CHAPTER 9 Multiprocessors and Multicomputers

9.1 Introduction

Most computers today use a single microprocessor as a central processor. Such systems—called *uniprocessors*—satisfy most users because their processing power has been growing at a compound annual rate of about 50%. Nevertheless, some users and applications desire more power. On-line transaction processing (OLTP) systems wish to handle more users, weather forecasters seek more fidelity in tomorrow's forecast, and virtual reality systems wish to do more accurate visual effects. If one wants to go five times faster, what are the alternatives to waiting four years ($1.5^4 \approx 5$ times)?

One approach to going faster is to try to build a faster processor using more gates in parallel and/or with a more exotic technology. This is the approach used by supercomputers of the 1970s and 1980s. The Cray-1 [31], for example, used an 80 MHz clock in 1976. Microprocessors did not reach this clock rate until the early 1990s.

Today, however, the exotic processor approach is not economically viable. Microprocessors provide the fastest computing engines by integrating tens (soon hundreds) of millions of transistors together with short wires to form complex pipelines and integrated memory systems. Designs are then carefully tuned and implemented with an expensive fabrication technology. A microprocessor's sales must reach millions of units to adequately amortize up-front costs. A customized microprocessor would not ship in enough volume to amortize the design and testing costs for complex pipelines and integrated memory systems. A processor that is not a microprocessor could use an exotic technology, but would have longer wires—and hence longer wire delays—and the same design and testing cost problems.

Therefore, the only viable alternative for going faster than a uniprocessor is to employ multiple microprocessors and connect them in some fashion so that they can cooperate on the same problem. Since uniprocessors have long separated memory systems and I/O systems, it should not be surprising that the two alternatives for connecting processors are via the memory system and via the I/O system. In either case, processors in these systems execute instructions independently, and, therefore, are *multiple instruction multiple data* (*MIMD*) systems, not single instruction multiple data (SIMD) systems, as were described in the previous chapter.

Shared-memory multiprocessors join multiple processors with a logically-shared memory. Hardware ensures that a store by a processor to a physical address will be visible to all other processors. Thus, normal loads and stores can be used for communication. Shared memory multiprocessors tend to also have a logically-shared I/O system. Implementing shared-memory multiprocessors is challenging, because good performance dictates that the logically-shared memory (and I/O) be physically distributed.

Multiple processors can alternatively be joined via the I/O system. With this approach each processor can reside in a conventional computer *node* augmented with a network interface (NI). The NI, which usually resides on an I/O bus, could be a standard NI (e.g., for Ethernet) that supports standard network protocols (e.g., TCP/IP) to connect nodes with a local area network or even the Internet. We do not examine this case further, but instead refer readers to networking texts, such as Peterson and Davie [30]

We will instead examine systems that connect computers via custom NIs and/or custom networks. The NIs and networks of these systems—sometimes called *multicomputers*—vary from specialized (e.g., for the Intel line of multicomputers) to something that is close to a local area network (e.g., Myricom Myrinet [8]). Unlike shared-memory multiprocessors, multicomputers do not use stores and loads for communication. Instead, hardware supports explicit mechanisms for passing messages between one node and another. Implementing high-performance multicomputers is challenging, because it is hard to avoid inordinate software overhead sending and receiving each message.

Parallel computing has had a significant impact on computing. Furthermore, we expect it to be even more important in the future. Traditionally, it has been a way to speed high-end applications at any cost. Today, we recognize that it can also be cost-effective [41]. A p-processor system is cost-effective relative to a uniprocessor if its speed improvement in executing a workload exceeds its cost premium versus the uniprocessor. In many cases, the cost premium for adding additional processors is low, because substantial costs go into the memory and I/O systems. In 1995, for example, an eight-processor Silicon Graphics Challenge XL with 1GB memory cost 2.5 times a comparable uniprocessor [41]. Thus, the 8-processor XL was cost-effective for all workloads it could speed up by more than 2.5 times.

The next sections discuss parallel computer software, shared memory multiprocessors, and multicomputers. More background on parallel software and hardware can be found in Almasi and Gottlieb [3] and Culler, Singh, and Gupta [13]. Both books significantly influenced the ideas presented here.

9.2 Parallel Computer Software

Modern parallel systems have three interface levels: *user programming model*, *application binary interface* (ABI), and the *hardware*. Compilers and libraries work to map applications to an ABI. ABIs are implemented on hardware via operating system services and device drivers. To design parallel hardware we must understand some about these interfaces and the software that converts between them.

9.2.1 Programming Models

For uniprocessors, programming discussions sometimes consider the relative merits of alternative programming languages, such as Fortran, C, C++, or Java. These languages are actually very similar and make similar demands on hardware. When programming parallel computers, however, a debate on Fortran vs. C is secondary to more fundamental issues such as whether data sharing is done by writing a variable or sending a message. These issues determine the *programming model* used.

The four most popular programming models are sequential, data parallel, shared memory, and message passing. With the *sequential* model, programmers write regular uniprocessor programs and rely on system software—mostly compilers—to parallelize their code and data . This approach relieves programmers of the burden of parallelization and is successful in some cases, but is not robust enough for general use despite decades of work.

With the *data parallel* model, programmers write code that is sequential in control flow but parallel in data operations (e.g., in High Performance Fortran). Assume that all the following variables represent matrices:

```
A = B + C

E = A * D
```

With data parallel, all processors would appear to execute the many additions in the matrix add before any began the multiplies and adds of the matrix multiply. In many ways, data parallel programming is an abstraction of SIMD processing. It can be wildly successful, but many other important programs work poorly with it or cannot be easily expressed.

The *shared memory* model is based on multiple sequential threads sharing data through shared variables. It provides the same model as a multi-tasking uniprocessor. It has achieved great success in small to medium-sized systems (less than 30 processors) for applications such as on-line transaction processing and scientific simulation. The greatest challenge of shared memory programming is including sufficient synchronization so that data communication is meaningful (e.g., a load does not read data before the store that writes it executes).

Finally, the *message passing* model allows programming with conventional uniprocessor languages augmented with explicit calls for communication. In most cases, communication is initiated with a send call that copies data from a buffer and sends it to a destination. An application usually obtains the communicated data with a receive call that copies data into a specified buffer. If the data is not yet available, the receive call can block or return a not-ready flag. The message passing model closely resembles the model used in network computing. Message passing has had many successes, but can make it difficult to correctly implement complex, pointer-based data structures.

9.2.2 Application Binary Interfaces (ABIs)

Over most of the history of parallel computing there has been no separation between programming model and hardware. Rather machines were characterized by the programming model that they supported. Data parallel or SIMD machines include the Illinois Illiac IV and Thinking Machines CM-1. Message passing machines include the Caltech Cosmic Cube, Intel Hypercube, and Intel Paragon. Shared memory machines include CMU C.mmp, Sequent Balance, and SGI 2000.

Today we understand that programming models and hardware are different. Any hardware can support any programming model. Modern systems use a compiler and libraries to map an application to an *application binary interface* (ABI). They then use operating system services and device drivers to implement an ABI on hardware. The challenge then is to identify ABIs and hardware mechanisms that work well.

Today's parallel systems have evolved to two classes of ABIs: shared memory and messaging. The *shared memory ABI* presents applications with an interface similar to a multi-tasking uniprocessor. Compiler and libraries map applications written in user programming models to the shared memory ABI. Mapping the sequential programming and data parallel programming models to a shared memory ABI requires extensive compiler support. Supporting shared memory is straightforward. Implementing message passing requires only a library that implements send and receive calls in shared memory.

The *messaging ABI* presents applications with an interface similar to communicating on a network. Nodes have separate memory and separate I/O spaces, but can exchange data through messages. Mapping user programming models to the message passing ABI is challenging. Implementing sequential and data parallel programming models on a message passing ABI requires even better compiler support than with a shared memory ABI, because the cost of unnecessary messages is usually higher than the cost of unnecessary memory references. Implementing the shared memory programming model on a message passing ABI is also hard when good performance is required. Supporting message passing program on a message passing ABI, however, requires only a straightforward library. The most popular such library for scientific programming is *Message Passing Interface* (MPI) [35, 18].

9.2.3 Hardware

Finally, one must implement the two ABIs, shared memory and messaging, on the two classes of hardware platforms—multiprocessors and multicomputers.

- *Shared memory ABI on a shared-memory multiprocessor.* The most significant challenge is developing an operating system that runs well on many processors.
- *Messaging ABI on a shared-memory multiprocessor.* Communication can be done with standard operating system mechanisms, provided they perform well enough.
- *Shared memory ABI on a multicomputer.* This is hard. Li and Hudak [28], discussed later, give some answers for sharing memory, but sharing I/O must also be implemented.
- *Messaging ABI on a multicomputer.* This can be implemented with a standard operating system with standard networking. The challenge is obtaining good messaging performance, especially if the operating system is involved on every message.

The relationships between programming model, ABI, and hardware can be confusing, but making these distinctions can be important. There are results, for example, that show that (a) message passing programming on a shared memory ABI on a multiprocessor can lead to higher performance than either (b) shared memory programming on a shared memory ABI on a multiprocessor or (c) message passing programming on a messaging ABI on a multicomputer.

Now for a quiz. What is the difference between (a) a shared memory program on a messaging ABI on a multicomputer and (b) a shared memory program on a shared memory ABI on a multicomputer? Answer. In (a) compilers or libraries would convert accesses to shared variables into messages (e.g., Berkeley Split-C [12]). In (b) the application actually uses load and store instructions and the system makes things work correctly (e.g,. Li and Hudak's Ivy [28]).

9.3 Shared-Memory Multiprocessors

This section examines the evolution of shared memory multiprocessors. In these machines a processor can use normal memory-referencing instructions (e.g., loads and stores) to access all memory and all processors can access the same memory location with the same address.

Most shared-memory multiprocessors allow processors to have caches. Caches work by making a copy of data that is still associated with the data's original address. Insuring that processors (and devices) obtain and update the most recent copy of data is the *cache coherence problem* and is usually solved by a *cache coherence protocol* (e.g., Goodman's write-once described in Chapter FOO). Multiprocessors that use cache coherence often get the designation *CC*, while those that don't are *NCC*.

Multiprocessors also differ on whether all accesses to memory encounter a similar delay (*uniform memory access* or *UMA*) or not (*non-UMA* or *NUMA*). This leads to four basic categories to multiprocessors: NCC-UMA, NCC-NUMA, CC-UMA, and CC-NUMA.[1] CC-UMAs are also called *symmetric multiprocessors* (*SMPs*).

9.3.1 Wulf and Harbison's "Reflections in a pool of processors/An experience report on C.mmp/

1. NCC-UMA and NCC-NUMA machines are often called UMAs and NUMAs, respectfully. We include the prefix NCC to avoid ambiguity.

Hydra"

Wulf and Harbison [42] summarize a seminal 1970s research project on multiprocessor hardware and operating systems. This project, begun at Carnegie Mellon University in 1972, developed the *C.mmp* hardware and the *Hydra* operating system. C.mmp provided uniform-delay access from all 16 DEC PDP-11 processors to 16 interleaved memory modules using a crossbar switch, but no caches. It pioneered a NCC-UMA architecture using of off-the-shelf components. Hydra was a symmetric operating system (no master-slave relationships) that also separated policy from kernel mechanisms and used capabilities.

Wulf and Harbison's paper is also notable as one of the most thorough and balanced research project retrospectives we have ever seen. If all research projects were as forthcoming about their technical and non-technical successes and failures as C.mmp, we could all learn much more from each other. Readers wishing to probe further into this era should also read about C.mmp's successor CMU CM* [36]

9.3.2 Lamport's "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs"

As architects began to consider performance optimization for shared-memory systems—such as write buffers and caches—it became useful to define exactly what correct shared memory should do. In a uniprocessor, a load to memory should return the value of the *last* store to the same address. In a multi-tasking uniprocessor, a load should return the value of the last store to the same address, but now this store could be from the load's thread or from another thread that was multiplexed onto the processor since the load's thread last stored this address.

Lamport uses this sort of reasoning to define *sequential consistency* (*SC*) [24]. Informally, a multiprocessor implements SC if it always behaves like a multi-tasking uniprocessor. Lamport formalizes this with a total order of all memory operations that respects the program order at each processor. Implementations of SC must keep performance optimizations—such as write buffers and caches—hidden from programmers.

To improve performance, other *memory consistency models* have been proposed and deployed. Some models expose first-in-first-out write buffers to programmers: *processor consistency, SPARC TSO,* and *Intel IA-32*. Other models allow some memory operations to be completely out of order: *weak consistency, release consistency, DEC/Compaq Alpha, IBM PowerPC,* and *Sun RMO*. Adve and Gharachorloo [1] provide a contemporary tutorial on alternative models, while Hill [21] argues that speculative execution should drive shared memory multiprocessors back to sequential consistency.

9.3.3 Snooping Cache Coherence

Goodman's "Using Cache Memory to Reduce Processor-Memory Traffic" [17] appears in Chapter 6 and is described more fully there. What is important for this chapter is Goodman's definition of the *write-once* protocol, the first snooping cache coherence protocol. On the first write to a block, the write-once protocol performs a write-through that also invalidates other cached copies. On subsequent writes, write-back is used. Write-once spawned a series of alternative snooping protocols that were then unified with the MOESI framework [37].

More importantly, snooping protocols have led to the most successful class of commercial multiprocessors: CC-UMAs or *symmetric multiprocessors* (SMPs). An early commercial SMP is the Sequent Balance [39]. An example of the state of the art is Sun UltraEnterprise 5000 [34]. Its bus supports up to 112 simultaneous transactions from up to 30 CPUs and updates coherence state immediately after a transaction begins and regardless of the order of data transfers.

9.3.4 Censier and Feautrier's "A New Solution to Coherence Problems in Multicache Systems"

Censier and Feautrier [10] present the first *directory* cache coherence protocol approach suitable for scaling to a large number of processors. This paper predates the invention of snooping, but is only now becoming important in commercial systems. Each block in memory has a directory entry that provides information on the block's caching status. The directory entry is at a known location, regardless of where the block is cached. Censier and Feautrier's directory entry has a bit to identify which processor caches a block. This scheme was later classified as Dir_nNB by Agarwal et al. [2]. In each cache, two state bits identify whether a block is modified (and therefore exclusive), valid (but unmodified and potentially shared), or invalid. On a processor read of an invalid block or a processor write of an unmodified block, a request is sent to the directory, which may in turn send messages to obtain the block or invalidate soon-to-be-stale copies.

Prior to Censier and Feautrier, Tang [38] developed a coherence protocol more suitable for a small number of processors, because it duplicated cache tags for all processors at the memory controller.

9.3.5 Lenoski, et al.'s "The Stanford DASH Multiprocessor"

A decade after Censier and Feautrier's directory proposal Stanford University implemented a refinement of the idea in the DASH shared memory multiprocessor prototype. Stanford DASH [27] connected up to 16 SGI four-way SMPs using two two-dimensional mesh network. It implemented shared memory with a CC-NUMA architecture using a distributed directory protocol. The project is notable for many research ideas on coherence protocols, memory consistency models, benchmarking, etc. Equally important, DASH "put it all together" to explore race-condition, deadlock, and other implementation issues. This exploration cleared the way for commercial follow-ons, such as the SGI Origin 2000 [25].

9.3.6 Hagersten, Landin, and Haridi's "DDM--A Cache-Only Memory Architecture"

Hagersten et al. [19] observe that CC-NUMA machines pay substantial memory cost to keep locally-cached copies of data from more distant memory modules. To counter this waste they propose to turn all memory into cache so that data used at one node need not also exist at a remote node where it is not used. This architecture was dubbed *cache-only memory architecture* (*COMA*). Key challenges for implementing COMA include finding data that has no permanent home and replacing data from a cache when it does not currently exist elsewhere in the system. These ideas were independently developed for the KSR-1, a commercial machine from Kendall Square Research [32]. To date, there have been no commercially-successful COMA machines, but researchers are still looking at ways to modify the idea to make it less complex to implement.

9.3.7 Multiprocessors Today and Tomorrow

Shared memory multiprocessors are now commonly used for servers and larger computers. Most of these are symmetric multiprocessors—CC-UMAs that use a bus. A few large machines, such as the Silicon Graphics Origin 2000, employ the CC-NUMA design. The Cray T3E is a NCC-NUMA [33]. We are not aware of any current NCC-UMAs.

For the cost-effectiveness arguments given we expect more desktop machines to become SMPs (CC-UMAs). SMPs will continue to flourish for low end servers, while directory-based CC-NUMAs become more widely deployed at the high-end. Exactly what size will divide SMPs from CC-NUMA will depend on the relative ingenuity of the engineers designing these systems. We expect NCC-NUMAs and NCC-UMAs to die out, because coherent caching is important for obtaining good performance on general purpose user and system software.

9.4 Multicomputing

This section concentrates on systems that do not globally share memory. Instead, nodes contain one or more processors, locally-shared memory, optional I/O devices, and an interconnection network interface. Processors use normal memory-referencing instructions to obtain data within a node, but obtain remote data with other mechanisms. At the extreme, a collection of hosts on the internet meets the above definition. We, however, will concentrate on systems that are more strongly coupled. These systems are sometimes called *multicomputers* or *NORMAs* (N*O Remote Memory Access*).

9.4.1 Athas and Seitz's "The Cosmic Cube"

Athas and Seitz [7] describes the Caltech Cosmic Cube. The Cosmic Cube pioneered the multicomputer hardware architecture and message-passing programming model. It included 64 nodes connected by a hypercube network. Each node contained an Intel 8086/8087 processor and 128K bytes of memory. It was programmed with message passing. With message passing, per-node programs are augmented with explicit `send()` and `receive()` calls to perform internode communication.

9.4.2 Multicomputer Follow-ons

The Caltech Cosmic Cube led to a series of Intel multicomputers, including the iPSC/1, iPSC/2 and Paragon, and influenced all other multicomputers vendors. These machines were employed by scientists at research labs and universities, but they did not enjoy broad commercial success. One problem is that message passing overheads have been large (>> 1 ms). A second is that these machines targeted message-passing exclusively, and this programming model is too limiting for many applications.

Many efforts have sought to reduce message passing overhead. We mention four here. The Thinking Machines CM-5 [26] was a commercial multicomputer that moved its network interface (NI) from the I/O to the memory bus. Furthermore, it mapped the NI into user-space so that user-level software could send and receive messages with uncached loads and stores. Protection was still maintained with a partitionable network and special context switch mechanisms. A spe-

cial network supported rapid global reductions for many associative operations (e.g., the sum of integers where each node contributes one number).

Berkeley active messages [40] was a software-only idea that sought to move the message abstraction down to something close to what hardware could actually implement. An active message contains a handler address and zero or more arguments. Messages arriving at their destination spawn a thread that begins execution at the handler address. There is no explicit receive() call.

The MIT J-Machine [29] explored connecting many small nodes (e.g., 1024). Each node contains 1 Mbyte of memory and a custom "message-driven processor." Particularly interesting is the processor's custom support for communication (message send instructions and a hardware receive/dispatch queue), synchronization (through tags), and naming (with instructions for loading and querying an associative table).

A final way multicomputers are evolving is toward greater exploitation of standard hardware and software. The IBM SP/2 [reference] uses workstation boards running IBM's standard Unix, but adds a custom NI and custom network and wraps everything in a custom box. Other systems use conventional workstations placed close together and connected with a custom local area network, such as Myricom Myrinet. Finally, one can run systems like Oak Ridge Parallel Virtual Machine (PVM) [15] to harness idle workstations from across your organization's desktops.

9.4.3 Li and Hudak's "Memory Coherence in Shared Virtual Memory Systems"

Li and Hudak [28] do not present a new multicomputer. Instead, they present Ivy—the first implementation of the shared memory ABI on a multicomputer (actually a network of workstations). This kind of a system—now called a *software distributed shared memory* system[1]—allows shared-memory application binaries to issues loads and stores and transparently insures that shared memory behaves correctly (e.g., sequentially consistently) across the system. The key is using standard virtual memory page protection—valid and writable—to simulate caching state. On a load or store to an inappropriate page, hardware generates a page fault that gets forwarded to Ivy so that it can send messages (following a directory protocol) to obtain the data and resume the application.

Substantial subsequent work [4, 5] seeks to ameliorate two key problems with Ivy: (1) false sharing due to maintaining coherence on whole pages and (2) long message delays between loosely-coupled machines. Some solutions blur the distinction between software distributed shared memory and COMA by maintaining coherence on cache blocks but naming locally-cached remote data with virtual memory hardware [20, 22]. Like COMA, DSM ideas cannot yet claim any substantial commercial successes.

1. Machines like the SGI Origin 2000 are called *hardware distributed shared memory* machines. Some use *DSM* to refer to both software and hardware distributed shared memory machines, while others reserve DSM for software distributed shared memory machines only.

9.4.4 Multicomputers Today and Tomorrow

Today the most successful multicomputers have nodes that are (or are similar to) workstations. The most successful product is the IBM SP/2. More widely deployed are hundreds of sites that employ "networks of workstations" [6]. These are not optimal computing systems, but the marginal cost for employing them can be nearly zero, since the workstations are already deployed on people's desks. In our opinion, however multicomputers will become less important than multiprocessors for large-scale computation, because the overheads of messages bleed away too much performance.

Multicomputers, however, are and will continue to flourish in *high-availability* computing. *Availability* is the probability at a given time that enough of the system is up to perform a given task. High-availability is important for web servers and critical for data bases. Multicomputers are a natural fit for high-availability, because it is relatively straightforward to isolate multicomputer nodes when a node crashes. This allows an n-node system to continue to be available at (n-1)/n throughput while another node is re-booting.

Highly-available multicomputers will continue to evolve. Today, most multicomputers are connected with standard local area networks (e.g., 100Mb/sec ethernet), but some use custom local area networks like Myricom Myrinet. Some people argue that custom local area networks will evolve into a new class of networks—called *system area networks*—that provide better bandwidth and latency than local area networks [23].

Finally, it is possible that there will be a blurring of the line between multicomputers and multiprocessors. One thrust toward making this happen is an effort to allow multicomputers to shared information without the operating system being involved with every message. Efforts, such as DEC Memory Channel [16] and Compaq/Intel/Microsoft's Virtual Interface Architecture [14], can move multicomputer communication bandwidths and latencies closer to those of multiprocessors. Alternatively, multiprocessors could be made more like multicomputers if parts the system can continue to be available even as other parts crash. Techniques like those used in Stanford HIVE [11] and DISCO [9] are steps in this direction.

9.5 References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[3] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin / Cummings Publishing Company, Inc., 1994.

[4] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwanepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[5] Jelica Protic andA Milo Tomasevic andA Veljko Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel & Distributed Technology, Systems, & Applications*, 4(2):63–79, Summer 1996.

[6] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[7] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, 21(8):9–25, August 1988.

[8] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[9] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[10] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[11] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teeodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 12–25, December 1995.

[12] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[13] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[14] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, March/April 1998.

[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. The book is available electronically, the url is t ftp://www.-netlib.-org-/-pvm3-/-book-/-pvm-book.-ps.

[16] Richard B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[17] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proc. Tenth International Symposium on Computer Architecture*, pages 124–131. NA, June 1983.

[18] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, 1998.

[19] Erik Hagersten, Anders Landin, and Seif Haridi. DDM–A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[20] Erik Hagersten, Ashley Saulsbury, and Anders Landin. Simple COMA Node Implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, January 1994.

[21] Mark D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, 31(8):28–34, August 1998.

[22] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[23] Robert W. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, 15(1):37–45, February 1995.

[24] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[25] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc.*

24th Annual International Symposium on Computer Architecture, pages 241–251, June 1997.

[26] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1993.

[27] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[28] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[29] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 224–235, May 1993.

[30] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 1996.

[31] Richard M. Russell. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[32] Rafael H. Saavedra, R. Stockton Gaines, and Michael J. Carlton. Micro Benchmark Analysis of the KSR1. In *Proceedings of Supercomputing '93*, pages 202–213, November 1993.

[33] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.

[34] Ashok Singhal, David Broniarczyk, Fred Cerauskis, Jeff Price, Leo Yaun, Chris Cheng, Drew Doblar, Steve Fosth, Nalini Agarwal, Kenneth Harvery, Erik Hagersten, and Bjorn Liencres. Gigaplane: A High Performance Bus of Large SMPs. In *Proc. IEEE Hot Interconnects*, pages 41–52, August 1996.

[35] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core*. MIT Press, 1998.

[36] Richard J. Swan, S. H. Fuller, and Daniel P. Siewiorek. Cm* – A Modular, Multi-Microprocessor. In *Proceedings AFIPS National Computer Conference*, pages 637–644, 1977.

[37] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[38] C. K. Tang. Cache System Design in the Tightly Compled Multiprocessor System. In *Proc. AFIPS National Computing Conference*, pages 749–753, June 1976.

[39] Thakkar, Gifford, and Fielland. Balance: A Shared Memory Multiprocessor System. In *Proc. 2nd Int. Conf. on Supercomputing*, pages 93–101, 1987.

[40] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[41] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.

[42] W. A. Wulf and S. P. Harbison. Reflections in a pool of processors/An experience report on C.mmp/Hydra. In *Proc. Nation Computer Conference (AFIPS)*. NA, June 1978.