Shared Versus Distributed Memory Multiprocessors*

Harry F. Jordan

ABSTRACT

The question of whether multiprocessors should have shared or distributed memory has attracted a great deal of attention. Some researchers argue strongly for building distributed memory machines, while others argue just as strongly for programming shared memory multiprocessors. A great deal of research is underway on both types of parallel systems. This paper puts special emphasis on systems with a very large number of processors for computation intensive tasks and considers research and implementation trends. It appears that the two types of system will likely converge to a common form for large scale multiprocessors.

*This work was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-18605 while the author was in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665, and in part by the National Science Foundation under Grant NSF-G87-17773.

What Are They?

The generic term parallel processor covers a wide variety of architectures, including SIMD machines, data flow computers and systolic arrays. The issue of shared versus distributed memory arises specifically in connection with MIMD computers or multiprocessors. These are sometimes referred to simply as "parallel" computers to distinguish them from vector computers, but we prefer to be precise and call them multiprocessors to avoid confusion with the generic use of the former word. Some similar sounding but different terms are often used in a confusing way. *Multiprocessors* are computers capable of running multiple instruction streams simultaneously to cooperatively execute a single program. *Multiprogramming* is the sharing of a computer by many independent jobs. They interact only through their requests for the same resource. Multiprocessors can be used to multiprogram single stream (sequential) programs. A *process* is a dynamic instance of an instruction stream. It is a combination of code and process state, e.g. program counter and status word. Processes are also called *tasks, threads*, or *virtual processors*. The term *Multiprocessing* can be ambiguous. It is either:

a) Running a program (perhaps sequential) on a multiprocessor or
b) Running a program which consists of several cooperating processes.

The interest here is in the second meaning of multiprocessing. We want to gain high speed in scientific computation by breaking the computation into pieces which are independent enough to be performed in parallel using several processes running on separate hardware units but cooperative enough that they solve a single problem.

There are two basic types of MIMD or multiprocessor architectures, commonly called shared memory and distributed memory multiprocessors. Figure 1 shows block diagrams of these two types, which are distinguished by the way in which values computed by one processor reach another processor. Since architectures may have mixtures of shared and private memories, we use the term "fragmented" to indicate lack of any shared memory. Mixing memories private to specific processors with shared memory in a system may well yield a better architecture, but the issues can be discussed easily with respect to the two extremes: fully shared memory and fragmented memory.

A few characteristics are commonly used to distinguish shared and fragmented memory multiprocessors. Starting with shared memory machines, communication of data values

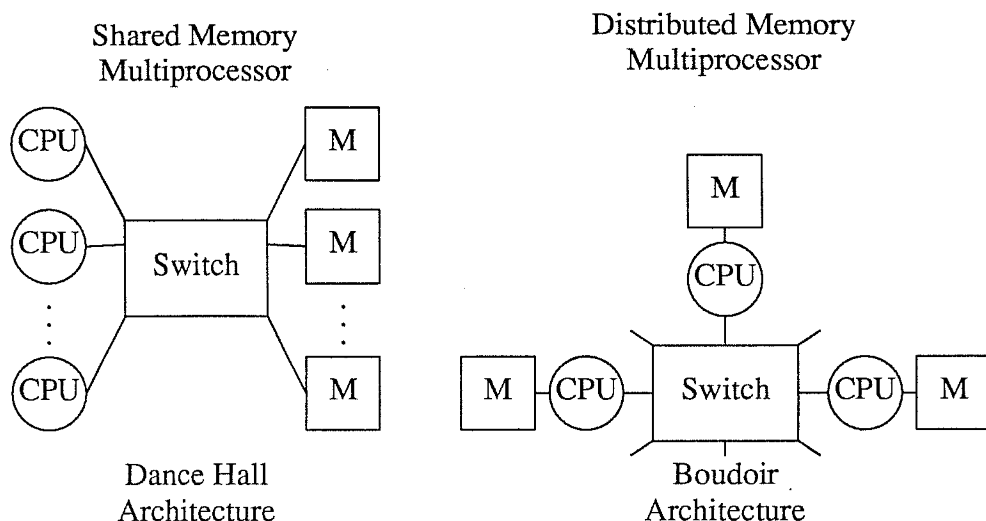


Figure 1: Shared and distributed memory multiprocessors.

between processors is by way of memory, supported by hardware in the memory interface. Interfacing many processors may lead to long and variable memory latency. Contributing to the latency is the fact that collisions are possible among references to memory. As in uniprocessor systems with memory module interleaving, randomization of requests may be used to reduce collisions. Distinguishing characteristics of fragmented memory rest on the fact that communication is done in software by data transmission instructions, so that the machine level instruction set has send/receive instructions as well as read/write. The long and variable latency of the interconnection network is not associated with the memory and may be masked by software which assembles and transmits long messages. Collisions of long messages are not easily managed by randomization, so careful management of communications is used instead. The key question of how data values produced by one processor reach another to be used by it as operands is illustrated in Fig. 2.

The organizations of Fig. 1 and the transmission mechanisms of Fig. 2 lead to a broad brush characterization of the differences in the appearance of the two types of architecture to a user. A shared memory multiprocessor supports communication of data entirely by hardware in the memory interface. It requires short and uniform latency for access to any memory cell. The collisions which are inevitable when multiple processors access memory can be reduced by randomizing the references, say by memory module interleaving. A fragmented memory switching network involves software in data communication by way of explicit send and receive instructions. Data items are packed into large messages to mask long and variable latency. Since messages are long, communications scheduling instead of randomization is used to reduce collisions. To move an intermediate datum from its producer to its consumer a fragmented memory machine ideally sends it to the consumer as soon as it is produced, while a shared memory machine stores it in memory to be picked up by the consumer when it is needed.

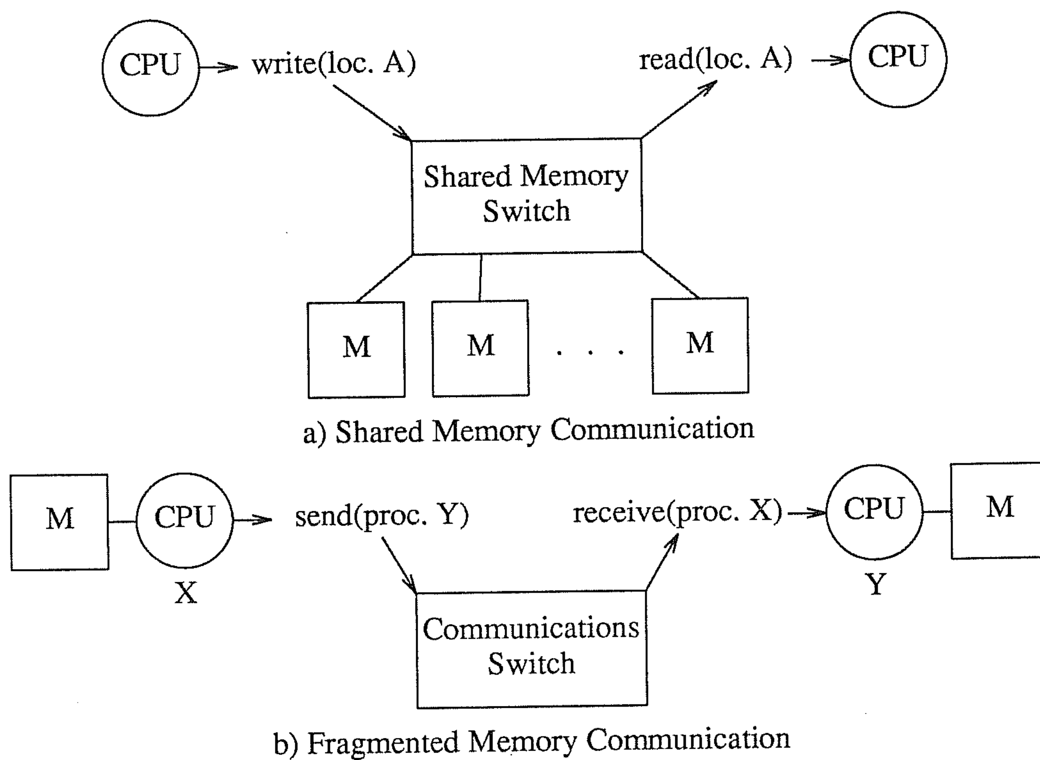


Figure 2: Communication of data in multiprocessors.

It can be seen from Fig. 1 that the switching network which communicates data among processors occupies two different positions with respect to the classical, von Neumann, single processor architecture. In shared memory, it occupies a position analogous to that of the memory bus in a classical architecture. In the fragmented memory case, it is independent of the processor to memory connection and more analogous to an I/O interface. The use of send and receive instructions in the fragmented memory case also contributes to the similarity to an I/O interface. This memory bus versus I/O channel nature of the position of the switching network underlies the naive characterization of the differences between the two types of network. A processor to memory interconnection network involves one word transfers with reliable transmission. The address (name) of the datum controls a circuit switched connection with uniform access time to any location. Since a read has no knowledge of a previous write, explicit synchronization is needed to control data sharing. In contrast, a processor to processor interconnection network supports large block transfers and error control protocols. Message switching routes the information through the network on the basis of the receiving processor's name. Delivery time varies with the source and destination pair, and the existence of a message at the receiver provides an implicit form of synchronization.

From the user's perspective, there are two distinct naive programming models for the two multiprocessor architectures. A fragmented memory machine requires mapping data structures across processors and the communication of intermediate results using send and receive. The data mapping must be available in a form which allows each processor to determine the destinations for intermediate results which it produces. Large message overhead encourages the user to gather many data items for the same destination into long messages before transmission. If many processors transmit simultaneously, the source/destination pairs should be disjoint and not cause congestion on specific paths in the network. The user of a shared memory machine sees a shared address space and explicit synchronization instructions to maintain consistency of shared data. Synchronization can be based on program control structures or associated with the data whose sharing is being synchronized. There is no reason to aggregate intermediate results unless synchronization overhead is unusually large. Large synchronization overhead leads to a programming style which uses one synchronization to satisfy many write before read dependencies at once. Better performance can result from avoiding memory "hot spots" by randomizing references so that no specific memory module is referenced simultaneously by many processors.

Why it Isn't That Simple

The naive views of the hardware characteristics and programming styles for shared and fragmented memory multiprocessors just presented are oversimplified for several reasons. First, as already mentioned, shared and private memories can be mixed in a single architecture, as shown in Fig. 3. This corresponds to real aspects of multiprocessor programs, where some data is conceptually private to the processor doing an individual piece of work. The program, while normally shared by processors, is read only for each and should be placed in a private memory, if only for caching purposes. The stack generated by most compilers normally contains only private data and need not be in shared memory. In addition, analysis done by many parallel compilers identifies some shared data as read only and thus cachable in private memory. Some multiprocessors share memories among some, but not all, processors. Examples are the PAX[1] and DIRMU[2] computers. These machines move intermediate data by having its producer place it in the correct memory and its consumer retrieve it from there. The transmission may be assisted by other processors if producer and consumer do not share a memory.

Not only may a multiprocessor mix shared and private memories, but the same memory structure may have different appearances when viewed at different system levels. An important early multiprocessor was Cm*[3], built at Carnegie Mellon University. An abbreviated

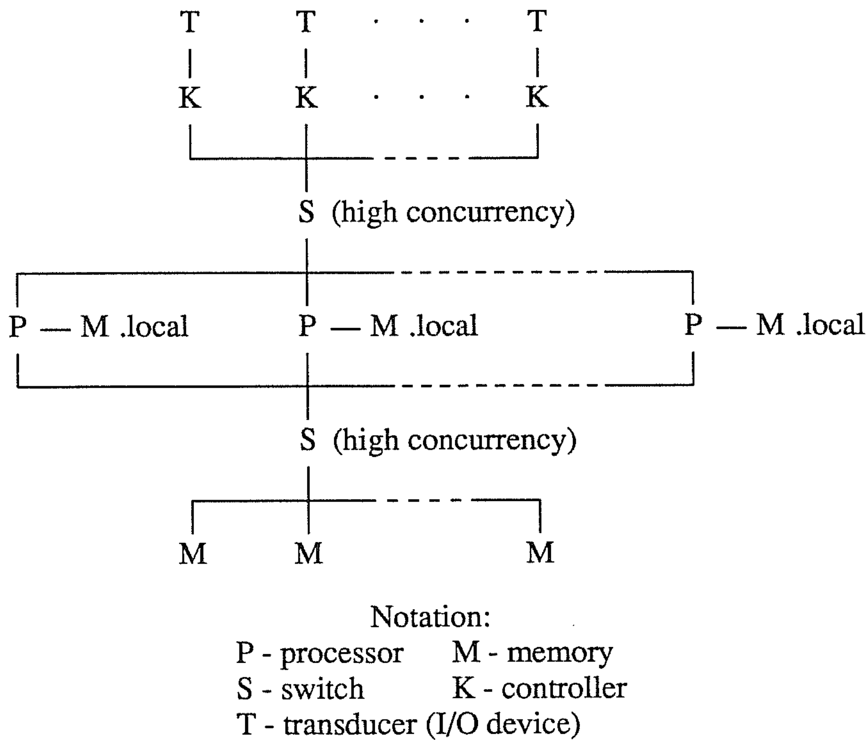


Figure 3: Shared plus private memory architecture.

block diagram of the architecture is shown in Fig. 4. Processors were attached by way of a local bus to memories and possibly I/O devices to form computer modules. Several computer modules were linked into a cluster by a cluster bus. Processors could access the memory of other processors using the cluster bus. Processors in different clusters communicated through interconnected mapping controllers, called *K.maps*. The name *K.map* and some of the behavior of Cm* are easier to understand in light of the fact that the PDP-11 had a very small physical address, so that address mapping was essential to accessing any large physical

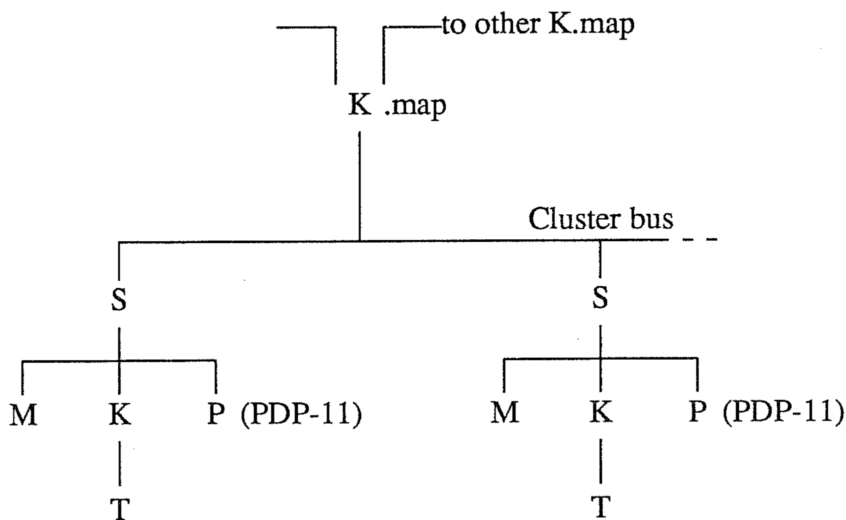


Figure 4: Architecture of the Cm* multiprocessor.

memory, shared or not.

Not only does Cm* illustrate a mixture of shared and fragmented memory ideas, but there are three answers to the question of whether Cm* is a shared or fragmented memory multiprocessor. At the microcode level in the K.map, there are explicit send and receive instructions and message passing software, thus making the Cm* appear to be a fragmented memory machine. At the PDP-11 instruction set level, the machine has shared memory. There were no send and receive instructions, and any memory cell could be accessed by any processor. The page containing the memory address had to be mapped into the processor's address space, but as mentioned, this was a standard mechanism for the PDP-11. A third answer to the question appeared at the level of programs running under an operating system. Two operating systems were built for Cm*. The processes which these operating systems supported were not allowed to share any memory. They communicated through operating system calls to pass messages between processes. Thus at this level Cm* became a fragmented memory machine once more.

Taking the attitude that a machine architecture is characterized by its native instruction set, we should call Cm* a shared memory machine. A litmus test for a fragmented memory machine could be the existence of distinct send and receive instructions for data sharing in the processor instruction set. The Cm* is an example of shared memory machines with non-uniform memory access time, sometimes called NUMA machines. If access to a processor's local memory took one time unit, then access via the cluster bus required about three units and access to memory in another cluster took about 20 units. Writing programs under either operating system followed the programming paradigm for fragmented memory multiprocessors, with explicit send and receive of shared data, but performance concerns favored large granularity cooperation less strongly than in a truly fragmented memory machine.

A more recent NUMA shared memory multiprocessor is the BBN Butterfly[4]. References to non-local memory take about three times as long as local references. The Butterfly processor to memory interconnection network also contradicts the naive characterization of shared memory switches. The network connecting N processors to N memories is a multistage network with $\log_2 N$ stages, and thus $(N/2)\log_2 N$ individual links. It thus has a potentially high concurrency, although collisions are possible when two memory references require the same link. Read and write data are sent through the network as messages with a self routing header which establishes a circuit over which the data bits follow. Messages are pipelined a few bits at a time, and long data packets of many words can use the circuit, once established. Thus, although single word transfers are the norm, higher bandwidths can be achieved by packing data into a multiword transmission. Messages attempting to reference a memory which is in use, or colliding with others in the switch, fail and are retried by the processor.

Finally, the naive view of the difference between implicit synchronization in fragmented memory and the need for explicit synchronization with shared memory should be challenged. A shared memory synchronization based on data rather than control structures is that of asynchronous variables. Asynchronous variables have a state as well as a value. The state has two values, usually called *full* and *empty*, which control access to the variable by two operations, *produce* and *consume*. *Produce* waits for the state to be empty, writes the variable with a new value, and sets the state to full. *Consume* waits for the state to be full, reads the value, and sets the state to empty. Both are atomic operations, or in general obey the serialization principle. *Void* and *copy* operations are often supplied to initialize the state to empty, and to wait for full, read and leave full, respectively. The HEP[5] and Cedar[6] computers supported these operations on memory cells in hardware.

When data is put in memory by one processor using *produce* and read by another using *consume*, the transaction behaves like a one word message from producer to consumer, with minor differences. The memory cell serves as a one word buffer, and may be occupied when

produce is attempted. The producer need not name the consumer; instead, both name a common item as when send and receive are linked to a common communications channel name. Another difference is that one *produce* and multiple *copy*s suffice to deliver the same datum to multiple receivers.

Abstraction of Characteristics

The essence of the problem to be addressed by the switching network in both shared and fragmented memory multiprocessors is the communication of data from a processor producing it to one which will use it. This process can slow parallel computation when either the producer is delayed in transmitting or when the consumer is delayed in receiving. This process can be abstracted in terms of four characteristics: initiation of transmission to the data's destination, synchronization of production and use of the data, binding of the data's source to its destination, and how transmission latency is dealt with. Table 1 summarizes these characteristics and tabulates them for the traditional views of shared and fragmented memory multiprocessors.

The initiation of data delivery to its consumer is a key characteristic and influences the others. Producer initiated delivery characterizes the programming model of fragmented memory multiprocessors. It implies that the producer knows the identity of the consumer, so that binding by processor name can be used, and provides the possibility of implicit synchronization when the consumer is informed of the arrival of data. If a producer in a shared memory multiprocessor were forced to write data into an asynchronous variable in a section of memory uniquely associated with the consumer, the programming model would be much the same as for fragmented memory. Consumer initiated access to data assumes a binding where the identity of the data allows a determination of where it resides. Since the consumer operation is decoupled from the data's writing by its producer, explicit synchronization is needed to guarantee validity. One can imagine a fragmented memory system in which part of a data item's address specifies its producer and a sharing protocol in which the consumer sends a request message to the owner (producer) of a required operand. An interrupt could cause the owner to satisfy the consumer's request, yielding a consumer initiated data transmission. Such a fragmented memory system would be programmed like a shared memory machine. Binding is by data name, and the consumer has no implicit way of knowing the data it requests has been written yet, so explicit synchronization is required.

Too many explicit synchronization mechanisms are possible to attempt a complete treatment, and sufficient characterization for our purposes has already been given. Since message delivery is less often thought of in terms of synchronization, Table 2 summarizes the types synchronization associated with message delivery. Different requirements are placed on the operating or run-time system and different precedence constraints are imposed by the possible combinations of blocking and non-blocking send and receive operations.

Types of binding between producer and consumer in fragmented memory systems include: source/destination pair, channel, and destination/type. In the case of

Characteristics	Fragmented Memory	Shared Memory
Initiation	Producer	Consumer
Synchronization	Implicit by message existence	Explicit
Binding	Processor name	Data name
Latency	Masked by early send	Consumer waits

Table 1: Data Sharing in Multiprocessors.

Message Synchronization	System Requirements	Precedence Constraints
Send: nonblocking Receive: nonblocking	Message buffering Fail return from receive	None, unless message is received successfully
Send: nonblocking Receive: blocking	Message buffering Termination detection	Actions before send precede those after receive
Send: blocking Receive: nonblocking	Termination detection Fail return from receive	Actions before receive precede those after send
Send: blocking Receive: blocking	Termination detection Termination detection	Actions before rendezvous precede ones after it in both processes.

Table 2: Summary of the types of message synchronization.

source/destination, the send operation names the destination and receive names the source. A message can be broadcast, or sent to multiple receivers, but not received from multiple sources. Source thus designates a single processor while destination might specify one or more. Message delivery can also be through a "channel" or mailbox. In this case send and receive are connected because both specify the same channel. A channel holds a sequence of messages, limited by the channel capacity. To facilitate a receiver handling messages from several sources, a sender can specify a "type" for the message and the receiver ask for the next message of that type. The source is then not explicitly specified by the receiver but may be supplied to it as part of the message. Binding in shared memory is normally by data location, but note that the Linda[7] shared tuple memory uses content addressability, which is somewhat like the "type" binding just mentioned.

The problem of latency in sharing data and how it is dealt with is the most important issue in the performance of multiprocessors. At the lowest level it is tied up with the latency and concurrency of the switch. Two slightly different concepts should be distinguished. If T_s is the time at which a send is issued in a message passing system and T_r is the time at which the corresponding receive returns data, then the latency is $T_L = T_r - T_s$. The transmission time for messages often has an initial startup overhead and a time per unit of information in the message, of the form $t_i + kt_u$, where k is the number of units transmitted. The startup time t_i is less than T_L, but is otherwise unrelated. In particular, if T_L is large, several messages can be sent before the first one is received. The granularity of data sharing is determined by the relationship of t_i to t_u. If $t_i >> t_u$ good performance dictates $k >> 1$, making the granularity coarse. If $t_i \sim t_u$ the fine granularity case of $k = 1$ suffers little performance degradation. Read and write in a shared memory switch must at least have small t_i so that data transmissions with small k perform well. A fine granularity switch with small startup t_i may still have a large latency T_L, and this is the concern of the fourth characteristic in Table 1.

Latency must grow with the number of processors in a system, if only because its physical size grows and signal transmission is limited by the speed of light. As the system size grows, the key question is how the inevitable latency is dealt with. An architecture in which latency does not slow down individual processors as the number of them increases is called scalable. Scalability is a function both of how latency grows and how it is managed. Message latency can be masked by overlapping it with useful computation. Figure 5 shows a send/receive transaction in a fragmented memory system. In part a) message latency is

Producer	Produce intermediate result	Send to consumer	Compute	
Channel			Message latency	
Consumer	Compute independent of producer			Receive intermediate result

Time →

a) Message latency well masked by computation.

Producer	Produce intermediate result	Send to consumer	Compute	
Channel			Message latency	
Consumer	Compute	Wait for message		Receive intermediate result

Time →

b) Poorly masked message latency.

Figure 5: Masking message latency by computation.

successfully overlapped by computation in the consumer whereas in part b) the consumer does not have enough to do before needing the data in order to completely mask the latency. In reference to Fig. 5, scalability is is a function of how the program doing the sends and receives is organized. The ratio of available overlapping computation to message latency decreases as system size grows, both because latency grows and because computation is more finely divided.

In shared memory multiprocessors the consumer initiation of access to data when needed eliminates the possibility of arranging the program so that sends occur early enough to mask latency. Latency can be managed in this case, as in the other, by reducing it or by masking it with useful computation. Latency reduction in the shared memory hardware regime is done by caching and latency masking by pipelining or multiprogramming. In the naive view, scalability is a hardware concern in shared memory but more a function of program structure in fragmented memory, leading to the notion of software scalability. Assuming infinitely fast transmission, networks with P processors and a reasonable number of switching nodes usually have latency on the order of $\log_m P$, where m is the number of input and output ports per switch node. If finite speed of signal transmission is an issue, latency is proportional to the cube root of P for a system buildable in three dimensional space and to the square root of P if messages occupy volume.

Concurrency of the switch also has an influence on latency. It must clearly have a concurrency much greater than one for any multiprocessor with more than a very few processors. Using a single bus for this switch is inadmissible in all but the smallest of systems. For scalability, concurrency should grow linearly with the number of processors; otherwise the lack of physical network paths will lead to long latencies when many processors use the switch simultaneously. Even with order P links, collisions between messages can occur under

unfavorable access patterns. The way to control collisions is a function of granularity. In a fine granularity network, randomization which distributes the small transactions uniformly over the network is usually appropriate. With large granularity transactions, randomization is less effective, and scheduling of the transactions may be required.

Thus the abstract differences between shared and fragmented memory multiprocessors rest on the four characteristics of Table 1, with the selection of producer or consumer initiation of data delivery having a strong influence on the other three. Consumer initiation is naively associated with explicit synchronization, data name binding, and latency reduction. Producer initiation suggests implicit synchronization, processor name binding, and latency tolerance by executing sends early.

Convergence

The direction of current developments in shared and fragmented memory multiprocessors is generally toward convergence. The desire to write programs with a shared name space for fragmented memory machines is supported both by research on virtual shared memory using paging techniques and by automatic compiled or preprocessed generation of sends and receives for remote data references. Multiprogramming the nodes of a fragmented memory multiprocessor can also increase the amount of computation available to mask latency. Virtual processors make use of the idea of parallel slackness, or using of some of a problem's inherent parallelism to control latency. In shared memory multiprocessors, considerable work is being applied to multiprocessor caching, which distributes shared data among processors to reduce latency. Hardware cache management, software cachability analysis, and correct placement and copying in NUMA machines have been considered. Much attention has been given to fast, packet switched, multistage interconnection networks for use in the processor to memory interface, and pipelining techniques have been applied to tolerate the inevitably large latency of such networks connecting many processors.

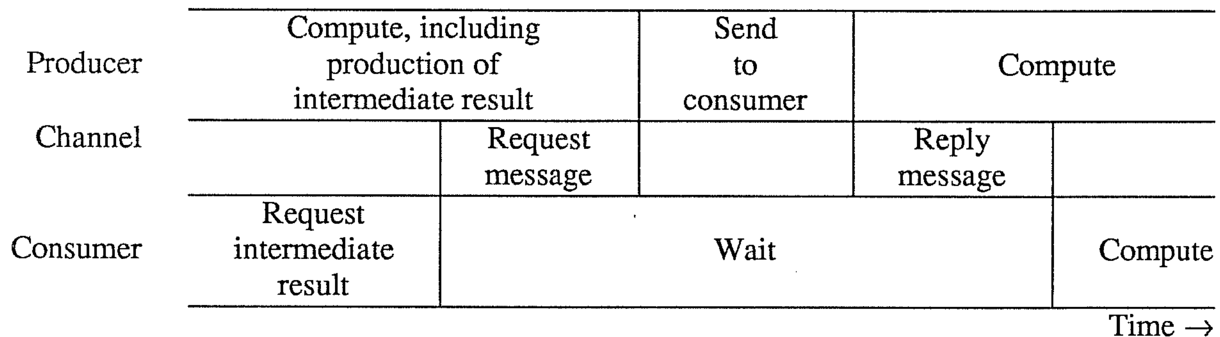
Support for a shared name space on fragmented memory multiprocessors takes several forms. Li[8] has considered using paging techniques to produce a shared memory address space on a fragmented machine. If the paging is heavily supported by hardware, convergence is easily seen between this work and the work on multiprocessor caching exemplified by [9]. Another approach uses program analysis to automatically generate the sends and receives required to move data from its producer to its consumer. For regular access patterns, the user can specify data mapping and a language like DINO[10] can generate message transmissions to satisfy non-local references. When regular access patterns are generated by loops in automatic parallelization of a sequential program[11], the more constrained structure allows even more of the mapping and data movement to be generated automatically by the compiler.

Automating data mapping across distributed memories has a long history and might be typified by the work of Berman and Snyder[12]. If access patterns are data dependent, as in computations on machine generated grids, they may still be constant over long periods. It may then be beneficial to bind addresses and generate data movement using a preprocessor[13] which acts at run-time, after data affecting addresses is known, but before the bulk of the computation, which is often iterative, is carried out. Preprocessor work can thus be amortized over many iterations with the same access pattern. Convergent work for shared memory has taken place in connection with NUMA architectures. The BBN Butterfly provides support for placement and copying to reduce the penalty for long memory references. Software places private data in the local memory of its processor and randomizes references to structures such as arrays over memory modules to avoid memory "hot spots"[14].

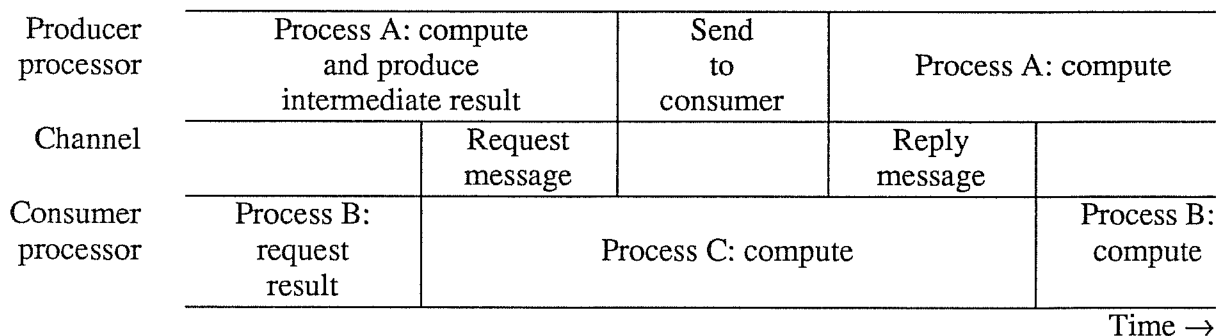
Finally, convergence in latency hiding techniques is seen between the use of virtual processors in fragmented memory and pipelining in shared memory multiprocessors. If we attempt to use consumer initiation in fragmented memory by interrupting the owner of a

datum with a request for transmission, we see a behavior like that of Fig. 6 a). In order to make use of the long wait resulting from consumer initiation of the delivery, the processor executing the consumer process can be switched to another process, as shown in Fig. 6 b). If the process is associated with a different program, we have the standard technique of masking latency by multiprogramming, which is used in masking disk latency in virtual memory systems. If the extra process is associated with the same parallel program, we have a partly time multiplexed form of multiprocessing often characterized by the name virtual processors. The use of virtual processors to enhance performance has recently been most frequently discussed in relation to an SIMD architecture, the Connection Machine[15], where it is important for masking latency arising from several different sources. If each processor of a fragmented memory multiprocessor time multiplexes several processes so that message latency in the communication network is overlapped with useful computation, a time snapshot of message traffic and processor activity might appear as in Fig. 7.

An early use of multiprocessing to mask memory latency, as opposed to I/O latency, was in the peripheral processors of the CDC 6600[16]. Ten slow peripheral processor memories were accommodated by time multiplexing ten virtual processors on a single set of fast processor logic. Process contexts were switched on a minor cycle basis. Later, the Denelcor HEP used fine grained multiprocessing to mask latency in a shared, pipelined data memory. The concept of pipelined multiprocessing is illustrated in Fig. 8. Round robin issuing of a set of process states into the unified pipeline is done on a minor cycle basis. Processes making memory references are queued separately to be returned to the execution queue when satisfied. Pipeline interlocks are largely unnecessary since instructions which occupy the pipeline simultaneously come from different processes and can only depend on each other through explicit synchronization.



a) Latency results in consumer wait.



b) Latency masked by multiprocessing.

Figure 6: Consumer initiated transmission in a fragmented memory system.

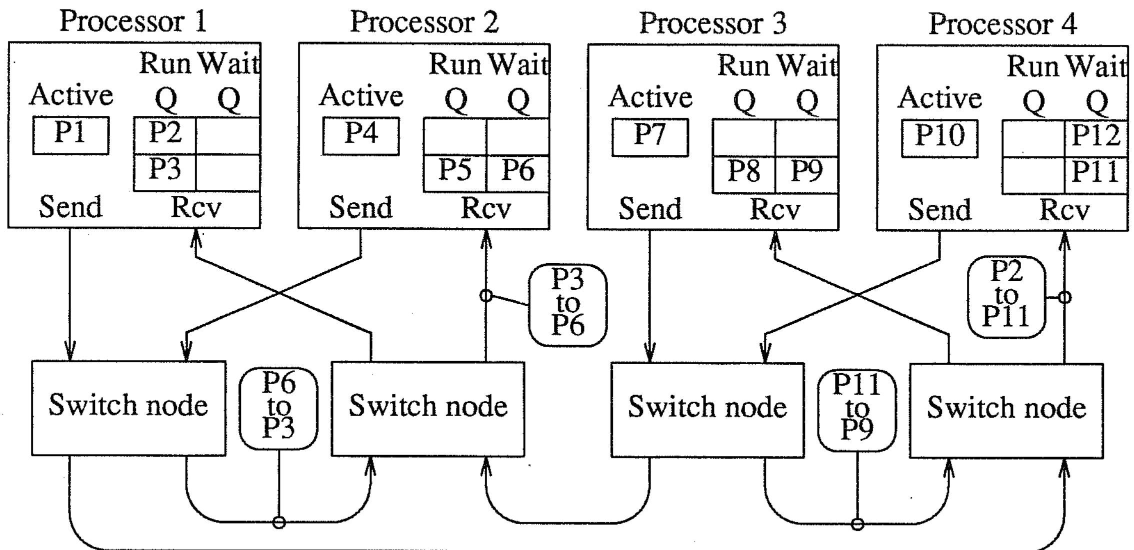


Figure 7: Masking Message Transmission with Multiprogramming.

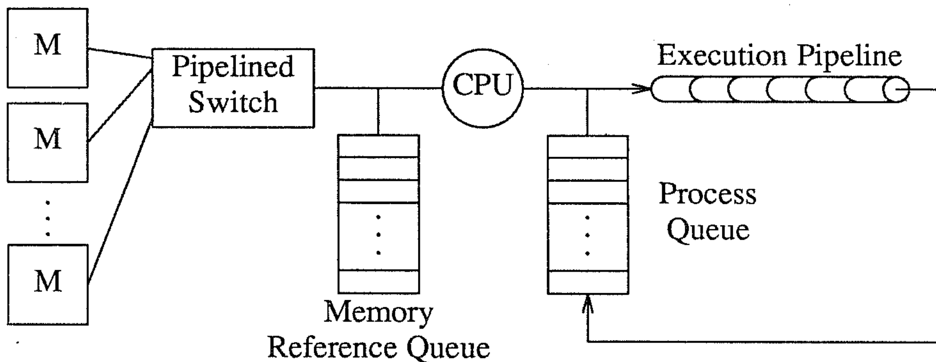


Figure 8: One execution unit of a pipelined multiprocessor.

For latency to be masked by satisfying requests at a higher rate than processor-memory latency would imply, many requests must be in progress simultaneously. This implies a pipelined switch between processors and memory, and possibly pipelining the memory also. Pipelining and single word access together imply a low overhead, message switched network. Variable traffic in the shared network requires a completion report for each transaction, regardless of whether it is a read or write. Whether the memory modules themselves are pipelined or not depends on the ratio of the module response time to the step time of the pipelined switch. If the memory module responds significantly slower than the switching network can deliver requests, memory mapping and address decoding are obvious places to use pipelining within the memory itself. Figure 9, which bears an intentional resemblance to Fig. 7, shows an activity snapshot in a system built of multiple pipelined multiprocessors which mask the latency of multiple read and write operations in the processor to memory switch.

Convergence can also be seen in switching network research. Packet switched processor to memory interconnections such as that in the NYU Ultracomputer[17] bear a strong resemblance to communication networks used in message passing distributed memory computers. Previously, the store and forward style of contention resolution was only seen in communications networks carrying information packets much larger than one memory word. There is also a strong resemblance between the "cut-through routing"[18] recently introduced in fragmented memory multiprocessors and the previously mentioned header switched connections

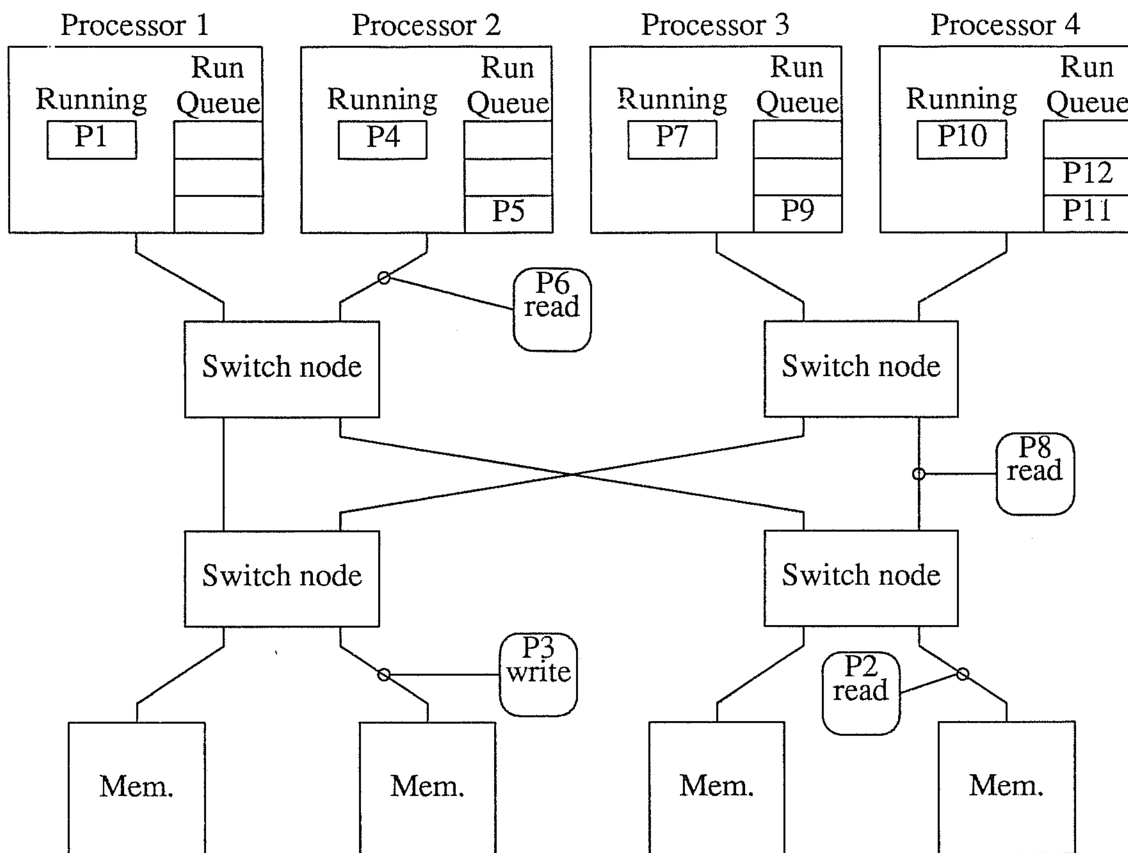


Figure 8: Pipelined Multiprocessors in a Shared Memory Multiprocessor System.

made by messages in the BBN Butterfly shared memory switch.

Conclusions

The question of what one concludes from all this is really a question of what one is led to predict for the future of multiprocessors. The predictions can be formulated as the answers to three questions: What will be the programming model and style for multiprocessors? How will the system architecture support this model of computation? What will be the split between hardware and software in contributing to this system architecture?

The programmer will surely reference a global name space. This feature corresponds too closely to the way we formulate problems, and too much progress has been made toward supporting it on widely different multiprocessor architectures, for us to give it up. It also seems that most synchronization will be data based rather than control based. Associating the synchronization with the objects whose consistency it is supposed to preserve is more direct and less error prone than associating it with the control flow of one or more processes. Programs will have more parallelism than the number of physical processors in the multiprocessor expected to run them, with the extra parallelism being used to mask latency.

Multiprocessor architecture will consist of many processors connected to many memories. A portion of the memory will be globally interconnected by way of a high concurrency switch. The switch will have a latency which scales as $\log_m P$ for moderate speed systems, with m probably greater than two. For the highest speed systems, the latency will scale as $P^{1/2}$. Multiprocessors will use a Harvard architecture, separating the program memory from data memory to take advantage of its very different access patterns. Data memory private to each processor will be used to store the stack, other process private data

and copies of read only shared data. Only truly shared data will reside in the shared memory.

A combination of software and hardware techniques will be used to mask the latency inherent in data sharing. Compiler analysis will be the main mechanism for determining what data is truly shared. It may even generate code to dynamically migrate data into private memories for a long program section during which it is not shared. The hardware will time multiplex (pipeline) multiple processes on each processor at a very fine granularity in order to support latency masking by multiplexed computation. Some of the multiprocessor cache research may find use in partially supporting the data migration with hardware, but a knowledge of reference patterns is so important to data sharing that it is unlikely that the hardware will forego the increasingly effective assistance available from the compiler.

In short, the hardware, assisted by the compiler, of multiprocessor systems can do much more than we currently ask of it. Moving software mechanisms into hardware produces a significant performance gain, and should be done when a mechanism is well understood, proven effective and of reasonably low complexity. Finally, although automatic parallelization has been poorly treated in this paper, it is perhaps possible to say that, in spite of the excellent work done in turning sequential programs into parallel ones, a user should not take great pains in a new program to artificially sequentialize naturally parallel operations so that they can be done on a computer.

REFERENCES

[1] T. Hoshino, "An invitation to the world of PAX," *IEEE Computer,* V. 19, pp. 68-79 (May 1986).

[2] W. Haendler, E. Maehle and K. Wirl, "DIRMU multiprocessor configurations," *Proc. 1985 Int'nl Conf. on Parallel Processing,* pp. 652-656 (Aug. 1985).

[3] E.F. Gehringer, D.P. Siewiorek and Z. Segall, *Parallel Processing The Cm* Experience,* Digital Press, Billerica, MA (1987).

[4] R.H. Thomas, "Behavior of the Butterfly parallel processor in the presence of memory hot spots," *Proc. of the 1986 Int'nl Conf. on Parallel Processing,* pp. 46-50 (Aug. 1986).

[5] J. S. Kowalik, Ed., *Parallel MIMD Computation: The HEP Supercomputer and its Applications,* MIT Press (1985).

[6] D. Gajski et al., "Cedar," *Proc. Compcon,* pp. 306-309 (Spring 1989).

[7] S. Ahuja, N. Carriero and D. Gelernter, "Linda and friends," *IEEE Computer,* V. 19, pp. 26-34 (1986).

[8] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors,* Ph.D. Thesis, Yale Univ. New Haven, CT (Sept. 1986).

[9] J.-L. Baer and W.-H. Wang, "Multilevel cache Hierarchies: Organizations, protocols, and performance," *J. Parallel and Distributed Computing,* V. 6, No. 3, pp. 451-476 (June 1989).

[10] M. Rosing, R.W. Schnabel and R.P. Weaver, "Expressing complex parallel algorithms in DINO," *Proc. 4th Conf. on Hypercubes, Concurrent Computers & Applications,* pp. 553-560 (1989).

[11] D. Callahan and K. Kennedy, "Compiling programs for distributed-memory multiprocessors," *J. of Supercomputing,* V. 2, pp. 131-169 (1988).

[12] F. Berman and L. Snyder, "On mapping parallel algorithms into parallel architectures," *Proc. 1984 Int'nl Conf. on Parallel Processing*, pp. 307-309 (1984).

[13] J. Saltz, K. Crowley, R. Mirchandaney and H. Berryman, "Run-time scheduling and execution of loops on message passing machines," *J. Parallel and Distributed Computing*, V. 8, pp. 303-312 (1990).

[14] R. Rettberg and R. Thomas, "Contention is no obstacle to shared-memory multiprocessing," *Communications of the ACM*, V. 29, No. 12, pp. 1202-1212 (Dec. 1986).

[15] L.W. Tucker and G.G. Robertson, "Architecture and applications of the Connection Machine," *Computer*, V. 21, pp. 26-38 (Aug. 1988).

[16] J. E. Thornton, *Design of a Computer: The Control Data 6600*, Scott, Foresman and Co., Glenview, Ill. (1970).

[17] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer—Designing an MIMD shared memory parallel computer," *IEEE Trans. on Computers*, v. C-32, No. 2, pp. 175-189 (Feb. 1983).

[18] W.J. Dally and C.L. Seitz, "The Torus routing chip," *Distributed Computing*, V. 1, pp. 187-196 (1986).