UNIX and the Connection Machine Operating System

Brewster U. Kahle

William A. Nesheim

Marshall Isman

Thinking Machines Corporation
Cambridge, Massachusetts

OS88-1

ABSTRACT

The UNIX[1] operating system is used on several different computers that comprise the Connection Machine®[2] system. This makes the Connection Machine operating system a simple distributed operating system optimized for large data applications. Common minicomputers are used as front ends for the data processors, with one controlling the disk system, and other computers used as I/O processors. The front end acts as a center of control for the other parts of the system, and is connected to the data processors via an interface on its I/O bus. Programs run on the front end, and instructions are sent to the Connection Machine (CM) processors to read, write, and manipulate data in CM memory. CM file transfers are set up and monitored by the front end and the disk controlling computers while the data is transferred over a high speed I/O bus. Application specific data transfers are handled by other I/O processors over a general purpose interface to the high speed data I/O bus. UNIX is used as the operating system on these components to provide a unified operating system model for the system.

This paper will illustrate how the pieces of the Connection Machine System are integrated to provide a distributed operating system based on UNIX. The structure and the integration of the overall system will be discussed without going into detail on the design and implementation of each. Furthermore, a prototype method of timesharing a massively parallel machine will be outlined. Finally, a list of interesting issues still to be addressed in operating systems for massively parallel computers will be presented.

1. Introduction

The Connection Machine system is a data parallel computer that is designed to run data intensive applications. This system consists of a conventional serial machine as a front end, many tens of thousands of data processors, and a selection of I/O devices and processors. A program executing on the front end commands all the components of the Connection Machine system including the data processors, disk system, and I/O devices. The user program is in many ways similar to a conventional serial machine program using UNIX like devices, but at run time many interconnected computers are used to perform operating system functions. Controlling these computers requires a distributed

---

[1] UNIX is a trademark of AT&T Bell Laboratories.

[2] Connection Machine is a registered trademark of Thinking Machines Corporation.

operating system. This paper describes the Connection Machine operating system.

The Connection Machine operating system is a hybrid of UNIX and CM specific code that controls the components of the system. Since the CM is quite different from conventional computer systems, its computational model will be briefly described. An overview of the hardware and software components of the system are given, and the interface between UNIX and the various portions of the Connection Machine system are described in some detail. Further work on the Connection Machine operating system is suggested.

2. A Data Parallel Computer System

Two distinct types of parallelism can be found in today's parallel computers. *Control parallel* computers achieve increased performance by taking advantage of parallelism found in the control structure of programs. The Cray X-MP, BBN Butterfly,[3] and CalTech Cosmic Cube[4] are examples of control parallel computers. In these machines, each processor executes a portion of the program. Consequently, each processor must have capabilities comparable to the processor of a serial computer on which the same program could be run.

Data parallel computers achieve increased performance by taking advantage of parallelism found in the data of a problem.[5] The Connection Machine system, DAP,[6] and Massively Parallel Processor[7] are examples of data parallel computers. Data parallel computers consist of a single instruction engine, and thousands of data processors, each having local memory and connected to a communications network over which they may exchange information with other processors. There are two reasons the factors affecting the design of data processors in a data parallel computer are quite different from those affecting the design of the processors in a control parallel computer. First, the control aspects of a program on the data parallel computer may be executed by the instruction engine. This means the data processors are not required to handle instructions, and may instead be optimized for data manipulation. Second, data parallel problems have tens of thousands of data elements which may be operated on simultaneously with minimal interprocessor interaction.

Programming a data parallel computer is more akin to programming serial machines than to programming control parallel machines. Data parallel programs have only one control sequence, and the program executes on one processor, the front end. Thus, the program is running in a familiar environment with familiar tools. The data resides in Connection Machine memory and is manipulated by the CM data processors, which can access the memory of other CM processors by using a high speed intercommunication network. Similarly the front end can access the CM memory easily and efficiently. A major difference between data parallel and serial code is that iteration over the data objects is unnecessary, as all data objects can be operated on at once.

3. Connection Machine System Components

The Connection Machine system can consist of several front end processors, a dividable block of data processors, DataVault[8] disk units, high speed graphic display systems, and various I/O computers. While many combinations are possible, this section will describe one configuration in order to illustrate the function of each component in the system (see figure 1).

---

[3] Bolt Beranek and Newman Inc. *Development of a Butterfly Multiprocessor Test Bed*, Report No. 5872, Quarterly Technical Report No. 1, March 1985.

[4] C. L. Seitz, *The Cosmic Cube*, Communications of the ACM, Vol. 28, No. 1, January 1985

[5] Thinking Machines Corporation, *Introduction to Data Level Parallelism*, Thinking Machines Corporation Technical Report 86.14, April 1986

[6] Flanders, P.M. et al, *Efficient High Speed Computing with the Distributed Array Processor*, High Speed Computer and Algorithm Organization, Academic Press, 1977, pp. 113-127.

[7] Batcher, Kenneth E. (1980). *Design of a Massively Parallel Processor*, IEEE Transactions on Computers, C-29 (9).

[8] DataVault is a tradmark of Thinking Machines Corporation.

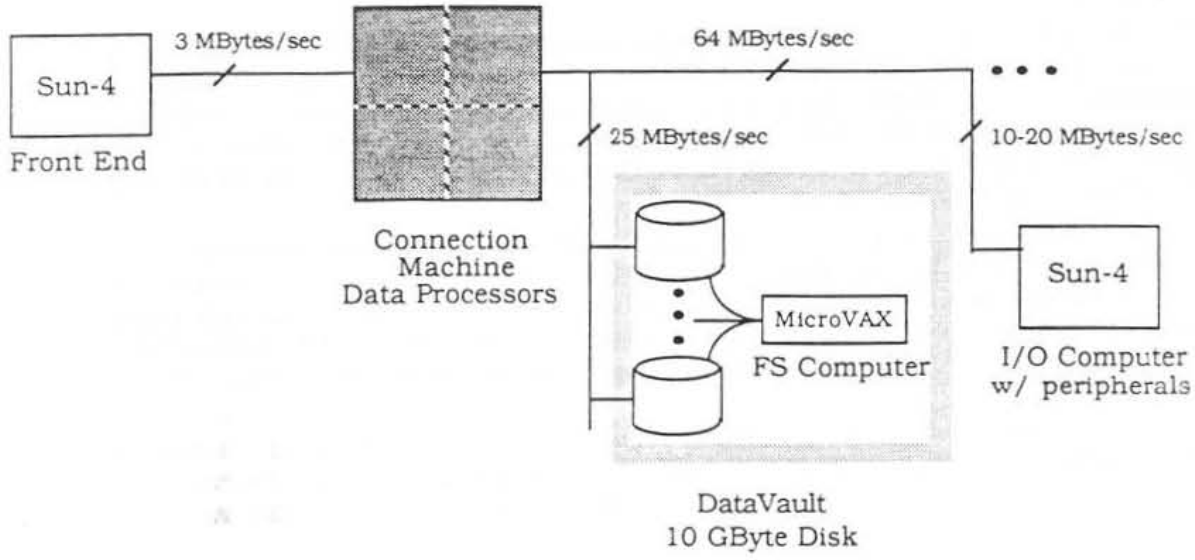

Figure 1: Example Connection Machine System

A Sun Microsystems®[9] Sun-4[10] workstation can be used as a front end to the Connection Machine processors in running user programs. This processor controls resource allocation, performs scalar computation and executes the control portions of Connection Machine programs. No programs are stored or executed in the CM data processors. The user programs run on the front end and issue instructions to the Connection Machine. Data may also be transferred between the front end and the Connection Machine processors via this interface.

The data processors store and process the larger data segments of an application. Each data processor operates on a different piece of data using the same instruction from the controlling front end processor. The data processors are very simple; a typical configuration consists of 16k - 64k processors containing 128 to 512 megabytes of memory.[11] A 64k processor machine performs a 4k by 4k matrix multiply at about 2500MFlops.[12] Each processor can efficiently access the memory of other processors and process the data within its local memory.

The connection between the scalar front ends and the data processors is a front end bus interface *(FEBI)*, a crossbar switch called the *Nexus*, and a *Sequencer* that takes macro-instructions and issues lower level instructions (nano-instructions) to the individual data processors. The FEBI is connected to the system I/O bus of the front end computer. Scalar transfers from the front end to the Sequencer run at approximately 2.5-3.5 MBytes/sec. Direct memory access block transfers are not supported on this interface because the bulk of the transfers are small instruction transfers from the front end CPU. The

[9] Sun Microsystems is a registered trademark of Sun Microsystems, Inc.

[10] Sun-4 is a trademark of Sun Microsystems, Inc.

[11] Thinking Machines Corporation, *The Architecture of the CM-2 Data Processor*, Thinking Machines Corporation Technical Report HA88-01, April 1988

[12] Thinking Machines Corporation, *Model CM-2 Technical Summary*, Thinking Machines Corporation Technical Report HA87-4, April 1987

Nexus connects up to 4 front ends to 4 sections of a CM to allow for flexible configuration. In our example system, we can assume that there is simply a direct connection from the Sun-4 to the CM. Data from the front end is written to a fifo in the Sequencer which, in turn, interprets these instructions calls or handles incoming data as appropriate. Results are relayed back to the front end through an output fifo. Each of these components also has a few status and configuration registers that are accessible from the front end.

The DataVault disk system consists a large number of standard disks operating in parallel, and a file server computer. The disk system can deliver data to the data processors or I/O interface computers at 25 MBytes/sec. The file server computer initiates data transfer operations, manages the CM file system, and provides facilities for disk maintenance and diagnostics. External data is available to the DataVault system through the I/O interfaces of the file server computer, or preferably via the 64Mbyte/sec CM I/O bus from an I/O processor.

The I/O interface system is a minicomputer with a VME bus interface, which is attached to the CM I/O bus via a special high speed interface. Data can be transferred between VME peripherals and the CM data processors or the DataVault under control of the Connection Machine operating system and the I/O interface computer. The speed of the connection depends on the peripherals and I/O processor involved, but 10-20 MBytes can be expected from optimized software and hardware combinations.

The resulting Connection Machine system can sustain high enough I/O bandwidths to keep the data processors busy, and have enough front end speed to handle several Connection Machine users. The remainder of the paper discusses the software and operating system tasks performed by these components.

4. The Connection Machine Operating System (CMOS)

The Connection Machine system operating provides many of the features found in a conventional operating system but implements them in a distributed manner. Parts of the CMOS run on each front end computer, the Sequencer, the file server computer and any other I/O interface computers. This section explains where the functions of resource allocation and management are performed. Further details on the components are discussed in later sections.

The CMOS functions generally are layered on top of the local operating system running on each computer within the CM system. Due to the number of different computers involved and the standard nature of the local operating system functions required, it does not make sense for the CMOS to be the native operating system on each computer in the CM system. Instead a common operating system, which is some variant of UNIX, is used on each computer. The front end computer is a Sun-4 running SunOS[13] or a VAX®[14] running ULTRIX[15], the file server computer is a MicroVAX running ULTRIX, and the I/O interface computer can be any VME based UNIX system. The Symbolics[16] Lisp Machine, as a front end, is the only non-UNIX computer that currently can be a part of the Connection Machine system. Each local UNIX is used to provide interprocess communication primitives needed to communicate among the distributed components of the CMOS. The local OS is also used to provide protection, local file storage, and switching between multiple local processes. The only modification to the local UNIX OS is a device driver for the CM specific device on each computer.

The *cmattach* command is used to obtain access to the CM system. This command is analogous to logging in to a UNIX system; once this operation is complete, use of the CM can begin. A *cmfinger* command, analogous to the UNIX command *who*, uses the same UNIX IPC mechanisms to list all current CM users.

---

[13] SunOS is a trademark of Sun Microsystems, Inc.

[14] VAX is a registered trademark of Digital Equipment Corporation.

[15] ULTRIX is a trademark of Digital Equipment Corporation.

[16] Symbolics is a trademark of Symbolics, Inc.

Process management and memory management are handled by a combination of the front-end computer and the Sequencer. All policy level decisions are handled on the front ends and some mechanism level functions are implemented by the Sequencer.

The Connection Machine File System (CMFS) provides a UNIX-like hierarchical file system with a programming interface that parallels the UNIX file system call interface. The file server runs on a MicroVax located inside the DataVault parallel disk subsystem. The file server daemon is an ULTRIX user process that takes advantage of UNIX facilities for path name resolution and communication operations to service CMFS operations. Portions of the file system code run on the front end computer, portions on the CM Sequencer, and portions on the file server computer. The front end and file server computers use UNIX IPC for communicating control information, and the high speed parallel CM I/O data bus is used for data transfers.

An I/O interface computer can also participate as one end of a file system transaction. This computer can act as a client of the DataVault, for transferring data to or from the disk subsystem, or as a server for the data needs of the CM data processors. This computer is interfaced to the high speed CM I/O bus via a VME interface board. This board allows a standard VME based UNIX host and all its peripherals to be a part of a CM system. The I/O interface computer uses UNIX IPC and other UNIX devices along with the CM specific hardware and software.

The CM graphics display consists of a CM specific board which resides in the CM backplane and a standard high resolution color monitor. The software necessary to display images runs on the front end and Sequencer. Access to this device is attained via a mechanism similar to that of *cmattach*.

Error handling facilities is distributed among all of the components of the CM system. There is communication among the components to pass error information and use of the underlying UNIX system to log errors and provide core dumps and debugging tools for user programs.

5. The Connection Machine Front End Subsystem

The control of the Connection Machine system is based in the front end processor. This section will describe the mechanism by which the front end system controls the CM data processors (see figure 2).

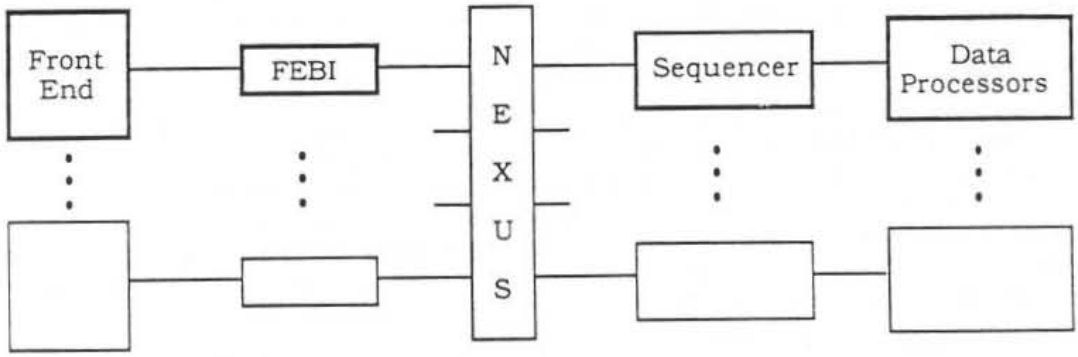

Figure 2: Front End Subsystem

The front end processor controls the front end bus interface (FEBI), the Nexus, and the Sequencer via 16 Connection Machine system registers. Four of these registers control the operation of the Nexus, while the remaining 12 access registers on the allocated Sequencer(s). Two of these registers are used to write to and read from fifo's in the Sequencer.

The ability to rapidly deliver instructions and get results from the Connection Machine is crucial to good application performance. For this reason the Sequencer registers are mapped directly into the address space of the front end system process currently "attached" to the Connection Machine system rather than using operating system calls to access the registers.

A process accesses CM resources by first gaining exclusive access to the FEBI. Access to the FEBI is controlled by the UNIX FEBI device driver. When a process desires access to the FEBI, it first opens an "indirect" CM device. This dummy device does not initially have access to the FEBI registers, and is in fact not associated with any hardware at all, but allows the user process to call driver *ioctl()* routines.

After successfully opening the indirect device, a process can issue a CONNECT_TO_INTERFACE *ioctl()* to request the driver to allocate a real FEBI. Blocking and non-blocking versions of this call take a list of requested FEBIs as an argument. Once successfully attached to a particular hardware interface, a process can map the FEBI CM registers into its address space. This mapping is accomplished on SunOS systems via a *mmap()* entry to the FEBI device driver, and via a special *ioctl()* in the driver on ULTRIX systems.

Gaining access to the Connection Machine processors requires the further step of setting up the Nexus to connect a given front end to the desired set of Sequencers and then initializing the CM processors, memory, and Sequencers. In some circumstances, new CM microcode is loaded into the Sequencers. For historical reasons, and for compatibility across our front-ends, the code which implements these functions is currently written in Lisp.

Users running Lisp under UNIX call C stubs to obtain access to the FEBI as described above. Then the Lisp process performs the Sequencer allocation and initialization functions. For non-Lisp access to the CM we have implemented a system which actually uses Lisp to do the allocation and initialization of CM hardware. The shell command *cmattach* calls the driver to access a FEBI. If one is not available, *cmattach* can optionally wait until a FEBI is available. Once the interface is allocated, a daemon process running a Lisp subprocess is called to set up and initialize the Connection Machine system. If the daemon indicates that the requested Sequencers are not available (for example if they are in use by a different front end), the cmattach command can sleep a short time and try the operation again.

Once the system is initialized, the cmattach program forks either an interactive sub-shell from which users execute their CM programs, or executes the user's program directly. The indirect device name is passed to the user program in an environment variable. When a CM application starts up, it opens the device named in the CMDEVICE environment variable, queries the driver via an *ioctl()* to make sure that the device is still connected to the interface, and then maps the FEBI registers into the process's address space.

Error handling is also done by the Lisp system. When a CM Exception is flagged by the FEBI, the runtime system calls the Lisp daemon to probe the hardware and interpret the error condition. Error information is written to a user definable error stream (generally *stderr*), and a user settable error function is called (by default *abort()*). Should an interface error occur, such as a parity error or bus timeout, the driver can modify the process's page table entries mapping the device to prevent further register accesses. The driver can also detach an indirect device from the hardware interface, allowing a process to be forceably detached from the CM.

Shell level commands are also provided for deallocating the CM *(cmdetach)*, finding out who is currently attached to or waiting for an interface *(cmusers)*, detaching another user or front end system from the CM *(cmdetach)*, and initializing the system *(cmcoldboot)*.

The front end system also acts as a center of control of CM I/O subsystem devices (figure 3). The simplest case is the Connection Machine graphics display system. The graphics display is connected to a interface board which resides in the Connection Machine system backplane. Under control of front end instructions, data is transferred directly from the Connection Machine processors to the display system. In order for a program to access the display system, the program must be attached to the section of the CM in which the frame buffer board resides. Thus allocation of graphics display systems is currently handled in the same manner as allocation of Sequencers and processors; the user either

gets exclusive access or no access at all.

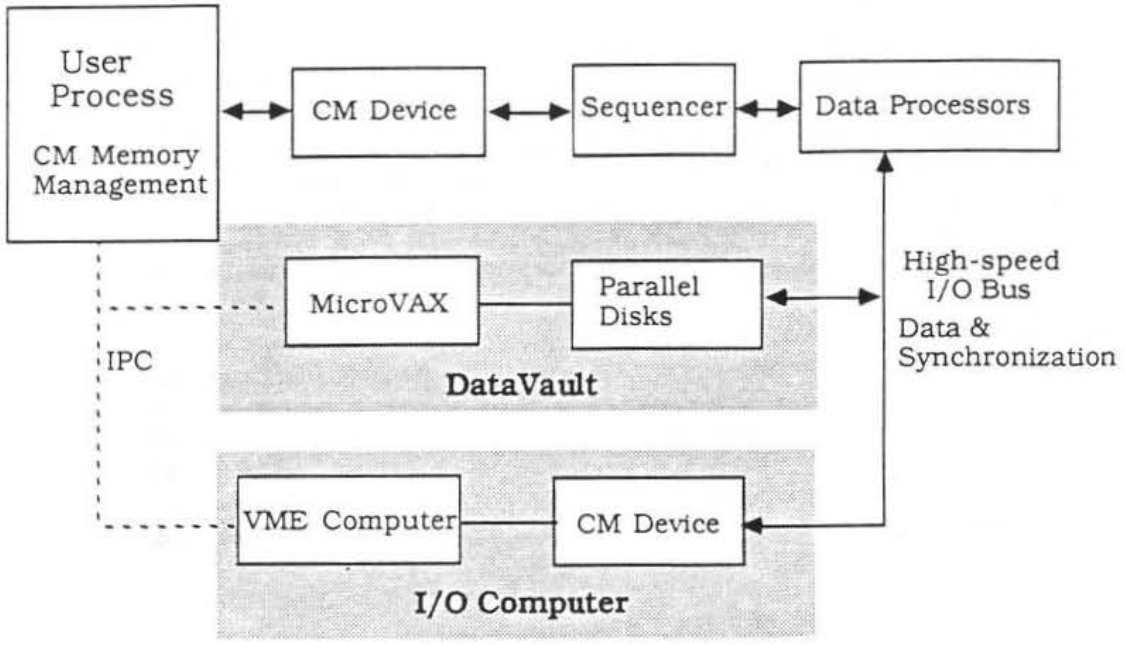

Figure 3: The front end and the I/O system.

I/O operations to the CM I/O bus are also initiated under front end control. Under front end control, the CM processors transfers data to the CM I/O controller board, and the I/O controller board is programmed to send or receive data on the CM I/O bus. Before a transfer on the bus is initiated, however, the front end system must arrange the other end of the transfer with the destination I/O subsystem controller host. These systems are are separate minicomputers running UNIX and in operations which transfer data between the CM and an I/O device, they operate in cooperation with the front end system.

When a process running on the CM needs to access data on the DataVault or retrieve data from a I/O processor, the front end system sends a command message to the I/O processor computer which then sets up the data transfer over the CM I/O bus. These command messages may be sent either via standard UNIX IPC mechanisms (TCP/IP networking), or via a command channel on the CM I/O bus. The I/O processor receives the message and passes it off to a CM I/O daemon server process. Another IPC message is sent back to the front end system to inform it of the success or failure of the operation. Data transfer is then initiated from the I/O processor to the CM via the CM I/O bus, freeing the I/O processor and the front end processor for other computations. Completion or error information is passed from the I/O processor.

6. User Process Memory Management

Memory allocation within a user process has some novel aspects that are appropriate for massively parallel machines. Data structures (memory fields) are allocated across all processors. This striping means that large arrays have elements across the entire machine. In fact, each element of a large array appears to have its own processor. Further, the dimensions of the data structures are not limited to the number of data processors because of a mechanism for simulating more processors. Currently, the CM does not support disk based virtual memory, so each user's entire memory space

must fit within the physical memory of the CM (.5 GBytes for a 64k processor machine). Memory allocation within a processor is performed by the run-time system in user specified increments rather than fixed pages. These increments are often small such as a 1 bit flag or a 32 bit number in each of the data processors. This section will address only those parts of memory allocation system that are necessary for understanding the operating system. For a further explanation of the programming model see *Introduction To Data Level Parallelism.*[17]

Since the memory structure in each data processor is the same, only one map to this structure must be kept for each process using the CM. This information, called the field table, is kept in the front-end processor. Some information about the most recently used fields is cached on the Sequencer for efficiency. All allocation and field management is done on the front-end by explicit calls from the user process. In fact, the table that stores all the field locations (memory stripes and their properties) is stored in the user process's address space. This is similar to heap and stack management within a language, but this functionality has been incorporated into the CM instruction set so that programs written in different languages can call each other. This field table is not used to share information between user processes.

The only restrictions that the current memory allocation procedure makes on the inter-user memory allocation organization is that each user must have a contiguous block of CM memory. User memory does not have to be zero based, so that a user can be located in any portion of memory. A user processes' memory is relocatable by looping through the field table, moving the data in the CM and updating the field table. The user does not have access to the physical location of the fields. Since the field table contains physical locations, relocation is expensive, but the run-time behavior is fast. Since the CM does not have hardware page tables, caching the addresses in this manner is desirable from a performance point of view.

7. Connection Machine Process Management

Currently, multiple users can access the Connection Machine system through a simple batch system. Under batch operation, a simple queue is maintained on the front-end and the data processors are completely reset between users. Each user gets sole use of the data processors, and all their associated memory and I/O devices. Under batch, only one front end process has access to the CM at a given time. Timesharing the Connection Machine between several users on the front end system can be achieved by allocating the FEBI device for short intervals to different processes as they are requested. Each process is given a fraction of the total CM memory, but when active, uses all the CM data processors. This section will describe this simple timesharing model.

Under a prototype timesharing implementation, each user is allocated a virtual CM with no interprocess communication, which is identical to the batch model of the machine with the single exception of having less memory available on each data processor (see figure 4). Timesharing of the data processors is done on top of the UNIX timesharing system of the front end. Only operations requiring the use of the CM data processors cause the system to determine whether the user can use the data processors at a given time. Manipulation of serial data, front end I/O, and user interaction all leave the CM data processors idle. Since most data parallel programs are actually a mix of serial and parallel code, most operations requiring the use of the CM data processors come in bursts. Because of this the CM process switching quantum can be quite large by traditional timesharing standards such as 1 second.

---

[17] Thinking Machines Corporation, *Introduction to Data Level Parallelism.* Thinking Machines Corporation Technical Report 86.14, April 1986

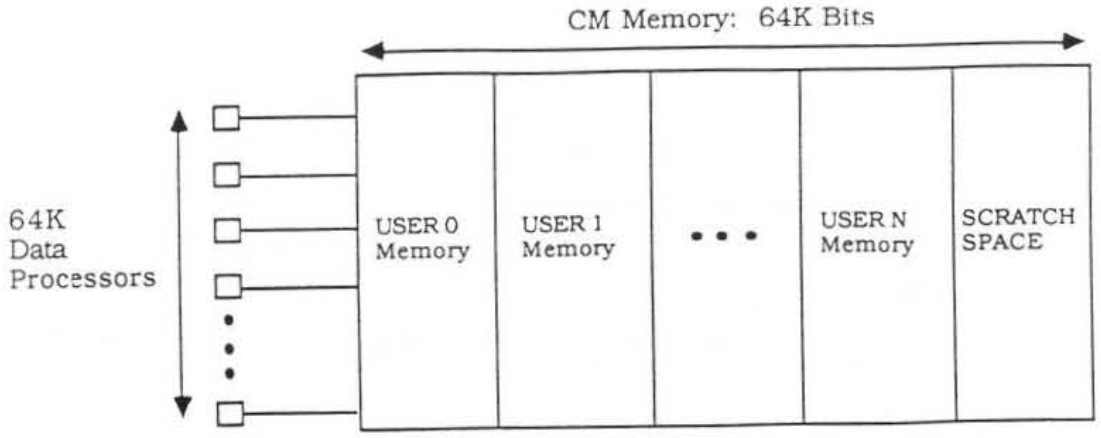

Figure 4: CM memory configuration under timesharing.

A prototype implementation of the CM timesharing system involves several pieces. Scheduling and resource management is handled by a daemon process. User processes requesting timesharing services use UNIX IPC primitives to communicate with the timesharing daemon. A structure is shared between the daemon and the user processes via UNIX shared memory facilities in order to implement an interlocking mechanism (see figure 5).

The timesharing daemon handles the registering of new users and switching between CM processes. A process gains access to the CM data processors by registering itself with the timesharing daemon. A UNIX IPC message is sent from the user process to the timesharing daemon requesting CM resources. The timesharing daemon sends a reply back to the user process indicating the success or failure of the request, and if successful, includes information on the amount of memory allocated to the user and a key for accessing a shared data structure. This data structure is used for most communication between the daemon and the user process.

The timesharing daemon can take the CM away from a user process anytime that process is not in a CM instruction. To implement this, all CM instructions check whether it is clear to run. If it is not clear to run then the process puts itself to sleep and waits for a signal from the daemon. If it is clear to run, then a flag is set in the shared data structure indicating that the user is actively using the CM. Once the CM instruction is completed, the flag is cleared. CM instructions are not interruptible by the daemon. To avoid a race condition, in fact, the instruction registers itself as using the CM before it checks to see if it has the right to actually use it, clearing the flag again if it is not clear to run. If the user process encounters an error on the CM it also gives up the machine, informing the daemon via another flag.

A further complication arises in handling errors that a user may have caused but not yet handled. A process may send many instructions to the CM before reading any results back. Error checking is generally done only at the time results are read back from the CM, so an error condition may exist for some time before a process notices it. In fact, one process may generate an error condition, and the CM switched to another process, before the first process is aware of the error. For this reason the timesharing daemon must check for any error conditions before switching CM processes. Since there are sophisticated error handling capabilities available to the user, an error does not lead to loss of state in the CM. The daemon uses another flag in the shared structure to indicate that an error has occurred in the background.

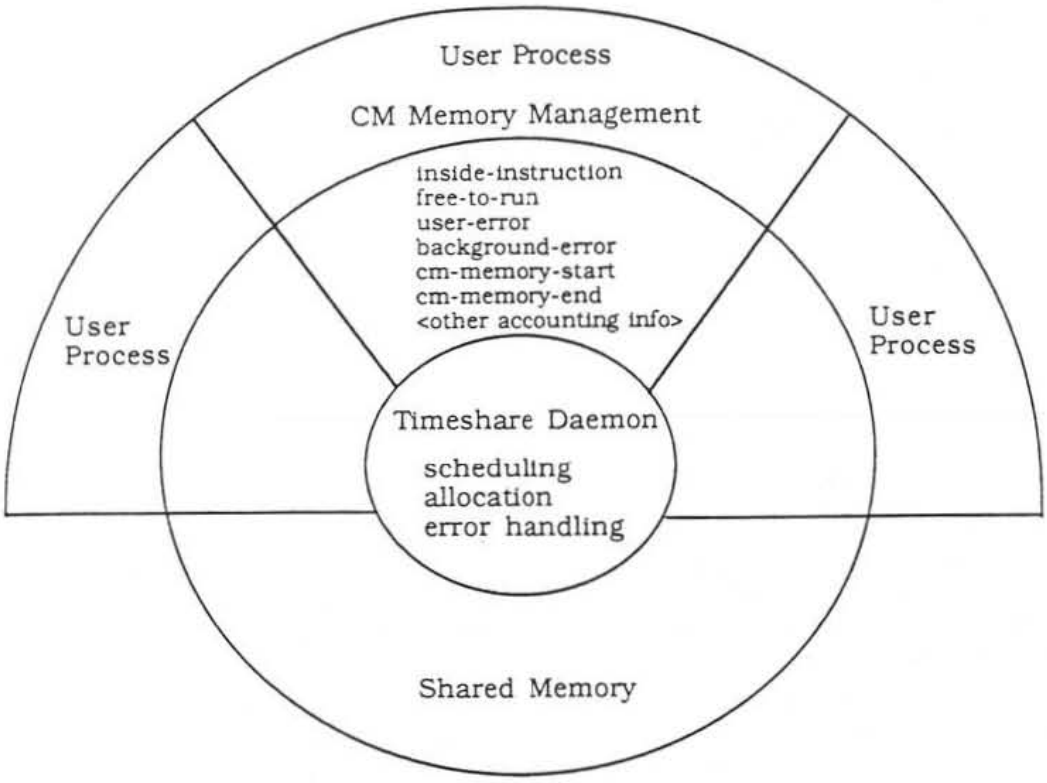

Figure 5: Timesharing control mechanism.

When more than one process is registered with the timesharing daemon, the daemon begins operation as a scheduler. Access to the CM data processors is granted in a simple round robin mechanism, with the daemon granting a process access to the CM by setting the appropriate flag bit in the shared data structure, and sending the user process a signal if it is waiting for CM resources.

Inter-user protection is accomplished by a variety of mechanisms. User state in the sequencer is unloaded and loaded again by the timesharing daemon process. User state in the CM memory is protected by hardware bounds registers that keep users from altering each others memory (either accidently or otherwise). I/O operations are left to complete before processes are switched, thus achieving safety and simplicity while sacrificing some performance.

As in all operating system code, code ordering is important to prevent race conditions. Since the master is not in the kernel at this point, many of these considerations are exacerbated. CM instructions are simple writes to the FEBI registers, so it is extremely unlikely that the UNIX system will block the current CM process for an extended period of time. Timesharing on a massively parallel machine needs further design and implementation work.

8. The Connection Machine File System (CMFS)

The Connection Machine File System is a high speed, parallel, hierarchical file system. It runs on a parallel disk subsystem called the DataVault which stripes bits across 32 data carrying drives and places error correcting code information on 7 additional drives. The software and hardware is designed

to maximize throughput for large data transfers. The programming interface is very similar to the UNIX file system with the read and write primitives causing data to be transferred from each virtual processor in the CM.

The CMFS has two main components. They are the file system library which is linked in with the user program and runs on the CM front end processor, and the CMFS file server processor which runs on the file server computer inside the DataVault. The system operates under a client/server model where state is maintained between calls to the file server. A simple file system protocol runs on top of UNIX IPC which provides the communication path between the clients and server which run on separate computers.

The file server provides pathname resolution, logical file to physical disk block mapping and block allocation. Although there can be more than one CM front end using a DataVault, the distributed file systems issues are substantially reduced by having a single server responsible for the file system. The division of labor between the client and server in the CMFS is similar to that in NFS[18]. The CMFS, however, uses a separate namespace for file names and uses a separate set of calls to access CMFS files. The separation was necessary to allow for large data transfers to occur in single I/O operations directly from the high speed CM I/O bus. The CMFS implements an extent based data layout. A file is made up of a variable number of extents where each extent is a variable sized, physically contiguous set of blocks on the disks. Only data is stored on the set of parallel disks while all directory and inode type information is stored in the UNIX file system on the file server computer.

9. General Purpose I/O Computer

The I/O Computer can be any VME bus based computer running UNIX. A VME interface board is used to transfer data from this system to the proprietary CM I/O bus. This interface (currently under development) allows standard computers and their peripherals to transfer data to and from the CM system at high data rates. By interfacing to a standard VME a wealth of peripherals available for those systems is made available to the Connection Machine system.

The software interface between the I/O processor and the CM system is the same as that used by the CM File System. When moving data to and from the DataVault the I/O interface computer acts like the client and executes almost the same code as the CM front end computer executes when using the DataVault. The I/O Interface Computer can also act as the server when exchanging data with the CM. In both cases UNIX IPC and other UNIX devices are used for the operations. Since the I/O bus speed of the CM is 64MBytes/sec, this interface is limited only by the speed of the VME bus and the peripherals involved in the transfer.

10. Future Work on Massively Parallel Operating Systems

Operating systems for massively parallel computers, like the Connection Machine, open many design issues. Many of the performance characteristics for paging and user loads, for instance, are different for these systems. Thus the opportunities for significant performance and functionality increases are unfolding. Some of the areas that need further study are:

[1] Sophisticated Timesharing

[2] Virtual memory: Paging and Swapping

[3] Integrated I/O subsystems

[4] Remote Access

Timesharing systems could, for instance, be extended to have a process model that supported IPC, forking, and shared memory, but the specifics on how this would perform or even be designed for such systems is not understood. Further, asynchronous I/O might boost system throughput of a timeshared system. Another issue is the desirability of tightly coupling the timesharing system with the front-end kernel to make smaller Connection Machine time quantums efficient. An efficiently timeshared parallel computer would greatly extend the number of potential users and make very large systems more

[18] NFS is a trademark of Sun Microsystems, Inc.

affordable. Accounting and performance analysis on a machine like the Connection Machine is an interesting problem because of the number of independent processors working on one program.

Paging and swapping could extend the data sizes users can easily compute on from .5 GigaBytes to 10-100 GigaBytes. How the page tables should be managed needs further study of existing programs based on how memory access patterns of data parallel algorithms. Caching performance is also effected in interesting ways in the common data parallel applications.

Various input/output devices can be used to make data-parallel machines more useful. One can imagine the uses for high speed general purpose networks, frame grabbers, music synthesizers, speaker systems, FAX machines, and the like, if they were cleanly interfaced and controlled by a large compute engine. Keeping integrated control over a large number of different high speed devices, some of which might be physically distant, is an interesting task of distributed control.

Using a high speed computers as a compute server in workstation environments opens interesting issues of data transfer, simulation, and splitting of user programs. Many institutions will have large parallel computers accessible on networks of various performances (from 100MBytes to 56k Bits). How users can best use these resources is not understood. Handling this smoothly will put demands on networks and operating system code.

Many of the system performance characteristics are different enough in a massively parallel computer, that the many operating system issues need to be re-examined. We look forward to research in this area of operating system design to increase the usefulness and performance of massively parallel computer systems.

11. Conclusion

UNIX is an important vehicle for unifying the distributed parts of the Connection Machine system. Instead of implementing a native UNIX for the CM, we have used serial machine's implementations on different platforms and have built what we need on top of it. Massively parallel computers share many of the system demands that a serial machines do, so we can use the interfaces and philosophy of UNIX in implementing operating system components for the Connection Machine system.

Further work needs to be done to make the Connection Machine operating system a true multi-user, virtual memory system. Since the control issues and memory use statistics are different from serial machines, some of the current OS work can be exploited directly while other aspects need to be examined anew.