A SHARED MEMORY ARCHITECTURE FOR DISTRIBUTED COMPUTING

Arvin Park
(Thesis)

CS-TR-127-87

December 1987

A Shared Memory Architecture For Distributed Computing

Arvin Park

A Dissertation
Presented to the
Faculty of Princeton University
in Candidacy for the Degree of
Doctor of Philosophy

Recommended for Acceptance by the
Department of
Computer Science

-December 1987-

Abstract

This thesis presents shared memory as a new paradigm for distributed computing architectures. It explains how shared memory can be used to simplify the programming of distributed applications, and how such systems can be efficiently implemented in a restricted domain of distributed computing known as *moderately-coupled* systems. The viability of moderately-coupled shared memory computing architectures is demonstrated through the implementation of ***Mind Meld***, a versatile interconnection which can be used to combine separate microprocessor systems into a single moderately-coupled shared memory distributed system.

Application codes written for ***Mind Meld*** have proven the system's ability to improve resource sharing and to enhance parallel processing capabilities. They have also demonstrated that shared memory can greatly simplify the task of distributed system programming.

Development of the ***Mind Meld*** system has generated new problems in multiprocessor system initialization. The *Processor Identity Problem* seeks to assign unique identities to a collection of identical processors which communicate through shared memory. Protocols that solve the *Processor Identity Problem* are developed and analyzed.

for Valerie

Acknowledgments

I would first like to thank my advisor Dick Lipton, a continual source of inspiration, who greatly enriched my work at Princeton. He taught me to ignore the superficial, and that the essence of research is inspiration not methodical plodding.

I'd like to thank Hector Garcia-Molina teaching me the craft of experimental research, and the craft of writing and presenting papers. I'd also like to thank Hector for suggestions which have greatly improved content and structure of this thesis.

I'd like to thank Rafael Alonso for his careful review of this thesis, and more importantly for his guidance through many of the pitfalls of graduate life and research.

I'd like to thank my office mate K. Balasubramanian for the many hours of conversation spent developing and refining ideas.

I'd Like to thank Kai Li for illuminating conversations which clarified my ideas on shared memory.

I'd like to thank Chris Zimmerman and Tarun Khanna two first rate hackers who programmed many of the interesting *Mind Meld* applications.

I'd like to thank Mike Orchard, Anna Karlin, and Mike Laszlo for the many weekend nights we spent together playing everything from Summertime to Hendrix to the Blues.

I'd like to thank Bill Lin and Abdou Youssef who helped me solve important proofs.

I'd like to thank Tim Snyder and Mark Weiss for a killer bachelor party, and for advice on thesis writing.

I'd like to thank Itzhak Brudney for hours of discussions on everything from nineteenth century Russian authors to Bach.

I'd like to thank Toshio Nakatani, Ken Salem, and E. S. Panduranga who greatly broadened my perspective on computer architectures, databases, and graphics.

I'd like to thank Daryl Porter, Ann Marie Spauster Luen Heng, and Boris Kogan for their company on hundreds of expeditions to Vesuvio's.

I'd like to thank Sharon Rodgers, Gene Davidson, and Gerree Pecht who were always on hand to oil the wheels of bureaucracy.

I'd like to thank my parents Sue and Kilho Park. Who have inspired me not merely through their words, but also by providing me with impressive role models to follow.

I'd especially like to thank my wife Valerie for companionship, constant support, and her careful revision of this thesis.

Contents

Chapter 1

Introduction

This thesis presents shared memory as a new paradigm for distributed computing architectures. It explains how shared memory can be used to simplify the programming of distributed applications, and how such systems can be efficiently implemented in a restricted domain of distributed computing known as *moderately-coupled* systems. The viability of moderately-coupled shared memory computing architectures is demonstrated through the implementation of *Mind Meld*, a versatile interconnection which can be used to combine separate microprocessor systems into a single moderately-coupled shared memory distributed system.

1.1 Shared Memory

Shared memory architectures differ from communication network architectures in a fundamental way. Typical communication network architectures are based on message passing between processors (see Figure 1.1.1). Shared memory architectures replace this message passing mechanism with a common shared memory through which processors communicate.

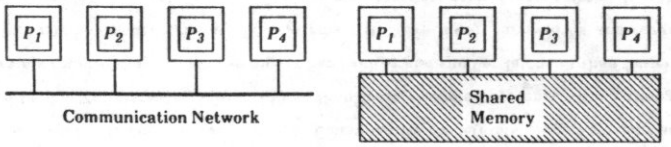

Figure 1.1.1: Communication Network Versus Shared Memory

There exist several variations of shared memory architectures. Some systems maintain a single common address space which is accessed by all processors. This memory may actually be distributed across all processors in the system, but it appears to be a single address space (see Figure 1.2.1a). Other shared memory architectures maintain separate local address spaces for each processor in addition to a common shared memory that can be accessed by all processors (see Figure 1.2.1b). In the most general case, processors possess sets of address spaces each of which can be shared by a subset of the processors in the system (see Figure 1.2.1c).

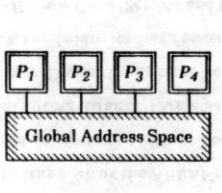

Figure 1.1.2a:
Single Addess
Space

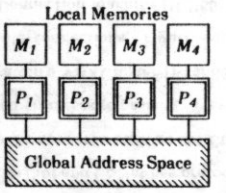

Figure 1.1.2b:
Local and Global
Adress Spaces

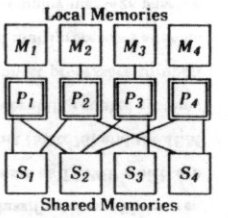

Figure 1.1.2c:
General Case

A number of recent tightly-coupled multiprocessor such as the Synapse [Fran84], and IBM RP3 [Bran87] have incorporated shared memory into their architectures. This is done by using multistage interconnection networks, or snooping caches to facilitate efficient sharing of memory between processors. This added design complexity is justified by a consequent simplification in programming for the multiprocessor systems. (This simplification is discussed further in Chapter 2). Some of the advantages which make shared memory attractive in tightly-coupled multiprocessor systems map into the domain of distributed computing.

This thesis extends the concept of shared memory architectures from its present tightly-coupled domain into a restricted class of distributed computing systems known as moderately-coupled distributed systems. Moderately-coupled systems lie between the extremes of tightly-coupled multiprocessors and loosely-coupled local area networks. Like local area networks they can connect a heterogeneous collection of independent processing

systems, and like tightly-coupled systems they can provide high bandwidth, low latency communication between processors.

Because of signal propagation delays it is impossible to connect processors over large distances with low latency interconnections. Consequently, moderately-coupled systems are limited to shorter interconnection distances than loosely-coupled systems. Even though moderately-coupled systems achieve higher bandwidths and smaller latencies than loosely-coupled systems, moderately-coupled systems still have lower bandwidths and larger latencies than tightly-coupled systems in which processors are packed more tightly. It is the contention of this thesis that moderately-coupled systems offer a valuable compromise between loosely-coupled and tightly-coupled computing systems.

1.2 Architectural Options for Multiple Processor Systems

The range of processor separations for different classes of multiple processor systems is illustrated in Figure **1.2.1**. Processors in tightly-coupled systems are packed as closely as

Wide Area Networks	10^3 to 10^7 meters
Local Area Networks	10 to 10^3 meters
Moderately-Coupled Systems	1 to 50 meters
Tightly-Coupled Systems	< 1 meter

Figure **1.2.1**: Proximities for Different Interconnection Classifications

possible to maintain low latencies and high bandwidth. They are usually separated by distances less than a meter. Moderately-coupled systems operate in the range of 1 to 50 meters of separation between processors. In this range they can connect collections of independent processors, while still maintaining some of the high bandwidth low latency characteristics of tightly-coupled computing systems. Local area networks such as the Ethernet [Metc76] connect machines separated by tens or hundreds of meters, in this range performance can still be provided to support sharing of printers and file servers between processors. Wide area networks such as the ARPANet [Klei75b] connect machines which are

separated by up to thousands of kilometers. These typically support applications such as electronic mail.

For each of these interconnection classifications, it is possible to employ either message passing or shared memory as the communication mechanism. The set of possible design options is illustrated in Figure **1.2.2**.

	Tightly-Coupled Systems	Moderately-Coupled Systems	Local Area Networks	Wide Area Networks
Shared Memory	*RP3* *Firefly* *CM**	*Mind Meld*	*Ivy*	?
Message Passing	*Cosmic Cube* *Connection Machine*	*VAXClusters*	*Ethernet* *IBM Token Ring*	*ARPANet* *CSNet*

Figure **1.2.2**: Shared Memory and Message Passing Architectures

Message passing has been used across the entire range of classifications. Wide area networks such as the ARPANet [Klei75b] and CSNet use protocols such as TCP/IP which are based on message passing. Local area networks such as the Ethernet [Metc76] and the IBM Token Ring [Brant87] are also based on message passing. New moderately-coupled systems such as Digital Equipment Corporation's VAXCluster [Kron86] connect processors through high bandwidth message passing interconnections. Tightly-coupled multiprocessors such as the Cosmic Cube [Seit85] and the Connection Machine [Hill85] also employ message passing as their communication mechanism.

Shared memory systems are not as common. A number of tightly-coupled systems such as the RP3 [Bran87], the Synapse [Fran84], and CM* [Gehr82] have been constructed. These systems all maintain some type of architectural support for shared memory: The RP3 uses a very expensive high performance multistage interconnection switch; the Synapse machine employs a set of snooping caches; the CM* system uses a set of hardware tables to map virtual addresses onto local processor memories. All of these machines maintain a common virtual address space which is accessible by all processors.

A prototype loosely-coupled shared memory system called *Ivy* has been implemented in software on top of the Apollo Domain system [Li86]. It allows a collection of loosely-coupled processors to share a single virtual address space. The *Ivy* prototype has demonstrated that several parallel applications of course granularity can achieve linear speedups on a set of loosely-coupled processors.

Shared memory wide area networks are represented by the box in the top right corner of figure **1.2.2.** Although it is theoretically possible to build a shared memory system across a wide area network, it will probably prove to be impractical. Wide area networks can potentially connect thousands of processors. Efficiently sharing an address space between this many processors may not be feasible. Also, the large distances traversed by communications across wide area networks produce large latencies. These latencies may make parallel processing tasks and sharing of resources (such as printer and file servers) impractical over such networks.

This thesis investigates the moderately-coupled shared memory multiple processor systems which lie in the shaded box of Figure **1.2.2**, and have until now been unexplored. A prototype moderately-coupled shared memory system called *Mind Meld* has been designed and constructed. This system connects a heterogeneous collection of microprocessor systems together into a single distributed computing system. This system provides high bandwidth, low latency interconnections between processors, and has made a wide range of new distributed computing applications possible.

This thesis argues that moderately-coupled systems can fill an important niche between loosely-coupled and tightly-coupled multiple processor systems where high performance communication is required between independent processors. It also attempts to show that shared memory is the best architectural paradigm to use in this domain.

1.3 Organization

This thesis is divided into 9 chapters: Chapter **1** is the Introduction; Chapter **2** compares shared memory with message passing; Chapter **3** describes how shared memory can be efficiently implemented in a moderately-coupled distributed system; Chapter **4** explores

the implications of moderately-coupled distributed systems; Chapter **5** describes the hardware implementation of the *Mind Meld* system; Chapter **6** documents performance of the *Mind Meld* system, as well as describing the innovative application programs which have been written for the *Mind Meld* system; Chapter **7** presents and solves the *Processor Identity Problem*. This solution enhances modularity in shared memory multiprocessing systems; Chapter **8** presents the design for a next generation shared memory system which provides better performance, and a cleaner shared memory interface than the present *Mind Meld* system; Chapter **9** presents conclusions of this thesis.

Chapter 2

Shared Memory Versus Message Passing

This chapter compares the relative merits of message passing and shared memory architectures for distributed computing. Shared memory can greatly simplify the task of parallel programming in a distributed system. It can also produce better system performance by eliminating the software layers used in message passing. Message passing insulates processors from one another. This simplifies protection and security, but hinders the sharing of data. Unlike shared memory systems, message passing systems can easily be extended to accommodate a large number of systems spread over large distances. These are a few of the issues which are discussed in this chapter.

2.1 Programming Simplification

A number of years ago, the major challenge in parallel processing was to build parallel processors. Many architectures were proposed, but a smaller number of them were built. Now that parallel machines exist, the major challenge is to program them [Kung87]. This has proven to be a cumbersome and time consuming task. This section demonstrates how programming a shared memory system can be easier than programming a message passing system.

Parallel machine programming typically consists of three phases. First an algorithm must be decomposed into parallel threads of execution. Some applications are relatively easy to parallelize, such as matrix multiply, and ray tracing [Pand87]. Other applications, such as solving for a non-linear recurrence relation with several variables have proven thus far to yield to no known parallel solutions [Smit86].

Next, the algorithm must be mapped onto the topology of a particular parallel architecture. Threads of execution must be dedicated to processors, and data motion between these processors must be routed through the data channels of a particular architecture. The routing operations occur under varying degrees of hardware control. Data transfers through intermediate nodes are sometimes automatically routed by destination tags or forwarding tables [Seit85] [Gehr82]. Other architectures require the explicit programming of data

motion between intermediate communication nodes [Barn68].

The last phase in a parallel programming task is to code the application in some type of parallel programming language. (There has been some research performed on compilers to parallelize serial code. However, these systems are still relatively primitive and inefficient.) Parallel programming languages are often standard programming languages which are augmented to support parallel processing. This usually means adding multiple process control primitives such as fork, join, and wait, as well as communication primitives such as send and receive.

Shared memory simplifies both the mapping phase and the coding phase of the parallel programming task. The mapping phase is less complicated because the system topology seen by the programmer is greatly simplified. All processors are assumed to access a common shared memory (see Figure **2.1.1**). If all processors have equal access to this shared memory, it makes no difference how parallel threads are mapped onto processors. (To improve performance, optimizations may be performed by the hardware or compiler, but these are transparent to the programmer.)

The coding phase is simplified because communication steps are not explicitly performed to move data between processors. Communication is accomplished through accesses to common data structures.

There is no concept of data movement between processors; all data appears to reside in shared memory. Of course, in reality, the data might reside in local snooping caches for performance reasons. However, this is all transparent to the programmer.

The advantages of parallel programming on a shared memory architecture are best illustrated by example. Consider the straightforward $O(N^3)$ Matrix Multiply algorithm. This is a rather simple parallel programming algorithm, but it demonstrates the programming advantages of shared memory. Other asymptotically faster algorithms exist that employ recursive decomposition techniques, but these will not be used here.

Assume there exist three N X N matricies which are labeled A, B, and C. We desire to multiply A and B together placing the result in C. Assume there exist M processors, where $M < N$ and M divides N. To parallelize this algorithm, simply dedicate each processor to a set of N/M rows of C and the corresponding N/M rows of A (see Figure **2.2.1**) (I will refer to these N/M row sections of matricies as slices.). Each processor multiplies its N/M row slice of A by the entire B matrix to produce a N/M row slice of C.

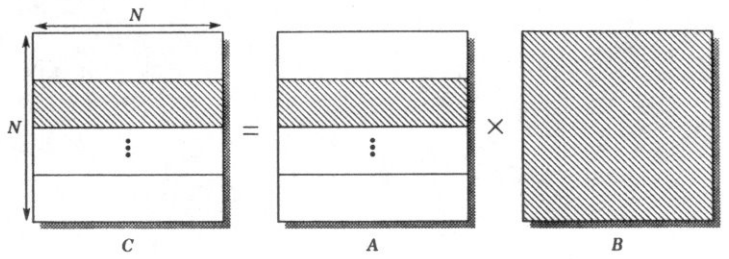

Figure 2.1.1: Parallel Matrix Multiply

Let us first program this using only message passing as the communication mechanism between processors. The data must reside in the local memory of one of the processors, call this P_0. The data must first be sent from P_0 to each processor. P_0 must then process its own small slice of the C matrix. After this , P_0 must receive results from the other processors and assemble the resulting C matrix. Pseudo code for this appears below:

Send(B to P_1)
Send(slice 1 of A to P_1)
Send(B to P_2)
Send(slice 2 of A to P_2)
⋮
Send(B to P_{M-1})
Send(slice M-1 of A to P_{M-1})
Process (slice 0 of C).
Receive(slice 1 of C from $P1$)
Receive(slice 2 of C from P_2)
⋮
Receive(slice M-1 of C from P_{M-1})
Assemble(C);

This task becomes much easier with shared memory. Assume the three matricies reside in shared memory. Each processor simply multiplies its own slice of A by the matrix B to produce its own slice of C. The code for this reduces to a single step:

Process(slice of C)

Even for this simple parallel matrix multiply algorithm, message passing communication operations complicate the programming. For more complicated parallel programs, which involve less uniform data motion, the complexity grows.

2.2 Protection

Shared memory systems provide easier access to shared data than message passing systems. This additional accessibility simplifies programming of parallel processing applications, however it also makes data security more difficult to ensure.

In message passing systems, remote data must be accessed through messages sent to the remote processor. To receive data from a remote processor, a request message must be sent to the remote processor. The remote processor then retrieves the data from its local memory which it then sends to the requesting processor in a reply message. All accesses to remote data must be channeled through the remote processor. This allows the remote processor to restrict access to certain portions of the remote data.

In shared memory systems, processors can directly access the shared memory without intervention of another processor. This is true even if the shared memory physically resides at a remote location. Since any single processor can read or write to shared memory without intervention of a remote processor, the data stored in this shared memory area is very hard to protect.

A protection mechanism can be incorporated into a distributed shared memory system by constructing a virtual addressing system for shared memory. In such a system each processor has its own mapping tables for shared memory. These tables map the processors addresses onto addresses in the physical shared memory. A processor can only access physical

shared memory which is mapped into its own virtual shared memory space. Portions of shared memory can be protected by only allowing a few processor to map to this shared memory. The design for such a facility is discussed in Chapter **3**.

2.3 Design Complexity

Shared memory systems are more complicated to build than message passing systems. Message passing can be implemented with a simple communication link, and appropriate hardware or software to maintain message buffers.

A distributed shared memory system must arbitrate between conflicting accesses to shared memory. In tightly-coupled shared memory systems, this is done by using multiple caches and some type of coherence protocol [Swea86], or by providing some type of blocking strategy in a multistage interconnection network [Bran87]. If shared memory is partitioned across a set of distributed processors, the shared memory system must map virtual addresses to different processors and local memory locations.

2.4 Shared Memory And Performance

Shared memory architectures can improve performance over message passing architectures by reducing latency for small data transfers. Because messages are typically of a fixed size, even single word data transfers require an entire message to be sent. A shared memory architecture manipulates individual words of data which may be quickly transferred between processors. (This is discussed further in Chapter **4**.)

Shared memory can also reduce software layers involved in message passing. These software layers provide services, such as assuring error free data transmission, establishing and maintaining communication links between processes, and providing high level abstractions for distributed applications programmers to use. However, these same layers also impede communication network performance. On message passing systems, the software tends to be the major performance bottleneck in point to point data transmissions. Shared memory provides a clean communications interface without the performance limiting

overhead of a large number of software layers. (Of course if shared memory is implemented through software layers itself, performance will still be limited. However, shared memory will still simplify programming.)

In typical message passing architectures, two types of data objects must be manipulated: memory words, and messages. In programming on a message passing architecture, one is continually loading memory words into messages, and messages into memory words. In a shared memory architecture there exists only data words (see Figure **2.4.1**). The shared memory analog of a message transmission is a block copy of data from one portion of memory to another portion of memory.

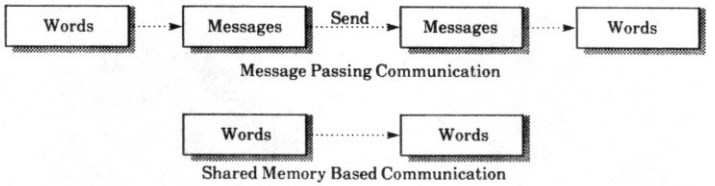

Figure 2.4.1: Message Passing vs. Shared Memory

To implement a high performance message passing or shared memory system special purpose hardware can be constructed. Note that if the hardware exists for a message passing system, it is possible to implement a shared memory system on top of this in software [Li86]. However performance will be sacrificed since transferring a single data word between processors will require sending an entire message. (Remember passing a message requires intervention of the remote processor and therefore involves longer latencies than directly accessing shared memory.)

It is easier to construct an efficient message passing system (in software) on top of a hardware shared memory system than it is to construct a shared memory (in software) on top of a hardware implementation of a message passing system. This is because words of memory are more atomic than messages. Data structures such as packets and queues of packets can be

efficiently constructed from smaller words of shared memory. It is not as easy to construct a system which efficiently manipulates shared memory words using packets.

2.5 Memory Mapping Distributed Devices

Memory mapping has been the most commonly used technique for integrating devices into tightly-coupled multiprocessor and uniprocessor systems. Hardware devices, and peripherals such as disk drive controllers, color graphics adapters, Ethernet taps, serial/parallel ports, and attached special purpose processors are all memory mapped. Placing hardware devices, and peripherals into the same address space presents processors with a uniform word-oriented view of the world.

Memory mapping reduces writing device drivers to standard programming tasks. The same programming language instructions that move data between main memory locations can move data between peripherals and main memory. This advantage has made memory mapping the most popular method of integrating tightly-coupled systems.

If the proper interfaces are provided, distributed devices could also be integrated into a memory mapped environment (see Figure 2.5.1). Processors would then have a uniform view of memory, local peripherals, and distributed devices. Distributed system programming would become identical to centralized system programming.

For example DMA from local disk controllers would be identical to DMA from remote file servers. In fact, the local DMA controller can be used to write to the shared memory of another processor. Other system tasks are similarly simplified.

Local Memory	Local Devices	Shared Memory	Distributed Devices
Local Data Structures	Disk Drive Controllers Attached Processors Parallel /Serial Ports	Common Data Structures	File Server Controllers Remote Processors

Figure 2.5.1: Memory Mapping Distributed Systems

2.6 Global State

Another advantage of shared memory is the existence of the global state which resides in shared memory. This global state can be used to simplify the programming of applications which require global system information.

Resource management becomes easier with global state. The status of each of the system resources can be kept in shared memory. Processors desiring to use a printer, paging disk, or memory server, can find load status information in the shared memory. Controllers for paging disks, memory servers, or printers are freed from the task of continually responding to requests of status information from each processor desiring service. The controllers only need to periodically update status information in shared memory.

This global status information can reduce communication operations required to implement load balancing algorithms [Powe85][Alon86]. Load balancing refers to the task of balancing processor load across a distributed computing system by migrating processes from heavily loaded processors onto lightly loaded processors.

Algorithms to balance the load usually have a distributed flavor. If a processor is heavily loaded it asks a number of other processors for load information. It then picks the least loaded processor to migrate processes to. Many communication operations are required to gain status on the load of other processors.

If shared memory exists, a more conventional centralized algorithm may be used. If a processor is heavily loaded, it places its excess jobs onto a global run queue in shared memory. Lightly loaded processors can then execute jobs from this global run queue.

Storing commonly referenced load distribution information in global shared memory can greatly reduce this communication overhead. Global status information such as load distributions can be obtained in a system without accessing state information stored in shared memory. The load information exists, but it is distributed across all of the processors. To extract this information without shared memory, it is necessary to communicate with all processors.

2.7 Limitations of Shared Memory Distributed Systems

Message passing architectures can be easily extended to accommodate thousands of processors. This may not be true for shared memory architectures. Protection and contention resolution become very complicated for a memory shared between thousands of processors. A shared memory for such a large system may have to be hundreds of megabytes in size which would exceed the addressing capabilities for many processors (The IBM PC/AT can only address up to 1 M bytes of memory).

Message passing architectures can also connect processors over large distances (even thousands of kilometers). These large distances incur large latencies for data transfer which makes parallel processing across such distances impractical. This diminishes the usefulness of shared memory.

2.8 Conclusions

This chapter discusses tradeoffs between shared memory and message passing architectures for distributed systems. Both types of architectures have advantages. Shared memory systems are easier to program than message passing systems since processors can manipulate common data structures in shared memory. Shared memory can also cut through the performance limiting software layers of message passing systems.

Message passing architectures are easier to implement in hardware and software than shared memory systems. Unlike shared memory systems, they are also extensible to large numbers of processors and large distances.

Given the tradeoffs that exist, shared memory can be an important architecture for a restricted domain of distributed computing known as the *moderately-coupled* systems. In this domain, the benefits of shared memory can be realized in a system which is composed of a collection of processors connected across modest distances (up to 50 meters).

Chapter 3

A Shared Memory Architecture

This chapter discusses issues in designing a shared memory distributed system. In constructing such a system a number of issues have to be dealt with: 1) Cache Coherency. 2) Protection. 3) Synchronization Primitives. 4) Compatibility between different processor systems. These are discussed below.

3.1 Cache Coherency

System performance can be increased if data items are cached in local processor memories. Caching improves performance in two ways: 1) The access time to local memory is generally shorter than access time to global memory. Thus, if most data references access the cache, performance is increased. 2) Caching eliminates read contention to global shared memory. If only one port exists to shared memory, and if many processors attempt to read shared memory simultaneously, some of the processors will be forced to wait. Caching decreases contention for global memory, and thereby provides higher performance.

Most shared bus multiprocessor systems provide local caches to eliminate contention for the global bus and shared memory. These caches are fairly large. For example, 1 megabyte for the XEROX Dragon [McCr84], and 128 K bytes for the Stanford MIPS X. They also implement various cache coherency protocols [Swea86]. Both their large size and the coherency protocols they use make these caches slower than uniprocessor caches, which are purely built to reduce memory access times.

Another way to eliminate contention for shared memory is to provide multiple ports to the shared memory . If each processor has a dedicated port to shared memory, processors are free to perform simultaneous reads to memory without contention (This assumes the memory chips are also multiported.). Simultaneous write operations to the same memory location can also occur freely. However, the resulting data value may be indeterminate.

Furnishing multiple ports to a shared memory is not easy. One option is to make each semiconductor chip multiported. However, pinout limitations and problems with multiported memory cell design constrain multiported chips to very low densities.

Another option is to use a very fast RAM to service many slower ports. Ports would share access to the RAM in round robin fashion. Each port would appear to have unrestricted access to a slower multiported memory (see Figure **3.1.1**). Like multiported RAMs, densities for very high speed RAMs also tend to be less than densities for standard memory chips. So memory size is again limited.

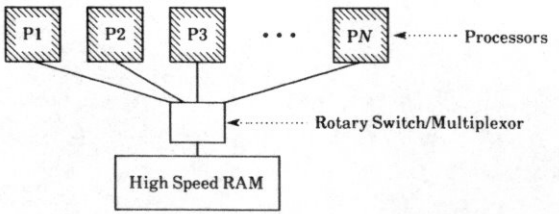

Figure 3.1.1: Implementing Shared Memory With A Single High Speed RAM

This round robin scheme can only accommodate a limited number of processors because a fast single ported memory can only provide a fixed amount of bandwidth. As this bandwidth is divided across larger numbers of processors, each processor sees less of the bandwidth.

To produce a shared memory that can service a large numbers of processors, it will be necessary to use a large number of memory modules. These memory modules can be connected to a set of processors through some type of interconnection network such as a Banyan or Benes network [Feng81] (see Figure **3.1.2**). Producing such a network can be expensive and complicated. Contention issues arise and choices have to be made between blocking and non-blocking switches. Global network control and routing is even more complicated [Yous87]. Large high performance interconnection networks exist today in such machines as the Butterfly [Crow85] and the RP3 [Brant87]. These extensible multistage interconnection networks can be applied to distributed shared memory architectures.

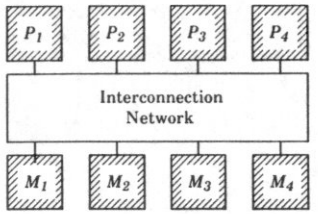

Figure 3.1.2:Interconnection Network

3.2 Multiported Cache

Using low density multiported or high speed RAM increases costs and limits the shared memory size. This can be a problem if large data sets are stored in shared memory. To increase effective memory size, the high speed or multiported RAM can be used as a cache for large amounts of slower single ported semiconductor memory (see Figure **3.2.1**). Note that no coherency problem arises because only one cached copy of the data exists.

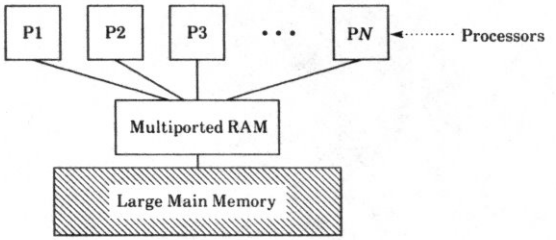

Figure **3.2.1**: Using Multiported RAM As A Cache for A Larger Shared Memory

This cache can similarly be shared in round robin fashion between the processors. Note that all processors will have to wait if a single processor generates a cache miss. This performance bottleneck can be alleviated by allowing other processors to access the cache while a single processor is waiting for a cache miss. In designing such a system care must be

taken to ensure that the cache block replacement process does not interfere with simultaneous cache accesses by other processors.

3.3 Protection

If all processors have access to shared memory, how can the system be protected from a malicious processor writing over important data structures in shared memory? Because of the large number of independent processors which share a distributed shared memory system, single processor failures can be relatively common. A protection facility is required to assure reliability in such an environment.

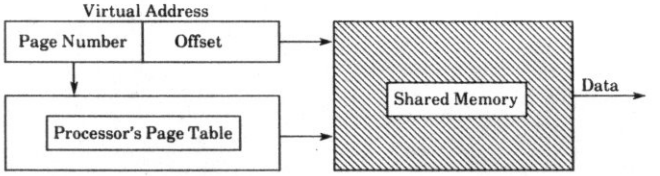

Figure **3.3.1**:Virtual Shared Memory System

One way to provide protection is through some type of virtual memory system (see Figure **3.3.1**). Mapping tables can be maintained in shared memory for each processor system. This is now commonly done in uniprocessor operating systems to provide insulation between separate processes. The mapping tables can be protected from processors that fail by allowing only one or a small group of processors to modify the mapping tables. (In uniprocessor systems, only system processes can access address mapping tables.)

This mapping mechanism will lengthen the access time to shared memory since an extra lookup must be performed to convert the virtual page number to the real page number. To improve performance page tables can be kept in a faster processor memory, or translation lookaside buffers can be provided to cache the most recently referenced real page addresses. This virtual memory hardware already exists for uniprocessor virtual memory systems [Digi81]. Much of this existing knowledge can be applied to the design of a virtual shared memory system.

3.4 Synchronization Primitives

In constructing a shared memory system, a test and set operation should be considered. A test and set operation allows a program to read the value of a variable and set it in a single atomic operation. No other processors may read or write to the variable between the test and set operations. Test and set variables can be used to implement monitors or semaphores to guarantee exclusive access to data structures or critical regions of code.

The test and set operation is included in many shared memory multiprocessor systems, but is harder to implement in a distributed system. In a tightly-coupled multiprocessor system, shared memory is often accessed through a bus. Extending a bus cycle to perform an atomic test and set operation is relatively simple, since bus arbitration logic typically guarantees that only a single processor can have control of the bus at any one time. Once this is assured, an extended bus cycle insures atomicity of a test and set operation.

This is not so easy to guarantee on a distributed system. If a multiported memory is used, hardware has to be provided at the memory chip or bus level to lock out all other ports when one processor performs a test and set operation. If a system consists of a heterogeneous collection of processors, it is unlikely that all processors include a test and set operation in their instruction sets. Adding test and set language primitives to these systems may require too much work to be practical.

Synchronization primitives can be provided in shared memory without test and set operations. The readers and writers protocol is one such example [Cour71]. For larger numbers of processors, it simply takes longer to test or set a lock. This added overhead may not significantly reduce performance because once a lock is placed, many operations are typically performed on the data protected by the lock. (A buffer might be copied, or a large data structure modified.) If this is this case, the overhead to test or set a lock is only a small fraction of the time to process the protected data.

For the reasons stated above the decision was made to not include a test and set operation in the design of the *Mind Meld* prototype. (The prototype is described in Chapter 5).

3.5 Compatibility Issues

Interfacing a heterogeneous collection of machines through a shared memory also involves other complications. Machines may have different byte orders. This can cause four consecutive bytes of memory to be interpreted as different integers by different machines. This permutation problem can be solved by choosing one byte order as the standard. Machines that do not conform to this standard byte order will have different hardware interfaces to shared memory that permutes bytes.

Differing word sizes can present compatibility problems to shared memory systems. Real number formats will be different between 64 bit and 32 bit machines. Also pointers to memory can vary in size between machines. A library of software tools can be made available to facilitate this conversion. The tools will convert real numbers and pointers to a single standard format before transmission to the shared memory. They will also convert from this standard format after data is retrieved from the shared memory.

This conversion process will be required for any distributed system composed of a heterogeneous collection of processors. The shared memory system will merely guarantee the transmission of binary data. Conversion between data formats will be left to compiler writers or application programmers who will be provided with a few simple tools.

3.6 Conclusions

This chapter has discussed design issues for a shared memory distributed computing system. A single multiply ported shared memory eliminates problems with multiprocessor cache coherency. This can be built by switching a single high speed memory between multiple ports. A virtual memory mapping mechanism can provide protection by restricting access to certain portions of shared memory. By interfacing processors at the byte level, compatibility can be maintained across a heterogeneous collection of processors.

A more detailed design for a shared memory distributed computing system is described in both Chapter 5 and Chapter 8.

Chapter 4

Moderately-Coupled Systems

This chapter investigates *moderately-coupled* distributed systems which lie between the tightly-coupled and loosely-coupled extremes. *Moderately-coupled* systems offer advantages over both tightly-coupled and loosely-coupled multiprocessor systems. Like loosely-coupled systems, *moderately-coupled* systems provide the ability to integrate heterogeneous collections of independent processing elements, file servers and printers. Like tightly-coupled systems they provide high bandwidth low latency connections between processors.

4.1 Loosely- and Tightly- Coupled Systems

Most existing multicomputer systems are either loosely-coupled or tightly-coupled. Loosely-coupled systems are characterized by physical separation of processing units, low performance interconnections, and distributed control. Tightly-coupled systems are characterized by close physical proximity of the processing units, high performance interconnections, and centralized control (see Figure **4.1.1**). These two types of systems reside at extreme ends of a spectrum of interconnection possibilities [Kron86].

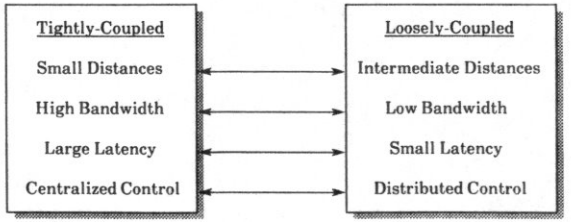

Figure 4.1.1 Characteristics of tightly-coupled and loosely-coupled systems

There is a great difference in bandwidths and latencies supported by loosely-coupled systems and tightly-coupled systems. A loosely-coupled system such as the Ethernet [Metc76] supports a point to point bandwidth of 500 K bits per second and an aggregate system bandwidth of up to 10 M bits per second (*Aggregate network bandwidth* refers to the maximum value for the sum of all simultaneous point to point bandwidths in an interconnection network.). Contrast this with the tightly coupled RP3 interconnection that can support 400 M bits/second point to point and 104.8 G bits/second aggregate bandwidth between 128 processor nodes [Pfis85]. They differ by four orders of magnitude!

Latencies vary greatly as well. The RP3 requires 500 nanoseconds to complete a single data transfer, while Ethernet can take milliseconds. Again, this difference is several orders of magnitude.

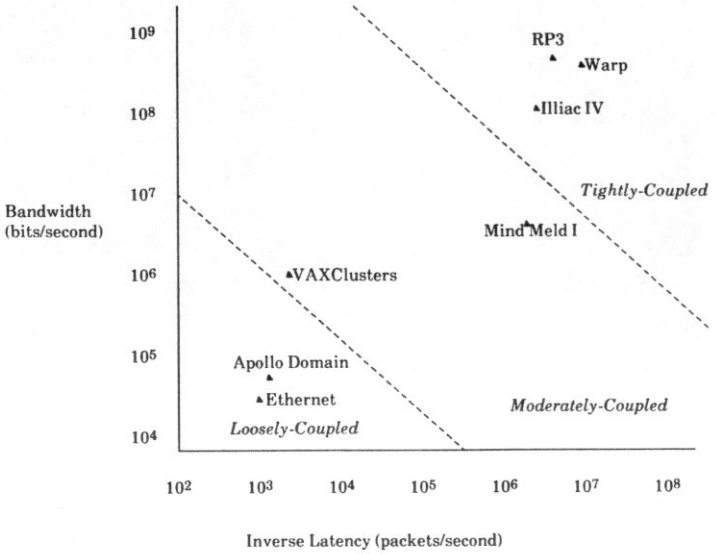

Figure 4.1.1 Bandwidth and Latency Metrics

A number of multicomputer systems are plotted on the graph of Figure 4.1.1. Maximum point to point bandwidth is represented on the vertical axis, and the inverse latency (packets/second) is represented on the horizontal axis. Tightly-coupled systems with high bandwidths and low latencies appear in the upper right corner of the graph, and low bandwidth high latency loosely-coupled systems lie in the lower left corner of the graph.

Moderately-coupled systems such as *Mind Meld* and VAXClusters fall into the intermediate range. Because moderately-coupled systems connect independent processors and workstations, they cannot support the extremely high bandwidth and small latencies of tightly-coupled systems in which all processors can be packed into a single box. However, moderately-coupled systems offer higher bandwidths and smaller latencies then loosely-coupled distributed systems. This added inter-processor communication performance can open up new capabilities for distributed computing systems.

These capabilities fall into three classes: 1) The first are strictly communication services such as mail and network news services; 2) Resource sharing capabilities have made loosely-coupled systems economically attractive. By increasing bandwidth and decreasing latency, new levels and types of resource sharing become possible; 3) Distributed computations will benefit through increased parallel processing capabilities (see Figure 4.3.1). These capabilities are discussed in the sections that follow.

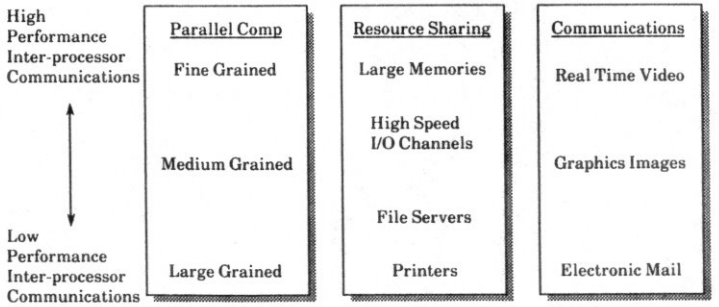

Figure 4.3.1 Applications for Ranges of Performance

4.2 Communication Services

Communication facilities such as electronic mail have made communication networks, such as the defense department's ARPANet and local area networks like the Ethernet, indispensable backbones for information transfer. Increasing the performance of communication links changes the the character of communication services.

If sufficient bandwidth is provided, text oriented data transmission can be augmented with graphics images and voice transmissions. Note that graphics images require large amounts of data. A 1K X 1K pixel black and white graphics image requires 128K bytes. A color image of the same size can take 3 M bytes. This is two to three orders of magnitude larger than typical electronic mail transmissions which consist of several thousands of characters of text.

Decreasing latencies and increasing bandwidths make real time sound and video transmissions possible. Telephone and picture phone services can be integrated into a computer communications systems. Integrating communication and computational services will change the nature of communication. Images and voices can be recorded digitally. Text, voice, video and data can be mixed into a single communication medium.

4.3 Resource Sharing

Resource sharing also changes. Performance can be increased for facilities which are presently shared, such as file servers and printers. Many workstations are now configured without local disks. They communicate with remote file servers for access to secondary storage. Even though secondary storage bandwidth is relatively low, the communication network is typically the bottleneck in transfers from file servers to workstations [Park87d].

For example, buffered file transfers from a DEC RA80 disk can proceed at 200 K bytes per second through the UNIX 4.3 BSD operating system. Remote file transfers from the same disk through the Ethernet/NFS proceed at one fourth of that speed. This remote file transfer facility can immediately benefit from up to a fourfold increase in bandwidth. Other higher performance disks and new parallel I/O systems can utilize even more bandwidth [Park86c].

Printers can use more bandwidth. Many text processing systems convert text to a symbolic form which is then broadcast to a printer. The printer then converts this text to the appropriate bit map before printing it. This conversion process can be off-loaded from the printer by simply sending bitmaps from the processor. Transmitting bitmaps across a low bandwidth interconnect is very inefficient. Higher bandwidth makes this more practical.

Increased bandwidths and decreased latencies provide the ability to share new types of resources. Large semiconductor memories may be shared between collections of processors. Instead of configuring each individual processor with large amounts of memory to service occasional memory bound tasks, a single processor can be equipped with large amounts of memory that can be shared between collections of processors.

A large shareable memory can be used two different ways. 1) It can be used as a buffer for file traffic between primary and secondary storage. This has been shown to greatly improve file system performance [Park87e]. 2) It can also be used to augment a processor's local memory. Note that accesses to this large semiconductor store will be slower than access to a local memory since additional propagation delays will increase latency. Accesses to local RAM might take 500 nanoseconds, while accesses to the large shareable memory might take twice as long. Although this large shareable memory may be slower than local memory, processors with little local memory will benefit greatly on memory bound tasks.

This large shareable memory is preferable to the alternative methods of increasing the usable program memory for individual processors. One can provide every processor with large amounts of local memory (which would be expensive), or one can use secondary storage to store a program's working set. Applications which use secondary storage to store their data sets can potentially run several orders of magnitude slower than programs that simply use memory [Garc87].

Fault-tolerance issues must be dealt with in constructing a shareable memory. If communication links to the shared memory fail, then mechanisms can be provided to prevent total processor failure. This problem has already been dealt with to some extent with processors accessing files on remote machines. If a communication request between processors fails, the system should attempt some type of retry. If this retry repeatedly fails, it will be impossible for the program to proceed and the program should trap to an error handler.

One would expect that communication link failures to a remote shareable memory will be more common than communication failures to local memory. For this reason it will probably be advisable not to run operating system code in the remote memory. If a user program is running in the remote memory and a communication link fails, then control can be passed to the operating system which is operating out of the local memory instead of simply halting the processor.

4.4 Distributed Computation

The third area that benefits from the improved performance of moderately-coupled systems is distributed computation. Lower latencies and higher bandwidths will make it possible to exploit parallelism of finer granularity.

Granularity refers to the size of units of work that can be performed in parallel before processors are forced to communicate. Some parallel tasks may be able to execute millions of instructions before processors are forced to communicate. Such applications are said to have large-grained parallelism. Other applications may have to communicate after just a few parallel instructions. These applications are said to possess fine-grained parallelism. As the granularity of an application becomes finer, communication delays become more significant. Processors can potentially spend all of their time communicating, and very little time computing.

Faster communication links between processors can efficiently support finer levels of parallel granularity. Parallel processors that efficiently support finer granularity can exploit more parallelism. Hence, architectures for parallel processing systems seek to minimize communication delays (latencies) and maximize communication bandwidths between processors. Consequently, processors in such systems typically reside on the same bus, circuit card, or even on the same VLSI chip [Lopr86].

Communication delays between processing nodes of a moderately-coupled system may not be as small as delays for a tightly-coupled system. However, the same parallel processing capabilities should be available at a slightly reduced performance level.

4.5 Conclusions

Moderately-coupled multicomputer systems fill an important niche between tightly-coupled multiprocessor systems and loosely-coupled local area networks. Like loosely-coupled systems, they can be used to connect collections of independent processors and workstations together. However they do not permit the flexibility of connecting processors over very large distances (>50 meters). Like tightly-coupled systems, they provide high bandwidth, low latency communication between processors which improves parallel processing performance, and permits new types of resource sharing for distributed computing systems. (Of course they cannot match the communication performance of tightly-coupled machines which are connected over much smaller distances.)

Chapter 5

Implementation

This chapter describes the implementation of *Mind Meld*, a hardware prototype of a shared memory distributed computing architecture. This prototype allows two separate microprocessor systems to communicate through a common shared memory. This implementation demonstrates the viability of a hardware supported shared memory interface between two microprocessor systems.

Since the prototype only allows pairs of processors to share a portion of real memory, multiple processors can only be interconnected using multiple shared memory interfaces. To fully exploit the capabilities of a shared memory distributed computing system one would ideally have memory which could be shared by a large collection of processors. The design of such a multiply ported shared memory system is described in Chapter **8**.

5.1 High Level Description

The *Mind Meld* prototype consists of a pair of IBM PC/AT circuit boards which are connected together by a 37 conductor cable (see Figure **5.1.1**). Each board fits into a separate

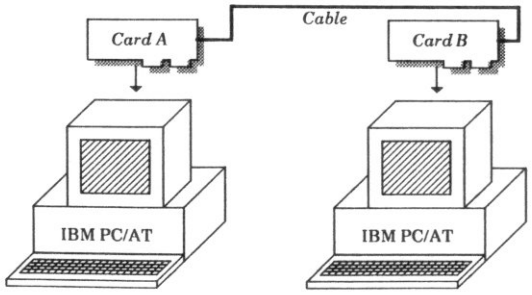

Figure **5.1.1**: Mind Meld Implementation

IBM PC/AT system allowing the two systems to share a portion of real memory. The boards are easy to install because they appear to be conventional memory boards. In reality the memory on these boards is shared between the two processors.

The boards can be configured to reside anywhere in the local PC/AT bus address space. These boards are typically placed at the high end of memory, below the system space as shown in Figure **5.1.2**. The system space is reserved for memory mapped peripherals and system ROM (read only memory). There exists a gap between the local memory and the system space. This gap can either be filled with additional local memory, or shared memory boards.

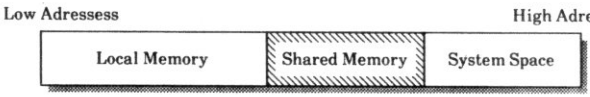

Figure **5.1.2**: Location of Shared Memory In PC/AT Address Space

Each pair of boards forms a point to point link between two IBM PC/AT systems. A number of these board pairs can be used to connect multiple systems together into a moderately-coupled multiprocessor computing system.

Consider each pair of boards to be an edge in a graph, and consider the processor systems to be nodes. Since each processor can accommodate up to five extra boards, an arbitrary graph of degree five can be constructed with a sufficient number of PC systems and links. A number of the topologies that have been considered appear in Figure **5.1.3**. Note that processors which are not directly connected have to communicate through intermediate processors.

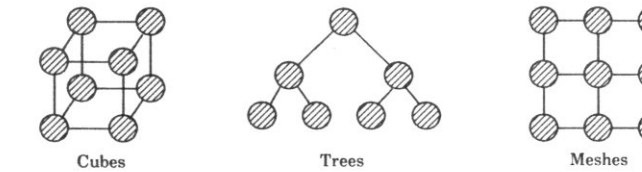

Figure **5.1.3**: Interconnection Possibilities

5.2 Block Level Description

The prototype consists of two different IBM PC/AT boards. One board actually contains the shared memory (call this board *A*). The other board must communicate over the inter-system cable to access the shared memory (call this board *B*).

5.2.1 Board A

Board *A* can be decomposed into five functional units as diagrammed in Figure **5.2.1**.

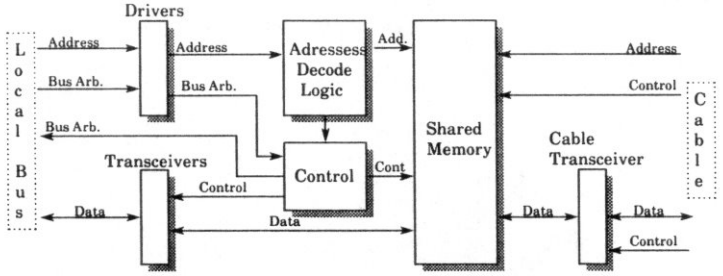

Figure **5.2.1**: Block Diagram of Board *A*

These units are: 1) Local bus drivers and transceivers. 2) Cable transceivers. 3) Local address decoding logic. 4) Shared memory. 5) Control logic.

The local bus drivers and transceivers pass data to and from the PC/AT bus. The drivers take signals from the bus and distribute them between multiple data inputs on board *A*. By presenting only one capacitive load to the bus, the drivers prevent overloading of the bus. They also effectively isolate board circuitry from possibly harmful bus conflicts. In the worst case, bus conflicts will only damage the bus drivers.

Local bus transceivers perform gating and buffering functions. On read requests, they gate and drive data from shared memory onto the local bus. On write requests, they gate and drive data from the local bus to the shared memory.

The cable transceivers gate data between shared memory and the inter-system cable. They perform the same gating and driving functions as the local bus transceivers; however, they are capable of sinking much more current to overcome the large capacitive load of the cable. In the present implementation, the cable can extend up to ten feet in length without limiting the data transmission rate. The cable length can be extended to 100 feet by placing active terminations on the signal lines as is discussed in section **5.4.2**.

The address decoding logic is used to determine if local bus addresses fall within the valid range of shared memory addresses. The starting address of this range can be set using DIP (Dual In line Package) switches to reside anywhere in the PC/AT bus's 16 M byte address space. The inputs of the DIP switches are fed into comparator circuits whose other inputs are taken from the address lines of the local bus. These comparator circuits generate a signal called *Address Valid* which indicates that a bus address falls within the legal range of shared memory addresses.

The shared memory sits between the local bus transceivers and the cable transceivers. It is composed of dual ported RAM. One port connects to the local bus transceivers and the other port connects to the cable transceivers. Read and write operations can occur simultaneously from both ports without conflict. The only possibly undefined situation occurs with simultaneous write operation from both ports (*A* and *B*) to the same memory location. In this case one of the sides (say side *A*) wins, and the write operation from side *B* fails without processor *B* being notified. (This outcome is the same as if processor *B* writes first and *A* subsequently overwrites the same location). Higher level software protocols are used to prevent the indeterminate outcome of simultaneous write operations.

The control logic takes in bus arbitration signals from the local bus, and the *Address Valid* signal from the address decoding logic. It then coordinates actions of the local bus, local bus transceivers, and the shared memory. This coordination process involves time critical propagation delays which must be carefully tuned. Coordination activities include; wait state generation, byte mode selection, and slave signal generation.

5.2.2 Board B

Board *B* is slightly less complex since it does not contain shared memory or two sets of transceivers. A block diagram for board *B* appears in Figure **5.2.2**. There are three major

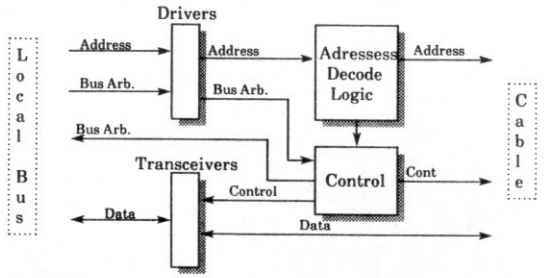

Figure **5.2.2**: Block Diagram of Board *B*

functional blocks on board *B*. They are: 1) Local bus drivers and transceivers. 2) Address decoding logic. 3) Control Logic. The functions are almost identical to the similarly labeled units on board *A*.

The local bus transceiver's and cable transceiver's functions have been combined into one unit. This set of transceivers gates and drives data from the local bus onto the inter-system cable. The local bus drivers are functionally identical to the board *A* drivers.

The board *B* address decoding logic is identical to the board *A* decoding logic. However, the DIP switches on board *B* may be set differently. This means the same shared memory (which is located on board *A*) can reside at different address locations in system *A* and system *B*.

The control logic on board *B* differs from the control logic on board *A*. The board *B* controller takes in local bus arbitration signals, and the *Address Valid* signal. It uses these inputs to generate read and write signals which are transmitted to the shared memory on board *A*. There is no special hardware to prevent write conflicts. All write signals are gated into the *B* port of the shared memory. Note that the shared memory has two independent data paths so write conflicts can only occur inside of the shared memory chips.

The control logic also synchronizes the cable transceivers on board *A* with the cable transceivers on board *B*. (The board *B* controller is able to control the transceivers and shared memory of board *A* through control lines of the cable.) For write operations the transceivers must gate data from board *B* onto board *A*. For read operations they must gate data in the opposite direction. Care must be taken to ensure that both sets of cable transceivers never try to drive the cable at the same time. The board *B* control logic also synchronizes the board *B* cable transceivers with local bus cycles of processor *B*.

5.3 Fabrication

Two implementations of the *Mind Meld* prototype were produced. The first was fabricated using wire wrap. The second was done on printed circuit boards. Both implementations used the same chips and circuitry with minor variations. SSI and MSI low power Schottky TTL chips were used to implement everything but the shared memory which consisted of dual ported RAM chips available through VLSI technologies (VT2131-PC). These VLSI RAMs support full dual ported access with 120 nanosecond worst case access time on read operations.

5.3.1 Wire Wrap Version

The wire wrap version was constructed on IBM PC/AT prototype boards available through Vector Electronics Inc. Two boards (A and B) were constructed. Each contained about 20 TTL and VLSI integrated circuit chips. These boards are connected with a heavily grounded 50 pin ribbon cable.

5.3.2 Printed Circuit Board Version

The second implementation was done on printed circuit boards. Like the wire wrap version, it also consisted of two boards (A and B). The schematics of the two boards were almost the same. Only a few connections were permuted, and more attention was paid to

electrical isolation. The cables and connectors were changed from non-standard 50 pin ribbon cables and connectors to standardized 37 conductor D type cables and connectors. The board layout, artwork, and fabrication was subcontracted to Precision Prototype of Garfield, New Jersey.

Ten pairs of boards were constructed in the first fabrication run. Fifteen of twenty boards worked immediately. In three of the five remaining boards, soldering errors were detected and easily fixed. The other two boards never worked. They probably have bad chips, but no time has been invested in chasing down these errors.

5.4 Hindsight

During the implementation of the *Mind Meld* prototype several design improvements have suggested themselves. These design improvements can be incorporated into future shared memory system designs.

5.4.1 Two Boards In One

Building two different types of circuit boards was a mistake. Most of the functional units of the *B* board are present on the *A* card. A single board could have been built which could act as either an *A* board or a *B* board. The mode (*A* or *B*) could have been selected using a jumper switch. This would have required additional jumper switches, multiplexers, and bus arbitration logic. About ten extra chips in all.

Instead of building two different twenty chip boards, A single board with thirty chips could have been built. This would have greatly reduced the time and cost involved in the printed circuit board fabrication. The two printed circuit boards required two sets of artwork, and two fabrication runs. Reducing this to a single, slightly more complicated printed circuit board would have reduced the work by almost fifty percent.

5.4.2 Active Terminations

I did not spend enough time considering transmission line effects on the 37 conductor cable which connects the two boards. I should have theoretically been able to extend this

cable about fifty feet without encountering problems with propagation delay. My boards can only operate at full speed with cable length of up to eight feet. This is because unmatched drivers and impedances cause tremendous transient line noise when driver outputs are changed.

These transmission line problems have well known remedies. Twisted pair cables and active terminations can reduce signal reflections [Ston82]. A new type of bus transceiver logic is also available which guarantees signal propagation without reflections [Bala86]. This standard transmission line lore should have been more carefully studied.

5.4.3 Address Space Problems

The *Mind Meld* boards were constructed to reside anywhere within the PC/AT bus's 16 M byte address space. I didn't realize at the time that only 1 M byte of this can be accessed by the segmented architecture of the processor. Out of this 1 M byte, only 640 K bytes can be used for programs and data. The rest is reserved by the system for use by memory mapped peripherals and the system ROM (see Figure **5.4.3**).

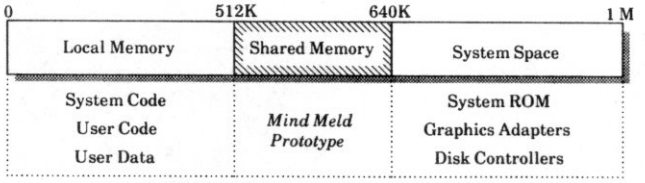

Figure **5.4.3**: Memory Map of the IBM PC/AT Bus

The *Mind Meld* boards presently occupy the region between 512 K and 640 K in the PC/AT address space. The extended memory that typically resides between 512 K and 640 K has been removed, reducing the real memory on the machine by 128 K bytes. It would have been preferable to locate the shared memory boards in the system space between 640 K and 1 M. This system space has many unused regions into which the *Mind Meld* boards could have been located. This was attempted, but the system initialization code presented an insurmountable obstacle.

At system boot time, the initialization code scans the area between 640 K and 1 M looking for memory mapped disk drive controllers and color graphics adapters. Any device that appears to be different from these (such as a *Mind Meld* board) causes an unrecoverable error. This code cannot be modified since it is burned into the IBM PC system ROM.

It is possible to get around the initialization code by designing the board so that it comes up in a disconnected state during the system power up processes. The board can be activated by writing to an I/O location which is mapped into an additional hardware location on the board. This technique is often used to place expanded memory boards in the IBM PC/AT system space.

5.5 Conclusions

The *Mind Meld* prototype has demonstrated that a shared memory interface can be constructed in hardware. Although this interface can only be used to connect pairs of processors through shared memory, large numbers of processors can be chained together using multiple shared memory links. Many unique applications programs have been developed for the *Mind Meld* system. These are described further in the next chapter.

Chapter 6

Application Codes And Performance Results

This chapter describes performance results of application codes written for the *Mind Meld* system. Besides providing high bandwidth, low latency communication between processors, the *Mind Meld* system presents a shared memory interface to distributed system programmers. This interface has simplified the programming task for a number of useful and imaginative programs that have been written for the *Mind Meld* system. These application codes demonstrate the power and flexibility of this high performance, shared memory distributed system.

6.1 Mind Meld Performance

This section compares performance between the *Mind Meld* and Ethernet [Metc76]. A comparison could have been made with a faster tightly-coupled system, but this would be less meaningful, since tightly-coupled systems do not have the ability to connect independent processor systems.

The *Mind Meld* system provides roughly an order of magnitude more bandwidth, and several orders of magnitude smaller latencies than the Ethernet (which is presently the most popular distributed system backbone). This added performance has been achieved at the cost of limiting the distances across which processors can be connected. The Ethernet can be used to connect processors distributed across hundreds of meters while *Mind Meld* can only connect processors connected across dozens of meters. Nevertheless, a comparison between *Mind Meld* and Ethernet is still significant since Ethernet (and other loosely-coupled systems) provide the only alternative method of connecting independent processors and workstations.

A comparison of the maximum bandwidths of *Mind Meld* and Ethernet appears in Figure **6.1.1a**. As this figure shows, the maximum possible bandwidth of the Ethernet system

is 10 M bits/second. The *Mind Meld* system's maximum bandwidth is over three times larger at 32 M bits/second.

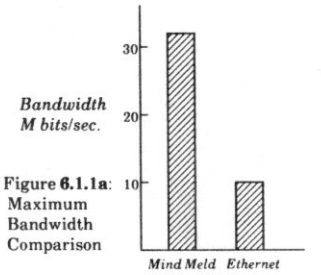

Figure **6.1.1a**: Maximum Bandwidth Comparison

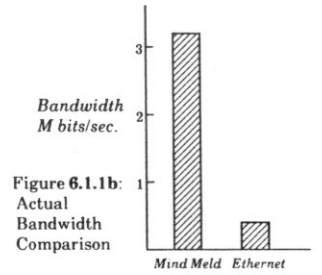

Figure **6.1.1b**: Actual Bandwidth Comparison

Maximum bandwidth may be interesting, but it is of little or no practical value. The actual point to point bandwidth that can be obtained through the software is the only significant metric for system users. A comparison of these point to point bandwidths appears in figure **6.1.1b**.

Note that the scale on this graph is in units of M bits/second instead of tens of M bits/second on the previous graph. Actual bandwidths are over an order of magnitude less than the maximum possible bandwidths. Many factors contribute to this performance loss. Among them are processor speed limitations, bus bandwidth limitations, and the overhead of software communication layers.

The actual bandwidth available to the programmer is 3.2 M bits/second for the *Mind Meld*, and 400 K bits/second for the Ethernet (The Ethernet bandwidth was measured between two VAX 11-750 processors running the UNIX 4.2 BSD operating system using the TCP/IP protocol). Even though Ethernet is only three times smaller in maximum bandwidth than *Mind Meld*, the actual bandwidth seen through the Ethernet software is eight times slower than for the *Mind Meld* Interface.

The Ethernet performs so poorly because of the software overhead involved in its message passing data transmission. In addition to the time required to send a message, time must be spent assembling, queuing and dequeuing messages. This extra message passing

overhead reduces the bandwidth of data transmission through the Ethernet by about an order of magnitude.

Message passing increases latencies dramatically. The *Mind Meld* system can transfer individual data words between machines at the memory cycle time of 1 microsecond. This latency time is a thousand times smaller than the Ethernet's millisecond latency through the UNIX-TCP/IP software.

Ethernet's latency is so much larger because it transmits data in large fixed sized messages. To transmit a single byte across the Ethernet, a large packet (of possibly thousands of bytes) must be constructed and transmitted between machines. Even reducing the message size does not decrease latency substantially. A message must still be constructed with the proper address header. It must then be passed to the software driver which sends the message to the input queue of the destination processor. The destination processor must then retrieve the message from the input queue. This process takes hundreds of times longer than the microsecond required for a *Mind Meld* data transfer.

The high bandwidth and low latency available through the *Mind Meld* system opens up new possibilities for distributed computing. These new possibilities and improved distributed system performance have been discussed previously in Chapter **4**. Application programs which exploit this added performance are described in following sections of this chapter.

6.2 Programming Primitives

A number of software primitives have been developed to access the *Mind Meld* shared memory. These are: 1) A message based interface. 2) An assembly language interface. 3) A C language interface. Out of these three, the C language interface is the simplest, and the most widely used. All three are described in the following subsections.

6.2.1 Message Based Interface

The first code written for the *Mind Meld* hardware was a set of message passing communications primitives. It was hoped that this high level abstraction would simplify

application programming for the *Mind Meld* system. Syntax for accessing these message passing primitives had the following flavor.

Send(message buffer , data pointer, number of bytes)
Receive(message buffer, data pointer, number of bytes)

These library functions transmit data through the shared memory between the source and destination processors. The message buffers are set up by the processors at system initialization time from a preset configuration file. The message buffer is located in shared memory along with associated synchronization primitives.

These synchronization flags were used to arbitrate data accesses between the reading and writing processors. For example, suppose processor *A* is writing to a buffer in shared memory and processor *B* is reading from the same buffer. If the data accesses are not arbitrated properly then processor *A* might overwrite the buffer while processor *B* is reading from it. To prevent this problem a single synchronization flag is used. This flag is initially set to 0 indicating that the buffer is empty. When processor *A* finishes filling the buffer it sets the flag to a 1 indicating that the buffer is full of data and is ready to be read. Processor *B* continues polling the synchronization flag until it detects a 1 and then reads the data from the buffer. After the data is read, processor *B* sets the synchronization flag back to 0 indicating that the buffer is again empty and ready for input from processor *A*. A diagram of this appears in Figure **6.2.1**.

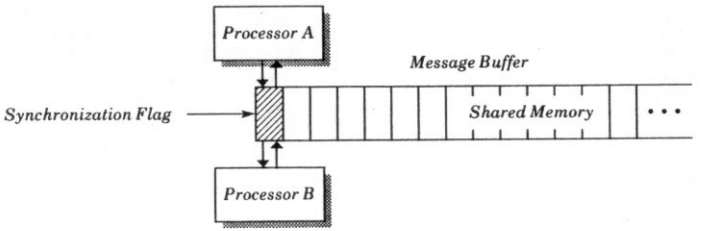

Figure **6.2.1**: Using Shared Memory As A Message Buffer

This message based interface has been used as the communication primitive in implementing the Screen Swap which is described in Section **6.6.1**. It proved relatively easy to use in this application. However, in writing these message passing primitives, we somehow lost sight of the advantages of a shared memory interface. This message passing interface introduced added data transmission overhead, and limited the flexibility of the shared memory system. Given the choice between using these message passing communication primitives or shared memory to program distributed applications, programmers overwhelming preferred the simpler shared memory interface.

6.2.2 Assembly Language Interface

Accessing the shared memory through assembly code is straightforward. The shared memory simply occupies a portion of the address space which can be accessed through real addresses.

The segmented architecture of the IBM PC/AT complicates this slightly. If the shared memory resides in a segment that is different from the program memory, segment registers must be manipulated to transfer data between the shared memory and the program memory.

The segment address of the shared memory is set using jumper switches on the *Mind Meld* board. Since the IBM PC/MS-DOS system does not support virtual addresses, this shared memory exists at the same segment address for all programs.

There can be possible conflicts if two processors try to modify the same piece of data at the same time. It is up to the programmer to guarantee that conflicting accesses do not occur. This can be done using dedicated synchronization variables which are stored in shared memory.

This method of ensuring conflict free access can work if all processors in the system cooperate. It may not be possible to assume cooperation in a much larger shared memory system. Issues concerning protection arise in such an environment. These are dealt with in the design of a future shared memory system in Chapter **8**.

6.2.3 C Language Interface

The main advantage of the C language interface is its simplicity. C language programmers are presented with a predefined array which is located in shared memory. This was accomplished by adding a single line "include" file to the C language library which defines a shared memory array.

```
/*SM.H: An include file which provides a C language interface to the shared memory*/
/*A character array/pointer is set to the hexadecimal address of shared memory*/

char *SharedMem = 0X9000;
```

The programmer can write the ASCII character 'A' to the number 3 byte of shared memory through the following C language program:

```
/*C program that writes a character to a location in shared memory*/
#include <sm.h>              /*include shared memory definitions file*/
main()
{
   SharedMem[3] = 'A';
}
```

A group of *Mind Meld* programmers were presented with the option of writing application programs with either the message based, assembly language, or C language interfaces. The C language interface was by far the most popular. It did not insulate the programmer from multiple processor synchronization; however, it was easier to understand, and much more flexible.

6.3 File Transfer Code

File manipulation commands were written to enable one processor to move files between its local disk drives and the disk drives of a remote machine. These commands are:

1) Copy 2) Erase. 3) Directory. 4) Rename. These commands are identical to their MS-DOS counterparts, except that they can operate on remote as well as local files.

These file access commands are transparent to the remote system. They can be used to access the remote disk drive even if the remote system is concurrently executing a program which accesses the same disk drive.

To implement this transparent operation, one would ideally run a background process which periodically polls the shared memory for file operation requests from remote processors. This was not possible because MS-DOS does not support multiple processes.

If the *Mind Meld* hardware included an interrupt facility, polling would not have been required. The design simplicity of excluding a hardware interrupt capability made up for the additional software overhead in polling.

6.3.1 Memory Resident Code

To provide the polling function described above from within MS-DOS, memory resident code was added to the operating system. On each timer interrupt, this code would poll the shared memory to see if any requests were pending. If so, the code would service these requests. The memory resident code also insured that local and remote disk accesses didn't interfere with each other.

This code was complicated by the fact that MS-DOS is not re-entrant. This means that certain MS-DOS functions cannot be called from within other MS-DOS functions. Since interrupts are themselves MS-DOS functions, this presented problems in writing file manipulation code from within the timer interrupt. This was remedied by hand-coding critical functions that would not work from within the timer interrupt.

All of this memory resident code was written in 80286 assembly language, which was a time consuming and painful process when compared with programming in a higher level language. The debugging tools available through MS-DOS were little help in trouble-shooting this interrupt driven code because the tools are themselves dependent on interrupts to function properly. (The memory resident code interfered with the interrupts used by the debugging tools.)

After a great amount of time and effort, all of these obstacles were overcome, and a working file transfer facility between remote processors finally came to life.

6.4 Matrix Multiply

A parallel matrix multiply program has been produced which effectively utilizes the processing power of four IBM PC/AT systems. The algorithm used is the standard $O(N^3)$ algorithm discussed in Chapter **2**. The program takes as input two N x N matricies, and produces an N x N product matrix. This program was run on four IBM PC/AT systems connected together in a linear chain using three pairs of *Mind Meld* printed circuit boards. (see Figure **6.4.1**).

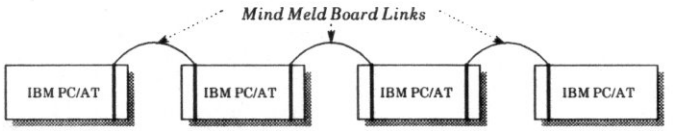

Figure **6.4.1**: Linear Chain Configuration

The problem was partitioned into four pieces, and each of the four processors was assigned to one of them. Unfortunately, the shared memory boards are only dual ported; data has to be transferred frequently through intermediate processors.

In a multiported shared memory system, this data transfer process would not have to occur. All of the data would have remained in shared memory. In spite of the complicated data transfer programming, the high performance *Mind Meld* links allowed rapid data transfer between computing systems.

A graph of Matrix Multiply performance results for different matrix sizes appears in Figure **6.4.2**. The matrix size is represented on the horizontal axis. A number 100 appearing on this axis refers to a 100 x 100 matrix. Speedup is plotted on the vertical axis. This is defined to be the running time of the best uniprocessor version of the algorithm divided by the running time of the best parallel version of this algorithm. (Some studies compare the running time of the parallel version of the program on a uniprocessor with the parallel version running on the parallel machine. I have used the most efficient algorithm for each

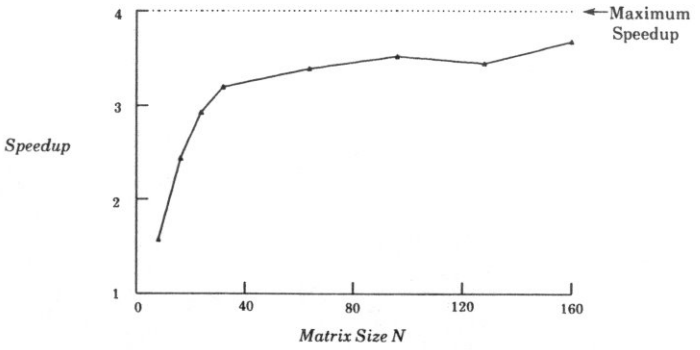

Figure **6.4.2**: Parallel Speedup for Matrix Multiply

machine.) This speedup has a maximum possible value of four since only four processors were used.

If the matrix is of size N x N, the matrix multiply algorithm involves $O(N^3)$ computation operations and $O(N^2)$ communication operations. As N grows larger, the data transfer time is dominated by the computation time. This effect can be seen as the speedup asymptotically approaches its maximum value of four as N increases.

Note the slight downward dip as the matrix size approaches 128 X 128. This is caused by the data set exceeding a single 64 K byte segment. At this point, both segment and address values have to be manipulated by the parallel program to access the data. This increases the running time of the parallel code. A similar effect takes place for the uniprocessor program at smaller values of N, but the resulting rise in speedup is less visible.

It is interesting to note that significant performance gains can be realized for very small matricies (Even of size 8 x 8). This demonstrates the *Mind Meld* system's ability to efficiently perform parallel computations of fine granularity. Matrix multiply can also achieve linear speedups in lower performance loosely-coupled systems [Li86]. However, this is not possible for smaller matrix sizes because of the large latencies involved in data transmission between loosely-coupled systems.

6.5 The Keyboard Switch

The keyboard switch code was written by Chris Zimmerman. It allows a programmer to operate multiple IBM PC/AT systems from a single keyboard. Keystrokes can be directed to either the local or remote processors. The keyboard is bound to a single processor until the "Alt" and "." keys are pressed simultaneously. This causes the keyboard output to be bound to the next processor in circular order. Up to four IBM PC/AT system screens can be comfortably viewed at once, so the keyboard switch facility can be used to allow a single user to operate four systems simultaneously. (The switch facility can be extended to more than four systems, however, it becomes hard to view more than 4 screens.) Keystrokes are transferred to the remote machine through the high speed *Mind Meld* shared memory interface. (This is diagrammed in Figure **6.5.1**) The Keyboard Switch code intercepts keystrokes through the

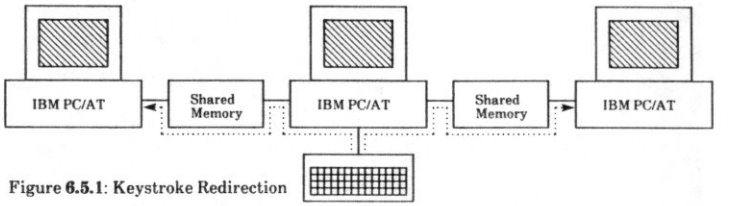

Figure **6.5.1**: Keystroke Redirection

low-level interrupt 9 of MS-DOS. It then directly writes to and manipulates system keyboard buffers. This code was carefully written so that applications which also use the keyboard interrupt are not disturbed by the keystroke redirection.

This keyboard switch facility has a number of useful applications. It provides an easy method for a single user to run multiple processes. By using the remote file transfer commands (described in Section **6.2**), executable files can be moved to and run on remote machines. This is very much like running a job as a background process in UNIX except that each process is running on a different CPU.

It is also possible to test compatibility between different versions of a program using this keystroke redirection. The keystroke program can be modified so that keystrokes from a single keyboard are broadcast to multiple processors. Each processor can be running a

different version of a program. If the versions are functionally the same, all the screens will remain identical. If not, errors will be easily detected. In this way, buyers of "look-alike" software can test compatibility between "look-alike" programs.

6.6 Multiple Display Applications

Besides providing additional computing power, multiple IBM PC/AT displays can be used in concert to view large data sets and graphics images.

The human mind is capable of assimilating more data than is available on a single IBM/PC display. Screens should be made larger, but large display tubes are bulky and hard to manufacture. Flat screen technology may make larger screens practical, but this is still many years away.

A typical screen can display about 2000 characters. People that need access to more information typically use multiple screens. A good example of this is a trading desk on the floor of one of the stock or bond exchanges. These desks are usually equipped with seven CRT monitors, three telephones, and a very hyperactive human being.

A number of applications which make use of multiple displays have been written for the *Mind Meld* system. These are described below.

6.6.1 Screen Swap

The first *Mind Meld* application to use multiple displays was a screen swapping facility that allows a user to swap screens between different IBM PC/AT systems. The IBM PC system appears to operate normally until a special key is pressed which causes the screens of the IBM PC/AT systems to be swapped.

This can be used to freeze a snapshot of the screen for later viewing. One can perform a directory listing on a collection of files, and then swap that listing onto the display of a second machine. The directory listing does not scroll off the screen as subsequent commands are executed on the first machine. Thus, the listing can be referred to without executing the directory command again.

The color displays on IBM PC/AT systems are memory mapped, so screen swapping simply entails moving data from the memory mapped screen buffer of one system, through the shared memory and into the screen buffer of the other system.

This screen swap involves moving 8 K bytes of data between the two systems. *Mind Meld* can do this in 1/50th of a second. (This is faster than the retrace rate for the video display.) The screens appear to swap instantaneously.

6.6.2 Banner

Banner is a program written by written by Chris Zimmerman and Tarun Khanna that utilizes multiple screens to display large graphics images. A graphics image can be painted across four screens. It then moves cyclically across the screens like a moving message display on Times Square.

The screen data is shifted one column at a time between the processors. The column which moves off the end of the display has to be transferred the entire length of the linear chain of processors to reach the start of the banner.

6.6.3 Balls

Balls is a program written by Chris Zimmerman and Tarun Khanna which displays a set of colored balls which follow trajectories across multiple screens. They are free to move anywhere within the confines of a rectangular box which spans the four contiguous screens. Balls bounce elastically against the borders of this box. A constant gravitational field is simulated which causes the balls to periodically bounce against the floor.

This program demonstrates *Mind Meld* system's ability to combine the processing power and screens of multiple systems to simulate and display real time motion of physical systems.

6.6.4 Wave

Wave is a program written by Chris Zimmerman and Tarun Khanna to display the propagation of a sine wave across a chain of jumping men which spans four displays. The wave propagation between processors is synchronized through shared memory variables. The jumping men were added for dramatic effect.

6.6.5 Video Game

A distributed video game has been written for the *Mind Meld* system by Chris Zimmerman. This video game enables four players sitting at separate IBM PC/AT systems to interact competitively in a single game.

The game's global state is kept at a central data base in one of the processors and is continually broadcast to the other machines utilizing the full 3.2 M bit/second bandwidth of the *Mind Meld* system. All four processors manipulate this data base without conflict at full animated video rates.

This application fully utilizes the computing power of a set of distributed processors. The high bandwidth, low latency communication capabilities of the *Mind Meld* system make rapid interactions between processors possible.

6.7 Future Mind Meld Applications

A number of *Mind Meld* applications are under developement. These include: 1) A distributed incremental compilation tool. 2) A computer aided design system which spans multiple workstations. 3) Using the memory of an IBM PC/AT system to store a large memory resident file system. (The extended memory of a single IBM PC/AT system can accommodate up to 15 additional megabytes of RAM based file system).

6.7.1 Parallel Compilation

This operates somewhat like the "make" utility of the UNIX operating system which compiles a group of source files, and links them into a single executable program. The "make"

utility supports incremental compilation which means that files are only recompiled if they are altered between compilations. The "make" utility also provides a natural partition of tasks because individual source files are compiled separately.

This natural partition of tasks can similarly be exploited to perform parallel compilation. A large program can be divided into a number of small source code files. These files can be sent to remote processors for parallel compilation, and then returned to a central machine to be linked into a single executable file. However, the linking process cannot be parallelized, so the parallel speedup is limited. A parallel compilation system could use two remote processors to decrease compilation time by about two thirds. This would decrease the total compiling and linking time by about fifty percent (Assuming a Microsoft C complier and linker).

6.7.2 Distributed CAD Tool

We are planning a distributed CAD tool that will allow designers sitting at multiple workstations to collaborate on a single large design project. The *Mind Meld* will allow multiple workstations to efficiently share and manage a large distributed database.

The majority of this data base can be kept on a massive memory machine which will be connected to the workstations through the *Mind Meld* interconnection. This project is best suited for the next generation *Mind Meld* machine which is described in the next chapter.

6.7.3 Memory Resident File System

This code would allow an IBM PC/AT system to use the memory of another IBM PC/AT system as a large memory resident file system. Code to implement a memory resident file system already exists (the VDISK utility in MS-DOS). One processor can be configured with a large main memory resident file system. This processor would have very little main memory left over for processing tasks so it would mainly function as a file server. The *Mind Meld* system can be used to allow another processor to access this main memory resident file

system. File transfer bandwidths to this main memory resident system will be many times faster than bandwidths to conventional disk-based systems.

Measurements made on a single IBM PC/AT system indicate that using a memory resident file system can improve file access speeds by a factor of four over slower disk based file systems. The *Mind Meld* system will be able to transfer files at these high bandwidths between machines.

6.8 Conclusions

Application programs written for the *Mind Meld* system have demonstrated that additional bandwidth and reduced latencies can improve distributed system performance: Multiple display applications, and remote file transfer programs have utilized increased bandwidth to rapidly transfer data between machines; the matrix multiply program has used decreased latencies to improve parallel processing performance.

The shared memory interface provided by *Mind Meld* has simplified the task of programming these applications. This simplification made it possible for a few programmers to easily write a large set of *Mind Meld* applications over the last half year.

One should note that the *Mind Meld* System is relatively inexpensive. *Mind Meld* board pairs cost about $200 each to produce. These low costs indicate that the Mind Meld system can provide improved distributed system performance in a cost effective manner.

Chapter 7

The Processor Identity Problem

This chapter poses the problem of establishing unique identities for a group of N identical processors which communicate through a common shared memory to which asynchronous read and write operations can be performed. We introduce a set of protocols which successfully assign unique integers from the set $\{0,1,2,\ldots,N\text{-}1\}$ to each processor.

By applying these simple protocols to the *Mind Meld* implementation, system modularity has been enhanced by eliminating the need for hardwired addresses or customized software for individual processor nodes.

7.1. Introduction

Mutual exclusion is one of the basic problems in computer science. In this problem a collection of asynchronous processes alternately execute a critical section of code. A solution to the problem must ensure that two processes never execute their critical sections concurrently. The problem has admitted to many solutions and refinements over the years [Dijk65] [Knut66] [Eise72] [Lamp86a] [Lamp86b]. However, all of these solutions have assumed that processes are initially assigned unique identities.

This chapter investigates the more fundamental problem of assigning these unique identities to processors which communicate through shared memory locations. A solution to this problem consists of a single protocol, run by N processors that communicate through shared memory, which produces a unique assignment of processors to the integers $\{0,1,2,\ldots,N\text{-}1\}$.

If two identical processors begin synchronously executing (synchronous execution is a special case of asynchronous execution) the same program at precisely the same time, it would be impossible to arrive at unique identities for each processor through shared memory communication. These two processors would proceed in lock step performing identical reads

and writes to the same memory locations. It would be impossible for one processor to detect the existence of the other processor let alone worry about distinguishing itself from the other processor.

To solve this problem an element of randomness must be introduced. We assume that each processor possesses a random number generator (this assumption is discussed in Section **7.5**). Using random number generators, we have developed a set of protocols which efficiently solve the *Processor Identity Problem*.

Section **7.2** presents a model for, and precisely states the *Processor Identity Problem*. Section **7.3** presents the *Random Key Protocol* which solves the *Processor Identity Problem* for the case of two processors. In section **7.4** this protocol is generalized to N processors. Section **7.5** discusses methods of producing different streams of random numbers from identical processors. Section **7.6** discusses applications of the *Processor Identity Problem* to shared memory multiprocessor systems such as the *Mind Meld* prototype. Section **7.7** presents conclusions.

7.2 Model

Our model consists of a collection of N processors which all access a common shared memory SM which consists of k words of size b bits. $SM[i]$ refers to the ith word of shared memory SM. Each processor can perform asynchronous read and write operations to fixed sized words of shared memory without conflict from the other processors. When more than one processor simultaneously writes to a single memory location two outcomes are possible. If all processors write the same bit pattern, the memory location will contain that bit pattern. If all processors simultaneously write different bit patterns to the same memory location, the contents of the memory location is indeterminate. The only way processors can communicate is through the shared memory SM (see Figure **7.2.1**).

Each processor possesses its own local memory which is not accessed by other processors. $LM[k]$ denotes the kth word of local memory LM. The goal of the identity problem solution is to produce a single protocol which is executed by every processor to produce a one-to-one

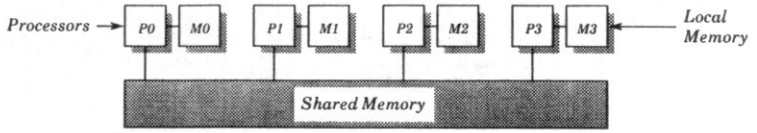

Figure 7.2.1: Shared Memory System Model

assignment of the N processors to the integers {0,1, 2, ... , N-1}. In solving this problem we will develop solutions for both the synchronous and asynchronous cases.

If all processors are active, this protocol must terminate with probability one in a finite amount of time. Even assuming that each machine possesses a true random number generator, it is impossible to guarantee that a solution can be arrived at in any fixed amount of time. We state this fact more precisely in the following theorem.

Theorem 7.2.1: *Assume that all processors start in identical states. For any fixed time t, no protocol exists which always solves the Processor Identity Problem within time t.*

Proof: Let us examine the case of two processors. Assume such a protocol exists. After a fixed length of time t, one processor is assigned to the number 1 and the other processor is assigned to the number 2. This means the two processors must have performed different assignment operations within the time t. Since both processors started in identical states, it is possible for both processors to remain exactly synchronized during the time t performing identical sequences of operations. Even queries to different local random number generators may produce the same results for any fixed length of time t. (Of course this probability rapidly drops to zero as the number of queries to the local number generator increases.) One can see that this proof also extends to the case of more than two processors. Therefore, no protocol exists which always solves the *Processor Identity Problem* in any fixed length of time t. □

7.3 Random Key Protocol: Case $N = 2$

The *Random Key Protocol* correctly solves the *Processor Identity Problem* with probability greater than 1-ε, where ε is an arbitrarily small real number less than 1. We first describe the *Random Key Protocol* for the case where the number of processors $N = 2$.

In the two processor *Random Key Protocol*, shared memory is divided into "bins". Each bin contains space for a large random number. (This random number may span one or more memory words.) Included in each bin is a "valid" bit which indicates if the contents of the bin is a valid random number (see Figure 7.3.1). The random key protocol for two processors uses two bins labeled bin 0 and bin 1.

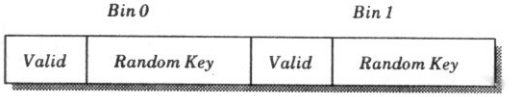

Figure 7.3.1: Bins In Shared Memory for 2 Processor Random Key Protocol

Each processor initially generates a large random number. These random numbers are assumed to be unique. If the random numbers are one hundred bits long, the probability that the two number will agree is 2^{-100}. Each processor now possesses a unique random number. One might argue that such a number constitutes a unique identity, but for purposes of parallel processing it is more useful if these unique tags come from a predetermined set of tags {0,1,2, ... , N-1}. Multiprocessing tasks can be specified in terms of such a set.

Each processor executes the protocol which is outlined below:

(1) Choose a large random number R.

(2) Initialize the valid bits of both bins to 0.

(3) Select a random bin i from the set.

(4) Set the valid bit of bin i, then write R to bin i.

(5) Read the contents of both bins.

(6) If bin i remains valid and contains R, and bin (i + 1) mod 2 is invalid go to step (5)

(7) If bin i is invalid or does not contain R, set valid bit of i to 0,

and start protocol over from step (3).

(8) If valid bit set in bin (i + 1) mod 2, read bin *i* again

If bin *i* remains unchanged, return *i*, else go to step (5)

(9) Go back to (5)

The processors each select a random bin form the set of two bins. If they choose different bins, the algorithm terminates, and they use their bin numbers as processor IDs. If they choose the same bins, they invalidate what they have written, and try to select different bins again.

The processors detect collisions when they both attempt to write their unique random keys to the same bin. One processor will eventually detect that its key has been overwritten. It will then invalidate the key which has been overwritten, and then pick another random bin. The remaining processor, will detect that its bin has been invalidated, and it will also choose another bin.

Processors only change bins if their valid bits and random numbers are overwritten by the other processor. If two processors ever simultaneously occupy different bins, they will never again change bins, since no processor will ever overwrite the other processors random number.

If a processor detects a valid bit set in the other bin, it first looks to see if its own valid bit and random key remains intact. If so, it knows the other processor must have written the valid bit to the other bin. It also knows the other processor will remain in possession of the other bin. It then terminates the protocol using the bin number as its unique processor ID.

This protocol is described more formally in the following C language routine:

```
/*solves Processors Identity Problem for the case of 2 processors*/
Solve()
{
    validbit[0] = 0;                          /*set valid bits to 0*/
    validbit[1] = 0;
    r = random();                             /*get random number*/
    while(1)
    {
        i = random() % 2;                     /*select random bin*/
```



```
        validbit[i] = 1;                      /*set valid bit of selected bin*/
        bin[i] = r;                           /*write random number to bin*/
        while((validbit[i] == 1) && (bin[i] == r)) /*repeat while your own bin is OK*/
        {
            if (validbit[(i+1)% 2] == 1)      /*read other valid bit*/
            {
                if((validbit(i) == 1) && (bin[i] == r))/*if your own bin remains in tact*/
                    return(i);                /*return unique processors ID*/
            }
        }
        valid[i] = 0;                         /*reset valid bit*/
    }
}
```

To prove the protocol solves the *Processor Identity Problem* we first establish that at least one processor will always finish the protocol in Lemma **7.3.1**.

Lemma 7.3.1: *With probability 1, at least one processor must eventually exit the two processor Random Key Protocol.*

Suppose neither processor leaves the protocol. If at least one processor is continually choosing a new random bin, the protocol will terminate as time goes to infinity because the processors will eventually occupy different bins in which case at least one processor will leave. If neither processor is changing bins then both processors must always see their own valid random number in their own bin and no valid random number in the other bin. If this is the case, both processors must occupy the same bin, therefore both processors must possess the same random number. Our assumption that no processor leaves the *Random Key Protocol* produces a contradiction. Therefore, at least one processor must eventually leave the two processor *Random Key Protocol*.□

We establish another useful lemma.

Lemma 7.3.2: *If both processors (A and B) initially choose distinct random numbers in the two processor Random Key Protocol, and processor A writes the valid bit of a bin, processor A will not touch the other bin unless its own bin is subsequently disturbed.*

Assume that processor A sets the valid bit of bin 0, and that bin 0 is never subsequently written to by processor B. Also assume that processor A subsequently writes to bin 1. Since processor A only writes to its own bin, processor A must have switched bins. Since processor A only switches bins if its own bin is disturbed by processor B, processor B must have disturbed bin 0 after the valid bit was set by A. This contradicts the assumption. □

Lemma 7.3.3: *If both processors initially choose distinct random numbers in the two processor Random Key Protocol, and one processor leaves the protocol, both processors must eventually leave the protocol.*

Assume that processor A has left the protocol and processor B remains executing the protocol. Processor A left the protocol after first making certain that the other bin contained a valid bit. Processor B must have set this valid bit because processor A would have reset this bit before changing bins. Processor A reads its own valid random number in its own bin before exiting the protocol. Both bins must now contain valid random numbers because processor A has not disturbed processor B's bin before exiting. And by Lemma **7.3.2** processor B will never disturb A's bin. However, if this were true than B will eventually leave the protocol. This contradicts our original assumption, therefore if one processor leaves the protocol, then both processors must leave the protocol. □

We next prove that the *Random Key Protocol* solves the *Processor Identity Problem* for the case of two processors.

Theorem 7.3.1: *If both processors initially choose distinct random numbers in the two processor Random Key Protocol, then the protocol will successfully solve the Processor Identity Problem.*

Proof: Suppose the *Random Key Protocol* does not solve the *Processor Identity Problem*. Lemmas **7.3.1** and **7.3.3** taken together say that both processors must eventually exit the protocol. If the protocol failed, they both must have exited with the same processor identification number, which means they both must have exited while occupying the same bin. Label one processor A and the other processor B. Processor A must have seen a valid random number in the other bin before leaving. This can only have been written by B since A invalidates bins before leaving them. Since the other bin is not disturbed in the exiting procedure, by lemma **7.3.2** processor B cannot have changed bins. If processor B has not changed bins, it exited the protocol with the other bin's number as a processor ID. This means both processors must have exited the protocol with unique identification tags. This contradicts our original assumption that the *Random Key Protocol* did not complete successfully. The *Random Key Protocol* therefore solves the *Processor Identification Problem* for the case of two processors. □

If both processors are actively performing the algorithm, then they will choose the same bin with probability 1/2 for each trial. Therefore, the program will halt with probability 1/2 after each iteration. The process can conceivably proceed for any fixed number of iterations but the expected number of trials can be calculated by considering each iteration to be a Bernoulli trial [Tuck80].

$$\textit{Expected number of trials} = \sum_{j=1}^{\infty} jP(j) = \sum_{j=1}^{\infty} j\left(\frac{1}{2}\right)^{j} = 2$$

The expected number of trails is shown to be two. We now extend this protocol to the general case where $N \geq 2$.

7.4 Random Key Protocol: Case $N \geq 2$

The *Random Key Protocol* for more than two processors is a generalization of the *Random Key Protocol* for two processors. The general protocol uses M bins (where $M \geq N$). As in the two processor case, each processor initially chooses a large random number (tag) which

is assumed to be unique. It then resets the valid bits in each of the M bins to 0. The processor chooses one of the M possible bins at random, and sets the valid bit in the bin before writing its tag to that bin. The processor remains on this bin unless the bin is disturbed by another processor.

The termination condition is harder to formulate in the general case. Even if a processor reads N unique valid tags in the M bins, it is not guaranteed that all processors have settled into different bins. A valid tag may be overwritten after it is read, but before all of the other bins have been read. This means reading a processor's valid tag does not guarantee that it will have settled into a unique bin after all M bins are read.

To solve this problem, each processor keeps a count of the number of times it has changed bins. It writes this number along with its tag into each bin that it occupies (see Figure **7.4.1**). If a processor reads all M bins twice, and if each occupied bin contains the same

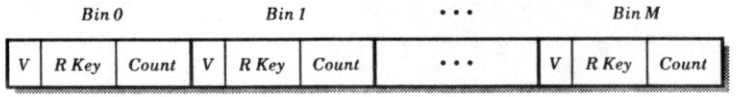

Figure **7.4.1**: Shared Memory Organization for N Processor *Random Key Protocol*

valid random number and count value, then it can be assured that for at least some point in time all M processors have settled into unique bins. Once they have settled into unique bins, no processor will move, because no processor will subsequently disturb another processor.

The general protocol is stated more formally below:

```
/*solves Processors Identity Problem for N processors*/
solve()
{
    set all valid bits to 0;
    r = random();                    /*get random number*/
    count = 0;                       /*initialize count*/
    while(1)
    {
        i = random() % N;  /*select random bin*/
        bin[i] = (1,r,count);        /*write (valid bit, random number, count) to bin i*/
```



```
        while(bin[i] = (1,r,count))
        {
            read all bins;
            if(N valid random numbers and bin[i] = (1,r,count) ){
                read all bins;
                if(no change between reads)
                    return(i);       /*return unique identity value*/
            }
        }
        bin[i] = (0,*,*);            /*reset only the valid bit of bin i*/
        count = count + 1;
    }
}
```

The count value makes it possible for the protocol to successfully determine the termination condition. However, it imposes a limit on the number of times a processor may change bins. If the count value is stored in L bits then the count value will cycle if it is incremented more than 2^L -1 times. If L is made sufficiently large, then all processors will successfully complete the protocol with probability greater than 1-α before changing bins 2^L times. (Where α is an arbitrarily small real number less than 1.)

If we assume that processors initially choose unique random keys, and that the *Random Key Protocol* will terminate before any processor's count value exceeds the maximum limit (2^L), then we can prove that the *Random Key Protocol* solves the Processor Identity Protocol. We first establish preliminary Lemmas.

Lemma 7.4.1: *A processor will only terminate the Random Key Protocol if each processor has a valid bit written to a bin, and no bin is occupied by more than one processor.*

Before exiting the protocol, a processor reads all of the bins in one pass, then reads the bins in a second pass. The processor only exits if it sees that no bins have changed between reads, and that N valid bits are set.

The count value ensures that the bin has not been altered between read operations. If the count value were not included, it would be possible for a valid key to be overwritten by one

processor and then rewritten by the original processor between read operations. This sequence of actions will change the count value, and this will be detected on the second read.

If the values remain the same between reads, then at some point in time between the two read passes N bins are guaranteed to contain N set valid bits (see Figure **7.4.2**). Since processors erase valid bits before changing bins, there can be at most one valid bit for each processor. The existence of N valid bits means the N processors must have each written one of these valid bits, and that no bin is occupied by more than one processor. □

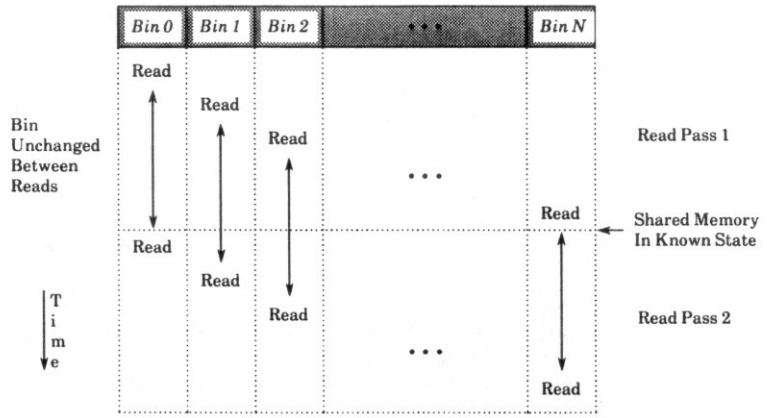

Figure 7.4.2: Termination Condition (Two Read Passes)

Lemma 7.4.2: *If each of the N processors has a valid bit written to a different bin in the Random Key Protocol, then no processor will ever subsequently change bins.*

Assume that each processor has a valid bit written to a unique bin, and that one or more processors subsequently move. One of these processors must be the first to move (processor A). (If more than one processor moves simultaneously, pick one of these at random and call it processor A.). Processor A must have been prompted to move because its bin was written to by another processor (call this processor B). This writing must have occurred

before all processors had valid bits written to different bins, because A was the first to move afterwards. However, processor B would have first erased A's valid bit before moving to its own bin. This means A must have subsequently rewritten its valid bit, and that processor B could not have subsequently interfered with A's bin. Our assumption that processors move produces a contradiction. Therefore, if each of the N processors has a valid bit written to a different bin in the *Random Key Protocol*, then no processor will ever subsequently change bins. □

We next prove that the all processors will eventually have valid bits written to unique bins.

Lemma 7.4.3: *If all N processors are actively performing the Random Key Protocol, then each processor will eventually write a valid bit to a unique bin.*

If all processors are active and fewer than N valid bits are written to the shared memory, then some processors do not have a valid tag in any bin. These processors will subsequently either write valid tags to empty bins (in which case N valid bits will be set) or they will write to occupied bins, causing collisions and more displaced processors. Even if we assume that each collision causes all processors to change bins, the protocol has a probability $P' = (M!/(M\text{-}N)!)/M^N$ of assigning each processor to a unique bin after each collision. The collisions cannot go on indefinitely because the unique assignment of bins to processors will take place with probability one as the number of collision events goes to infinity. □

Theorem 7.4.1.: *If all processors remain active, the Random Key Protocol will produce a one-to-one assignment of the N processors to the integers {0,1,2, ... ,N-1}.*

Proof: Lemmas **7.4.2** and **7.4.3** taken together say that each processor will eventually settle into its own bin and remain there. The processors will then eventually read N set valid bits on two separate scans of the bins without any of the bins changing. All processors will then terminate while occupying different bins. This implies each processor will choose a

unique number from {0,1,2, ... ,N-1}. The *Random Key Protocol* will therefore produce a one-to-one assignment of the N processors to the integers {0,1,2, ... ,N-1}. □

An issue to be concerned about is the probability of two processors choosing the same random key at the beginning of the protocol. If the random numbers are L bits long, the probability that N processors choose unique random numbers is $(2^L)!/(2^L\text{-}N)!/(2^L)^N$. As in the two processor case, this number can be made arbitrarily close to one by increasing the size of L.

In order to estimate the running time we define the term *trial*. Each processor moves a certain number of times during the protocol. The maximum of this number over all processors will be the number of trials for the protocol. In the worst case all processors will be forced to move with each new trial. In this case the protocol will succeed with probability $N!/N^N$ in each iteration. This means it terminates with probability one as time goes to infinity. The success probability can be increased by using M bins in shared memory where $M \geq N$. Each processor picks one of the M bins at random and writes its random number tag to that bin. The rest of the protocol remains unchanged.

To produce an identification tag, each processor counts the number of occupied bins with lower shared memory addresses, and uses this as an identification tag. This succeeds with probability $P' = (M!/(M\text{-}N)!)/M^N$, The expected number of trials is then.

$$\textit{Expected number of trials} = \sum_{j=1}^{\infty} jP(j) = \sum_{j=1}^{\infty} jP'\left(1-P'\right)^j = \frac{1}{1-(1-P')} = \frac{1}{P'}$$

For example, if $M = 20$ and $N = 8$ the expected number of trials is five.

7.5. Generation of Random Numbers

It is no trivial task to generate a stream of random numbers from a deterministic computing system. Random number generators are typically deterministic programs which take as input a random seed. Finding unique seeds for all processors is not a trivial task. If each element of the computing system were truly deterministic, it would clearly be

impossible. There are, however, channels through which random processes from the physical world can be tapped from within a computing system.

On system power up, the contents of random access memory assume a somewhat random state. Processors typically initialize the entire memory to a known state before proceeding. Before this initialization takes place, a random number can be generated as the result of a function performed on the contents of each local memory. (Remember that each processor possesses its own local memory.) One must make certain that correlations due to spatial locality do not affect the final random number. For instance, one might exclusive-or all bits whose addresses are multiples of the ith prime number to produce the ith bit of a random number.

A disk drive is a physical device that is affected by random processes. Variations in the power supply, vibrations, variable air friction, and rotational velocity all affect seek and latency times. The time for any given disk seek will be randomly distributed around an expected value. One can use deviations from the expected time of a disk operation as a source for random numbers.

Even if a processor has no disk drives or uninitialized RAM locations, processors can reference their system clocks for unique random number generator seeds. If all processors are started at different times, and some amount of clock skew is present, then system clocks will suffice for seed generation.

Clock skew may also be used to generate unique random number seeds. The following program uses clock skew between processors to arrive at unique random number generator seeds. It requires one "bit" in shared memory, and one "count" value in local memory.

```
/*function returns random number generator seed*/

sharedmem bit = 0;          /*a bit in shared memory*/

getseed()
{
    int count;
    count = 0;
    while (bit != 1){
        if (count = N) then bit = 1;
```

```
        count = count + 1;
    }
    return(count);
}
```

The first processor to terminate will take on the seed value N. Slower processors will take on lower values.

Even if all processors are tied to the same clock, and memory is assumed to be initialized, one can provide each processor with a large unique random number seed which is set in hardware. This is not the same as assigning each processor a name from a small set of names that can be referred to by a program. Two processors will not have the same random number seed in practice. Processors can then be replaced without reconfiguration work, or modifications to global tables of processor to name assignments.

7.6 Applications of the Processor Identity Problem

The *Processor Identity Problem* was encountered while writing communication primitives for the *Mind Meld* prototype described in Chapters **5** and **6**. The *Mind Meld* system allows processors to communicate by performing asynchronous read and write operations to a common shared memory. To implement these primitives it is necessary to assign each processor to a unique task and set of memory locations. Our protocols allow a set of identical processors running the same program to arrive at such a unique assignment.

Of course this problem may be trivially solved by loading a different program into each processor. But we have found that this entails a great amount of inconvenience. As updated versions of the software are created, the software must be custom tailored for each processor. The programs must subsequently be individually loaded into each processor. It is more efficient to maintain a single version of a program which can be broadcast in parallel to all processors.

Another option is to provide each processor with a special file, jumper configuration, or ROM location which contains a unique identification tag. (Ethernet socket addresses are

maintained this way [Metc76].) This solution requires the overhead of maintaining global tables of unique addresses, as well as extra processor configuration work.

A third option is to provide the system with an atomic read-modify-write or test-and-set operation. This capability will allow processors to serially grab control of a single shared variable and thereby establish an ordering of processors. This type of architectural feature must be built into the hardware of the machine, and it can require extra design time. (Especially if arbitration circuits must be constructed for multiple data paths.)

Instead of accepting one of these alternatives, we produced a simple protocol that could be used by a group of identical processors running identical programs to agree on an assignment of unique identities for each processor. This protocol has eliminated the need for special processor configuration or development of hardware test-and-set instructions for the *Mind Meld* prototype.

7.7 Conclusions

This chapter has presented and analyzed a set of protocols which solve the *Processor Identity Problem*. These protocols produce an assignment of N processors (which communicate through a common shared memory) to the integers $\{0,1,2,\ldots,N\text{-}1\}$. This assignment of identities is a necessary precondition for solutions to the mutual exclusion problem. These protocols can also be used to enhance system modularity by reducing configuration work associated with installing or replacing individual processor nodes in a parallel processing system such as *Mind Meld*. For a more detailed examination of this problem refer to [Lipt87].

Chapter 8

Design for the Next Generation System

The *Mind Meld* system has provided fertile ground for the development of new types of distributed computing applications, however, it has not fully realized the potential for shared memory in distributed computing architectures. The present *Mind Meld* memory can only be shared between pairs of processors. Unless two processors are directly connected by a pair of *Mind Meld* boards, inter-processor communication is only possible by passing data through intermediate processors.

This chapter sketches the design for a high performance distributed system which allows a collection of heterogeneous processors to efficiently share a single shared memory. This system can provide a backbone for an efficient distributed shared memory computing system.

8.1 The Design

The next generation system is designed around a shared memory box with up to sixteen independent ports. This is diagrammed in Figure **8.1.1**. Processors connect to this shared

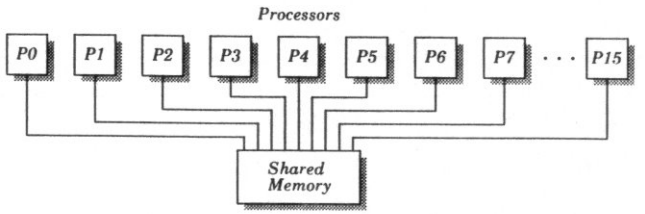

Figure **8.1.1**: Multiported Shared Memory

memory box through high speed fiber optic or coaxial cables. Each processor system is provided with a memory-mapped interface board which allows it to access the shared memory

box through the cable. This memory-mapped board appears to be a standard memory extension board. In reality, the memory mapped board transmits read and write requests over the cable to the shared memory box.

Each of the ports provides unrestricted read and write access to all portions of the shared memory. No test and set hardware is provided to ensure exclusive access to data items or critical sections of code. This task is left to the programmer who can construct the appropriate exclusive access mechanisms in software if necessary [Dijk65]. Eliminating this test and set feature eliminates hardware complexity at the cost of additional programming time and software complexity.

Many shared memory systems employ multiple caches to reduce contention for access to the shared memory [Fran84]. A multiply-ported shared memory eliminates this contention problem by providing each processor with its own dedicated channel to shared memory. Providing these multiple ports increases the cost of shared memory, but also eliminates the the need to build multiple caches. A multiply-ported shared memory may not be easily extended to a large number of ports. This scalability problem also confronts multiple cache shared memory architectures.

Having a single shared memory box limits the distance of the cables which connect a processor to shared memory. As these cables increase in length, the latency for data transfers to and from the shared memory become longer. To maintain low latencies, cables will have to be restricted to a maximum length.

This shared memory box can be connected to a heterogeneous collection of processors by constructing interface boards for several industry standard busses (VME, PC/AT, Multibus). Specifications for this shared memory architecture are listed in Figure **8.1.2.**

16 M bytes of Shared Memory
16 Ports
4 M bytes/sec Bandwidth Per Port
1 Microsecond Access Time
Fiber Optic/Coaxial Cables
Interfaces to Industry Standard Buses

Figure **8.1.2**: Shared Memory System Specifications

The shared memory box contains 16 M bytes of memory. Ideally, one would have as much memory as possible in this box, but if high speed RAMs are used, large amounts of shared memory can become quite expensive. Many processors are also unable to address more than 16 M bytes of data because of CPU and operating system limitations (For example the Multibus I systems such as the SUN II can only address up to 16 M bytes of data.). A 16 M byte shared memory should provide enough storage for a wide range of applications without a large cost. If future processor architectures can address more memory, and memory prices decrease, shared memories can also increase in size.

Architectural options for multiply-ported memory boxes have been described in Chapter **3**. We have chosen to use the simplest possible rotary switch design in which a single very high speed memory is switched between sixteen slower speed ports. If 50 nanosecond RAMs are used, it should be possible to provide microsecond access times to each of the 16 ports. (Actually, twenty RAM cycles can occur in a microsecond, but some time will be lost in switching.). If four bytes are transferred during each memory cycle, each port can provide 4 M bytes/second bandwidth to the shared memory.

The rotary switch can be implemented with a high speed multiplexer design as is diagrammed in Figure **8.1.3** for 3 ports. In this design, each port gets an equal slice of time to access the fast RAM in round robin fashion. Buffers are used to latch the slower address and data signal from the ports for the faster RAM chip to service.

During a read cycle, an address is first latched from the port into the address buffer. When the port finally receives its time slice, the address is gated onto the RAM chip. The RAM chip returns a data word which is latched into the data buffer. The data buffer then broadcasts this data item to the port.

Switching the shared memory between different ports must occur rapidly to maintain high bandwidth and low latency. A high speed technology such as ECL will have to be used to assure rapid switching and signal propagation through the multiplexers and buffers.

To extend this design to more ports and larger distances, one would probably use some type of multistage interconnection network. These are described further in section **8.6.3**. Our goal, however, is not to create an idealized paper design, but to produce a practical design that can be built successfully. This rotary switch design provides a high level of system

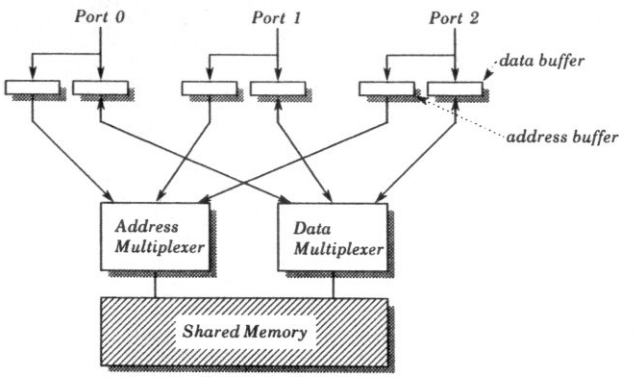

Figure **8.1.3**: High Speed Rotary Switch

performance with a minimal amount of system complexity. This is something we can conceivably build in the near future.

8.2 Operating System Interface

Processors will generally interface to the shared memory box by mapping the shared memory into the virtual address space of a program. (If the processors do not have virtual addressing, programs can be mapped into the real addresses as was done in the *Mind Meld* prototype.). The shared memory can be incorporated into a standard programming language by the introduction of a new shared memory data type, or by use of a predefined shared memory array as in the *Mind Meld* implementation.

Adding a shared memory data type will require minor modifications to existing compilers. These compilers can generate a mapping of virtual addresses to shared memory addresses. This mapping can be stored in the program's executable file and used by the operating system to allocate shared memory at program run time.

8.3 The Interconnection Cable

A number of options have been considered for the connection between shared memory and the memory-mapped processor boards. The present *Mind Meld* implementation uses a cumbersome 37 pin interconnection that limits the length of *Mind Meld* cables because of reflection problems and crosstalk. Instead of this 37 wire parallel interconnection, a pair of coaxial or fiber optic links can be used to transmit data over longer distances. Pairs of coaxial or fiber optic links will also be easier to manipulate than the large 37 conductor linkages.

These cables will be unidirectional. One cable of the pair will be used to transmit data from the shared memory box to the processor. The other cable will be used to transmit addresses and data from the processor to the shared memory (See Figure **8.3.1**).

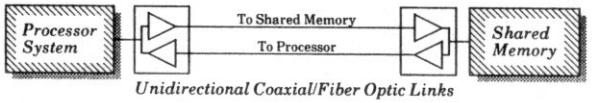

Figure **8.3.1**: Pairs of Unidirection Cables

Recent advances in fiber optics make this data transmission medium more attractive: Off the shelf transducers are now available that convert between digital computer hardware signals and analog fiber optic transmissions; new modular connectors allow fiber optic lines to be easily connected and disconnected. Fiber optic linkages also provide extremely high data transmission bandwidths, as well as electrical isolation between systems.

Coaxial cables are also a possibility. They do not provide the extremely high bandwidths or electrical isolation properties of fiber optics. However, they are easier to build and more cost effective.

8.3.1 Communication Formats

A uniform communication format must be decided upon to transmit data to and from the shared memory. Word sizes and byte orders vary between different machines. It is

probably most practical to choose one byte order and word size as the transmission standard (say 4 byte VME), and then adapt the rest of the interface boards to this standard.

8.4 The Interface Boards

We would like to produce interface boards for several industry standard buses. In doing so, it is desirable to minimize design time for each different board. This can be done by producing a standardized design for all interface boards.

These interface boards can be decomposed into two functional units: 1) Interface to the local bus. 2) Interface to shared memory. The boards should all have similar interfaces to shared memory, but different interfaces to each bus.

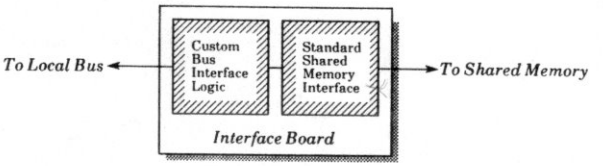

Figure **8.4.1**: Standardized Interface Design

By dividing the board design into two functional units, the same shared memory interface logic can be used for all systems. Only customized bus logic must be designed for each particular system. Even this customized logic can be standardized somewhat by using programmable logic arrays to perform most of the bus interface functions. In this way, designing an interface for a new bus can be reduced to programming a logic array on a pre-existing design. See Figure **8.4.1**.

8.5 Other Features

We have excluded some beneficial architectural features from the design in the interests of simplicity. These features include: 1) Extending the architecture to accommodate hundreds of processors. 2) Providing fault tolerance to the shared memory system. 3) A two

level hierarchy for the shared memory. 4) A virtual memory based protection mechanism for the shared memory.

8.5.1 More Processors

A shared memory system can be extended to possibly hundreds of processors by using a multistage interconnection network (this has been discussed in Chapter **3**). A number of tightly-coupled systems of this type have already been constructed [Bran87] [Crow85].

Extending this type of interconnection network to a distributed system will entail providing longer communication links between processors and the interconnection network. As one would expect, this added communication distance will reduce the bandwidth and latency available through such a system. Nevertheless, this type of interconnection can allow an unbounded number of processor to connect to a single shared memory (which is composed of multiple memory modules).

Extending any shared memory system beyond hundreds of processors is impractical for two reasons. 1) Processors architectures limit the amount of shared memory that can be addressed. This effectively limits the shared memory size, and hence the number of processors which can efficiently share the memory. 2) It may prove impossible to manage thousands of processors accessing the same shared memory area. Unexpected memory contention issues might arise.

8.5.2 Fault-Tolerance

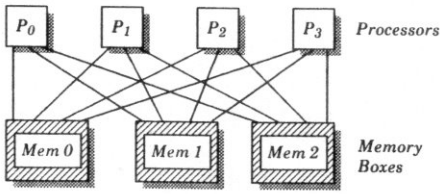

Figure **8.5.2**: Fault-Tolerant Shared Memory Architecture

Fault tolerance can be provided by using multiple shared memory boxes. A fault-tolerant system with three memory boxes is diagrammed in Figure **8.5.2**. The three boxes form a single shared memory. Each box contains a contiguous portion of the shared memory address space. If a single box fails, the other two will remain functional. The data contained in the failed box will be lost, but a communication link between processors through the shared memory system will still be maintained.

This fault-tolerant system can be constructed from three of the shared memory boxes described in previous sections of this chapter. No hardware modifications have to be made. However, each box will have to be configured to reside in a different contiguous portion of the shared memory, and the operating system will have to be modified to accommodate three times as much shared memory.

8.5.3 Two Level Hierarchy

The shared memory size is limited by the high cost and low densities of the high speed RAMs used to implement the multiported memory. Even producing the desired 16 M byte shared memory can be very expensive. To expand this memory size further, the shared memory can be split into a two level hierarchy, with a small high speed cache, and a slower large memory (This has been discussed previously in Chapter **3**. A diagram of this appears in Figure **8.5.3**). This would allow processors to share large data structures, and file systems

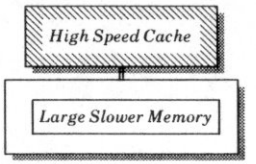

Figure **8.5.3**: Two Level Hierarchy for Shared Memory

that reside in shared memory. The design for this caching system can be adapted from existing uniprocessor designs.

8.5.4 Protection (Memory Controller)

Protection is another important issue. As the shared memory system is designed now, a single malfunctioning processor can overwrite any portion of the shared memory. To prevent this, a virtual memory interface can be provided for the shared memory (This type of system was described previously in Chapter **3**.) Processors would access virtual memory through the ports of the shared memory box. The mapping from virtual shared memory to the real shared memory would occur inside the box.

This system would maintain a memory mapping table for each processor. For security reasons this table would be maintained by a dedicated microprocessor which resides in the shared memory box. Only this processor would update mapping tables.

The controlling processor can restrict access to portions of shared memory by restricting memory mapping table entries for these blocks of shared memory. These blocks of shared memory would only be made available to processors with special access privileges.

8.6 Performance

Several factors will improve performance for the next generation implementation: 1) Fiber optic links will eliminate line noise and reduce propagation delay times. 2) Faster processors will transfer data more rapidly to the shared memory. 3) Multiple ports will eliminate the need for processors to communicate through intermediate processors.

This next generation implementation will provide ten times the bandwidth of the present *Mind Meld* system, and 80 times the bandwidth of the Ethernet. This new system is contrasted with existing loosely-coupled and tightly-coupled systems on the graph of Figure **8.6.1**. (A similar graph appeared in Figure **4.1.1**.)

Note that the new system provides a couple of orders of magnitude more bandwidth and less latency than conventional loosely-coupled systems. Tightly-coupled systems achieve higher bandwidths and lower latencies, but they cannot be used to connect independent processor systems together.

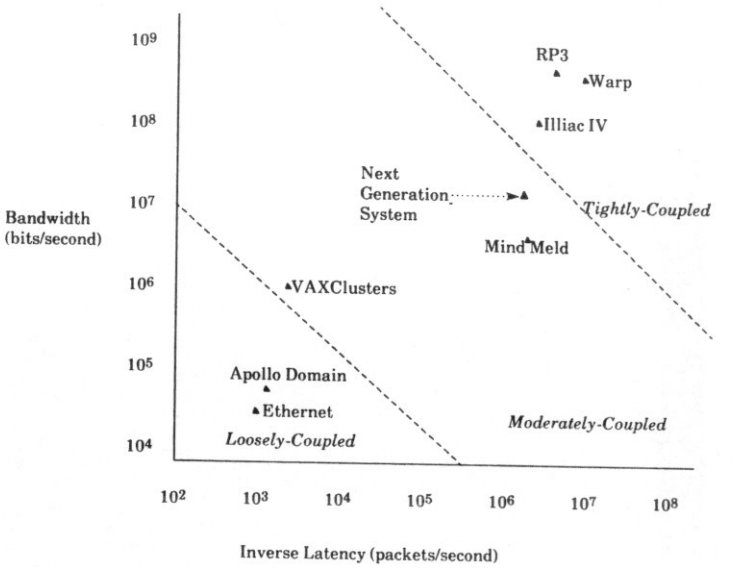

Figure 8.6.1 Performance

8.7 Conclusions

This chapter has outlined the design for a future shared memory distributed system architecture. This system connects up to 16 processor systems together through a common shared memory box. This system does not match the extensibility of loosely-coupled systems, or the high bandwidth and low latency of tightly-coupled systems. It does, however, fill an important niche between tightly-coupled and loosely-coupled systems. Through this system, a collection of independent processors can be integrated into a single multiple processor system which nearly achieves tightly-coupled performance.

Chapter 9

Conclusions

This thesis has presented a moderately-coupled shared memory architecture for distributed computing. The shared memory architecture simplifies the programming of distributed applications. And the high bandwidth and low latencies of this system brings high performance communication to computing systems composed of independent processors and workstations.

9.1 Research Results

This shared memory architecture presents the programmer with a uniform view of local memory, local memory mapped devices, shared memory, and distributed devices. Standard programming languages can therefore be used to control and transfer data between these entities. In this way, both system level and application level programming is simplified. The architecture also provides high bandwidth, low latency communication between processors. This improves distributed computing operations such as remote file transfer (which is presently constrained by communication network performance [Park87d]).

Additional performance also makes it possible to share new types of distributed resources such as large semiconductor memories, and high bandwidth communication lines which span long distances. New applications such as real time transfer of high resolution animation between processors also become possible. Lower latencies improve performance on parallel programs with fine granularity.

A shared memory distributed architecture can be implemented by using a multiply-ported shared memory box. These multiple ports eliminate access conflicts by providing each processor with a dedicated channel to the shared memory. This architecture can effectively provide high bandwidth, low latency communication between dozens of processors spread throughout a small locality (building or machine room).

The prototype *Mind Meld* system allows separate microprocessor systems to communicate through a shared memory interface which provides 8 times the bandwidth, and 1000 times smaller latencies than the Ethernet. The shared memory architecture has provided *Mind Meld* programmers with a clean, powerful interface providing the flexibility to program a diverse collection of applications.

A number of application programs have been written for the *Mind Meld* system. These include system level programs which allow processors to transfer files between local and remote disk drives, and interrupt driven code which allows a single keyboard to operate multiple machines. Other application programs combine several system monitors to display large amounts of data in several imaginative ways. These applications have demonstrated that distributed applications can effectively utilize the added bandwidth and latency of the *Mind Meld* system. They have also demonstrated that a shared memory interface can be effectively used to program these distributed applications

The *Processor Identity Problem* was posed and solved in Chapter 7. The solution to this problem assigns unique identities to a collection of identical processors which communicate through shared memory. The solution to this problem is a necessary precondition for solutions to the mutual exclusion problem. The solution also simplifies system initialization for shared memory multiprocessor systems.

The design for a future shared memory system has been presented in Chapter 8. This design uses fiber optic links to provide an order of magnitude more bandwidth, and half the latency time of the present *Mind Meld* implementation. More importantly though, it allows sixteen separate processing systems to directly access the same shared memory eliminating the need for processors to communicate through intermediate systems (as in the *Mind Meld* prototype).

9.2 Future Research

A number of areas merit further investigation. These include: 1) Developement of a new type of operating system for a shared memory-based distributed computing system. 2) Expanding the present architecture to accommodate hundreds and thousands of machines. 3)

Providing a shared memory caching mechanism which allows high performance access to shared memory over longer distances.

A shared memory-based high performance distributed architecture will make a different type of distributed operating system possible. This type of operating system will be a hybrid between operating systems for centralized computing systems, and operating systems for distributed computing systems.

The new operating system will perform the same functions of a centralized operating system except that system functions will be executed by a group of distributed processors instead of a single central processor. It will control and allocate distributed system resources such as printers, file servers, and memory servers, as well as balance processor load across a distributed system. No distributed operating system of this type has been proposed, yet certain system functions such as load balancing are beginning to be implemented. Designing this new type of operating system for a shared memory architecture is bound to uncover interesting research issues.

Shared memory distributed architectures can be expanded to accommodate hundreds of processors. This can be done by investing in large interconnection switches (which have been discussed in Chapter **3**), or mapping a large shared virtual space onto a large number of local processor memories (as has been done by Kai Li [Li86]).

To maintain performance over larger distances, a caching mechanism may have to be incorporated into a shared memory architecture. Such mechanisms are already widely used in shared bus multiprocessor systems [Swea86]. These multiple-cache systems maintain coherence between different copies of the same data through various data ownership protocols. Standard mechanisms to maintain coherence do not work efficiently over large distances, so new methods of maintaining coherence will have to be developed.

References

[Alon86] R. Alonso, P. Goldman, P. Potrebic, "A Load Balancing Implementation for a Local Area Network of Workstations", *1986 IEEE Workstation Technology and Systems Conference*, March 18-20, 1986, Atlantic City, New Jersey.

[Barn68] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, R. A. Stokes, "The Illiac IV Computer", *IEEE Transactions on Computers*, Volume C-17, Number 8, August 1968, pp. 746-757.

[Bala86] R. V. Balakrishnan, "Backplane Transceiver Logic - A New Standard For Bus Transceivers", Proceedings of Buscon '86 West, January 1986, pp. 168-175.

[Bran87] William Brantley, Personal Communications, June 1987.

[Cour71] P. J. Courtois, F. Heymans, D. L. Parnas, "Concurrent Control With Readers and Writers", *CACM*, Volume 14, Number 10, October 1971, pp. 667-668.

[Crow85] W. Crowther, J Goodhue, E. Starr, R. Thomas, W. Milliken, T. Balckadar, "Performance Measurements on a 128-Node Butterfly Parallel Processor", *Proceedings of the 1985 International Conference on Parallel Processing*, 1985, pp. 531-540.

[Digi81] Digital Equipment Corporation, *VAX Hardware Reference Manual*, Digital Equipment Corporation Press.

[Dijk65] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *CACM*, Volume 8, Number 9, September 1965, pp.569.

[Eise72] M. A. Eisenberg, M. R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Control Problem", *CACM*, Volume 15, Number 11, November 1972, pp. 999.

[Feng81] T. F. Feng, "A Survey of Interconnection Networks", *Computer*, December 1981, pp. 12-27.

[Fran84] S. J. Frank, "Tightly Coupled Multiprocessor Speeds Memory-Access Times", *Electronics*, January 12, 1984, pp. 164-169.

[Garc84] H. Garcia-Molina, R. J. Lipton, J. Valdes, "A Massive Memory Machine", *IEEE Transactions on Computers*, Volume C-33, Number 5, May, 1984, pp. 391-399.

[Garc87] H. Garcia-Molina, A. Park, L. R. Rogers, "Performance Through Memory", *Proceedings of the 1987 ACM - SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1987.

[Gehr82] E. F. Gehringer, A. K. Jones, Z. Z. Segall, "The Cm* Testbed", Computer, October 1982, pp. 40-53.

[Hill85] Daniel M. Hillis, *The Connection Machine*, MIT Press, 1985.

[Klei75a] L. Kleinrock, *Queuing Systems, Volume 1: Theory*, John Wiley & Sons, New York, 1975.

[Klei75b] L. Kleinrock, *Queuing Systems, Volume 2: Applications*, John Wiley & Sons, New York, 1975.

[Knut66] D. Knuth, "Additional Comments on a Problem in Concurrent Program Control", *CACM*, Volume 9, Number 5, May 1966, pp.321-322.

[Kron86] N. P. Kronenberg, H. M. Levy, W. D, Strecker, "VAXclusters: A Closely-Coupled Distributed System", *ACM Transactions on Computer Systems*, Volume 4, Number 2, May, 1986, pp. 130-146.

[Kung87] H. T. Kung, The Warp Architecture, *Princeton University Distinguished Lecture*, May 1987.

[Lamp86a] L. Lamport, "The Mutual Exclusion Problem: Part I - A Theory of Interprocess Communication", *JACM*, Volume 33, Number 2, April, 1986, pp. 313-326.

[Lamp86b] L. Lamport, "The Mutual Exclusion Problem: Part II - Statement and Solutions", *JACM*, Volume 33, Number 2, April, 1986, pp. 327-348.

[Li86] Kai Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, PhD Thesis, Yale University, September, 1986.

[Lipt87] R. Lipton, A. Park, "The Processor Identity Problem", *Technical Report CS-TR-088-87*, Department of Computer Science, Princeton University, Princeton, NJ, 1987.

[Lopr86] Daniel Lopresti, *Discounts for Dynamic Programming on VLSI Processor Arrays*, PhD Thesis, Princeton University, December 1986.

[McCr84] E. McCreight, *Dragons Come Slowly But Relentlessly*, Presentation at XEROX Palo Alto Research Center, 1984.

[Metc76] R. M. Metcalfe, D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, Volume 19, Number 7, July, 1976, pp. 395-404.

[Pand87] E. S. Panduranga, *Reflections On Curved Surfaces*, PhD Thesis, Princeton University, August 1987.

[Park87a] A. Park, K. Balasubramanian, R. J. Lipton, "Array Access Strategies for Block Storage Memory Systems", *IEEE Transactions on Computing*, to appear, 1987. (Also published in: Proceedings of the 1986 Allerton Conference on Communications, Controls, and Computing).

[Park87b] A. Park, K. Balasubramanian, "Improved Sorting Algorithms for Parallel Computers", *Journal of Parallel and Distributed Computing*, to appear, (Also published in: Proceedings of the 1987 ACM Fifteenth Annual Computer Science Conference, 1987).

[Park87c] A. Park, K. Balasubramanian, "Providing Fault Tolerance in Parallel Secondary Storage Systems", *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, submitted.

[Park87d] A. Park, R. J. Lipton, "Models and Measurements of File System Performance", *Proceedings of the 18th Annual Pittsburgh Conference on Modeling and Simulation*, April 1987.

[Park87e] A. Park, R. J. Lipton, "IOStone: A Synthetic File System Benchmark Program", *Performance 87*, submitted.

[Pfis85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Klienfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, J Weiss, "The IBM Research Parallel

Processor Prototype (RP3) Introduction and Architecture", *Proceedings of the 12th Annual Symposium on Computer Architecture*, May 1985, pp. 764-769.

[Powe83] M. L. Powell, B. P. Miller, "Process Migration in DEMOS/MP", *Proceedings of the Ninth Symposium on Operating System Principles*, 1983, pp. 110-119.

[Seit85] C. L. Seitz, "The Cosmic Cube", Communications of the ACM, Volume 28, Number 1, January 1985, pp. 22-33.

[Smit86] Burton J. Smith, Private Communications, 1986.

[Ston82] H. S. Stone, Microcomputer Interfacing, *Addison-Wesley*, 1982.

[Sweaz86] Paul Sweazy, Tutorial Seminar on Fundamentals of Cache Memories, BUSCON 86 West Conference, January 1987.

[Tuck80] A. Tucker, *Applied Combinatorics*, John Wiley & Sons, New York, 1980.

[Yous87] Abdou Youssef, *Structure Control and Functionality of Multistage Interconnection Networks*, PhD Thesis, Princeton University, August 1987.