Appears in: "ASPLOS VI," Oct. 1994.

Where is Time Spent in Message-Passing and Shared-Memory Programs?*

Satish Chandra James R. Larus Anne Rogers†

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
{chandra,larus}@cs.wisc.edu

Department of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08544 USA
amr@cs.princeton.edu

Abstract

Message passing and shared memory are two techniques parallel programs use for coordination and communication. This paper studies the strengths and weaknesses of these two mechanisms by comparing equivalent, well-written message-passing and shared-memory programs running on similar hardware. To ensure that our measurements are comparable, we produced two carefully tuned versions of each program and measured them on closely-related simulators of a message-passing and a shared-memory machine, both of which are based on same underlying hardware assumptions.

We examined the behavior and performance of each program carefully. Although the cost of computation in each pair of programs was similar, synchronization and communication differed greatly. We found that message-passing's advantage over shared-memory is not clear-cut. Three of the four shared-memory programs ran at roughly the same speed as their message-passing equivalent, even though their communication patterns were different.

1 Introduction

Parallel machines rely on two distinct mechanisms—message passing and shared memory—for communication and coordination. Proponents of each mechanism have offered many arguments in favor of their approach. This paper analyzes the strengths and weaknesses of these two mechanisms by measuring and comparing equivalent, well-written message-passing and shared-memory programs running on detailed simulators of comparable message-passing and cache-coherent shared-memory machines.

This paper reports detailed measurements of the execution time of highly-tuned message-passing and shared-memory programs that use the same algorithms and run on detailed architectural simulators with a common hardware base. We accurately measured the time that each program spent computing, communicating, and synchronizing. Because of the commonalities, we can compare where these pairs of programs spend their time.

To ensure that our measurements are comparable, we produced two carefully tuned versions of each program. Two of the four programs (Gauss and MSE) started as message-passing programs. From them, we wrote shared-memory programs. One program (LCP) started as a shared-memory program. From it, we wrote a message passing program. The final program (EM3D) started as a Split-C [3] program, which was the basis for message-passing and conventional shared-memory programs. In tuning a program, we frequently found insights and techniques from one version helpful in improving the other.

Both machine simulators are based on the Wisconsin Wind Tunnel [19]. We used its Dir_nNB protocol simulation [1] as our cache-coherent shared-memory machine. We built a simulator of a message-passing machine *similar* to a CM-5, which was detailed enough to execute directly a slightly modified version of the Thinking Machine's active message library (CMAML) and an unmodified copy of the TMC message-passing library (CMMD). The simulator predicted execution times for three programs within 27% of an actual machine.

We were surprised to find no clear performance advantage for message passing. Three of four shared-memory programs ran at roughly the same speed as their message-passing equivalent. The time each pair of programs spent computing was very close, al-

*This work is supported in part by NSF PYI/NYI Awards CCR-9157366 and CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, and donations from Digital Equipment Corporation, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

†On leave at the University of Wisconsin.

though the overhead of managing buffers for message passing was visible. Communication was a different story. In three programs, message-passing communication was slightly faster and in another program it was significantly faster. Message passing permitted programs to control the timing and protocol for data transfer, provided bulk data transfer, and eliminated separate synchronization. Partially offsetting these advantages, the message-passing programs spent a significant amount of time in communication library routines, despite the CM-5's efficient network interface. In three cases, these factors cancelled and shared memory and message passing both performed well. Only when the cache-coherence protocol clearly did not match an application's intensive communication did a large difference emerge.

In MSE, a program dominated by computation, shared memory's higher cost to obtain remote data was offset by not having to manipulate buffers. The Gauss program implemented reductions and broadcasts in software. The shared-memory implementation performed as well as the more complex message-passing version, because of the high latency of handling messages in software. In the LCP program, message-passing communication was slightly more expensive, but this additional cost was more than offset by the additional synchronization that shared memory required. Finally, EM3D benefited the most from message passing. $Dir_n NB$'s invalidation-based cache-coherence policy proved to be an expensive way (in time and number of messages) of transferring data between a producer and consumer [6]. The four programs used the communication mechanisms in very different ways, which (except for EM3D) did not demonstrate a convincing advantage for either approach.

This comparison of message passing and shared memory should be useful to designers and users of parallel machines. Designers need to know which approach offers the highest performance. Recent evidence [12] (and our results) suggests than neither approach dominates the other, which argues for incorporating both in a machine. When this occurs, programmers and compiler writers will need to make frequent choices between alternative mechanisms. To choose, they need an accurate understanding of message passing and shared memory's strengths and weaknesses. The tools and techniques developed for this study can be used to compare many other aspects of message passing and shared memory. The range of possibilities studied in this paper was necessarily limited.[1]

[1] For example, we did not studying the benefits of prefetching in shared memory or DMA transfers in message passing.

The rest of the paper is organized as follows. The next section discusses related work. Section 3 describes the techniques used to simulate both systems. Section 4 describes details of the shared memory and message passing machines that we simulated. Section 5 presents the results of our experiments. Section 6 concludes the paper.

2 Related Work

Previous papers that compared message passing and shared memory fall into two groups. The first group compared a shared-memory program against a similar program written with a message-passing library that was implemented in shared memory on the same machine. Lin and Snyder [15] compared shared-memory programs written under a naive model and a more accurate model. They found that the latter programs performed better on both a cache-coherent and a NUMA shared-memory machine. Ngo and Snyder [18] compared several shared-memory programs against message-passing versions running on the same shared-memory machine. The message-passing programs, again written to distinguish local and non-local data, performed better.

These papers compared two styles of writing a program. Although the experiments provide strong evidence that shared-memory programmers should be aware of locality, they compare neither machine implementations nor programming styles. The performance of a message-passing library simulated on a shared-memory computer is likely to differ substantially from the (far more complex) library on message-passing hardware. In addition, the two sets of programs are not comparable since the shared-memory codes were written naively. Also, the programs were executed on a real machine, which limited the comparison to elapsed time and speedup.

Martonosi and Gupta [16] compared, on a shared-memory machine, a variety of shared-memory and message-passing implementations of the Locus-Route standard cell routing program. They measured performance in terms of message traffic, execution time, and solution quality. The best message-passing implementation reduced message traffic significantly, which improved execution time, with a small degradation in solution quality. This work explored parallel programming techniques and ways by which programmers can manage parallel data efficiently.

LeBlanc and Markatos [13] studied load balancing for message-passing and shared-memory programs. They again simulated message passing on a shared-memory machine and compared a light-weight thread model (in shared memory) against a static partitioning model in message passing. These experiments did not compare the performance of these two types of machines or programs directly.

The second group of papers used simulation to compare machines. This approach opens the possibility of more detailed measurements, but requires two accurate, comparable simulators. Klaiber and Levy [11] compared message traffic in message-passing and shared-memory programs using a combination of direct execution and simulation. They compiled data-parallel programs, written in C*, with a compiler that invoked a run-time communication library. Their instrumented libraries produced a trace of off-processor references for message-passing and shared-memory simulators. Their work differs from this paper in three respects. First, their programs were not written, compiled, or optimized for any particular type of machine, so the measured behavior is dependent on arbitrary decisions by the compiler. Our programs were carefully tuned for the machines on which they ran and so are more typical of programs run on parallel computers. Second, their programs did not either access memory directly or send messages, but rather invoked a machine-independent library for communication. This library is potentially misleading, both because it misses significant overheads (and protocol-specific traffic) introduced by message-passing libraries and because it introduces unnecessary overheads in shared memory. Finally, their simulators reported only message traffic (both number and amount), not execution time. The relationship between traffic and elapsed time is unclear. Our measurements focused on execution time, although we also collected traffic information.

Kranz et al. [12] explored the use of the message passing integrated in the MIT Alewife shared-memory machine and discussed how messages improved the performance of several common operations, such as barriers, thread invocation, thread scheduling, and bulk data transfer. They measured several versions of a simple Jacobian SOR code on an Alewife simulator, which supported both shared memory and message passing. Their results agreed with our finding (Section 5) that message passing and shared memory can perform equally well.

Woo et al. [23] studied the implications of adding a message passing-like block transfer facility to shared memory. They added this feature to an architectural simulator of the shared-memory Stanford FLASH machine and modified five programs to use it. They found that block transfer was difficult to use effectively in these programs and provided only a small performance advantage in limited cases. This result accords with our findings that, in many situations, shared-memory communication is effective and has overhead as low as message-passing communication.

3 Simulation Methodology

We used the Wisconsin Wind Tunnel (WWT) to simulate the message-passing and shared-memory machines accurately and efficiently [19]. WWT is a system for virtual prototyping of parallel computers. It directly executes a parallel program written and compiled for a proposed parallel computer (the *target*). WWT accurately calculates the target program's execution with a distributed, discrete-event simulation running on a parallel *host* computer, in our case a CM-5.

WWT runs an instrumented SPARC binary on CM-5 processors. The Wind Tunnel system provides two key functions: the ability to invoke simulation code a miss in the target machine's cache and a mechanism for deterministically simulating interactions among nodes in a parallel computer. For more details, see the paper by Reinhardt et al. [19]. The shared-memory simulation uses the first feature to detect and model both local and shared address cache misses. The message-passing simulation uses it only for local misses. Both simulators use the discrete event simulation to ensure causality among processors and compute a program's execution time.

For this paper, we modified the shared-memory WWT simulator to simulate a message-passing machine. For the most part, this change involved disabling the shared address space, adding calls into WWT to simulate the memory-mapped locations in the CM-5 network interface, and using WWT's event mechanisms to simulate message transmission. Our message-passing simulator is similar to LAPSE [5], a direct execution simulator for message-passing machines that runs on the Intel Paragon. One difference between the two simulators is that LAPSE models network contention.

To provide a communication library for message-passing programs, we ported the Active Message [22] layer from Thinking Machines' CMMD library to the Wind Tunnel. The complete CMMD library runs as part of the target program, as it does on a CM-5. Since the active message code is heavily-optimized assembly code that violates the SPARC ABI convention in ways that conflict with WWT's violations of the SPARC ABI, we had to rewrite portions of this code to run on WWT. We also changed instructions that referenced memory locations in the network interface to invoke WWT routines that simulated the network interaction.

4 Simulation Details

This section describes the shared-memory and message-passing machines that we simulate in this

Table 1: Common hardware characteristics

Cache	256 KB, 4-way set assoc, random replacement
Block size	32 bytes
TLB	64 entries, fully associative, FIFO replacement
Page size	4 KB
Message latency	100 cycles remote
Barrier latency	100 cycles from last arrival
Private Cache miss[a]	11 cycles + replacement cost, if a block is replaced
DRAM access	10 cycles

[a]does not include DRAM access

study. Both machines are close to the hardware of a Thinking Machines CM-5 [9]. Our message-passing interface matches the CM-5 data network interface closely enough that we ran the full TMC message-passing libraries. The shared-memory system uses a full-map write-invalidate protocol. More importantly, both machines share common assumptions about the base hardware.

This base consists of workstation-like nodes connected by a point-to-point interconnection network. Table 1 summarizes hardware characteristics common to both machines. We assume a machine with a 30ns cycle time. Each node contains a SPARC processor, a large cache (256K bytes), and local DRAM. Nodes communicate by sending and receiving short messages. The message-passing machine's messages are limited to 20 bytes, as on the CM-5,[2] but the shared-memory machine uses 40 byte messages.[3] Programs on the message-passing machine directly access a memory-mapped network interface. Programs on the shared-memory machine cannot send messages directly. In this study, we assume constant (100 cycle) network latency and do not model network contention. We can simulate systems of 1–128 processors. In this paper, all experiments use 32 processors. Both machines provide a hardware barrier similar to the CM-5. Unlike the CM-5, our simulation does not support broadcast or reduction hardware in either machine. This change makes comparison with shared memory easier and enables us to study the cost of implementing these operations in software (as is necessary on other machines [10]).

4.1 Message Passing

The message-passing machine sends and receives messages through a memory-mapped network interface.

[2]TMC's CMMD library is implemented to use 20 byte packets.

[3]The 40 byte size corresponds to the cache block size plus some control information.

Table 2: Hardware in the message-passing machine

Replacement cost	1 cycle (inf. write buffer)
NI status word access	5 cycles
NI write tag + destination	5 cycles
NI send 5 words	15 cycles (including stores)
NI receive 5 words	15 cycles (including loads)

Our network interface models the CM-5 data network interface, except for the kernel interaction with interrupt handling (see below). The interface is accessible from user programs and contains incoming and outgoing FIFOs for receiving and sending messages up to 20 bytes long (along with a tag). The processor moves data in and out of this interface with explicit load and store instructions. There is no DMA-like message transfer. A status register in the interface indicates if an incoming packet is queued. The register also indicates if the previous 20-byte packet was successfully dispatched (in our simulation, a send always succeeds since we do not model network contention). Furthermore, the interface's interrupt mask controls if the processor will be interrupted when a message with a particular tag(s) enters the queue. Table 2 summarizes the hardware and overheads of various operations (in cycles).

Our simulated message-passing machine is different from a CM-5 in several respects. Its cache is larger than the actual machine (256KB, 4-way set associative vs. 64KB, direct-mapped) and its network is not modeled. Also, message interrupts on the real machine trap to the kernel, which invokes a user-level handler in a new register window. Our simulator directly invokes the handler, without simulating the kernel code or its effect on machine state. Fortunately, the CMMD library polls heavily, so the discrepancy should not affect our results. To gain confidence in our simulation, we compared its times against actual runs on a CM-5. As would be expected from the differences listed above, two programs LCP and EM3D ran 14–27% faster on the simulator than the real machine. A third program MSE ran 15% slower on the simulator, which probably reflects inaccuracy in accounting for dynamic floating point instruction times in this computationally-bound program.

The message-passing programming model is exactly the same as on the CM-5. A program can either use the low-level, 20-byte memory-mapped messages, or the vendor-provided Active Message and CMMD libraries. The CMMD library provides commonly-used synchronous and asynchronous message sends and receives.

The CMMD library maintains a collection of send and receive data structures on each node, called channels.[4] The channels are initialized with information on destination, number of bytes to be transmitted, starting address of the data to be transmitted, and the address at which to store the received data. A channel send routine breaks up the data to be send into packets and injects them into network; on the receiving side, each packet is pulled from the network by a data packet handler, which is invoked either by an interrupt service routine or by explicit polling. The handler reads the data from the network interface and stores it into the correct place in memory. Counters in these data structures keep track of the transmission in progress. High level send and receive functions initialize the appropriate send/receive data structures and handshake to exchange the receiver's channel number.

4.2 Shared Memory

The shared-memory machine uses the invalidation-based Dir_nNB cache-coherence protocol [1]. Each processor node's local memory locations have global addresses, so they can be referenced by other nodes. A node maintains a directory that manages coherence for shared cache blocks in its local memory. The data transfer and coherence unit is a cache block (32 bytes in our experiments). If a processor accesses a block that is not in its local cache, the processor's shared-memory hardware sends a request message to the block's home node. That node uses information in its directory to determine if other processors hold copies of the block, possibly invalidates these copies, and returns the block to the requesting node. Our implementation of the Dir_nNB protocol provides sequentially consistent memory (and ensures the fewest possible invalidation messages).

In addition to the base hardware of the message-passing machine, each processor node contains a directory and cache controllers. The directory controller maintains coherence for the node's cache blocks. The cache controller receives invalidation requests and data messages (cache blocks) from directories and sends messages containing write-back data, miss requests, and invalidation acknowledgements to directories. To avoid consistency problems, cache blocks in the shared data segment do not pass through a write buffer on write-back. Dirty cache blocks in the private data segment always go to an (infinite) write buffer. For synchronization, the machine provides the hardware barrier and an atomic swap instruction for locks. Table 3 summarizes the hardware and overheads of various operations (in cycles).

Table 3: Hardware in the shared-memory machine

Message to Self	10 cycles
Shared Cache miss	19 cycles + replacement cost, if block is replaced
Cache Invalidate	3 cycles + replacement cost, if block is replaced
Replacement Cost	1 cycle if block private, 5 cycles if shared but clean, 13 cycles if shared and dirty
Directory	10 cycles + 8 if cache block is received + 5 if message is sent + 8 if cache block is sent

Shared-memory programs use the `parmacs` macros. In this system, memory from the shared address space is allocated by the `gmalloc` routine, which uses round-robin allocation across processors. At the beginning, only node 0 executes. After preliminary initialization it invokes the `create(f)` routine, which duplicates its data segments and starts subroutine `f` on all other nodes. The `lock` and `unlock` operations use MCS locks [17].[5] The `barrier` function uses the hardware barrier.

5 Results

This section discusses our measurement of four programs. Although a small number of examples, the programs' behavior varies dramatically enough to demonstrate clearly many advantages and disadvantages of message passing and shared memory.

5.1 Microstructure Electrostatics

Our first benchmark is a program that computes boundary integral solutions of the Laplace equation arising from simulating microstructure electrostatics [21]. Both the shared-memory (MSE-SM) and message-passing (MSE-MP) versions were written and optimized by researchers in the University of Wisconsin Chemical Engineering department. The program solves an N body system, where each body is discretized into M boundary elements. Since the system matrix is of size $(NM)^2$, it cannot be stored completely in memory and is recomputed as needed. The program solves the equations with parallel asynchronous Jacobi iterations. It updates a global solution vector according to a predetermined sched-

[4] Programs that repeatedly transmit a fixed amount of data between nodes can use channels directly to improve performance. We use this optimization is some of the programs (see Section 5.)

[5] MCS locks avoid excessive network traversals by having each processor spin on a separate, locally cached shared memory location. The relinquisher of the lock passes it on to the next acquirer by terminating its spin through a single remote write.

ule. This benchmark is an example of a highly-tuned, computation-bound application used in computational science research.

The following table breaks down time for the message passing program running with 256 bodies, 20 boundary elements per body, and 20 iterations. The cycle times reported represent an average over all processors.

4: Microstructure Electrostatics Message Passing (MSE-MP)		
Category	Cycles (M)	%
Computation (incl. Start-up)	1115.9	90%
Local Misses	53.0	4%
Communication	72.2	6%
Lib Comp	69.9	6%
Lib Misses	0.2	0%
Network Access	2.1	0%
Total	1241.1	100%
Relative to Shared Memory		98%

For each category, we present a cycle count in millions of cycles (M) and a percentage with respect to the total (for example, computation takes 1115.9M cycles in MSE-MP, which is 90% of the total cost of 1241.1M cycles). MSE-MP, like other programs, spends its time on computation; local cache misses; and communication, which can be further divided into time spent in communication library routines (*Lib Comp*), servicing cache misses to local data from within these library routines (*Lib Miss*), and accessing the network. The categories for shared memory are slightly different. MSE-SM spends its time on computation; misses to shared data; and synchronization, which can be further divided into time spent at barriers and start-up.

5: Microstructure Electrostatics Shared Memory (MSE-SM)		
Category	Cycles (M)	%
Computation	1043.8	82%
Cache Misses	62.7	5%
Synchronization	160.2	13%
Barrier	80.2	6%
Start-up Wait	80.0	6%
Total	1267.8	100%
Relative to Message Passing		102%

Our second set of tables present *per-processor* counts for a variety of events.

6: Microstructure Electrostatics Message Passing (MSE-MP)	
Local Misses	2.4M
Messages sent	1271
Bytes Transmitted	1.1M
Data	0.8M
Control	0.3M
Computation Cycles Per Data Byte Transmitted	1452

For message-passing programs, we report number of local cache misses, message sent, and bytes transmitted. Counts, except for message counts, are millions of misses or bytes transmitted. Bytes transmitted is broken down into data bytes and control bytes. This breakdown is not completely accurate as data bytes are occasionally sent in control packets. For shared-memory programs, we report number of cache misses, write faults, and bytes transmitted. Cache misses are categorized as misses to private or shared data. Shared misses are separated into local and remote misses. A write-fault occurs when a processor attempts to write to a read-only cache block. The number of bytes transmitted is broken down into data bytes and control bytes, which arise from requests for data, invalidations, and acknowledgements. The tables also include an estimate of the computation per byte transmitted, which is a crude measure of a program's communication intensity.

7: Microstructure Electrostatics Shared Memory (MSE-SM)	
Cache Misses	
Private Misses	2.5M
Shared Misses	0.04M
Local	0.01M
Remote	0.03M
Write Faults	774
Bytes Transmitted	2.4M
Data	1.0M
Control	1.4M
Computation Cycles Per Data Byte Transmitted	985

MSE was designed carefully at both algorithm and implementation level, so computation dominates both implementations (90% for MSE-MP and 82% for MSE-SM). Elapsed computation times differ slightly, because MSE-SM performs a small amount of initialization, which is part of a larger initialization phase, on Processor 0, while other processors sit idle (80M cycles). In MSE-MP, every processor participates in the initialization.

This program manages communication carefully. It controls updates to the solution vector with schedules that rely on an insight about the physical structure of the problem to decrease communication. Since distant bodies interact less strongly, they need to exchange solutions less frequently. This drastically reduces communication and only slightly increases the iterations until convergence.

Once past initialization, communication in this program passes through the solution vector, which records body positions. In MSE-SM, this vector resides in the shared address space and processors update it according to their schedule. In MSE-MP, each processor keeps a local copy of the solution vector. When a processor's schedule calls for updates, it sends an asynchronous request for current values and awaits replies. Each processor services these requests asynchronously.

In MSE-SM, 0.04M of the 2.5M cache misses (per processor) are to shared data, which, in part, communicate data between processors. Although shared misses incur a sharply higher cost than local misses, MSE-SM's total cache miss cost is only 9.7M cycles higher than MSE-MP's local cache miss cost because shared misses are a small fraction of total misses. Synchronization in MSE-SM consists of a single barrier between the initialization and main computation loop. This barrier costs 80M cycles, because of a load imbalance among the processors. In MSE-MP, the communication cost as well as the waiting time due to load imbalance manifests itself as library computation time (69.9M cycles).

5.2 Gaussian Elimination

The *Gauss* program solves a linear system of equations using Gaussian elimination. We adapted the message-passing code (Gauss-MP) from an iPSC version from Syracuse University and wrote the shared-memory code (Gauss-SM). The algorithm is a straightforward forward elimination phase followed by a backward substitution phase. The program divides work among processors by distributing rows of the coefficient matrix blockwise. Each processor fills its rows with random numbers and solves the equations using a known vector. In the forward phase for a column, processors select a pivot by computing the maximum from among each processor's local maxima. After selecting a pivot, a processor subtracts a factor of the pivot row from its rows. In the backward substitution phase, processors start from the last row and, as a variable's value becomes known, subtract the appropriate factor from their rows. The programs do not explicitly redistribute the rows as they are pivoted to avoid load imbalances. Instead, each processor maintains a local mask array that tracks the global position of each row.

Work in Gaussian elimination falls into two broad categories of computation and data management. The following pair of tables break down costs for the two implementations running with 512 variables.

8: Gauss Message Passing (Gauss-MP)

Category	Cycles (M)	%
Computation	40.8	58%
Local Misses	0.04	0%
Broadcast/Reduction	30.1	42%
Lib Comp	23.6	33%
Lib Misses	0.03	0%
Barriers	1.7	2%
Network Access	4.7	7%
Total	71.0	100%
Relative to Shared Memory		98%

Again, running times of the two programs are nearly identical. Since the two programs use the same algorithm, their computation costs are very similar. Gauss-MP performs slightly more computation, to manage buffers for communication. The programs differ primarily in the organization and cost of communication.

9: Gauss Shared Memory (Gauss-SM)

Category	Cycles (M)	%
Computation	39.5	54%
Cache Misses	17.1	23%
Synchronization	16.1	22%
Reductions	4.5	6%
Barriers	11.6	16%
Totals	72.7	100%
Relative to Message Passing		102%

Communication in this program is either one-to-many or many-to-one. First, in selecting a pivot, processors perform a reduction to choose a maximum pivot element. Next, the pivot element is sent to all processors, so the processor that owns the pivot row can identify itself. Third, this processor sends the pivot row all processors. Finally, in the backward substitution phase, as the value for each unknown becomes available, it is sent to each processor. The first communication is a reduction and the others are broadcasts. Most of the effort in optimizing this program was obtaining efficient implementations of these primitives.

Gauss-MP uses carefully tuned reduction and broadcast routines. Both routines are based on a lop-sided tree, whose superior performance was suggested by the LogP model [4] under the assumption that message send and receive overhead are higher than network latency. We experimented with several approaches before settling on lop-sided trees. Our initial attempt used a binary reduction tree and a flat broadcast, in which the initiator of a broadcast sent a message to every other processor. This was very slow (119.3M cycles for both the broadcasts and reduction). Next we tried a binary tree-based broadcast, but still used CMMD-level messages to transmit data. This was also slow (40.9M cycles). Although our final lop-sided trees use active messages and channels to improve performance further (30.1M cycles), their time is still not comparable to hardware. The difficulty in all three implementations is the high latency of sending and receiving a message. A node several levels down in a tree (or late in a flat broadcast) waits a long time. The lop-sided tree is most efficient, because its structure minimizes the effect of this latency. Active messages also help reduce this latency.

The following table presents per-processor event counts for Gauss-MP. Bytes transmitted is reported in millions. The remaining counts, for local misses and number of messages sent, are quite small.

10: Gauss Message Passing (Gauss-MP)	
Local Misses	3,489
Message Counts	
Channel Writes	511
Active Messages	1534
Bytes Transmitted	0.7M
Data	0.5M
Control	0.2M
Computation Cycles Per Data Byte Transmitted	78

The low number of computation cycles per data bytes transmitted shows that Gauss-MP is communication intensive.

Gauss-SM reductions and broadcasts exploit shared-memory's fine-grain, low-latency communication. Reductions use the same approach as the upward phase of MCS barriers [17] and account for only 6% of total time. Gauss-SM broadcasts a value by letting all processors read it. Since all processors wait at a barrier until the write completes (11.3M cycles), these invalidates are on the program's critical path. However, they occur at hardware, not software speed. After the barrier, processors read the shared data (16.7M cycles). Read requests encounter contention in the network and directory. In these simulations, we do not model network contention. We do, however, model directory contention. The per-processor event counts in the following table give us some insight into the problem.

11: Gauss Shared Memory (Gauss-SM)	
Cache Misses	
Private Misses	92
Shared Misses	23,590
Local	781
Remote	22,809
Write Faults	946
Bytes Transmitted	1.8M
Data	0.8M
Control	1.0M
Computation Cycles Per Data Byte Transmitted	47

Although the number of cache misses is low, the program spends 23% of its time on cache misses. The average cost of a miss to shared data is 700 cycles, which is roughly 450 cycles higher than the cost of servicing a miss to idle data in the absence of contention. The average queuing delay at a directory is 200 cycles. These delays, which are lower than the software latencies in the message passing code, will become untenable for larger systems.

5.3 EM3D

EM3D models propagation of electromagnetic waves through objects in three dimensions [3]. The problem is framed as a computation on a bipartite graph with directed edges from E nodes, which represent electric fields, to H nodes, which represent magnetic fields, and vice versa. At each step in the computation, new E values are first computed from the weighted sum of neighboring H nodes, and then new H values are computed from the weighted sum of neighboring E nodes. Edges and their weights are determined statically.

The initialization phase of EM3D builds the graph and does some precomputation to improve the performance of the main loop. To build the graph, each processor allocates a set of E nodes and a set of H nodes. Edges are randomly generated using a user-specified percentage that determines how many edges point to remote graph nodes. We refer to edges with a source on one processor and a sink on another processor as *remote* edges.

Work in EM3D can be classified into three broad categories: computation, data access, and synchronization. Tables 12 and 14 break down costs for initialization, main loop, and the whole program. We ran 50 iterations with 1000 E nodes and 1000 H nodes per processor. All nodes have an outdegree of 10 and 20% of the edges are remote. A production run would perform more iterations and so the main loop would dominate the computation.

The major result in these tables is that EM3D-MP is substantially faster (86.4M vs. 172.1M cycles), although its computation is more expensive (50.5M vs. 43.7M cycles). To understand why communication is much more costly in EM3D-SM (109.8M vs. 36.0M cycles), we need to examine the programs in detail. The next section discusses the program's data structures. Then, we examine the performance of the initialization phase and main loop.

5.3.1 Data Structures

Before analyzing performance results, we need to explain the program's data structures. The Split-C version [3] heavily influenced our implementation of EM3D-MP; both use ghost graph nodes to shadow remote source nodes. A ghost node holds a local copy of a remote node's value. EM3D-MP differs slightly from the Split-C code in this respect. Instead of maintaining a single ghost node for each remote node, EM3D-MP uses one ghost node for each remote edge, which simplifies initialization substantially and only slightly increases the data transferred. Before each half-step, a processor sends a message to the other processors that it is connected to by graph edges. The message contains values that enable the remote processors to update their ghost nodes to the current values of source nodes. Remote values are batched and transmitted in a single bulk message. Ghost nodes make remote and local data accesses uniform and remove all communication from the main loop.

12: EM3D Message Passing (EM3D-MP)						
	Initialization		Main Loop		Total	
Category	Cycles (10^6)	%	Cycles (10^6)	%	Cycles (10^6)	%
Computation	18.2	91%	32.3	49%	50.5	58%
Local Misses	1.3	6%	13.7	21%	15.0	17%
Communication	0.5	3%	20.5	31%	21.0	24%
Lib Comp.	0.4	2%	16.4	25%	16.8	19%
Lib Misses			0.3	0%	0.3	0%
Network Access	0.1	1%	3.8	6%	3.9	5%
Total	20.0	100%	66.5	100%	86.4	100%
Relative to Shared Memory						50%

13: EM3D Message Passing (Main Loop only)	
Local Misses	643,436
Message Counts	
Channel Writes	200
Bytes Transmitted	2.0M
Data	1.6M
Control	0.4M
Computation Cycles Per Data Byte Transmitted	20

14: EM3D Shared Memory (EM3D-SM)						
	Initialization		Main Loop		Total	
Category	Cycles (10^6)	%	Cycles (10^6)	%	Cycles (10^6)	%
Computation	17.2	41%	26.5	20%	43.7	25%
Data Access	15.7	37%	94.1	72%	109.8	64%
Shared Misses	13.4	32%	83.6	64%	97.0	56%
Write Faults	1.8	4%	10.4	8%	12.2	7%
TLB Misses	0.6	1%	0.1	0%	0.7	0%
Synchronization	9.0	22%	9.4	7%	18.4	11%
Sync Comp	1.2	3%			1.2	1%
Locks	6.9	16%			6.9	4%
Barriers	0.9	2%	9.4	7%	10.3	6%
Total	42.1	100%	130.0	100%	172.1	100%
Relative to Message Passing						200%

15: EM3D Shared Memory (Main Loop only)	
Cache Misses	
Private Misses	109
Shared Misses	330,044
Local	10,818
Remote	319,226
Write Faults	24,975
Bytes Transmitted	22.9M
Data	11.9M
Control	11.0M
Computation Cycles Per Data Byte Transmitted	2

EM3D-SM does not use ghost nodes because shared-memory caching exploits temporal locality by making a copy of a remote node in a processor's cache. Note that this caching comes at a high cost because each update requires four messages (two to invalidate, one to request a value, and one to send it) [6, 20]. The main loop references only the value field from remote nodes. The rest of a node's data is unnecessary. EM3D-SM improves spatial locality by allocating nodes' value fields in a separate vector.

5.3.2 Initialization

The EM3D programs demonstrate some of the complexities and overheads in building a complex, pointer-based data structure in a parallel machine. Initialization in EM3D-MP runs almost twice as fast as EM3D-SM (21.6m vs. 41.2m) and is computationally bound (84% of its time). EM3D-SM runs slower because of sharply higher data access costs and additional synchronization costs. In fact, the initialization phase of EM3D-MP spends 5% more time on computation, because of the cost of setting up calls to communication routines.

The largest difference between the programs, however, are data access costs. In EM3D-MP initialization, most access costs are due to local cache misses (1.3M cycles), which occur because data does not fit in the cache (573,212 bytes, of which, 81,040 bytes are ghost-node related). The remainder is due to sending local values to remote processors (0.5M cycles). In EM3D-SM initialization, data access costs are much higher, primarily due to the cost of shared cache misses (13.4M cycles), but also due to write faults (1.8M cycle) and TLB misses (0.6M cycles).

In both implementations, processors transmit information about remote edges from their source nodes to their sink nodes to build a reverse edge graph. This edge information is referenced twice in the initialization code, once to compute the in-degree of each node and then to record pointers from sinks to sources (these are the pointers that are modified to point to ghost nodes in EM3D-MP).

In EM3D-MP, this information is transmitted between each pair of processors in a single bulk message and stored in a processor's local memory, where it can be reused without further communication. By contrast, in EM3D-SM, remote data accesses require locks and remote writes because each processor updates incoming edge counts and pointers for remote sinks. These remote writes likely incur a cache miss every time because another processor reads or writes, and hence invalidates, a cache block before it can be reused.

EM3D-SM uses synchronization, the final category, explicitly and it represents 22% of overall time. EM3D-SM uses both locks and barriers. Locks, which protect updates to remote nodes, account for the largest portion of synchronization costs (6.9M cycles) and have no analogue in EM3D-MP, since its writes are local. In EM3D-SM, the few barriers that prevent premature access to shared data also have no direct analogue in EM3D-MP.

Much of the shared-memory penalty is due to the way that the E-H graph is constructed. We faithfully followed the Split-C approach, which is much better suited to message passing. In a real application, this graph would be read in (not randomly generated) and so could be stored in a more appropriate form.

5.3.3 Main Loop

Like the initialization, EM3D-MP's main loop spends more time computing than EM3D-SM (32.3M vs. 26.5M cycles), because of the cost of managing calls to communication routines (measured at 5.4M cycles). Again, EM3D-SM's sharply higher data access costs far outweigh this small communication cost. To understand why data access is more expensive in EM3D-SM, we need to examine how the two programs manage data.

Updating ghost nodes at each half-step is the only communication in the main loop of EM3D-MP. To update another processor's ghost nodes, a processor collects its values into a buffer and sends them, in bulk, over a virtual channel. This transfer is very efficient for three reasons. First, the sender initiates the transaction, which eliminates a request message. Second, data is transferred in bulk, which significantly reduces the overhead per byte transmitted. And third, communication is static, which allows us to use virtual channels to reduce communication handshaking.

By contrast, EM3D-SM uses a standard shared-memory (request-response) protocol, which performs very poorly for this producer-consumer communication, because it requires four messages to update a remote value. Consider an H node, h on Processor P that has an edge to an E node on Processor Q. In the first iteration, Processor Q incurs a remote miss to retrieve the initial value of h (two messages). When Processor P is ready to compute a new value for h, it invalidates the copy of h in Processor Q's cache (two messages), which causes Processor Q to miss on its reference to h at the beginning of the next iteration.

Data access costs for EM3D-SM are very high in comparison with the costs in EM3D-MP. Event counts in Tables 13 and 15 point out two factors that account for this difference. First, the absolute number of misses is high. And second, many misses are expensive remote misses, although a processor mainly references data that it allocated. One problem is that the per-processor working sets are too large for the 256KB caches that we simulate. The following table breaks down costs in the main loop of EM3D-SM running with a 1MB cache.

16: EM3D-SM 1MB Cache Main Loop

Category	Cycles (M)	%
Computation	26.5	43%
Data Access	33.1	54%
Shared Misses	22.1	36%
Write Faults	10.9	18%
TLB Misses	0.1	1%
Synchronization	1.5	2%
Barriers	1.5	2%
Total	61.0	100%

Notice that the total cost drops below message passing and the number of misses and data access cost drop to a third of their values with a 256KB cache. Not surprisingly, the computation cycles per data byte transmitted metric increases to 7. The large number of remote misses can be traced to our round-robin memory allocation policy. The following table breaks down costs in the main loop for EM3D-SM using a local allocation policy (with a 256KB cache).

17: EM3D-SM Local Allocation Main Loop

Category	Cycles (M)	%
Computation	26.5	31%
Data Access	58.9	68%
Shared Misses	52.3	61%
Write Faults	6.5	8%
TLB Misses	0.1	0%
Synchronization	0.9	1%
Barriers	0.9	1%
Total	86.3	100%

The local allocation version runs in two thirds the time of the round robin version. Remote misses to shared data drop from 97% of misses to 10% of misses and the computation cycles per data byte transmitted metric improves to 16.

As in the initialization phase, only EM3D-SM requires synchronization, which represents 11% of its running time. The main loop of EM3D-SM uses barriers to separate half-steps and prevent a process from accessing a remote value before it is computed.

5.3.4 Discussion

EM3D-SM's performance could be improved by a variety of techniques. In the example above, Processor Q could flush its copy of a remote H node from its cache, thereby changing a 2-message invalidate into a single-message cache replacement operation.[6] In addition, Processor Q could hide some latency by prefetching. The cooperative prefetch operation in CSM [8] works well in this situation, because a consumer need not worry about issuing a prefetch too

[6] As the data set outgrows a cache, the likelihood that a line will still be in the cache when the remote write occurs diminishes, which reduces the value of flushes.

early. In addition, the Stache policy [20] would reduce the cost of the cache replacements by storing data in local memory, rather than returning it to its home node.

A better approach is to replace the invalidation-based cache-coherence protocol with a bulk update protocol, which requires only a single message to transmit new values from Processor P to Processor Q. Falsafi et al. replaced an invalidation-based protocol with a bulk update protocol in a shared-memory version of EM3D [6]. The resulting program performed equivalently with EM3D-MP. Similar protocol changes could benefit other programs, most notably the broadcasts in Gauss.

5.4 Linear Complementarity Problem

Our final application solves the linear complementarity problem (LCP) using a multi-sweep synchronous successive over-relaxation algorithm [14]. The linear complementarity problem finds a vector z that satisfies the equations $Mz + q \geq 0$, $z \geq 0$ and $z(Mz + q) = 0$. In our case, M is a symmetric sparse matrix, q is a (dense) vector, and the problem has 4096 variables. We wrote the shared-memory version (LCP-SM) and derived a message-passing version (LCP-MP) from it.

The algorithm statically divides the matrix into equal sized blocks of rows, which are distributed to processors.[7] At each step, processors perform a specified number (5) of Gauss-Seidel sweeps on their local rows using a *local* copy of the solution vector. Each sweep updates a portion of the local vector. At the end of a step, processors update the global solution vector and test for convergence. The two implementations handle the solution vector differently. LCP-SM uses a single global solution vector, but each processor in LCP-MP keeps its own copy.

Most communication in this algorithm arises from updates to the global solution vector. In LCP-MP, updating the local copies requires all-to-all communication. We do this with log(NPROC) steps of point-to-point exchanges across CMMD channels. In LCP-SM, processors compute their portion of the new solution vector into a local buffer. To update, they copy values from the local buffer into the global vector.

A reduction in the convergence test requires additional communication. LCP-MP's reductions use the active message trees discussed earlier. LCP-SM uses MCS style reductions.

The following tables breaks down costs for LCP-MP and LCP-SM.

[7]This does not guarantee load balancing, but the sparse matrices that we used had uniform non-zero elements per row.

18: LCP Message Passing (LCP-MP)

Category	Cycles (M)	%
Computation	41.1	73%
Local Misses	0.06	0%
Communication	15.6	27%
Lib Comp	12.6	22%
Lib Misses	0.02	0%
Network Access	2.7	5%
Barrier	0.3	0%
Total	56.8	100%
Relative to Shared Memory		86%

19: LCP Shared Memory (LCP-SM)

Category	Cycles (M)	%
Computation	41.3	63%
Cache Misses	13.4	20%
Synchronization	11.3	17%
Sync Comp	3.2	5%
Sync Miss	0.1	0%
Barrier	8.0	12%
Total	66.0	100%
Relative to Message Passing		116%

The tables show that, although both the message-passing and shared-memory versions spend the same amount of time computing, shared-memory communication is costlier. LCP-MP spends 15.6M cycles communicating, but LCP-SM spends 24.7M cycles, of which cache misses cost 13.4M cycles and synchronization costs 11.3M cycles. These cache misses, which arise mainly from references to the solution vector, are costly because the invalidation-based coherence protocol is ill-suited to the producer-consumer communication needed for the global solution vector.

De Leone et al. showed that faster convergence results if updates to the global solution vector become available to other processors as soon as they are computed [14]. This approach, however, increases the amount of communication. We implemented this alternative as well. ALCP-SM writes new values directly to the global solution vector. Processors synchronize every five iterations to test for convergence. The message-passing version (ALCP-MP) sends bulk updates asynchronously (in a star communication) after each Gauss-Seidel sweep.[8]

20:Asynchronous LCP Message Passing (ALCP-MP)

Category	Cycles (M)	%
Computation	32.9	35%
Local Misses	0.09	0%
Communication	59.8	64%
Lib Comp	46.5	50%
Lib Misses		
Network Access	12.9	13%
Barrier	0.3	0%
Total	92.7	100%
Relative to Shared Memory		94%

[8]We explored active message puts that update the global vector as each value was computed, but this proved prohibitively expensive.

21: Asynchronous LCP Shared Memory (ALCP-SM)		
Category	Cycles (M)	%
Computation	32.0	32%
Cache Misses	62.9	64%
Synchronization	3.8	4%
Sync Comp	1.6	2%
Sync Miss	0.1	0%
Barrier	2.2	2%
Total	98.7	100%
Relative to Message Passing		106%

Both asynchronous versions required fewer time steps (from 43 down to 34 in ALCP-SM, 35 in ALCP-MP), which reduced computation costs by about 23% (from 41.1M to 32.9M cycles for message passing). Increased communication (from 15.6M to 59.8M cycles for message passing), however, swamped this gain, and the programs run slower overall. This example illustrates the tradeoff between computation and communication. From the event count tables below, we can see that the computation per data byte transmitted dropped drastically from 29 to 6 in message passing and from 26 to 4 in shared memory. Our results differ from those of De Leone et al., because they ran their experiments on a Sequent Symmetry, which has much lower costs for remote cache misses and coherence traffic.

22: LCP Message Passing		
	Synchronous	Asynchronous
Local Misses	3,873	4,345
Message Counts		
Channel writes	220	5,425
Active messages	90	74
Bytes Transmitted	1.8M	6.9M
Data	1.4M	5.6M
Control	0.4M	1.4M
Computation Cycles Per Data Byte Transmitted	29	6

23: LCP Shared Memory		
	Synchronous	Asynchronous
Cache Misseses		
Private Misses	56	60
Shared Misses	48,411	206,615
Local	1,528	6,140
Remote	46,883	200,475
Write Faults	1481	15,814
Bytes Transmitted	3.7M	17.0M
Data	1.6M	7.4 M
Control	2.1M	9.6M
Computation Cycles Per Data Byte Transmitted	26	4

6 Conclusion

An important contribution of this paper is the identification of where message-passing and shared-memory programs spend their time. Despite vast differences in their communication mechanisms, both types of program spent roughly the same amount of time computing. The differences in execution time arose from communication. Moreover, the behavior of our four pairs of programs varied greatly, which strongly suggests that parallel computers should provide a range of communication mechanisms. The most unexpected result was that shared memory performed very well in three of the four programs.

Although we simulated a CM-5-like machine, which has an efficient user-level network interface, message-passing programs paid a high cost in moving data in and out of the network. This cost appeared both in increased computation time to manage buffers and the 3–42% of program time spent in communication library routines. Many alternatives—such as faster hardware [2], faster libraries, and protocol compilers [7]—could reduce this overhead.

Software overhead in processing low-latency (fast-turnaround) messages is a major weakness of the CM-5 message-passing system. These messages are fundamental to performing reductions or broadcasts in software (i.e., in Gauss). Hardware implementations of these operations (e.g., as on the CM-5) run much faster, but are not always appropriate, which necessitates software implementations.

Our measurements also confirm the widely believed advantage of message passing in transferring a large volume of data between a producer and consumer. EM3D-MP sends a couple hundred messages to transfer the data that requires several hundred thousand cache misses and many times that many protocol messages. Mechanisms for bulk data transfer and more efficient protocols have been proposed [23, 20].

We identified two major sources of overhead in shared-memory programs. First is the cost of moving large quantities of data with a request-response shared-memory protocol and of updating these values with an invalidation-based protocol.[9] Second is the cost of synchronization, which, in many cases, has no analogue in a message-passing program. In programs such as MSE, in which computation dominates, these inefficiencies have little effect on the program's running time. In other programs, such as EM3D, these inefficiencies have a major impact on program performance.

Shared memory, however, also offers fine-grain, low-latency access to remote data. Programs that exploit this feature, such as Gauss, can perform better than equivalent message-passing code, which interposes a level of software to respond to data requests.

Another important contribution of this research is development and demonstration of the tools to study

[9] Kranz et al. [12] show that with prefetching this cost can be decreased.

the tradeoffs between message passing and shared memory. By using a pair of closely-related simulators, we were able to control many of the independent variables (such as processor architecture, network structure, etc.) that affected previous comparisons. This technique has a wide range of applications beyond the direct comparison in this paper.

Acknowledgements

This work was performed as part of the Wisconsin Wind Tunnel project, which is co-lead by Profs. Mark Hill, James Larus, and David Wood and funded by the National Science Foundation. We are indebted to the members of the Wind Tunnel project for help with intricacies of the WWT system and for reading early versions of this paper. Steve Dirkse, Frank Traenkle, and Sangtae Kim wrote some of the applications. Babak Falsafi, John Reppy, Steve Reinhardt, P. Subbarao, and T.N. Vijaykumar offered many helpful comments. Finally, the reviewers provided many insightful comments.

References

[1] 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.

[2] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[3] 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.

[4] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Toward a Realistic Model of Parallel Computation. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–12, May 1993.

[5] Phillip M. Dickens, Philip Heidelberger, and David M. Nicol. A Distributed Memory LAPSE: Parallel Simulation of Message-Passing Programs. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, pages 32–38, July 1994.

[6] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing 94*, November 1994. To appear.

[7] Edward W. Felten. Protocol Compilation: High-Performance Communication for Parallel Programs. Technical Report 93-09-09, Department of Computer Science, University of Washington, September 1993.

[8] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in ASPLOS V, Oct. 1992.

[9] W. Daniel Hillis and Lewis W. Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993.

[10] Intel Corporation. Paragon Technical Summary. Intel Supercomputer Systems Division, 1993.

[11] Alexander C. Klaiber and Henry M. Levy. A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 94–105, April 1994.

[12] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

[13] Thomas J. LeBlanc and Evangelos P. Markatos. Shared Memory Vs. Message Passing in Shared-Memory Multiprocessors. In *Fourth IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, December 1992.

[14] R. De Leone, O.L. Mangasarian, and T.-H. Shiau. Multi-Sweep Asynchronous Parallel Successive Overrelaxation for the Nonsymmetric Linear Complementarity Problem. *Annals of Operations Research*, 22:43–54, 1990.

[15] Calvin Lin and Lawrence Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing (Vol. II Software)*, pages II–163–170, August 1990.

[16] Margaret Martonosi and Anoop Gupta. Tradeoffs in Message Passing and Shared Memory Implementations of a Standard Cell Router. In *Proceedings of the 1989 International Conference on Parallel Processing (Vol. III Algorithms and Applications)*, pages III88–96, August 1989.

[17] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[18] Ton A. Ngo and Lawrence Snyder. On the Influence of Programming Models on Shared Memory Computer Performance. In *Scalable High Performance Computing Conference (SHPCC '92)*, April 1992.

[19] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[20] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[21] F. Traenkle. Parallel Programming Models and Boundary Integral Equation Methods for Microstructure Electrostatics. Master's thesis, University of Wisconsin–Madison, 1993.

[22] 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.

[23] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors. Technical Report CSL-TR-93-593, Department of Computer Science, Stanford University, November 1993. To appear in ASPLOS VI.