
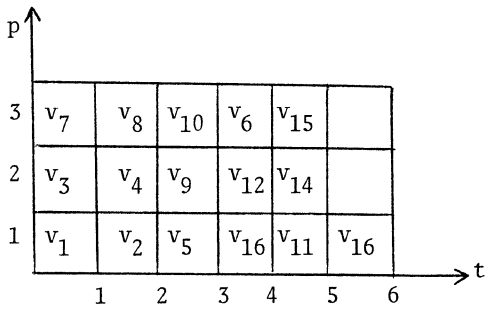


Table 2

label (r)	operation on	d_r	u_r
1	a_{15}	1	2
2	a_{55}	2	3
3	a_{23}	1	1
4	a_{33}	2	2
5	a_{36}	3	3
6	a_{66}	4	4
7	a_{46}	1	2
8	a_{47}	1	3
9	a_{66}	2	3
10	a_{67}	2	4
11	a_{76}	2	5
12	a_{77}	2	4
13	a_{57}	3	4
14	a_{77}	4	5
15	a_{67}	5	5
16	a_{77}	6	6

Table 2



An optimal schedule of operations of matrix A using 3 processors.

Figure 3

Order of matrix: 500
Number of nonzero elements: 2060
Density: 0.82%

Operation weight	100.0	100.0	1.0	0.01
Depth weight	0.0	0.1	1.0	100.0
Fill-ins	2675	2630	2866	3594
Total nonzeros	4735	4690	4926	5654
Total operations	25943	24811	26895	36308
Depth	95	94	91	93
B_m	1143	1089	1138	1395
B_u	573	575	604	754
B_f	341	329	378	523
Average: Total op./D	273.1	263.9	295.5	390.4

Table 3

Order of matrix: 500
Number of nonzero elements: 1952
Density: 0.78%

Operation weight	100.0	100.0	1.0	0.01
Depth weight	0.0	0.1	1.0	100.0
Fill-ins	933	982	1159	1519
Total nonzeros	2885	2934	3111	3471
Total operations	4519	4761	5532	7508
Depth	69	62	53	52
B_m	611	674	759	884
B_u	168	163	316	451
B_f	80	101	159	257
Average: Total op./D	65.5	78.9	104.4	147.8

Table 4

Order of matrix: 500
Number of nonzero elements: 2060
Density: 0.82%

Operation weight	100.0	100.0	1.0	0.01
Depth weight	0.0	0.1	1.0	100.0
Fill-ins	2675	2653	2638	3294
Total nonzeros	4735	4713	4698	5354
Total operations	25943	25287	24835	31019
Depth	95	92	94	95
B_m	1143	958	1089	1177
B_u	573	546	551	611
B_f	341	347	329	419
Average: Total op./D	273.1	274.9	264.2	326.5

Table 5

Order of matrix 500
Number of nonzero elements: 1952
Density: 0.78%

Operation weight	100.0	100.0	1.0	0.01
Depth weight	0.0	0.1	1.0	100.0
Fill-ins	933	972	994	1374
Total nonzeros	2885	2924	2946	3326
Total operations	4519	4729	4788	6437
Depth	69	65	59	43
B_m	611	667	700	877
B_u	168	171	193	440
B_f	80	93	112	281
Average: Total op./D	65.5	72.8	81.2	149.7

Table 6

VECTOR REDUCTION OPERATIONS ON CRAY-1 AND THEIR PERFORMANCE

T. L. Jordan
C-Division
Los Alamos Scientific Laboratory
Los Alamos, NM 87545

Summary

The CRAY-1 computer architecture has a conventional scalar vocabulary plus a limited yet powerful vector vocabulary. Although the scalar performance of CRAY is about twice that of the CDC-7600 or IBM 360-195, vector performance is typically at the 40 to 70 megaflop rate with some important applications such as matrix multiplication and matrix inversion performing at demonstrated rates of 100 to 140 megaflops.

The aforementioned vector rates are applicable to functions or operations which produce vector valued results from vector operands or mixed scalar and vector operands. When the results are scalar and functions of vector operands, i.e., reduction operations, then the hardware is not so explicitly equipped to provide the kind of speeds shown above. For the more important functions of the latter type, summing a vector, computing a dot product and searching a vector for its maximum or minimum, one asks to what extent performance can be improved over optimal scalar code through the use of the vector hardware. For some reduction operations, CRAY-1 has a peculiar instruction that explicitly assists the implementation the first two functions. No similar instruction is available for finding the maximum (minimum) value of a vector and its index. Hence, in addition to describing the implementations we have used for the first two functions, we present in some detail a vector search algorithm. The performance as a function of vector length is also given.

The logic for handling the vector sum problem and the dot product are intrinsically the same. One divides up the vectors involved into m segments including the residue segment, i.e., n = 64 (m - 1) + RESIDUE, where n is the total vector length. Operating on these segments as vectors one obtains 64 or less partial sums. These partial sums are then collapsed to 8 partial sums through the use of the aforementioned special recursive hardware instruction. Finally, one then forms a scalar sum from 8 elements to produce the desired sum. The asymptotic time to do this is 1 cycle per element for the sum and 2 cycles per element for the dot product. However, for smaller vector lengths this degrades to scalar rates.

Despite the absence of explicit hardware aids, excellent performance for large vectors can be obtained through software for the problem of finding the index of the maximum (minimum) value of a vector. The asymptotic rate at which each comparison can be made is between 2 and 3 machine cycles per compare. A detailed description of the software technique used to achieve this speed is presented.

IMAGE MAGNIFICATION

J. M. Vocar
Digital Technology
Goodyear Aerospace Corporation
Akron, Ohio 44315

Summary

The processing necessary to convert raw input data into a convenient form which can be analyzed by man or machine generally requires many calculations for each input pixel. With every pass of an observation satellite, thousand upon thousands of such input pixels are being received. The demand to convert this data into a usable form is stressing the capacity of existing digital (sequential) processors.

This ever increasing amount of input data is forcing the image processing community to examine hardware in which many parameters are treated simultaneously; i.e., parallel processors, to handle the processing load. While the maximum processing capability of these processors is usually impressive, the capability can be realized only if the data can be delivered to the processing elements efficiently.

To determine the suitability of a STARAN[(a)] parallel processor in the area of digital image processing, a cubic convolution interpolation algorithm for image magnification was implemented on a 2-way STARAN-B series machine. The program magnified an arbitrarily sized, arbitrarily-located rectangular subset of 8-bit pixels imbedded within a 512 x 512 input image into a 512 x 512 output image. Independently specified, noninteger X and Y magnifications were performed using two-dimensional cubic convolution resampling procedures.

The developed magnification program has practical value. Magnification is an essential requirement in the areas of scene examination and finger print analysis. The cubic convolution reconstruction filter can also be used as a prelude to further image enhancement techniques.

When an image is magnified, the spaces in between the given pixels must be filled in. These intermediate pixel values are generally supplied using one of the three following most popular methods: 1) pixel replication, 2) bilinear interpolation, or 3) cubic convolution.

This paper presents a brief discussion of the advantages and shortcomings of each of these three approaches, including aliasing and roll-off effects. The cubic convolution approach is shown to be clearly superior to the other two methods. The use of STARAN to perform cubic convolution, including algorithm implementation and the program performance is described.

It was found that the magnification process could be performed in less than 0.4 seconds. Approximately 30 add or multiply operations were required for each output pixel, and about 7.5 million of such operations were performed during the magnification process.

The results of the work described show STARAN to be a highly efficient and effective tool in processing image data, e.g., in the areas of resampling and reconstruction. The processing power of STARAN, with its flexible routing and high band width, make it particularly adaptable to digital image processing. References [1]-[7] detail the STARAN organization and several processing techniques.

References

[1] K. E. Batcher, The Multi-Dimensional Access Memory in STARAN, 1975 Sagamore Computer Conference on Parallel Processing.

[2] K. E. Batcher, The Flip Network in STARAN, 1976 International Conference on Parallel Processing.

[3] R. Bernstein, Scene Correction (Precision Processing) of ERTS Sensor Data Using Digital Image Processing Techniques, Third ERTS Symposium, Vol. I, Section A, NASA SP-351, December 10-14, 1973.

[4] Samuel S. Rifman, Evaluation of Digital Correction Techniques for ERTS Images - Final Report, TRW Systems Group Report 20634-6003-TU-00, March 1964.

[5] J. M. Vocar, Image Magnification: Elementary with STARAN, Goodyear Aerospace Corporation, GER-16342, August 1976.

[6] APPLE Programming Manual, Goodyear Aerospace Corporation, GER-15637B, Revision 2, December 1975.

[7] STARAN Reference Manual, Goodyear Aerospace Corporation, GER-15636B, Revision 2, December 1975.

(a) T.M. Goodyear Aerospace Corporation, Akron, Ohio 44315

ALGORITHM DEVELOPMENT FOR PIPELINED PROCESSORS

P.M. Kogge
IBM Federal Systems Division
Owego, NY 13827

Summary

Although processors using pipelined techniques have been available for over a decade (for example, the IBM System/360, Model 91), only recently have they become sufficiently general, with flexible enough controls over the pipeline itself, to allow development for a single pipeline of a whole range of complex algorithms, such as the Fast Fourier Transform, optimal filter derivation, interpolations, etc. Examples of such processors include the IBM 3838 the Proteus Arithmetic Element, the IBM 2938, and others. However, the development of efficient algorithms for such processors is greatly different than for conventional processors. This paper discusses some of the tradeoffs involved in the development of such algorithms.

The typical system hierarchy of such processors includes a pipelined arithmetic unit containing adders, multipliers, etc. Operands for this unit come from a small high-speed "local store". Since there is never enough memory in the local store to hold the largest problems, there is typically a large "main store" with some kind of "storage-to-storage transfer" unit between it and "local store". Pipelining exists in at least three different levels: within the arithmetic modules themselves, within the interconnection of arithmetic modules and the "local store", and with the overlap of arithmetic operations and data transfers between memories. Efficient algorithms must consider all three levels.

Clearly the major constraint on the total speed of an algorithm is the rate at which the innermost loop of calculations can be performed. For a pipelined processor, however, we must carefully choose how many calculations to include in the inner loop to maximize performance. In most cases it is necessary to "balance" the number of operands accessed from the "local store" with the number of each type of operation occuring in one iteration of the inner loop in the arithmetic processor unit. As an example, the inner loop of an FFT is often considered to be what is known as a "2-point Butterfly," which is simply repeated over and over with different data. On many pipelines, however, this calculation uses up all the "local store" bandwidth while leaving the arithmetic unit only 50% utilized. However, by expanding the inner loop to include four butterflies in a certain pattern both the arithmetic modules and the "local store" can be 100% utilized, with twice the performance of the simpler inner loop.

Another related problem occurs when the inner loop involves a recurrence, that is, when a series $x_1, \ldots x_n$ is to be computed, and x_i depends on $x_{i-1}, \ldots x_{i-m}$. Direct implementations of this on a pipeline are inefficient since the calculation of x_{i+1} cannot start until x_i is complete. There are techniques, however, that allow many different recurrences to be "backed up" or "overlapped" to the point where a pipeline can more quickly compute them (cf. Kogge [1,2]).

Once the exact form of the inner loop has been defined, the pipelined arithmetic unit can be programmed to execute them. Typically this involves starting the next iteration of the inner loop as soon as the current iteration has freed up the first stage in the pipeline. However, there are often "collisions" where more than one iteration attempt to use the same stage at the same time. When this occurs it is necessary to insert non-compute delays into the program that actually lengthen the time per iteration, but have the opposite effect of bringing the total number of iterations executed per second back up to the theoretical maximum. Such delays are typically implemented by leaving results in internal registers for short periods of time. Davidson, et al [3] has developed some good techniques for determining optimal delay placement.

Finally, the choice and implementation of the inner loop cannot proceed without concern for the uppermost level of pipelining - namely the overlap of data transfers between memories with the arithmetic computations performed in the arithmetic processor. Data sets must be segmented into pieces small enough to fit into "local store," but large enough so that the overhead involved in synchronizing the transfers with the computations is kept to a minimum. Additionally, the existence of data segmenting at all may force a complication to the inner loop of the calculations to allow carry-over of intermediate results from segment to segment.

References

[1] Kogge, P.M. 1973. "Maximal Rate Pipelined Solutions to Recurrence Problems", First Annual Symposium on Computer Architecture, University of Florida, Dec. 1973, pp 71-76.

[2] Kogge, P.M. 1974. "Parallel Solution of Recurrence Problems," IBM Journal of Research and Development, March, 1974, pp 138-148.

[3] Davidson, E.S., et al 1975. "Effective Control for Pipelined Computers", IEEE Computer Society Conference COMPCON Feb. 1975, pp 181-184.

PARALLEL PREFIX COMPUTATION†

Richard E. Ladner and Michael J. Fischer
Department of Computer Science
University of Washington
Seattle, Washington 98195

Abstract - The prefix problem is to compute all the products $x_1 \circ x_2 \circ \ldots \circ x_k$ for $1 \le k \le n$, where $\circ$ is an associative operation. Using a recursive construction, we obtain a product circuit to solve the prefix problem which has depth exactly $\lceil \log_2 n \rceil$ and size bounded by $4n$. An application yields fast, small Boolean circuits to simulate finite state transducers. By simulating a sequential adder, we obtain a Boolean circuit for n-bit binary addition which has depth $2\lceil \log_2 n \rceil + 2$ and size bounded by $14n$. The size can be decreased significantly by permitting the depth to increase by an additive constant.

1. Introduction

Let $\circ$ be an associative operation on a domain D. The prefix problem is to compute, for given $x_1, \ldots, x_n \in D$, each of the products $x_1 \circ x_2 \circ \ldots \circ x_k$, $1 \le k \le n$.

By analogy with Boolean combinational circuits [6], [7], we consider product circuits, which are directed acyclic oriented graphs. Each node of in-degree 2 represents a product of its two inputs. All other nodes have in-degree 0 and are labelled with an integer between 1 and n. These are the input nodes. With each node v we associate an element of D in the obvious way.

We consider two complexity measures on a product circuit N. $C(N)$, the size, is the number of product nodes in N, and $D(N)$, the depth, is the maximum number of product nodes on any directed path in N. For example, the circuit of Figure 1 has depth 3, size 4 and computes $x_1 \circ x_3 \circ x_3 \circ x_2 \circ x_3$. Note that it also computes $x_1 \circ x_3 \circ x_3 \circ x_2$, $x_1 \circ x_3$ and $x_3 \circ x_2$.

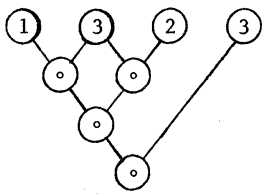


Figure 1. A product circuit. (All arcs are directed downwards.)

The depth of a circuit corresponds to the computation time in a parallel computation environment, whereas the size represents the amount of hardware required. For the prefix problem, it is straightforward to construct a circuit of size $n-1$, the minimum possible, but its depth is also $n-1$. Similarly, it is not difficult to find a circuit of depth exactly $\lceil \log_2 n \rceil$, the minimum possible, but the immediate recursive construction yields a circuit of size $\Omega(n \log n)$.[(a)] In Section 2, we find a solution to the prefix problem of minimum depth $\lceil \log_2 n \rceil$ and size $< 4n$.

The reader familiar with logical networks will notice many similarities between our methods and those used in constructing fast, linear-size circuits for binary addition, such as the "carry-lookahead" method [9]. Indeed, our methods, applied to the binary addition problem, yield a circuit which has linear size and depth $2\lceil \log_2 n \rceil + 2$. Brent has an adder of depth $\log_2 n + O(\sqrt{\log_2 n})$ but has size $\Omega(n \log_2 n)$ [2]. Krapchenko has a linear size adder of depth only $\log_2 n + O(\sqrt{\log_2 n})$ [5], [7]. It appears, however, that our circuit is quite competitive with Brent's and Krapchenko's circuits for small values of n.

The construction involves two steps. First, given an arbitrary finite state transducer, we obtain in Section 3 a circuit for simulating the machine on inputs of length n which has depth $O(\log n)$ and size $O(n)$. Applying that construction to the simple machine for binary addition of Figure 2 and analyzing the constant carefully yields the desired result. The details are presented in Section 4.

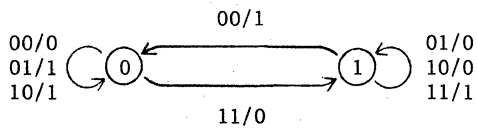


Figure 2. A sequential adder.

2. Circuits for the Prefix Problem

In this section, we define a family of circuits $P_k(n)$ for solving the prefix problem on n inputs. For each k, the depth $D(P_k(n)) =$

† This research was supported in part by NSF Grant No. DCR-12997-A01 through a subcontract from M.I.T. and by NSF Grant No. GJ-43264.

(a) $f = \Omega(g)$ iff $g = O(f)$.

$k + \lceil \log_2 n \rceil$. The size, $C(P_k(n)) < 2(1 + 1/2^k)n$ for all $n \geq 0$ and $0 \leq k \leq \lceil \log_2 n \rceil$. For small n , the size is substantially smaller than this bound would suggest.

The recursive construction of $P_0(n)$ is shown in Figure 3, and the construction of $P_k(n)$ for $k \geq 1$ is shown in Figure 4. When $n = 1$, $P_k(n)$ is simply a single input node and contains no products. In the figures, circles represent concatenation nodes.

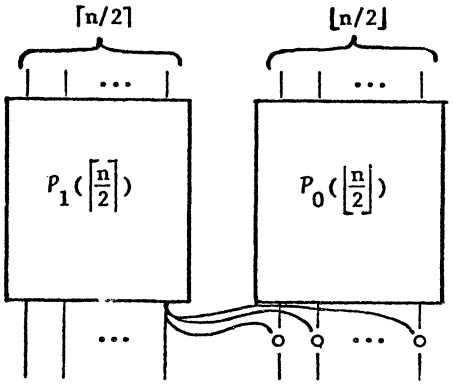


Figure 3. The construction of $P_0(n)$.

That the constructions achieve the desired depth follows easily by induction given the additional fact, also proved by induction, that the last output in $P_k(n)$ has depth exactly $\lceil \log_2 n \rceil$, even when $k > 0$. The correctness of the construction is also easily shown by induction and is left to the reader.

Let $S_k(n) = D(P_k(n))$. Then S satisfies the following recurrences:

$$S_0(n) = S_1(\lceil n/2 \rceil) + S_0(\lfloor n/2 \rfloor) + \lfloor n/2 \rfloor , \quad n \geq 2 ;$$

$$S_k(n) = S_{k-1}(\lceil n/2 \rceil) + n - 1 , \quad n \text{ even and } n \geq 2 , k \geq 1 ;$$

$$S_k(n) = S_{k-1}(\lceil n/2 \rceil) + n - 2 , \quad n \text{ odd and } n \geq 3 , k \geq 1 ;$$

$$S_k(1) = 0 , \quad k \geq 0 .$$

In case n is a power of 2 , we get exact solutions

$$S_0(n) = 4n - F(2 + \log_2 n) - 2F(3 + \log_2 n) + 1 ,$$

$$S_1(n) = 3n - F(1 + \log_2 n) - 2F(2 + \log_2 n) ,$$

and more generally,

$$S_k(n) = S_0(n/2^k) + n \cdot (2 - 1/2^{k-1}) - k$$
$$= 2(1 + 1/2^k)n - F(2 + \log_2 n - k) - 2F(3 + \log_2 n - k) + 1 - k$$

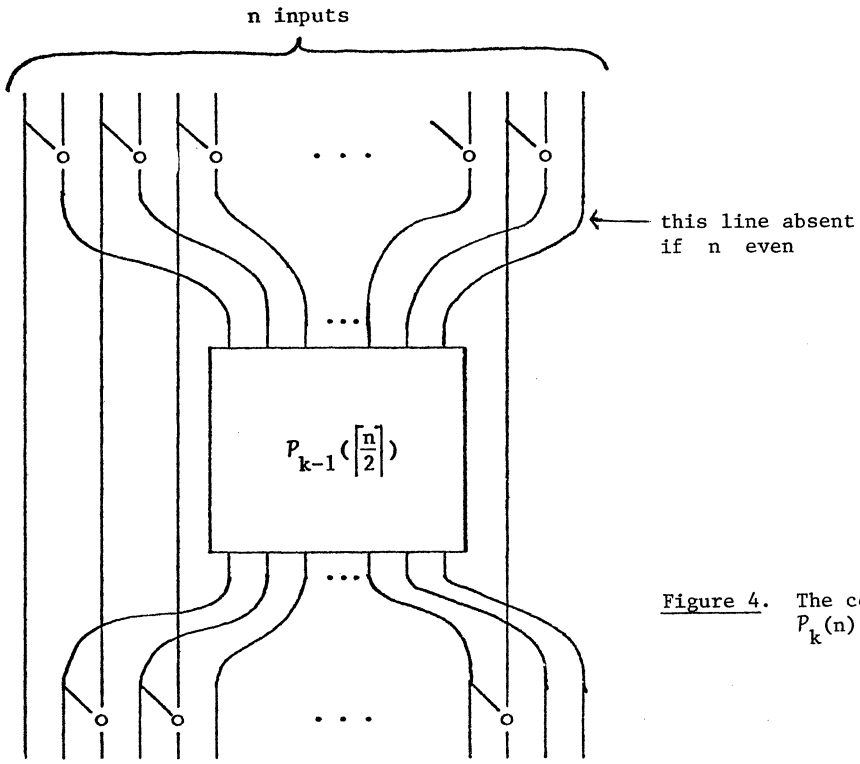


Figure 4. The construction of $P_k(n)$, $k \geq 1$.

holds for all k, $0 \le k \le \log_2 n$. Here, $F(m)$ denotes the m^{th} Fibonacci number, and $F(m) = \frac{\phi^m - \hat{\phi}^m}{\sqrt{5}}$, where $\phi = \frac{1 + \sqrt{5}}{2}$ and $\hat{\phi} = \frac{1 - \sqrt{5}}{2}$ (cf. [3]). Thus, for large n and fixed k, $S_k(n)$ is bounded by $2(1 + 1/2^k)n - a_k \cdot n^{0.69424\ldots}$, where $a_k > 0$ is a constant depending only on k. Some values of $S_k(n)$ are shown in Figure 5.

n \ k	0	1	2	3	4	5	6	7
1	0							
2	1	1						
4	4	4	4					
8	12	11	11	11				
16	31	27	26	26	26			
32	74	62	58	57	57	57		
64	168	137	125	121	120	120	120	
128	369	295	264	252	248	247	247	247

Figure 5. $S_k(n)$ for n a small power of 2.

When n is not a power of 2, we do not have an exact solution, but it is easily verified by induction that $S_k(n) < 2(1 + 1/2^k)n - 2$, $n \ge 1$. In fact we know that $P_k(n)$ is not optimal for n not a power of two. For example, $C(P_0(9)) = 13$, but the circuit of Figure 6 has size only 12 since $S_1(8) = 11$, and it also has minimal depth 4.

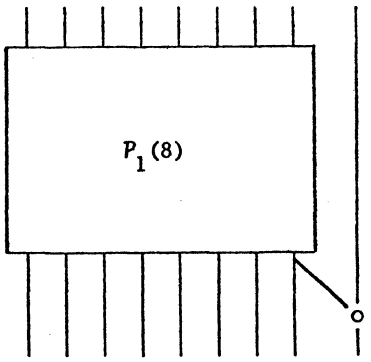


Figure 6. A solution to the 9-input prefix problem.

It is an open problem to determine just how to split the circuit to optimize the construction using the methods of Figures 3 and 4.

There is an analogy between product circuits and addition chains [8], [4]. Let D be the natural numbers, $\circ$ be ordinary addition, and fix each input to 1. Then the minimum size circuit to compute a number n is exactly the length of the shortest addition chain for m. A prefix circuit on n inputs under this interpretation constructs each of the integers from 1 to n. Unlike most of the work on addition chains, we are interested in the depth as well as in the size. As with addition chains, analysis becomes much more difficult for n not a power of 2.

3. Application to Finite State Machines

A classic example of a sequential process is a finite state transducer (cf. Booth [1]). Given an input of length n and an initial state we show below how to compute in parallel the output and final state. This method leads to the construction of fast Boolean circuits that simulate finite state transducers.

We use the Mealy model of finite state transducer which is a five-tuple $M = (Q,\Sigma,\Delta,\delta,\gamma)$ where Q is a finite set of states, Σ is the input alphabet, Δ is the output alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function and $\gamma : Q \times \Sigma \to \Delta$ is the output function.

For each input symbol a we define a function $M_a : Q \to Q$ by $qM_a = \delta(q,a)$. (The argument to M_a is on the left.) Given an input word $a_1a_2\ldots a_k$ the state $qM_{a_1} \circ M_{a_2} \circ \ldots \circ M_{a_k}$ is the state of M after reading $a_1\ldots a_k$ starting in state q, where $\circ$ denotes functional composition.

A parallel algorithm to compute the output and final state given the input $a_1a_2\ldots a_k$ and the initial state q_0 is:

1. Compute $M_{a_1}, M_{a_2}, \ldots, M_{a_n}$ in parallel.
2. Compute $N_1 = M_{a_1}$, $N_2 = M_{a_1} \circ M_{a_2}$, $\ldots$, $N_n = M_{a_1} \circ M_{a_2} \circ \ldots \circ M_{a_n}$ by a parallel prefix algorithm.
3. Compute $q_1 = q_0N_1$, $q_2 = q_0N_2$, $\ldots$, $q_n = q_0N_n$ in parallel.
4. Compute $b_1 = \gamma(q_0,a_1)$, $b_2 = \gamma(q_1,a_2)$, $\ldots$, $b_n = \gamma(q_{n-1},a_n)$ in parallel.

The output is $b_1b_2\ldots b_n$ and the final state is q_n.

Let $c_1(d_1)$ be the size (depth) of computing M_a, $c_2(d_2)$ be the size (depth) of computing functional composition, $c_3(d_3)$ be the size (depth) of computing functional evaluation, and $c_4(d_4)$ be the size (depth) of computing $\gamma(q,a)$. Given an input of length n and an initial state, the size and depth to compute the output and final state is

$$\text{SIZE} \le c_2\, c(n) + (c_1 + c_3 + c_4)n$$
$$\text{DEPTH} \le d_2\, d(n) + d_1 + d_3 + d_4 ,$$

where $c(n)$ ($d(n)$) is the size (depth) of a product circuit to solve the prefix problem. (Note: we assume the state q_0 can be coded or decoded at no cost.)

There are several ways of obtaining Boolean circuits from this method. One simple way is to represent the M_a's as $s \times s$ Boolean matrices where s is the number of states. Functional composition is Boolean matrix multiplication and functional evaluation is the Boolean product of a matrix and a vector. For this representation using the standard matrix multiplication algorithm and the prefix circuit P_0 (or P_k for fixed k) we can construct a Boolean circuit for inputs of length n with linear size and depth $(1 + \log_2 s)\log_2 n + d$ where d is a constant depending only on M.

4. Application to Binary Addition

Consider the finite state transducer A of Figure 2. There are three functions A_{00}, $A_{01} = A_{10}$, A_{11} on states which are closed under composition. We represent them by a pair of bits g,p (for generate and propagate, respectively) as shown in Figure 7. The composition table is shown in Figure 8, and the evaluation table in Figure 9.

input x y	function g p
0 0	0 0
1 0	0 1
0 1	0 1
1 1	1 0

$g = x \wedge y$

$p = x \oplus y$

Figure 7. Computation of the function from the inputs.

function g_1p_1 \ function g_2p_2	0 0	0 1	1 0
0 0	0 0	0 0	1 0
0 1	0 0	0 1	1 0
1 0	0 0	1 0	1 0

$g = g_2 \vee (g_1 \wedge p_2)$

$p = p_1 \wedge p_2$

Figure 8. Composition table.

state s \ function g p	0 0	0 1	1 0
0	0	0	1
1	0	1	1

$t = g \vee (s \wedge p)$

Figure 9. Evaluation table.

From Figure 7 the inputs can be represented by the initial g,p pair, so we get the output table shown in Figure 10.

state t \ input g p	0 0	0 1	1 0
0	0	1	0
1	1	0	1

$z = t \oplus p$

Figure 10. Output table.

By observation we can calculate the constants

$c_1 = 2$ $\quad d_1 = 1$
$c_2 = 3$ $\quad d_2 = 2$
$c_3 = 2$ $\quad d_3 = 2$
$c_4 = 1$ $\quad d_4 = 1$

The basic costs for addition are $SIZE \le 3c(n) + 5n$ and $DEPTH \le 2d(n) + 4$. There are certain refinements that can be made.

1. Let the input state be the constant 0. The evaluation table reduces to $t = g$. There is no "evaluation" so there is no need to compute p at the last level before step 3. This results in a total savings of $3n$ in size and 2 in depth, so $SIZE \le 3c(n) + 2n$ and $DEPTH \le 2d(n) + 2$.

2. We may obtain an n-bit adder with the state as an additional "carry-in" input by forming an $(n+1)$-bit adder which starts in state 0 and uses the lowest order bits to simulate the incoming state. This observation leads to an adder of $SIZE \le 3c(n + 1) + 2n$ and $DEPTH \le 2d(n + 1) + 2$.

3. These techniques can also be used to construct ones-complement adders. Because of the "end-around" carry the input state is a function of the input numbers. The input state is computed in step 2 which makes it available for step 3 where it is used. In this case the adder has $SIZE \le 3c(n) + 5n$ and $DEPTH \le 2d(n) + 4$.

Using the results of Section 2 and the observation 1 above there exists Boolean circuits to compute n-bit sums (with no carry in) of

$$SIZE \le (8 + \frac{6}{2^k})n \quad \text{and}$$

$$DEPTH \le 2\log_2 n + 2k + 2$$

for $0 \le k \le \log_2 n$.

Notice that if we set $k = \log_2 n$ then we obtain a circuit of $SIZE \le 8n + 6$ and $DEPTH \le 4\log_2 n + 2$. These bounds are similar to those obtained for the "carry-lookahead" adder [9]. We believe that our circuit $P_k(n)$ for $k = \log_2 n$ is essentially the same as the "carry-lookahead" adder.

The table of Figure 11 illustrates the trade-offs that can be made between size and depth in small adders. The numbers of Figure 11 are based on those of Figure 5 together with observation 1.

		k = 0		k = 1		k = 2		k = 3	
		DEPTH	SIZE	DEPTH	SIZE	DEPTH	SIZE	DEPTH	SIZE
number of bits	4	6	20	8	20	10	20		
	8	8	52	10	49	12	49	14	49
	16	10	125	12	113	14	110	16	110
	32	12	286	14	250	16	238	18	235
	64	14	632	16	539	18	503	20	491
	128	16	1363	18	1141	20	1048	22	1012

Figure 11. DEPTH and SIZE of small adders.

References

[1] T. L. Booth, Sequential Machines and Automata Theory, John Wiley and Sons, Inc., New York, (1967).

[2] R. Brent, "On the addition of binary numbers," IEEE Transactions on Computers, vol. C-19, no. 8, (1970), pp. 758-759.

[3] D. E. Knuth, The Art of Computer Programming, Volume 1, Addison-Wesley, Reading, Mass., (1968).

[4] D. E. Knuth, The Art of Computer Programming, Volume 2, Addison-Wesley, Reading, Mass., (1969).

[5] V. M. Krapchenko, "Asymptotic estimation of addition time of a parallel adder," Engl. transl. in Syst. Theory Res., Vol. 19, (1970), pp. 105-122; orig. in Probl. Kibern. 19, pp. 107-122.

[6] M. S. Paterson, "An introduction to Boolean function complexity," Société Mathématique de France Astérisque 38-39, (1976), pp. 183-201. Also appeared as technical report STAN-CS-76-557, Computer Science Department, Stanford University, (August, 1976).

[7] J. E. Savage, The Complexity of Computing, John Wiley and Sons, New York, (1976).

[8] A. Schönhage, "A lower bound for the length of addition chains," Theoretical Comp. Sci. 1, (1975), pp. 1-12.

[9] C. Tung, "Arithmetic," in Computer Science, A. F. Cardenas, L. Presser, M. A. Marin, eds., Wiley-Interscience, New York, (1972).

TOWARD AN ARITHMETIC FOR PARALLEL COMPUTATION

Jerome Rothstein
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210
614-422-7027

Abstract

A new positional binary number system was devised in an attempt to avoid carrying in addition. It originated from the groupoid string formalism, previously shown to have the computation universality of Turing machines and the parallel capabilities of cellular and bus automata. It is uniquely defined by the natural conditions (a) a number is doubled by adding a copy, shifted one place left, to the original number, where digits are added mod 2, and (b) adding 1 to any number adds 1 mod 2 at precisely one place. Arithmetical, combinatorial, and parallel computational properties of the binary system are discussed and some properties of similar systems with higher radix briefly noted.

I. Introduction

At the 1976 International Conference on Parallel Processing the writer [1] developed a groupoid string formalism (based on earlier work on patterns [2]) and proved that groupoid string systems (a) had the computation universality of Turing machines, (b) corresponded in a simple detailed manner to one-dimensional cellular automata, and (c) had a natural parallel computation potentiality which led to developing bus automata which actually realized it to a considerable degree. The success of the groupoid string approach, both in effectively speeding up Turing machine algorithms and in recognizing many kinds of formal languages, led to the hope that arithmetic computations might also be sped up if they could be properly formulated as groupoid string processes.

The first problem is to find a number system, compatible with a groupoid string viewpoint, with some kind of parallel possibility inherent in its "local" structure. For example, addition might be done everywhere along a string, in parallel, without having to worry about carry chains. A positional notation seems mandatory to keep string length within reasonable bounds. We sought a binary system based on addition mod 2 as the underlying groupoid. Having the one dimensional infinite shift register in mind as the "active tape" suggested that integers correspond to finite strings over (0,1), with infinite strings of 0's understood as preceding and following the integer. The set of positive integers and zero, {I}, is then, in a regular notation in which the infinite 0-strings are omitted,

$$I = \Lambda + 1 + 1(0+1)^*1 \qquad (1.1)$$

Here the null symbol corresponds to 0, 1 to 1, and the remaining positive integers correspond biuniquely to members of the infinite set of strings $1(0+1)^*1$ in a manner to be determined. Word length is required to increase monotonically with increase in size of the integer represented, with doubling an integer increasing the length by 1. It turns out that forming a daughter string, earlier used to generate patterns in the plane [2] where the groupoid operation is ⊕ (addition over (0,1) mod 2), corresponds exactly to adding a left shifted copy of the original string to itself without carrying. We take this daughter string as representing the integer which is twice the integer represented by the original string. We then obtain the powers of 2 as successive rows of Pascal's triangle taken mod 2. To avoid carry chains it is certainly necessary that addition of unity initiate no chain. The number must then change in precisely one digit, i.e., the number system is a grey code.

These properties define the number system uniquely (up to direction of reading). However, it was so awkward to work with at first that it seemed doomed to remain a mere curiosity. For example, given an integer in this system, there was no easy way to find the next higher integer. But eventually properties of the system began to emerge suggesting that it could be useful in handling some combinatorial problems by parallel computation, even if arithmetic remained difficult. It began to seem likely that a dimly perceived inversion of what was easy and what was hard between this number system and conventional ones had deep roots and could lead to interesting developments in computability theory generally and in parallel computation in particular.

A particularly intriguing feature of this kind of number system is the beautiful way in which arithmetic properties of numbers are associated with symmetries of patterns generated by the daughter string process. A new kind of geometry of numbers, profoundly different from Minkowski's, which has algebraic and combinatorial features and is also related to the architecture of cellular and bus automata may emerge from this line of research. The possibility of arithmetizing the generation of geometrical pattern, and the emergence of patterns as properties of classes of strings again suggest that this number system deserves prolonged study even if it should be utterly useless for numerical computations of conventional kinds.

In the following sections we develop enough of the groupoid formalism to show how it connects both with algorithmic pattern generation and with parallel computation via cellular and bus automata. The two naturally meet in Pascal's triangle, from which multiplication by 2 arose naturally. We then derive the number system for

the integers and extend it to numbers of the form $N2^{-n}$ for all integer N and n. This gives binary approximations for all reals to any desired accuracy. These non-integers have a possibly null non-repeating part followed by an infinite periodic string, the number of digits in the period being a power of two. Some geometric and combinatorial aspects of the number system are presented, and it is shown that many peculiarities making the system awkward for conventional computations are helpful from the viewpoint of parallel operations on cellular and bus automata. After a discussion of ternary and higher bases, a tentative assessment of the significance of these number systems is made.

II. Groupoid String Systems, Patterns, and Bus Automata

A groupoid (G, o) is a set G closed under a binary operation o, i.e., given arbitrary elements a and b of G, their combination by means of that operation yields an element of G, say c. In symbols

$$a \circ b = c \qquad (2.1)$$

or

$$o: \; G \times G \to G \qquad (2.2)$$

A finite groupoid is conveniently specified by its multiplication table.

If the symbols representing groupoid elements are taken as an alphabet and used to generate strings by concatenation, we can form a daughter string $\ldots d_i \; d_{i+1} \ldots$, from any string which can be called the parent string $\ldots p_i \; p_{i+1} \ldots$ by

$$d_i = p_i \circ p_{i+1} \qquad (2.3)$$

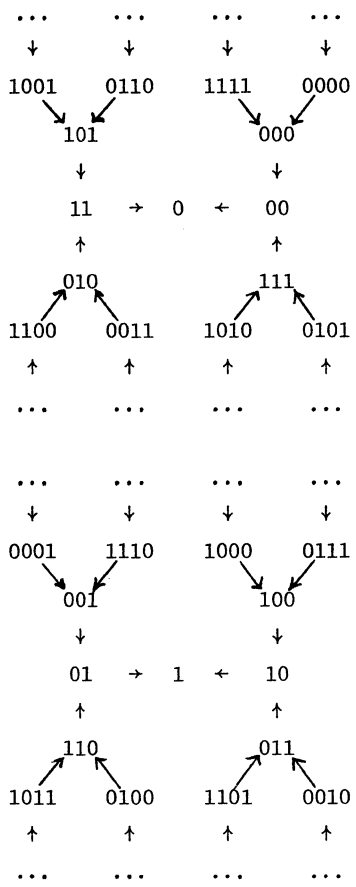
The strings can be finite, infinite to the left, right or both. We denote the set of all possible strings by G^{ω}, the finite strings by G^{*}, the left and right semi-infinite strings by $G^{-\infty}$ and G^{∞} respectively, and the set of two-way infinite strings by $G^{\pm\infty}$. A groupoid string system is a subset of G^{ω} closed under the (unary) daughter operation (or relation). The descendant relation between strings is the transitive closure of the daughter relation, and its converse is the ancestor or ancestral relation. A string system is often conveniently represented as a set of directed trees. Nodes are strings and edges are drawn only from parent to daughter. Fig. 2.1 illustrates the system of finite binary strings over {0,1} under addition modulo 2. Flow is to the root, and there are two distinct trees with roots 0 and 1.

A two-sided identity (unit, neutral element) and/or an annihilator (zero) can always be adjoined to any groupoid if not already present, and each is unique. Denoting an identity by e and an annihilator by z, they satisfy

$$e \circ a = a \circ e = a \qquad (2.4)$$

$$z \circ a = a \circ z = z \qquad (2.5)$$

and uniqueness follows from $e \circ e' = e' = e$ and



⊕	0	1
0	0	1
1	1	0

Fig. 2.1 Groupoid String System Over $(\{0,1\},\oplus)$ Consisting of All Finite Strings, Where Arrows Point from Parent Strings to Daughter Strings

$z \circ z' = z = z'$. The string system of Fig. 2.1 is embeddable in $G^{\pm\infty}$ over (z,0,1) by matching the strings in that system to the strings in $G^{\pm\infty}$ consisting of those strings preceded and followed by infinite "rays" of z's, and defining z as the daughter of any element of G. The daughter of a finite string with z at one end and e at the other has the same length as the parent. Such systems are ultimately periodic under iterated daughter transformations, as are infinite periodic strings. The latter give rise to an infinite set of "wallpaper" designs [2], the former to shift-register sequences. With infinite strings of e's preceding and following a word whose end letters are not e, each generation increments the length of the non-e positions by unity. Omitting the semi-infinite e-strings gives the generalization of Pascal's triangle for addition, starting from 1, to the general groupoid starting with an arbi-

trary groupoid string. Figures 2.2 and 2.3 illustrate the process for the cyclic groups of order 2 and 3 respectively, starting from 1, giving Pascal's triangle modulo 2 and 3.

1
11
101
1111
10001
110011
1010101
11111111
100000001
1100000011
10100000101
111100001111
1000100010001
11001100110011
101010101010101
1111111111111111
10000000000000001
11000000000000011
10100000000000101
11110000000000001111
10001000000000010001
11001100000000110011
10101010000000001010101
11111111000000011111111
10000000100000001000000001
11000000110000001100000011
10100000101000001010000101
11110000111100001111000011111
10001000100010001000100010001
11001100110011001100110011001
10101010101010101010101010101
11111111111111111111111111111111
10000000000000000000000000000001

Fig. 2.2 Pascal's Triangle Mod (2)

Computation of daughter strings is performable in parallel on a one-dimensional cellular automaton (CA), essentially a "shift-register accumulator" whose "logic" embodies groupoid multiplication; see Fig. 2.4 (applied to Equation (2.3)). For discussion of how the groupoid formalism covers the general CA, thus achieving computation universality via simulation of an arbitrary Turing machine, see Rothstein [1]. That paper also describes the genesis and parallel speed-up aspect of bus automata (BA), giving further references to both the CA literature and the work of the writer and his former students, C.F.R. Weiman and J.M. Moshell.

As noted, the CA can compute a complete daughter string, in parallel, in the time needed for one operation. To compute a parent string from a daughter string requires a BA; see Fig. 2.5. The BA is here one-dimensional, i.e., it utilizes a linear array of similar finite state automata A_0, A_1, A_2, ... A_k, ... The states of individual automata, s_0, s_1, ... s_m, are labeled by the groupoid elements, which, in this case, are also the input and output alphabets. Ordinarily the state, input, and output alphabets

1
11
121
1001
11011
121121
1002001
11022011
121212121
1000000001
11000000011
121000000121
1001000001001
11011000011011
121121000121121
1002001001002001
11022011011022011
121212121121212121
1000000002000000001
1100000022000000011
1210000002120000000121
1001000002002000001001
1101100002202200011011
1211210002122120001211 21
1002001002001002001002001
1102201102201102201102201 1
1212121212121212121212121 21
1000000000000000000000000001
1100000000000000000000000011
1210000000000000000000000000121
1001000000000000000000000001001
1101100000000000000000000011011
1211210000000000000000000000121121

Fig. 2.3 Pascal's Triangle Mod (3)

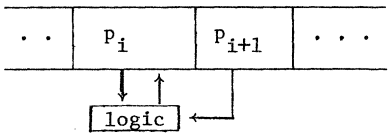


Fig. 2.4 Shift Register Computation of Daughter String

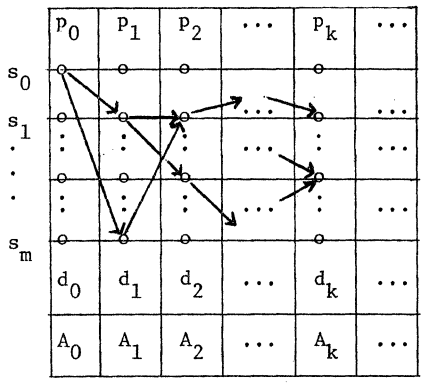


Fig. 2.5 Bus Automaton Explained by Means of a Connected Linear Array of Automata $\{A_i\}$

are designated by distinct sets of symbols, e.g. respectively as

$$K = \{s_0, s_1, \ldots s_m\} \tag{2.6}$$

$$\Sigma = \{\sigma_0, \sigma_1, \ldots \sigma_n\} \tag{2.7}$$

$$\Theta = \{\theta_0, \theta_1, \ldots \theta_r\} \tag{2.8}$$

The effect of input σ_j on state s_i is to induce a transition to state s_k, and produce output θ_k, which can be written

$$s_i\sigma_j \rightarrow (s_k, \theta_k) \tag{2.9}$$

In the present case we can both take

$$K = \Sigma = \Theta = G \tag{2.10}$$

and replace the resulting right side of (2.9), namely (d_k, d_k), simply by d_k. We then have

$$p_i \circ p_j = d_k \tag{2.11}$$

where p_i, p_j, d_k are elements of G replacing s_i, σ_j, and (s_k, θ_k) respectively. We use the symbol for the groupoid operation in (2.11) instead of the concatenative notation of (2.9) to stress that each automaton A_i is designed to embody the groupoid multiplication table in its state transition function. The arrows drawn in Fig. 2.5 signify that (2.9) applies in the sense that the tail of an arrow at s_i of A_i is drawn with its head at s_j of A_{i+1} if p_i is the input of A_i, d_i the output (a Moore machine is here chosen for definiteness, a similar discussion applies to Mealy machines). The BA concept combines the automata A_i with communication busses controlled locally by them for calculation of a parent string in terms of a daughter string. The arrows become bus sections. They are hooked up as part of the setting-up condition in which the d_i string is used as input. The p_i string is now the output activated by a continuous path (through arrows) between end markers or the like bounding the d_i string. As semigroups are associative groupoids, this leads to immediate recognition of regular languages and thus to Turing machine speed-up in a number of steps one more than the number of tape turnarounds [1].

III. The Binary Parallel Number System

We now derive the number system briefly described in I, beginning with a statement of the fundamental theorem.

Theorem 3.1 There exists a binary number system unique up to reversal of reading direction, satisfying (i) 1 designates the integer one; (ii) the double of any integer is obtained by adding a copy of that integer, shifted one place to the left, to the original integer, where addition in each place is modulo two; (iii) the number of digits representing any integer can not exceed the number of digits representing a larger integer; and (iv) addition of one to any integer changes one and only one digit.

The proof depends on a string of subsidiary results which will be stated and proved as theorems because of their interest both for the number system and for string patterns. In the sequel integers are understood to be in this system.

Theorem 3.2 The powers of two are given by successive rows of Pascal's triangle mod 2 and are palindromes.

Proof: The rule for constructing Pascal's triangle is clearly expressible as the daughter string algorithm for addition; for mod 2 addition it is the doubling rule (ii) of the hypothesis of Theorem 3.1. The symmetry of Pascal's triangle makes the strings read the same backwards and forwards, i.e., they are palindromes. Fig. 2.2 gives the powers of 2 from zero to 32.

Theorem 3.3 For any integer N, the integers 2^n N, n = 0, 1, 2, ..., are given by the successive rows of the trimmed Pascal triangle whose first row is N; equivalently the product of N and 2^n is calculated as in conventional binary except that the final additions are performed mod 2.

Proof: The first half of the theorem is established as in the proof of Theorem 3.2; it is simply the doubling rule iterated. Before establishing the second half, we refer to Fig. 3.1,

3	111	1	1
6	1001	11	2
12	11011	101	4
24	101101	1111	8
48	1110111	10001	16
96	10011001	110011	32
192	110101011	1010101	64

```
  11011 = 12
×1111  =  8
 11011
  11011
   11011
    11011
10011001 = 96
```

Fig. 3.1 Three Times a Power of Two

illustrating the case N = 3 (proof that 3 is uniquely 111: (a) it can have no more than three digits because 4 is 101; (b) it must have more than two digits because 2 = 11 is the only permissible two digit number as 01, 10, and 00 would be partially or totally "absorbed" in the semi-infinite 0-strings; (c) the only possibility left is 111). To see that the method of the illustrative multiplication example is again but another form of the doubling rule, compare doubling and quadrupling as illustrated by Fig. 3.2. The number $x_n\ x_{n-1} \ldots x_o$ is shown with a copy left shifted and added to the original string to double it, and this result is again duplicated as shown to quadruple the number. The unshifted

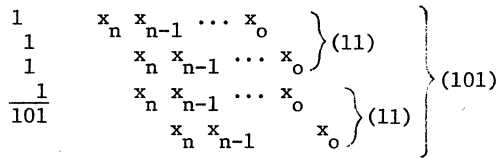


Fig. 3.2 Multiplication by a Power of Two

member of the shifted double clearly lines up with the shifted member of the unshifted double. They "cancel" in mod 2 addition, leaving only the result of adding a copy shifted two places left to the original string as the quadruple of the original number. The asserted result now follows by induction.

Theorem 3.4 The doubles of two numbers which differ in only one place differ in two adjacent places.

Proof: Let the numbers be $x_n\ x_{n-1} \cdots x_k \cdots x_o$ and $x_n\ x_{n-1} \cdots \bar{x}_k \cdots x_o$, where $\bar{0} = 1$ and $\bar{1} = 0$, and call them N and $\bar{N}$ respectively. We have

$$\begin{array}{rl} & x_n\ \ x_{n-1} \cdots x_k\ \ x_{k-1} \cdots x_o \\ & \underline{\quad x_n \cdots x_{k+1}\ x_k \cdots x_1\ x_o} \\ 2N = & y_{n+1}\ y_n \cdots y_{k+1}\ y_k \cdots y_1\ y_o \end{array}$$

$$\begin{array}{rl} & x_n\ \ x_{n-1} \cdots \bar{x}_k\ \ x_{k-1} \cdots x_o \\ & \underline{\quad x_n \cdots x_{k+1}\ x_k \cdots x_1\ x_o} \\ 2\bar{N} = & y_{n+1}\ y_n \cdots \bar{y}_{k+1}\ \bar{y}_k \cdots y_1\ y_o \end{array}$$

where we have used the easily proved theorem for addition mod 2 that

$$x \oplus y = z \text{ implies } x \oplus \bar{y} = \bar{z} \qquad (3.1)$$

This theorem clearly generalizes by induction to multiplication by 2^n as stated in the next theorem.

Theorem 3.5 The multiples by 2^n of two numbers which differ in one place differ in at most (n+1) contiguous places, namely those corresponding to 1's in the number 2^n, where the rightmost 1 of 2^n lines up with the x_k in which the two original numbers differ.

Proof: Form the trimmed Pascal's triangle starting from $N = x_n \cdots x_o$, and put a bar over x_k as in the proof of Theorem 3.4. Then the bars propagate just as the 1's do in Pascal's triangle, for (3.1), via commutativity of $\oplus$, shows that

$$x \oplus y = z \text{ implies } \bar{x} \oplus \bar{y} = z, \qquad (3.2)$$

i.e., two bars (1's) combined give none (0). But this is just what the theorem states.

A similar proof, which we omit, proves the further generalization stated in the next theorem.

Theorem 3.6 If two integers are right justified and comparisons of digits in the various positions are encoded as 1 when they differ and 0 when they are the same, then the differences in their multiples by 2^n are encoded as the multiple by 2^n of the first encoding.

Theorem 3.4 is needed to construct the number system, while 3.5 and 3.6 are of interest in bus automata and groupoid pattern investigations.

We now prove Theorem 3.1. We already know that 1, 2, 3, 4 are respectively 1, 11, 111, 101, and that given the doubling rule, if we know the integers 1 through N, the even numbers from N to 2N are determined, leaving only the odd numbers between N and 2N to be found. By Theorem 3.4 two successive evens differ in two adjacent places, so by (iv) of the hypothesis of Theorem 3.1 the odd number between them differs in precisely one of those two places from each of the two evens. We need only show that all the numbers from (N+1) to 2N, for N a power of 2, have one more digit than N, and give the string for one odd integer, e.g. N+1, to conclude that all the representations of the integers are uniquely determined to 2N. Uniqueness would then follow, by induction on n, $N=2^n$, for all the integers; the base of the induction having already been amply provided by the truth of the theorem for n=0,1,2. Note that 5, 6, 7, 8 must all have four digits and begin and end in 1, just as in the argument leading to 111 for 3. End 0's are excluded; no three digit strings are available for 5; 6 and 8 have four digits by the doubling rule, so 7, which can not have fewer digits than 6 nor more than 8 also has four; 5 can have no more than four and must have more than three, and so exactly four. As 6 and 8 are 1001 and 1111 respectively, 5 and 7 are 1101 and 1011, not necessarily respectively. Note that 5 has only the two possibilities of prefixing or suffixing a 1 to 4 = 101, that 3 can be regarded as either prefixing or suffixing a 1 to 2 = 11, and that the two cases are mirror images. In accordance with the custom of forming larger numbers by adjoining digits on the left we take 1101 for 5, thus having only the choice 1011 for 7. The four four-digit numbers exhaust all possibilities for filling the two center places. The same situation recurs in the next "octave", 9 through 16. The doubling process always converts strings beginning and ending in 1 into strings beginning and ending in 1, the interpolated odds must do likewise, and by Theorem 3.4 adoption of the prefix convention for 5 forces it for 9, and by induction for 2^n+1, $n > 3$. The 3 interior places in the five-digit numbers use all possible "fillings" for the 8 integers they represent. The next octave of integers thus must again all have one more place, doubling the numbers of "fillings" which again exactly accommodate the new integers. By induction this is always the case, whence the representations of all the integers are unique up to a mirror reflection of the entire system. The choice is a reading direction convention common to all number systems.

This completes the proof of Theorem 3.1, but armed only with that result one might despair of ever being able to count in this number system. The only simply specified odd numbers are $\{2^n + 1 \mid n = 0,1,2 \ldots\}$. One would then have to find consecutive evens in each of their octaves

between which each one could fit, the other of the two possibilities then being the right one for that slot. Then one would have to find another place where this last number might have fitted, plug the other possibility in that place and so on, until all slots are filled. However, observe that the arguments given to build up the integers from 1 to 2^n, with the list, in order and right justified, can be applied to the succession of digit changes needed to count from 2^n+1 to 2^{n+1}. The only novelty is that now the left justified reversed list is added mod 2, digit by digit to 2^n, where the list lines up its left-most digits one place to the left of the left-most digit of 2^n. Fig. 3.3 illustrates the process applied to 1 through 4 to obtain 5

1	1		1101	5	1011	11001	13
2	11	→	1001	6	1001	→ 11101	14
3	111		1011	7	1101	10101	15
4	101		1111	8	1111	10001	16
	101				1111		

Fig. 3.3 Counting in Octaves

through 8, and the latter reversed to obtain 13 through 16 via addition of 8. The process is amenable to parallel operation, e.g., in a planar BA, the 1's in 2^n complementing the digits of the corresponding columns of the list, 0's leaving them unchanged (blanks are interpreted as 0's). The above can be viewed as an extension of the doubling rule, applied to a power of 2, to addition of a power of two to any number not exceeding it. We state it as Theorem 3.7.

Theorem 3.7 To find the sum of any integer and a power of two not less than that integer, shift the reversed integer one place left from where it would be if left justified with the power of two and sum digit by digit mod 2.

Applied interatively this immediately proves the correctness of the following algorithm to convert conventional binary to parallel binary.

Theorem 3.8 To find the representation of an integer given in conventional binary (a) left-shift the smallest power of 2 indicated by it with respect to the left justified list of the remaining powers of 2; (b) add the first two numbers; (c) reverse the result and add to the next; (d) repeat a, b, c, alternating reversals and additions until one number remains.

This algorithm can be inverted to convert a given number to conventional binary.

Theorem 3.9 To convert a parallel binary number with n digits to conventional binary, first add 2^{n-1}, digit by digit, mod 2. If the result is a string of n 0's the number is 2^{n-1}. If not, add in 2^{n-1} again, restoring the original number, and repeat the process with 2^{n-2} right justified, writing 1 in the 2^{n-2} position of the sought conventional binary number. The procedure is then repeated with the reversed shortened string of (n-k) digits (add 2^{n-k-1}, right justified, mod 2 in each place, etc.), giving further digits of the binary number sought, and so on until the string has been reduced to a power of 2.

Proof: The procedure reverses the steps of the previous theorem, for what was called addition is also subtraction mod 2. It alternately tests whether the given or modified string is a power of two and subtracts an appropriate power of two.

A procedure similar to that of Theorem 3.7 works to add 2^n to all numbers from 2^n+1 to 2^{n+1}

Theorem 3.10 To add 2^n to any number from 2^n+1 to 2^{n+1} shift the integer left one place from where it would be if right justified with the power of 2, and sum digit by digit mod 2.

Proof: The sums sought are the numbers from $2^{n+1}+1$ to 2^{n+2}. From the way they would be constructed by the octave counting method it is clear that the procedure given is equivalent to doubling the 2^n part first and then adding from 1 to 2^n.

We list a few useful miscellaneous results in the next theorem, most proved already.

Theorem 3.11 All numbers begin with 1 and end with 1, and have an even number of 1's if they are even and an odd number of 1's if they are odd. The largest number with (n+1) digits is 2^n, the smallest is $2^{n-1}+1$.

Proof: The assertions needing proof are those on the number of 1's. Let N have k 1's. Then 2N will have (2k-2k') 1's, where k' is the number of places in which 1's of the shifted copy are above 1's of the original copy. Then 2N, for arbitrary N, has 2(k-k') 1's, i.e., all even numbers have an even number of 1's. Odd numbers differ by one, in their number of 1's, from the evens, by (iv) of Theorem 3.1.

The next theorem is of particular interest as it shows that mere counting by octaves to 2^n also computes the set of binomial coefficients $\binom{n}{r}$, inaugurating the study of "parallel combinatorics".

Theorem 3.12 If the integers from 1 to 2^n are tabulated, either left or right justified, then the number of digit changes (alternations) in the r^{th} column from either end (blanks are counted as 0's) is precisely $\binom{n}{r}$.

Proof: The theorem is true by inspection of Fig. 3.3 for $n \leq 2$. Assume it is true for n=k and consider the right justified list of integers from 1 to 2^{k+1}. By the discussion in connection with Fig. 3.3 which led to Theorem 3.7, the number of digit changes in the first k columns of the second half of the list is the same, column by column, as those of the first half. We thus form the total number of changes for case k+1 by adding the changes for two contributions, offset by one, of case k. But this is identical with the rule by which Pascal's triangle is constructed, whence the numbers of changes are given by $\binom{k+1}{r}$ and the theorem is established.

Fig. 3,4 gives a Markov algorithm for doubling a number and illustrates its use. As the markers move in one direction their action can be simulated by a finite state machine, showing once again that doubling is "immediate" on a BA. As only one marker at most is ever present

$\alpha 0 \rightarrow 0\alpha$	1101 = 5
$\alpha 1 \rightarrow 1\beta$	α1101
$\beta 0 \rightarrow 1\alpha$	1β101
$\beta 1 \rightarrow 0\beta$	10β01
$\beta \rightarrow .1$	101α1
$\alpha \rightarrow .\Lambda$	1011β
$\Lambda \rightarrow \alpha$	10111 = 10

Fig. 3.4 Markov Algorithm for Doubling

the only critical features of the priority ordering of productions are that marker introduction has lowest priority, and that all conclusive productions, as a group, have next lowest priority. Markov algorithms for daughter strings over general groupoids are readily transcribable from their multiplication tables. For example, in G^*, where daughters are one letter shorter than their parents, with $G = \{g_1, g_2, \ldots, g_n\}$ and $g_i \circ g_j = g_k$, we can write

$$\begin{aligned} \alpha_0 g_i &\rightarrow g_i \alpha_i \\ \alpha_i g_j &\rightarrow g_k \\ \alpha_i &\rightarrow .\Lambda \\ \Lambda &\rightarrow \alpha_0 \end{aligned} \qquad (3.3)$$

for the algorithm which computes the daughter for all finite strings of lengths greater than one, and leaves strings of length one unchanged. There are (n+1) markers, but if the groupoid has an identity e, n markers suffice as α_0 can be replaced by α_e. The productions above are multiple, except for the last, the first, second and third standing for n, n^2, and n different ones, the second representing the multiplication table, essentially.

The algorithm of Fig. 3.4 is easily modified to multiply by 2^n: replace the last rule by $\Lambda \rightarrow \alpha_f \alpha\alpha \ldots \alpha$, keeping the other rules unchanged and also applicable to α_f and β_f except that the conclusive productions for α and β cease to be so and become conclusive only for α_f and β_f. Similarly we can easily write a groupoid n^{th} generation descendant Markov algorithm. The markers still move in one direction, so the computation is still immediate on a suitable BA. This can be generalized to the equivalent of a Turing machine speed-up theorem distinct from the one cited earlier.

We close this chapter with some observations on addition and multiplication in parallel binary. First the simple rules for adding sufficiently high powers of 2 to a number or to multiply by a power of 2 fail in general. For example, if one computes 3 × 3 by the rule of Theorem 3.3, namely as in conventional binary except that the final additions are mod 2, the answer is 15 while 3 × 5 gives 27. Addition is particularly frustrating, for given some big number, there still seems to be no simple way even to tell what digit to change in order to add 1 to it (short of finding where it occurs in the list of integers and going to its successor). The difficulty stems from the way powers of 2 are folded into an integer; adding two numbers containing a common power of two still seems to require replacing the two contributions of that power by the next higher power. Carrying has thus not been avoided, in essence. It may well be that Theorems 3.8 and 3.9 will have to be utilized in some form to do many ordinary arithmetical tasks, even on a BA, but conversion is rapid both ways and may often be necessary only in part. Hopefully results like Theorem 3.12 will ultimately abound and have important practical applications.

IV. Extension to Fractions $2^{-n}N$

It is known that the direct product of any number of cyclic groups of order 2 is a group (hence a groupoid) satisfying the "self-solving" conditions

$$a \circ b = c \Rightarrow a = b \circ c \Rightarrow b = c \circ a \qquad (4.1)$$

as well as being commutative and unipotent. This means that in Pascal's triangle mod 2 not only are successive rows daughters of their immediate predecessors (remember that the rest of the half-plane is covered by 0-strings) but strings parallel to the sides, terminating on the sides of the original triangle are parents of the parallel strings immediately above them. As the original sides, 111 ... , give 1 when doubled, we immediately deduce that the successive semi-infinite strings parallel to one side, starting with that side represent the successively higher negative powers of 2. Using (4.1), rewritten in the notation of (2.3), permits us to write

$$p_{i+1} = d_i \circ p_i \qquad (4.2)$$

and to interpret the resulting recursive algorithm for finding the parent string as division by 2.

Fig. 4.1 shows how both of the foregoing developments can be combined in an extended Pascal half-plane pattern. The horizontal arrow points to the vertex of the original Pascal triangle, the vertical arrow to that of the Pascal triangle made up of inverse powers of 2 calculated by (4.2). The infinite half-plane to the left of the common side of the two triangles is covered with 0's, and an infinite triangle of 0's fills the rest of the half-plane containing the two triangles.

For $N = 2^{n_2} 3^{n_3} 5^{n_5} \ldots p^{n_p}$ the halving process will not give infinite strings (as usual we suppress infinite 0-strings) until the n_2 powers

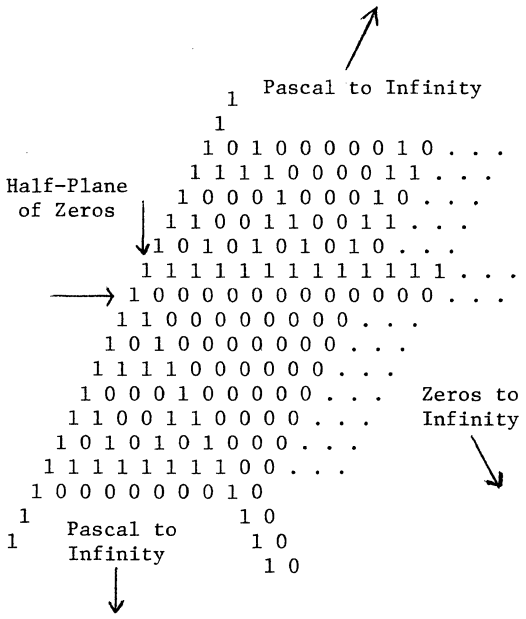


Fig. 4.1 Positive and Negative Powers of Two

of 2 are eliminated, i.e., the new N is odd. Eventually the process produces a trimmed triangle from which the periodic infinite part of $2^{-n}N$ is read off parallel to its sides. See Fig. 4.2, which shows the process for 9 = 11111. The vertical arrow indicates the left end digit

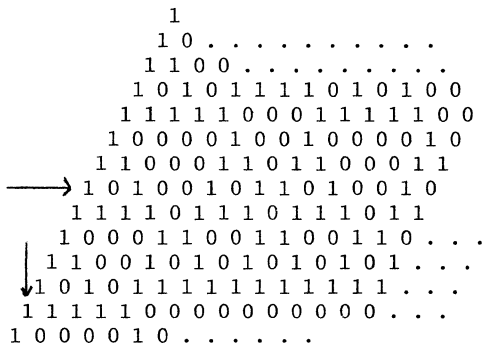


Fig. 4.2 $2^{-n}N$ for N = 9

of 9. The horizontal arrow indicates the 1, which together with the 1 beginning the infinite string of 1's of 9/2, bounds the "top", 11001, of a trimmed Pascal triangle whose rows are 2^n13. The lines parallel to its bottom side give the periodic parts of $2^{-n}9$. The 9 is also the top of a trimmed triangle (whose rows are 2^n9). This relation between 9 and 13 is reciprocal for $2^{-n}13$ has its periodic part given by the side strings of 2^n9. Every odd number is clearly paired in this way with an odd number with the same number of digits. Some, like 1, 3, 11 and 15 (respectively 1, 111, 10011 and 10101), pair with themselves.

The connection between Pascal's triangle and the negative powers of a binomial is old. Just as its rows give the coefficients of $(1+x)^n$, so do the infinite sequences ("sides") parallel to one side give the coefficients occurring in the series expansions of $(1-x)^{-n}$. Proofs of many results of this chapter can be devised by using such standard theorems with the binomial coefficients taken mod 2.

We now designate a power of 2 by T, with superscript (positive or negative integer or zero) when needed, whether the string be finite (integer, 2^n), or infinite to the right (fraction, 2^{-n}). We have $T^0 = 1$, $T^1 = 11$, $T^n = (n+1)^{th}$ row of Pascal's triangle, $T^{-1} = 1111 \ldots$, $T^{-2} = 10101 \ldots$, and so on, with $T^{-n} = (n+1)^{th}$ "side" of Pascal's triangle. The difference between 2^n and T^n is that the latter admits an interpretation as an operator on all binary strings, while the former is a number which is $T^n(1)$. The set of operators $\{T^n \mid n = \ldots -2, -1, 0, 1, 2, \ldots\}$ is clearly the free commutative group on one generator isomorphic to integer addition, and $T^n(1) = 2^n$. Explicitly,

$$T^m T^n = T^{m+n} \tag{4.3}$$

$$T^m T^{-m} = T^0 \tag{4.4}$$

$$T^m(1) = 2^m \tag{4.5}$$

The operation T is essentially the daughter algorithm for 2-way infinite strings, with finite and semi-infinite strings understood as having two or one semi-infinite string(s) of identities in the remaining places. It thus has meaning for an arbitrary groupoid. The interpretation of T^n as an operator on strings over an arbitrary groupoid is not generally the corresponding descendant or ancestor function, encoded as a binary word. The operator T^2, for example, encoded as 101, can be interpreted as element by element groupoid multiplication of the operand string with a copy of itself shifted two places left. For the daughter function (2.3) it is natural to take the first "factor" from the shifted string. For the general groupoid string $\ldots a_i\ a_{i+1}\ a_{i+2} \cdots$ the "grand-daughter" element is $(a_i \circ a_{i+1}) \circ (a_{i+1} \circ a_{i+2})$, not $(a_i \circ a_{i+2})$ as in $\{(0, 1), \oplus\}$ and as called for by T^2.

One might question the need for introducing T-notation; except in the case at hand, where it appears to be only multiplication by a power of 2, it has limited applicability, and so little interest. However, the notation makes it clear that substituting T^m for the 1's of T^n for all n, i.e., writing T^m for the 1's of Pascal's triangle generalized to the half-plane transforms that half-plane into itself by a "translation" of m along the "row coordinate". The interpretation of the $(T^m, 0)$-strings as appropriately shifted $\oplus$-addition of as many copies as there are occurrences of T^m in the string verifies their equivalence to the 2^{n+m} rows of the Pascal half-plane. But now, for any interpretation of the 1's of Pascal's triangle as T^m's, for fixed positive or negative m, we can carry through the entire chain

of reasoning by which the number system was constructed. We can therefore take the entire list of positive integers, replace the 1's throughout by any one T^m, perform the indicated shifted $\oplus$-additions, and come out with the list of integer multiples of $T^m(1)$. In short, we obtain $2^m N$ whether m be negative or positive.

We sum up the main results obtained in this chapter by stating them as theorems.

Theorem 4.1 The negative powers of 2 are right semi-infinite binary strings lying parallel to one side of Pascal's triangle and beginning with an element of the other side. The side itself represents 2^{-1} and successive contiguous parallel strings represent successively higher powers, the n^{th} being 2^{-n}. When Pascal's triangle is augmented with the strings produced successively by the parent string algorithm, with the initial 1's of the set of ancestor strings continuing the alignment of the side of the original triangle containing the initial 1's of the positive powers of 2, then a half-plane is covered by three regions (a) the original Pascal triangle, (b) a triangle of 0's, (c) a Pascal triangle whose left side is parallel to the rows of the original triangle, and whose right side is a continuation of the left side of the original triangle.

Theorem 4.2 The string representing $2^{-n}N$ for any positive integer N, and for m any positive or negative integer or zero, is obtained by multiplying the strings representing 2^{-n} and N as in ordinary binary except that the final additions are mod 2.

V. Related Number Systems

In this chapter we summarize our initial results on two kinds of number system, or better, two compatible methods for constructing number systems related to the parallel binary system discussed heretofore, and make some observations about possibilities for more general systems.

The first is the natural generalization from base 2 to any base. The second, based on the observation that one penalty for requiring both monotonic increase in number of digits with increasing integer size and the grey code property was loss of the neat multiplication algorithm of powers of 2, requires instead that multiplication of any two integers be done the same way as it is done for the base. There may be many other properties, the presence of which would partially or completely define a complete unambiguous representation of the number system. One must procede very cautiously, however, for it is easy to demand properties which are, in fact, incompatible. We close with some remarks on generalization of the underlying groupoid from cyclic groups to general groups, semigroups, quasigroups, and general groupoids.

From Fig. 3.4 and the Markov algorithm (3.3) it is easy to see that the daughter string algorithm is easily adapted to the design of a base k number system. It multiplies a string by k. The rows of Pascal's triangle mod k represent the powers of k, and inverse powers, together with the positive powers, fill out a half-plane as

```
                1
               1 . . .
             1 0 0 1 0 0 0 0 0 2 0
           1 1 0 1 1 0 0 0 0 2 2
         1 2 1 1 2 1 0 0 0 2 1
       1 0 0 2 0 0 1 0 0 2 0 . . .
     1 1 0 2 2 0 1 1 0 2 2
   1 2 1 2 1 2 1 2 1 2 1 2
 1 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0
1 2 1 0 . . .
```

Fig. 5.1 3^{-n} in Base 3

before. Fig. 2.3 gives that part of Pascal's triangle representing 3^n for $0 \le n \le 33$ and Fig. 5.1 gives a portion of the half-plane representing 3^{-n}, $0 \le n$. Note that the triangle above the triangle of 0's has 1212 ... for one of its sides; only in base 2 does it coincide with the original Pascal triangle. In base k that side becomes 1(k-1) 1(k-1) .... We have proved uniqueness of the system of integers in base 3, assuming monotonicity and the grey code property as before. The argument is much more laborious than for binary (omitted for lack of space) because the rule for multiplication by 3 leaves two empty slots between the triples of consecutive integers. Uniqueness was proved by showing that only one way of building the system "from the ground up" avoided inconsistencies later on. In the resulting system negative numbers "mirror" the positives in a pleasing fashion. Using the build-up process "backwards" through 0 gives, for the negative of a b c ... the string $\bar{a}$ $\bar{b}$ $\bar{c}$ ..., where barred elements are inverses of unbarred ones, $\bar{0} = 0$, $\bar{1} = 2$, $\bar{2} = 1$. Fig. 5.2 gives the Markov process for multiplication by the base k and a part of the number system in base 3. While there is little doubt that parallel systems in any base can be constructed, our experience in proving uniqueness for base 3 and the large number of possibilities for base 4 led to low priority for attempting to prove uniqueness for all bases, particularly in view of the possibility of non-uniqueness suggested by the second class of number systems, to which we now turn.

As the multiplicative properties of the powers of 2 are simple (as in standard binary, except that the final sums are performed mod 2), but complicated if neither factor in a product of two factors is a power of 2, we examined the consequences of requiring that multiplication always be performable in that fashion. It turns out that "gaps" and inversions appear (inversion here means having the smaller of two integers with a larger number of digits). The gaps, naturally, are filled in some fashion by assigning primes to them in some fashion, and there seems to be nothing yet visible to restrict how one should assign strings to primes ("in order" no doubt, but what is that order). Unique factorization exists as does a simple algorithm to test divisibility, and all of the foregoing apparently applies to any base.

To take a specific case, take 1 for 1, and 11 for 2 as before, getting the same strings as

(a) 1. $\alpha_0 0 \to 0\alpha_0$ ... $\alpha_i 0 \to i\alpha_0$ ...

$\alpha_0 1 \to 1\alpha_1$ ... $\vdots$

$\vdots$ $\alpha_i j \to j\alpha_{i+j}$ ...

$\alpha_0(k-1) \to (k-1)\alpha_{k-1}$

2. $\alpha_i \to .i$ 3. $\Lambda \to \alpha_0$

(b)

$V = \{0,1,2\}$

$\{N\} = \Lambda+V+(1+2)V^*1$

$\alpha 0 \to 0\alpha$
$\alpha 1 \to 1\beta$
$\alpha 2 \to 2\gamma$
$\beta 0 \to 1\alpha$
$\beta 1 \to 2\beta$
$\beta 2 \to 0\gamma$
$\gamma 0 \to 2\alpha$
$\gamma 1 \to 0\beta$
$\gamma 2 \to 1\gamma$
$\alpha \to .\Lambda$
$\beta \to .1$
$\gamma \to .2$
$\Lambda \to \alpha$

1	1
2	21
3	11
4	211
5	221
6	201
7	101
8	111
9	121
10	2121
11	2221
12	2011
13	2001
14	2201
15	2101
16	2111
17	2011
18	2211
19	1211
20	1011
21	1111
22	1121
23	1021
24	1221
25	1201
26	1101
27	1001

Fig. 5.2 (a) Markov process: multiplication by k in base k
(b) Base 3

before for 2^n. We can take 111 for 3 and thus get the same expressions for $2^n 3$ as before. But now 9 must be 10101; compare it with 8, which is still 1111, and we see the grey code property is gone. If we choose the previous number for 5, 1101, then 15 is now 100011, whereas 16 is 10001, and the monotonicity property is also gone. Our investigation to date suggests that these infelicities are invariable companions of the simplified multiplication rule. There is consolation in that division is easy: the recursive relations involved in computing reciprocals are easily solved, and dividing by n is the same as multiplying by 1/n. To illustrate we compute 1/3, 3 = 111. Let the string sought be $a_1 a_2 a_3 \ldots$. Then we must have

$$\begin{array}{r} a_1a_2a_3a_4 \cdots a_{i+2} \cdots \\ a_1a_2a_3 \cdots a_{i+1} \cdots \\ a_1a_2 \cdots a_i \quad \cdots \\ \hline 1\ 0\ 0\ 0 \ \ldots\ 0 \quad \cdots \end{array}$$

from which we easily find that 1/3 is 110110110.... This string is periodic; the 2-way infinite string is one of many studied earlier in connection with patterns [2]. Indeed, if a daughter element be written below and between the two parent elements of which it is the product so that all three are at the vertices of an equilateral triangle, then in the periodically covered part of the plane the 1's are on the vertices of a network of hexagons with 0's at their centers (Fig. 5.3). Our research has thus come full circle: study of groupoid string patterns [2] has led to "number systems" which may well become powerful tools for investigating

Fig. 5.3 Periodic Covering of the Plane by 110110...

aspects of pattern, symmetry, and geometric configuration.

The last possibility for generalization considered here is that of admitting more general groupoids than cyclic groups. If associativity, unique solvability, or commutativity are sacrificed one might think that all chances of building a number system have been lost. Many might think that even abandoning cyclic groups (single generating element) for more general groups would have similar effects. However, we have shown that many different quasigroups (groupoids) with unique two-sided solvability) can generate precisely the same totality of patterns in the plane. A single pattern-equivalent class can contain associative, non-associative, commutative, and non-commutative quasigroups, and direct products of as many different ones as desired. Considered as algebraic systems, they are often strongly non-isomorphic. It would be rash, therefore, to deny the possibility that such systems might be useful for parallel computation. Semigroups, too, in spite of the fact that inverses of elements usually can not be defined, should not be ruled out either, both because of the intimate connection between them and finite state machines and because at least one cyclic group is associated with every idempotent, be it an identity or an annihilator.

VI. Concluding Remarks

Although a truly parallel arithmetic has not yet been constructed, those found may have pattern and combinatorics so firmly woven into their very structure that they may permit new attacks to be mounted on some very difficult problems. They have intriguing "nesting" possibilities, like the substitution of T^m for the 1's of Pascal's triangle, and a totally new mix of "local" and "global". They approach additive and multiplicative properties of numbers more from the multiplicative side, in contrast to traditional mathematics (e.g., Peano's axioms). We therefore feel they have great interest and much promise, despite the profound problems they present.

References

1. J. Rothstein, "On the Ultimate Limitations of Parallel Processing", Proc. 1976 International Conf. on Parallel Processing, pp. 206-212, (IEEE and Wayne State U., Detroit, Aug. 1976).
2. J. Rothstein, "Patterns and Algorithms", Proc. 9th Symp. on Adaptive Processes (IEEE and U. of Texas, Austin, Dec. 1970).

RESPONSE TIME OF PARALLEL PROGRAMS

Richard J. Lipton and Frederick G. Sayward
Department of Computer Science
Yale University
New Haven, Connecticut 06520

Abstract -- The response time of a parallel program is defined to be the maximum delay between successive activities of an event. Response times are dependent on two factors: the parallel program's structure and the program's scheduler policies. It is shown that under weak assumptions about the scheduler policy, the imposition of an N-fair policy in which each event gets a chance to execute at least every N scheduler steps, the response time becomes dependent only on program structure: either the response time is infinite or it is linear in N (i.e., $\leq cN$ for some $c>0$). Also presented are decision procedures for determining whether or not the response time is infinite and for determining the exact linear relationship in N (i.e., the minimum c).

1.0 *Introduction*

The response time of a parallel program is the maximum time that an event in the program may ever wait for a chance to execute. Response time is clearly important in realtime programs: large or unbounded response time may cause the program to fail. Even nonrealtime programs may be seriously degraded if the response time is too large.

Response time of a parallel program is not easily computed. Often it is only determined by empirical observation. The fundamental question addressed in this paper is:

> How can one compute the response time of a parallel program?

Previous studies of this question [1,4,8] have shown that, under certain assumptions about how programs are scheduled, one can show that particular events execute infinitely often. While this type of information is useful, there are situations where it is inadequate: e.g., in a realtime program for data acquisition, knowing that an event will eventually execute does not guarantee that data (for example) will not be lost.

In order to get a more useful analysis of the response time of a parallel program we will make stronger assumptions about how our parallel programs get scheduled. In all of the previous work very weak scheduling assumptions have been made. Here we will assume instead that we have scheduling where each event of the parallel program gets a chance to try to execute at least every N scheduling steps ($N>0$), in the worst case. To avoid scheduling anomalies, it is necessary that N be at least as large as the number of events in the program. Even in this case, of course, some events may get their chances faster than others; however, no event ever waits longer than N steps to be "looked" at by the scheduler.

Clearly, an event may *not* be able to execute every N steps; it may have to wait for some other event to occur. In particular, if r is the response time of some event, then $r > N$ is possible. A basic question is then:

> As a function of N, what values can r take?

For example, can r be N square, i.e., can r grow non-linearly in N? The answers to these questions are contained in the *response time theorem*: as N grows, either

(1) the response time of an event becomes infinite (i.e., in the worst case it can wait forever), or
(2) the response time is linear in N (i.e., it is bounded by cN, for some constant c).

To fully describe the response time behavior of a parallel program we must consider the question of how one can compute the smallest such c (our main theorem gives an upper bound) after having determined that the response time is finite? The answers to these questions are given by providing decision procedures for the following questions:

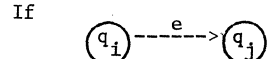
(1) given an event e of a parallel program, can e ever have infinite response time?
(2) given a constant $c>0$, is the response time of $e \leq cN$ for all N?

These questions are reduced to questions about suitably encoded vector addition systems [10].

The remainder of this paper has the following organization. In section 2 we give a formal model of parallel programs and their computations and show how this model relates to the parallel programming notations found in the literature. The scheduler of a parallel program is presented in section 3. A scheduler is shown to be just an alternative characterization of a program's computations. In section 4 we introduce the scheduler restrictions necessary for our main result. The response time definition and the response time theorem are given in section 5. In section 6 this result is shown to include as special cases several schedulers used in actual systems. In section 7 the afore-mentioned decision procedures are presented.

2.0 *Parallel Programs*

A *parallel program* P is a finite directed graph G, a distinguished node q_1 of G, and edges which are labelled with elements from a finite set E. Intuitively, the nodes of G are the *states* of P. If



is an edge, then we have the folowing semantics:

> "If P is in state q_i, then event e can occur resulting in P going into state q_j."

Clearly, so far, P is nothing more than a finite state diagram. As an aside, while our main theo-

This work is sponsored in part by the Office of Naval Research Grant N00014-75-C-0752.

rem depends on the assumption of finite states, our basic definitions and indeed several of our results can be generalized to allow infinite state parallel programs.

Formally, a *parallel program* P is a 4-tuple $P = (Q,E,q_1,\tau)$ where:

(1) Q is a finite set of *states*, denote $Q = \{q_1,q_2,\ldots,q_n\}$.

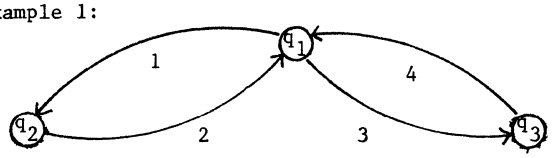
(2) E is a finite set of *events*, denote $E = \{1,2,\ldots,m\}$.

(3) q_1 is a distinguished *start state*.

(4) τ is the *state transition function*: $\tau:Q \times E \rightarrow Q$.

It should be noted that this definition is by no means novel. It links well with path expressions [1] and many other such definitions. Also, note that we have deliberately defined a parallel program to be a rather unstructured object. The usual notions of process, semaphores, instruction counters and so forth, are implicit rather than explicit.

As an example of a parallel program, consider the following directed graph, which we will call example 1:



This corresponds to a parallel program represented in the semaphore notation of [5] as follows:

```
semaphore s (initially 1)
    parbegin
        repeat 1:  P(s);  2:V(s) forever
        repeat 3:  P(s);  4:V(s) forever;
    parend;
```

Indeed, our model of parallel programs is capable of representing the control aspects of any parallel program which uses bounded value semaphores.

2.1 *Parallel Program Computations*

In order to study the response time of parallel programs, it is necessary to introduce the notion of an event blocking. Thus our definition of a parallel program's computations must include both event execution and event blocking. To this end, let the elements of E be called *event executions*. Then the elements of $E'=\{e' \mid e \in E\}$ are called *event blockings*. The elements $EA=E \cup E'$ are called *event activities*.

We will define a parallel program's computations to be certain finite and infinite strings over EA. Intuitively, an event e may execute whenever the program's control is in a state q where e is eligible to execute (i.e., $\tau(q,e)$ is defined). An event e may block whenever the program is in a state where e cannot execute, the program has passed through a state where e could have executed but didn't, and e hasn't executed in the meantime. To formally define those strings over EA which satisfy this intuitive notion, we introduce the following function on EA*:

Definition: The function *state*:EA* ---> Q is defined as follows:

(1) *state*$(\wedge) = q_1$

(2) For $e \in E$ and $x \in EA^*$, *state*$(xe) = \tau($*state*$(x),e)$

(3) For $e' \in E'$ and $x \in EA^*$, *state*$(xe') =$ *state*(x) only if
   (a) $\tau($*state*$(x),e)$ is undefined, and
   (b) for some event f, $x=yfz$ such that $\tau($*state*$(y),e)$ is defined and *not* *substr*(fz,e).

where *substr* is the usual substring predicate.

Note that the ways in which *state* can be undefined correspond to illegal event executions and blockings. For example, if P is in state q and $\tau(q,e)$ is undefined, then e is not eligible to execute. Likewise, if $\tau(q,e)$ is defined, then e can't block. We are now ready to define the computations of a parallel program.

Definition: The *computations* of a parallel program P are members of the set C, the union of the following two sets:

(1) $CF=\{x \in EA^* \mid$ *state*(x) is defined$\}$.

(2) $CI=\{x$ an infinite string over $EA \mid$ *state*(y) is defined for all finite prefixes y of x$\}$.

The set CF is called the set of *finite computations* of P and CI the *infinite computations*. Note that CI may be empty and that C is closed under finite prefix.

For later use, we distinguish a (possibly empty) subset of the finite computations:

Definition: A finite computation $x \in CF$ is called *terminating* if for all events e of P both *state*(xe) and *state*(xe') are undefined. A computation terminates when no further event activity is possible.

The following are examples of legal and illegal computations, in terms of regular expressions, for the parallel program presented above.

Legal

(1) $(12 + 34)^*$ - no event ever blocks

(2) $13'2(12)^*$ - event 3 remains blocked forever

(3) $123(1')^*$ - event 1 is forever blocking

Illegal

(1) $(2' + 4')^+$ - events 2 and 4 may never block

(2) $12341'$ - event 1 is ineligible to block since it can execute.

Note that example 1 has no terminating computations.

2.2 *Parallel Program Total State*

At any point during the execution of a parallel program P a (possibly empty) subset of the events will be blocked. We define the total state of the program to be the state of P's control coupled with the subset of currently blocked events.

Definition: A *total state* of a parallel program P is a member of the set $T=\{(q,B) \mid q \in Q$ and $B \subseteq E\}$.

Note that T is finite for finite state para-

llel programs. Given any computation we can compute the total state via the following function:

Definition: The *total state function tstate*:C---> T is defined as:

(1) $tstate(\wedge) = (q_1,\phi)$

(2) Let $xf \in C$, $B \subseteq E$, $q \in Q$ and $tstate(x)=(q,B)$.
   (a) If $f=e$ then $tstate(xf)=(\tau(q,e), B-\{e\})$.
   (b) If $f=e'$ then $tstate(xf)=(q,B \cup \{e\})$.

3.0 *Parallel Program Schedulers*

We have defined the computations of a parallel program P to be sequences of event executions and blockings. Which particular computation is produced by the execution of P is determined by the decisions made in an agent entirely external to P; namely, by the *scheduler*. The scheduler maintains a data structure that contains information such as the state in which P's control lies and the blocking status of P's events. We will call this data structure the *scheduler state*. A *scheduler step* consists of the scheduler determining which events are eligible for event activity, using a *scheduling policy* to determine which one of those events will execute or block, and then reflecting this decision by appropriate changes to the data structure (i.e., making a scheduler state transition). The scheduler repeats this cycle as long as there are events eligible for event activity.

In this section, we will formally define the scheduler of a parallel program independently of any scheduling policies. We show that this is just an equivalent characterization of a parallel program's computations. Thus, in subsequent sections when scheduling policies are introduced, we will be effectively restricting the computations that parallel programs produce.

3.1 *Scheduler State*

Let P be a parallel program having n states and m events. The scheduler state will consist of three types of information:

(1) The *program state*.
(2) For each event, a *delay* which indicates the number of scheduler steps which have passed since the event's last activity.
(3) An *event status set* which indicates whether or not an event is eligible to *block*.

Accordingly, we have the following formal definition:

Definition: Let P be a parallel program having m events. A *scheduler state* S is an element of the set $SS = Q \times D \times B$ where:

(1) Q is the state set of P.
(2) $D=NN \times NN \times \ldots \times NN$ (m times) where $NN=\{0,1,2,\ldots\}$.
(3) $B=\{0,1\} \times \{0,1\} \times \ldots \times \{0,1\}$ (m times).

The *i*th member of D indicates the delay of the *i*th event and the *i*th member of B indicates the blocking status of the *i*th event, with 0 indicating ineligibility.

In order to facilitate future presentation we now introduce several projection functions on scheduler states. Let $S=(q;d_1,d_2,\ldots,d_m;b_1,b_2,\ldots,b_m)$ be an arbitrary element of SS. We have

(1) *pstate*:SS ---> Q by $pstate(S) = q$.
(2) *delay*:SS × E ---> NN by $delay(S,i) = d_i$.
(3) *blocked*:SS × E ---> {*true*,*false*} by $blocked(S,i)=\{$if $b_i=1$ then *true* else *false*$\}$.
(4) *blockedset*:SS ---> 2(E) by $blockedset(S) = \{i \mid blocked(S,i)\}$ where 2(E) is the power set of E.
(5) *totalstate*:SS ---> T by $totalstate(S) = (q,blockedset(S))$.

3.2 *Schedules and the Scheduler*

A *schedule* for a parallel program P will be the non-empty sequence of scheduler states that correspond to a particular computation of P and the *scheduler* of P will be all schedules. We will show that P's scheduler is isomorphic to P's computation set.

Definition: Let P be a parallel program which has m event. Let $Z=S_1,S_2,\ldots$ be a finite or infinite sequence of scheduler states. Then Z is a *schedule* for P if and only if

(1) $S_1=(q_1;0,0,\ldots,0;0,0,\ldots,0)$ (2m zeroes)
(2) For $i>1$, let $S_i=(q;d_1,\ldots,d_m;b_1,\ldots,b_m)$ and $S_{i+1}=(q';d_1',\ldots,d_m';b_1',\ldots,b_m')$. Exactly one of the following two cases must hold:
   (a) There is an event e in P such that
      (i) $\tau(q,e)=q'$.
      (ii) $d_j'=\{$if $j=e$ then 0 else $d_j+1\}$
      (iii) $b_j'=\{$if $j=e$ then 0 else {if $\tau(q,j)$ is defined then 1 else b_j}}.
   In this case we say e *executes* and denote by S_i R(e) S_{i+1}.
   (b) There is an event e in P such that
      (i) $\tau(q,e)$ is undefined and $q'=q$.
      (ii) $d_j'=\{$if $j=e$ then 0 else $d_j+1\}$.
      (iii) $b_j'=\{$if $j=e$ then 1 else $b_j\}$.
   In this case we say e *blocks* and denote by S_i R(e') S_{i+1}.

Definition: Let P be a parallel program. Then the *scheduler* for P is the set $S=\{Z$ a sequence of scheduler states$\mid Z$ is a schedule for $P\}$.

Theorem: Let P be a parallel program. Then the set of P's computations C is isomorphic to P's scheduler S.

Proof: We only sketch the proof. Define the function *makesch*:C ---> S as follows:

(1) $makesch(\wedge)=(q_1;0,0,\ldots,0;0,0,\ldots,0)$
(2) For $xf \in C$ where $f \in EA$ and $makesch(x)=S$, $makesch(xf)=S'$ such that S R(f) S'.

It should be clear that *makesch* is well-defined, one-to-one, and onto.

Notation: We let *makecomp* denote the inverse function of *makesch*.

As with computations, we will talk of finite, infinite, and terminating schedules.

4.0 *Initial Scheduler Policy*

In this section we introduce three scheduling policies, the first two are common in the literature -- the third new, which allow us to develop our concept of response time.

4.1 *The Busy Wait Free Policy*

Recall that in example 1 we had z=123(1')* as a legal computation in which event 1 is forever blocking. Although the program is technically executing, it is essentially doing nothing. This phenomenon has been dubbed *busy wait* [5] and great care has been taken to avoid it in the design of operating systems [3,6,9]. Hence, our first scheduling policy will be a "busy wait free" policy.

Definition: Let $Z=S_1S_2\ldots$ be a schedule for a parallel program P. Z is called *busy wait free* if for all $i\geq 1$, S_i R(e') S_{i+1} implies *not blocked* (S_i,e).

Intuitively, under the busy wait free policy once an event e blocks e may not block again until e has executed at least once. This rules out 123(1')* as a computation but 1231'(43)* is still legal. The busy wait free scheduler for P is then

Definition: The set $SF=\{Z\in S \mid Z \text{ is busy wait free}\}$ is called the *busy wait free scheduler* for P.

and the allowable computations under the busy wait free policy are

Definition: The members of the set $CF=\{makecomp(Z) \mid Z\in SF\}$ are called the *busy wait free computations* of P.

The following result is immediate from the definitions of busy wait free schedules and computations.

Lemma 1: Let $w\in C$. Then $w\in CF$ iff. for all events e and decompositions w=xe'ye'z, we have *substr*(y,e).

4.2 *The Release Policy*

As noted above, even with the busy wait free policy we have z=1231'(43)* as a legal computation for example 1. In z event 1 blocks but is never released (i.e., it never executes again even though it is capable of doing so). This is, in general, unacceptable. For example, event 1 could represent a data recording process and we would want it to eventually be executed if it has data to record. Satisfying this criterion has been called showing that an event executes "infinitely often" (if it is capable of doing so) [1,4,11]. Necessary for showing that an event executes infinitely often is the imposition of a "release" scheduling policy.

Definition: Let $Z=S_1S_2\ldots$ be a busy wait free schedule for a parallel program P. Z is called a *release* schedule if for all $i\geq 1$ and arbitrary distinct events e and f, we have S_i R(e) S_{i+1}, *not blocked*(S_i,e), and *blocked*(S_i,f) imply (*pstate*(S_i),f) is undefined.

Intuitively, under the release scheduling policy when there is a choice between executing either blocked or non-blocked events a blocked event is chosen. Thus 1231'(43)* is ruled out as a computation for example 1 since for the second and subsequent executions of event 3 the blocked event 1 could have been executed. We now have

Definition: The set $SFR=\{Z\in SF \mid Z \text{ is a release schedule}\}$ is called the *release scheduler* for P.

and the allowable computations under the release scheduling policy are:

Definition: The members of the set $CFR=\{makecomp(Z) \mid Z\in SFR\}$ are called the *release computations* of P.

The following result is immediate from the definitions of release schedules and computarions:

Lemma 2: Let $w\in CF$. Then $w\in CFR$ iff. for all distinct events e and f and decompositions w=xey, we have $xfy\in CF$ and *blocked*(x,f) imply *blocked*(x,e).

Here, *blocked*(x,e) is the expected predicate on *tstate*(x).

4.3 *The N-Fair Policy*

Under the release scheduling policy we still have z=(123(43)*4)* as a legal computation for example 1. In z event 1 executes infinitely often but from any execution of 1 to its subsequent execution an arbitrary number of scheduler steps may pass. In certain applications this would be intolerable. To remedy this situation we introduce an "N-fair" scheduling policy.

Definition: Let $Z=S_1S_2\ldots$ be a release schedule for a parallel program P. Let N be a fixed integer ≥ 1. Z is called an *N-fair schedule* if for all $i\geq 1$ and all $e\in E$, *not blocked*(S_i,e) implies *delay*$(S_i,e) \leq N$.

Intuitively, under the N-fair scheduling policy events which are not blocked will undergo event activity (execute or block) within N scheduler steps from the point of their last execution. Of course, blocked events may have to wait longer than N scheduler steps or forever, depending on the structures of the particular program. Thus in z event 1 would wait for at most N/2 executions of event 3 since the N-fair policy would force the scheduler to consider event 1 at that time. We now have the following definitions:

Definition: For fixed $N\geq 1$, the set $SN=\{Z\in SR \mid Z \text{ is an N-fair schedule}\}$ is called the *N-fair scheduler* for P.

and the allowable computations under the N-fair scheduling policy are:

Definition: For fixed $N\geq 1$, the members of the set $CN=\{makecomp(Z) \mid Z\in SN\}$ are called the *N-fair computations* of P.

The following results are immediate from the definitions of N-fair schedules and computations:

Lemma 3: Fix N≥1 and let w∈CFR. Then w∈CN iff. for all events e∈E and decompositions w=xyz with |y|>N, *not substr*(y,e), and *not substr*(y,e'), we have *blocked*(x,e).

Here |y| denotes the length of the string y.

Lemma 4: Fix M>N≥1. Then CN⊆CM.

Pefore proceeding, we present a lemma which will be crucial in proving our response time results.

Lemma 5: For a fixed N≥1, let x∈CN with the following properties:

(1) x=yz with |z|=M>N.
(2) *tstate*(y)=*tstate*(yz)

Then for all i≥1, x_i∈CM where x_i=yzz...z (i copies of z).

Proof: Note that by the determinism of τ we have *tstate*(x)=*tstate*(x_i) for all i≥1. For i=1, x_i=x∈CM by Lemma 4. Fix i>1.

(1) If x_i is not in C, then we contradict x being in C.
(2) If x_i is not in CF, then we contradict Lemma 1.
(3) If x_i is not in CR, then we contradict Lemma 2.
(4) Suppose that event e is the reason why x_i is not in CM. There are three subcases:
    (a) If *substr*(z,e), then we contradict Lemma 3.
    (b) If *not substr*(z,e) and *blocked*(x,e) then we contradict Lemma 3.
    (c) If *not substr*(z,e) and *not blocked* (x,e), then we contradict x being in CN.

4.3.1 *Implementation Considerations*

Implementing the busy wait free and release scheduling policies is a rather trivial task: the decisions to be made in a scheduler step can be determined entirely from the scheduler state independently of past or future decisions (i.e., the scheduler would be a Markov process). Note, however, that this is not true when an N-fair policy is in effect. When making a decision on event activity the scheduler must consider not only past decisions (i.e., event delays) but also the structure of the parallel program under consideration since a faulty decision might make violation of the N-fair policy inevitable. Thus, some degree of "lookahead" must be done. While this can always be done for finite state parallel programs, there will be some infinite state programs which require infinite lookahead and thus N-fair scheduling becomes impossible.

As can readily be seen in example 1, low values of N can severely restrict the scheduler. For example, under 2-fair scheduling there are only four computations: 12, 13', 34, and 31'. Since each computation is non-terminating, we have an anomalous situation. In general, we should choose N at least as large as the number of events in the program.

5.0 *Response Time of Parallel Programs*

Recall in the computation z=(123(43)*4)* for example 1, under N-fair scheduling once event 1 executes it will wait at most N scheduler steps to execute again. We call this time of waiting the *response time* of an event. We will be concerned with the *worst case* response time of an event for all possible schedules since a parallel program with acceptable worst case behavior is acceptable in general.

In most applications we would like all events to have finite response times. Moreover, we would like these finite response times to be "acceptable" in some sense. Suppose we have a two event parallel program *P* in which it is known that both events, say e and f, have finite response time for all values of N. Suppose further that event e has acceptable response time r(e) for N=t but f's response time is unacceptable for N<5t. Hence, we must adopt a 5t-fair scheduling policy to have any hope that both events will have acceptable response time. A basic question is: how is event e's response time affected by this increase in N? In this section we answer the question by showing that e's response time will increase only linearly in N.

We have the following definitions:

Definition: Let e be any event of a parallel program *P* and for N≥1, let Z=S_1S_2... be in SN. The *response time* of e in Z, denote r(e,N,Z), is

case 1: Z is a terminating schedule with S_n the final scheduler state and *blocked*(S_n,e). Then r(e,N,Z) is infinity.

case 2: Otherwise, r(e,N,Z)=max{*delay*(S_i,e)|i≥1}.

The *N-response time* of e, denote r(e,N), is r(e,N)=max{r(e,N,Z)|Z∈SN}.

Hence, there are two ways that r(e,N) might be infinite: the program could terminate with e blocked, or e might block and never execute again in spite of the fact that the program never terminates. This latter condition has been defined as "individual starvation" [7].

The following result is immediate from the definitions:

Lemma 6: Let e be any event of a parallel program *P* and for N≥1, let Z=S_1S_2... be in SN. If r(e,N,Z) > N then e is blocked in Z for r(e,N,Z) consecutive scheduler steps.

5.1 *Response Time Theorem*

We first prove the following lemma, which holds for general string systems.

Lemma 7: Let A be a finite set, w∈A*, and N≥2. If |w|≥2|A|N+2, then there exists an a∈A such that w can be decomposed as w=xayaz with |y|>N.

Proof: (by induction on |A|) If |A|=1 then |w|≥2N+2. The form of w must be w=aya where |y|≥2N>N.

Assume the result holds for |A|<k, for fixed k≥2. If |A|=k then |w|≥2kN + 2>2N + 2. Assume a is the first character of w and decompose w as

w=axy where |x|=N + 1. We have two cases:

(1) If *substr*(y,a) we are done.
(2) If *not substr*(y,a) then y is in (A-{a})* and |A-{a}|=k - 1. We have |a| + |x| + |y|≥2kN + 2. Thus |y|≥2kN + 2 - 1 - N - 1=2kN - N. Since 2 - N≤0, we have |y|≥2kN - N + 2 - N = 2(k-1)N + 2. By the induction hypothesis on y, there is b∈A-{a} such that y can be decomposed as y=x'by'bz' and |y'| > N.

We are now ready to prove our main result.

Response Time Theorem: Let P be a parallel program having n states and m events. For any event e of P either:

(1) There exist N≥2 such that for all M≥N r(e,M) is infinity, or
(2) For all N≥2 there is a constant c>0 such that r(e,N)≤cN.

Proof: Assume that r(e,N) is finite for all N≥2. Suppose there is an M≥2 such that for all constants c>0 we have r(e,M)>cM.

Let $d=n2^m=|T|$ be the number of total states of P and look at the constant c'=2d + 1. There must be a schedule Z in SM such that r(e,M,Z)>c'M. Let $Z=S_1S_2\ldots$ and let Y=*totalstate*(S_1)*totalstate*(S_2)....

By Lemma 6 we can decompose Y as $Y=X_1X_2X_3$ where $|X_2|$=c'M and e is always blocked in X_2. Thus, $|X_2|$=c'M=(2d+1)M=2dM+M≥2dM+2. Applying Lemma 7 to X_2, there is a total state X such that $X_2=X_4XX_5XX_6$ and $|X_5|$>M. Clearly, e is blocked in X.
Rewriting Y, we have $Y=X_1X_4XX_5XX_6X_3$ as the sequences of P's total states which correspond to the schedule Z.

Let L1=$|X_1X_4X|$ - 1≥0 and L2=$|X_5X|$ - 1>M. Look at the computation corresponding to the schedule Z: z=*makecomp*(Z). We can decompose z as $z=z_1z_2z_3$ where $|z_1|$=L1 and $|z_2|$=L2.

By the above arguments we have *tstate*(z_1)=*tstate*(z_1z_2), *blocked*(z_1,e), and e doesn't execute in z_2 (i.e., *not substr*(z_2,e)). Also, z_1z_2∈CM and $|z_2|$=L2>M. Hence, by Lemma 5, $z_1z_2z_2z_2\ldots$ is in CL2 and it follows that r(e,L2) is infinity. Thus, with this contradiction, r(e,M)≤cM for all M≥2.

As an interesting sidelight of this proof we have established an upper bound on c to be $1 + n2^{m+1}$.

6.0 *Additional Scheduler Policies*

We have defined only the minimum amount of scheduler policies needed to prove the response time theorem. Observe that the N-fair scheduler will make an arbitrary choice when more than one blocked event is capable of executing. Because of this, it possible that an event may have an infinite response time even though it is always capable of executing. A way to avoid this is by a *FIFO scheduling* policy, as has been suggested in [6,9]. We have

Definition: Let P be a parallel program and for fixed N≥2, let $Z=S_1S_2\ldots$ be in SN. Z is called an *N-fair FIFO schedule* if for all i>1 and arbitrary distinct events e and f, the following holds: S_i R(e) S_{i+1}, *blocked*(S_i,f), and τ(*pstate*(S_i),f) defined imply *delay*(S_i,e)≥*delay*(S_i,f).

In this definintion note that the release policy guarantees *blocked*(S_i,e). Intuitively, in FIFO scheduling when there is a choice of executing several blocked events, an event which has been blocked for a maximum number of scheduler steps is chosen. Since the FIFO policy is a restriction of N-fair scheduling, we have the following corollary:

Corollary 1: The response time theorem holds under an N-fair FIFO scheduling policy.

In certain applications it is desirable that the choice among blocked events be made on the importance of the events rather than on the egalitarian FIFO rule. This is called *priority scheduling* [3]. Each event is given a priority as follows:

Definition: Let P be a parallel program. A *priority function* is a total mapping θ:E ---> NN.

When several blocked events are capable of executing, the choice is made on the basis of maximum priority:

Definition: Let P be a parallel program with priority function θ. For fixed N≥2, let $Z=S_1S_2\ldots$ be in SN. Z is called an *N-fair θ priority schedule* if for all i≥1 and arbitrary distinct events e and f, the following holds: S_i R(e) S_{i+1}, *blocked*(S_i,f), and τ(*pstate*(S_i),f) defined imply θ(e)≥θ(f).

Note, unlike FIFO scheduling, it is possible under priority scheduling for an event to have infinite response time even though it is always capable of executing. Since priority scheduling is a restriction of the N-fair policy, we have

Corollary 2: The response time theorem holds under an N-fair priority scheduling policy.

7.0 *Response Time Decision Procedures*

There are two additional questions we must answer in order to completely describe the response time behavior of a parallel program:

(1) Given an event e of a parallel program P, is the response time of e ever infinity?
(2) What is the minimum constant c>0 which describes the linear growth of r(e,N) as N grows?

We will answer these questions by providing decision procedures for the following:

(a) Does there exist an N≥2 such that r(e,N)

is infinite?

(b) Given $c>0$ is $r(e,N)\leq cN$ for all $N\geq 2$?

The answer to question (1) follows directly from (a). Question (2) is answered by (b) and the observation in section 5 that the minimum constant is bounded above by $1 + n2^{m+1}$.

The decision procedures for (a) and (b) will be by reduction to questions about suitably encoded vector addition systems.

7.1 *Vector Addition Systems*

In this section we briefly review the definition of vector addition systems [10], their decision procedures that we will use, and relate these systems to our definition of a parallel program's scheduler.

Definition: A *vector addition system* of degree k, denote VAS, is a 2-tuple W=(v,V) where:

(1) The *start vector* $v\epsilon NN^k$=NN X NN X...X NN (k times).

(2) V is a finite set of *vectors*, each in ZZ X ZZ X...X ZZ (k times) where ZZ={...,-2,-1,0,1,2,...}.

Definition: The *reachability set* of a VAS W, denote R(W), is a subset of NN^k recursively defined as follows:

(1) $v\epsilon R(W)$.

(2) for $x\epsilon R(W)$ and $w\epsilon W$, $x+w\epsilon R(W)$ iff. $x+w\geq 0$.

We will be using the following two problems which are concerned with the reachability set of a VAS.

Definition: The *boundedness problem*: given an arbitrary $x\geq 0$ is there a $y\epsilon R(W)$ such that $y\geq x$?

Definition: The *reachability problem*: given an arbitrary $x\geq 0$, is $x\epsilon R(W)$?

A decision procedure for the boundedness problem can be found in [10]. The decidability of the reachability problem has recently been claimed in [13].

A link between a parallel program's scheduler and vector addition systems is that a VAS can represent any finite state control activity [10].

7.2 *A High Level VAS Language*

Rather than work with vectors of integers, it will be more convenient and convincing to give the VAS reductions in terms of a "high-level" nondeterministic VAS language. This approach has been previously used in [12].

There are five statement types in the language: initialization of variables, assignment, nondeterministic branch, testing the finite state control, and updating the finite state control. All but the first statement types may have a statement label. The syntax and semantics are as follows:

Initialization

var $v_1=a_1,v_2=a_2,\ldots,v_n=a_n$

The distinct variables $v_1,v_2,\ldots,v_n$ are initialized to the respective natural numbers $a_1,a_2,\ldots,a_n$. Variables not initialized start at zero.

Assignment

$v_1 \leftarrow\!\!- v_1 + c_1,\ldots,v_n \leftarrow\!\!- v_n + c_n$

where the v's are distinct variables and the c's are integers. The assignment can take place only if $x_i + c_i\geq 0$ *for all* i. Otherwise, the VAS computation terminates.

Guessing

guess$(s_1,s_2,\ldots,s_n)$

This statement causes a nondeterministic branch to one of the statements labelled $s_1,s_2,\ldots,s_n$. If n=1 then the branch is deterministic.

Testing Event Activity

event(character)

This statement is used to see which events are eligible for event activity. It returns a list of eligible events, each prefixed by the supplied character. If no event activity is possible (i.e., the parallel program has terminated), then the list consists exclusively of the supplied character. For example, if events 1 and 2 can execute and event 3 can block, then *event*(s) returns s1,s2,s3. If no event activity can take place, the list would be s. *Event* is always used in conjunction with the *guess* statement, e.g., *guess*(*event*(s)).

Testing for a Blocked Event

blocked(e,s)

This statement causes a branch to statement s if event e is blocked. If e isn't blocked then the statement acts like a no-op.

Updating the Control

update(f)

Here f is either an event execution (e) or an event blocking (e'). This statement reflects in the finite state control the result of event activity f.

Globally, VAS programs are listed one statement per line and execution commences at the first statement. Execution proceeds sequentially until a *guess* is encountered, whence several nondeterministic computations may be spawned. A computation may terminate in the ways listed above or by executing the last statement in the list (when it is not a *guess* statement). Although not listed above, we also have a *no-op* statement with the obvious semantics.

7.3 *VAS Reductions*

We now show that the questions posed above are reducible to questions about suitable VAS systems.

Theorem 2: Let P be a parallel program having n states and m events. For an event e of P the question of whether or not there is an $N\geq 2$ such that r(e,N) is infinity is reducible to a boundedness question.

Proof: The vector addition system will have the following coordinates:

<finite state control, local control, $d_1,\ldots,d_m,M_1,\ldots,M_m$,B>

To facilitate the presentation of the VAS program we will employ obvious abbreviations described in comments and the following two macros:

zerodelay(<1>,<2>)
1: d<1> <--- d<1> - 1,M<1> <--- M<1> + 1
guess(1,<2>)

This macro takes two string inputs and does the usual concatenation. Its function is to try to set the delay counting variable d<1> to zero.

sucdelay(<1>)
blocked(<1>,1)
d<1> <--- d<1> + 1,M<1> <--- M<1> - 1
1: *no-op*

The purpose of this macro is to increase the delay count variable d<1> of a nonblocked event.

The VAS program is as follows:

var $M_1=1,M_2=1,\ldots,M_m=1$

comment: guess N

10: M_1 <--- M_1 + 1,...,M_m <--- M_m + 1
guess(10,20)

comment: Simulate P - guess to start phase 3 whenever e blocks

20: *guess*(*event*(2))
2: *guess*(20)

comment: the following group of statements is repeated for $1\le i\le m$.

2i: *zerodelay*(i,2ii)
2ii: *update*(i)

comment: the following statement is repeated for each j not equal to i.

sucdelay(j)
guess(20)

comment: the following group of statements is repeated for each i not equal to e.

2i': *zerodelay*(i,2ii')
2ii': *update*(i')

comment: the following statement is repeated for each j not equal to i.

sucdelay(j)
guess(20)

2e': *zerodelay*(e,2ee')
2ee': *update*(e')

comment: the following statement is repeated for each j not equal to e.

sucdelay(j)
guess(20,30)

comment: simulate P assuming that e never executes again.

30: B <--- B + 1
guess(*event*(3))
3: *guess*(30)

comment: as in phase 2, the following group of statements is repeated. However, here it is repeated for each i not equal to e.

3i: *zerodelay*(i,3ii)
3ii: *update*(i)

comment: the following statement is repeated for each j not equal to i.

sucdelay(j)
guess(30)

comment: the following group of statements is repeated for each i not equal to e.

3i': *zerodelay*(i,3ii')
3ii': *update*(i')

comment: the following statement is repeated for each j not equal to i.

sucdelay(j)
guess(30)

comment: by busy wait free, 3e' is impossible.

3e: *guess*(3e)

The result follows since r(e,N) is always finite iff. B is bounded.

Several comments are in order about this VAS program. In phase 1, by nondeterminism, every value of $N\ge 2$ is considered. In phase 2, the N-fair execution of the parallel program is simulated. An event activity as dictated by the finite state control is nondeterministically chosen and appropriate event delay counts are performed on the d variables. The variable pairs d_i and M_i play a crucial role in that they force only N-fair computations to be considered. Note that $d_i + M_i=N$ is invariant. When an event is executed or blocked we try to set d_i to zero. The crucial part of the simulation is to observe that even if d_i isn't set exactly to zero (it will be in some computation) we still get N-fair computations since M-fair computations are N-fair computations for M<N. Similarly, d_i is increased by 1 whenever event i is blocked and another event activity takes place.

Phase 3 is started nondeterministically whenever event e blocks in phase 2. The purpose of phase 3 is to assume e will never execute again and reflect this in variable B. If e does execute, then phase 3 loops forever and B is bounded. If the parallel program terminates (i.e., no event activity with e blocked) or e is never executed again, the B grows unboundedly.

For the second VAS reduction we will simply modify the above VAS program.

Theorem 3: Given c>0, whether or not $r(e,N)\le cN$ for all $N\ge 2$ is reducible to a reachability question.

Proof: A variable D is added to the above VAS program with an initial value of D=c + 1. Statement 10 is changed to
10: M_1 <--- M_1 + 1,...,M_m <--- M_m + 1,D <--- D + c
Hence, after phase 1 completes D has a value of cN + 1.

A fourth phase is added at the end of the program as follows:

40: D <--- D - 1,B <--- B - 1
guess(40)

The purpose of the fourth phase is to see if B ever is ≥ D. If so, then there must be some VAS computation in which B=D and thus r(e,N)>cN. This happens only when D=B=0 can be reached. It remains only to make changes to the VAS program to nondeterministically start phase four. They are:

(1) Change statement 2 to 2: *guess*(40).
(2) Change each *guess*(30) in phase 3 to *guess*(30,40).

Hence, r(e,N)>cN iff. D=B=0 is reached.

8.0 *Conclusions*

We have introduced the notion of an N-fair scheduling policy as a condition which allows the development of theoretical results on the response time behavior of parallel programs. We have shown that for any event either the response time is infinite or it is linear in the choice of N, that one can determine which is the case, and that one can compute the exact linear relationship in the finite case.

Although the methods used would seem to indicate that computing the exact response time behavior of a parallel program is an intractable task, the development of heuristics for computing good upper bounds on response time is under investigation.

References

[1] J. Cadiou and J. Levy.
"Mechanizable Proofs about Parallel Programs."
Fourteenth Symposium on Switching and Automata, October 1973.

[2] R. H. Campbell and A. N. Haberman.
"The Specification of Process Synchronization by Path Expressions."
Proc. Int. Symp. on Operating Systems Theory and Practice, April 1974.

[3] E. G. Coffman and P. J. Denning.
Operating Systems Theory.
Prentice Hall, Englewood Cliffs, 1973.

[4] E. S. Cohen.
"A Semantic Model for Parallel Systems with Scheduling."
Proc. Second Symp. on Principles of Programming Languages, 87-94.

[5] E. W. Dijkstra.
"Cooperating Sequential Processes."
In *Programming Languages*, ed. F. Genuys, Academic Press, New York, 1968, 43-112.

[6] E. W. Dijkstra.
"The Structure of the THE Multiprogramming System."
CACM 17,10 (May 1968) 341-347.

[7] E. W. Dijkstra.
"Hierarchical Ordering of Sequential Processes."
ACTA Informatica 1,2 (1971) 115-138.

[8] P. J. Gilbert and W. J. Chandler.
"Interference Between Communicating Parallel Processes."
Comm. of the ACM 15,6 (June 1972) 427-437.

[9] C. A. R. Hoare.
"Monitors: An Operating System Structuring Concept."
CACM 17,10 (October 1974) 549-562.

[10] R. Karp and R. Miller.
"Parallel Programming Schemata."
J. Computer and Systems Science, May 1969, 147-195.

[11] R. J. Lipton.
"On Synchronizing Primitive Systems."
Proc. Sixth Annual Symp. on the Theory of Computing, May 1974.

[12] R. J. Lipton.
"The Reachability Problem Requires Exponential Space."
Research Report #62, Dept. of Computer Science, Yale University, January 1976.

[13] G. S. Sacerdote and R. L. Tenney.
"The Decidability of the Reachability Problem for Vector Addition Systems."
In *Proc. of the Ninth Annual ACM Symp. on the Theory of Computing*, ACM (1977), 61-76.

Algorithmic Analysis Of Control Structure Behavior

by

R.M. Mattheyses and S.E. Conry*

Clarkson College, Potsdam, NY 13676

Summary

It is frequently convenient to view an asynchronous digital system involving parallel processing as being comprised of two parts: a control structure and a device structure. In this paper we are primarily concerned with analysis of the control structure for such a system.

The control structures studied in this paper are modular in nature. Their primitive elements are control modules which behave in much the same manner as many which have been previously proposed [1-4]. We have incorporated these modular control structures in a model for parallel processing systems and obtained necessary and sufficient conditions under which the control structure of such a system is well behaved. (Similar problems have been investigated by others [2,5].) When presented with a control structure of significant size, one immediately asks whether or not that control structure is well behaved. In this paper, three algorithms for determining whether or not a control structure is well behaved are presented and analyzed.

The first algorithm discussed is based directly on the necessary and sufficient conditions for "good behavior" that have been obtained. The two other algorithms are based on a more generative approach to the analysis of control structures. We define what is effectively a reduction system for control structures and show that a control structure is certainly well behaved if it reduces to a singleton node using the reduction rules given. This approach to the analysis of control structures is very similar in flavor to the analysis of control flow graphs for sequential programs presented in [6,7].

Some well behaved control structures exhibit peculiar structural configurations which cannot be analyzed directly using the reduction rules given. In these cases, some transformation of the control structure is necessary if reduction to a singleton is to be achieved. Two algorithms based on the "reduction rules" approach to control structure analysis are presented and compared. Each incorporates a different approach to analyzing the graph when structural peculiarities are present.

Both of the "reduction rules" algorithms begin by applying these rules until either a singleton node is produced or no further reduction can be done. If a singleton is produced, both algorithms terminate and return the result that the control structure is well behaved. If, on the other hand, no further reductions are possible, there are two possibilities.

The first "reduction rules" algorithm applies the direct algorithm to the remainder of the control structure at this point. The second algorithm performs local transformations on the remaining control structures and proceeds, attempting to reduce still further.

* Formerly Susan Conry Meyer.

This work was supported in part by the National Science Foundation under grant number MCS76-07681A01.

References

[1] J.B. Dennis, Modular, Asynchronous Control Structures for a High Performance Processor, Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, June 1970, pp. 55-80.

[2] J. Bruno and S.M. Altman, A Theory of Asynchronous Control Networks, IEEE Trans. Computers C-20, 6 (June 1971), pp. 629-638.

[3] M. Yoeli and J.A. Brzozowski, A Model of Parallel Computation Structures, Report CS-76-43, University of Waterloo, October 1976.

[4] R.M. Keller, Towards a Theory of Universal Speed Independent Modules, IEEE Trans. Computers, C-20, 1974, pp. 21-33.

[5] O. Herzog and M. Yoeli, Control Networks for Asynchronous Systems, Part I, Technical Report #74, Dept. of Computer Science, Technion, Haifa, 1976.

[6] R. Farrow, K. Kennedy, and L. Zucconi, Graph Grammers and Global Program Data Flow Analysis, Proceedings of the 17th Annual Symposium on Foundations of Computer Science, October 1976, pp. 42-55.

[7] K. Kennedy and L. Zucconi, Applications of a Graph Grammer for Program Control Flow Analysis, Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, January 1977, pp. 72-85.

[8] J. Hopcroft and R. Tarjan, Efficient Algorithms for Graph Manipulation, Comm. ACM, 16, 6, June 1973, pp. 372-378.

1977 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

L I S T O F R E F E R E E S

T. Agerwala
Arvind
K. Batcher
G. Baudet
P. Borgwardt
D. Boyd
P. Bruce Berra
M. Chandy
I. N. Chen
T. C. Chen
L. Cheung
E. Coffman, Jr.
J. Cornell
M. Daya
R. DeMillo
J. Dennis
H. Downs
C. Ellis
P. Enslow
D. Farber
T. Feng
E. Feustel
M. Flynn
C. C. Foster
G. Foster
M. Freedman
M. Freeman
W. Gaertner
O. Garcia
M. Gonzalez
K. Gostelow
I. Greif
W. Grossky
M. Hack
B. Hays
L. Higbie
J. Howard
M. Hu
K. Irani
C. Jensen
J. Jensen
R. Johnson
R. Jump
D. Kafura
T. Kehl
R. Keller
K. Kim
T. Kimura
A. Klayton
V. Klee
L. Kleinrock
P. Kogge
H. Kung
L. Lamport
D. Lawrie
E. Lazowska
L. Levy
D. Lipkie
J. Lipovski
C. Liu
M. Liu
H. Love
W. Meilander
S. Meyer
R. Miller
D. Misunas
G. Nutt
S. Oleynick
D. Padua
F. Preparata
L. Presser
C. V. Ramamoorthy
R. Rao
S. S. Reddi
O. Reimann
S. Robertson
J. Rothstein
S. Saunders
K. Schaffer
H. Schmitz
H. Shapiro
A. Shaw
J. Shore
H. Siegel
D. Siewiorek
A. J. Smith
E. Stabler
K. Thurber
G. Tjaden
F. Tung
O. Wing
D. Wise
R. Wishner
M. Wolfe
A. Yao
R. Zingg

1976 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

A U T H O R I N D E X

1976 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

A U T H O R I N D E X