Appears in *Proceedings of the 4th Int'l Symposium on High Performance Computer Architecture*, Las Vegas, NV, Feb 1-4, 1998.

The Sensitivity of Communication Mechanisms to Bandwidth and Latency

Frederic T. Chong[†], Rajeev Barua[⋆], Fredrik Dahlgren[‡], John D. Kubiatowicz[◇], and Anant Agarwal[⋆]
[⋆]Massachusetts Institute of Technology
[†]University of California at Davis
[‡]Chalmers University of Technology
[◇]University of California at Berkeley

Abstract

The goal of this paper is to gain insight into the relative performance of communication mechanisms as bisection bandwidth and network latency vary. We compare shared memory with and without prefetching, message passing with interrupts and with polling, and bulk transfer via DMA. We present two sets of experiments involving four irregular applications on the MIT Alewife multiprocessor. First, we introduce I/O cross-traffic to vary bisection bandwidth. Second, we change processor clock speeds to vary relative network latency.

We establish a framework from which to understand a range of results. On Alewife, shared memory provides good performance, even on producer-consumer applications with little data-reuse. On machines with lower bisection bandwidth and higher network latency, however, message-passing mechanisms become important. In particular, the high communication volume of shared memory threatens to become difficult to support on future machines without expensive, high-dimensional networks. Furthermore, the round-trip nature of shared memory may not be able to tolerate the latencies of future networks.

1 Introduction

Shared memory and message passing offer two different mechanisms for communication on distributed memory multiprocessors. Each mechanism has further variants such as shared memory with prefetching, message passing using polling, interrupts, and bulk data transfer. Because the effectiveness of communication mechanisms and their use by parallel applications has a first-order effect on the performance of parallel applications, much research in the past decade has focused on their implementation and evaluation. We now understand to a much greater extent than before the implementation tradeoffs. Many research and commercial machines also sport various combinations of mechanisms. For example, machines such as the BBN Butterfly have long supported shared memory and bulk transfer, the Cray T3E [27] supports both shared memory and messaging styles of communication, the Stanford Dash [18] supports shared memory and prefetching, MIT Alewife [1], Fugu [20], and the Wisconsin Typhoon [25] support several variants of shared memory and messaging styles.

The availability of machines with multiple mechanisms has led to an increasing amount of insight on the effectiveness of the various mechanisms for different applications [6] [11] [28] [30] [15] [8]. Message passing mechanisms, usually in the form of user-level active messages and efficient bulk-transfer of data, offer good performance on programs with known communication patterns since data can be communicated when produced rather than when requested by the program, thereby allowing the program to hide communication latency with useful computation. Messaging also allows combining synchronization with data transfer, and provides support for fast message-passing based synchronization libraries. However, it suffers from higher overhead for fine-grained data transfers. Bulk transfers, in turn, often requires expensive copying to and from buffers when transfer data is not consecutive. Cache-coherent shared memory provides efficient fine-grained (cache-line sized) data transfer and re-use of remote data using automatic caching, but suffers significant overhead when shared data is frequently modified on different processors, rendering caching ineffective, and causing a large amount of cache-coherency related invalidations and updates.

Most of the available insight on the relative performance of communication mechanisms, however, is specific to a given machine or a given application. Furthermore, because of the difficulty of porting applications and building machines, simulators, or emulators with many mechanisms, relative comparisons are often available only for a subset of the mechanisms. Because the relative effectiveness of communication mechanisms is also tied to basic machine parameters such as available bandwidth and latency, not surprisingly, various studies offer differing conclusions on the relative effectiveness of the mechanisms on the same application. For example, the simulation study of Chandra, Rogers, and Larus [6] using a basic machine model similar to the CM5 found that message passing EM3D performed roughly a factor of two better than the shared memory version. The two mechanisms were more or less indistinguishable on Alewife for the same application. Consequently, more general insights on the relative merits of the mechanisms have been elusive.

1.1 Objective

The goal of this paper is to provide insight into the relative effectiveness of various communication mechanisms for several applications over a modest range of communication latencies and bandwidth. The results are obtained with real applications on a real machine, whose parameters are varied through carefully designed scaling experiments. In a sense, we are using the machine as an emulator for other hypothetical machines. In particular, we analyze the impact of communication mechanisms on performance by evaluating four programs that are written using a variety of programming techniques on the MIT Alewife machine with 32 processors. The programming styles include shared memory with and without prefetching, message passing with interrupts, with polling, and with bulk transfer via DMA. To help explain the relative performance of various mechanisms, we also present the breakdowns of the relevant constituent components

This work was done while the authors were at MIT and was funded in part by NSF grant # MIP-9504399, in part by ARPA contract # N00014-94-1-0985, and in part by a NSF Presidential Young Investigator Award to Anant Agarwal. Contact: **chong@cs.ucdavis.edu**.

for each communication mechanism.

Our sensitivity experiments attempt to capture the communication performance of applications on machines with different design points, such as machines with different processor speeds, network latencies, and network bandwidth. We change processor clock speeds keeping the network latency constant to understand the effect of network latency. We also use context-switching and delay loops to study the relative effect of varying communication latency even further. We emulate machines with different bisection bandwidth by introducing cross-traffic from IO nodes to vary bisection bandwidth. This experiment also provides insights on how applications behave in multiprogrammed systems with background traffic from other applications.

Although several of these mechanisms have been studied in isolation, such as a comparison of shared memory and message passing barriers in terms of speeds of the barriers themselves, and a simulation-based study on the impact of integrating bulk transfer in cache-coherent multiprocessors (see Section 6 for details), this paper is the first to study real, hardware-based, efficient implementations of all these communication mechanisms on real applications in exactly the same framework. The sensitivity study in this paper also helps relate previous results presented by other researchers for specific design points, and also provides insights into the relative performance of communication mechanisms for other machine design points.

1.2 Preview

Our results confirm that the relative performance of several parallel communication mechanisms can be highly dependent upon machine parameters. For example, we find that shared memory performance is sensitive to the ratio of network bisection bandwidth and processor speed, while message passing performance is largely insensitive. This result is important because this sensitivity occurs in the range of ratios of network bisection and processor speeds exhibited by several extant research and commercial machines. For EM3D, this sensitivity results in the performance of shared memory varying by about 30 percent; for high ratios both shared memory and message passing have roughly equal performance, while that of shared memory degrades by 30 percent when the ratio is reduced by 60 percent.

In general, we find that shared memory offers better or comparable performance to message passing for machine parameters in the range of contemporary machines. The performance of message passing, on the other hand, is more robust to variations in the ratios of processor to network latencies and bandwidth. Although preference of programming styles might be the ultimate factor, messaging works well even on machines with lower bisections and higher latencies, and thus might be the mechanism of choice for low-cost machines.

The rest of the paper is as follows. Section 2 begins with some intuition about how we expect communication mechanisms to be affected by bandwidth and latency. Section 3 describes the hardware platform used in this study and the parallel communication mechanisms studied. While Section 4 discusses the four applications and the results obtained on them on the unaffected MIT Alewife multiprocessor, Section 5 makes a parametric comparison which varies bandwidth and latency. Section 6 relates our results to previous work, while section 7 concludes.

2 Performance Impact of Programming Models

We would like to begin this study by developing an intuition about how various communication mechanisms (and their concomitant programming models) are affected by variations in network bandwidth and latency. In the next section, we will spend time discussing the experimental platform and communication mechanisms explored for this study. For now, however, we consider *shared memory* and *message passing* in general terms.

In this paper, "shared memory" refers to the presence of hardware which automatically translates load or store instructions to shared data into messages which fetch this data. As an artifact of the communication model, shared-memory references typically incur round-trips latencies in the network. *Prefetching* provides one standard technique for tolerating network latency: it increases performance by requesting data before it is needed, thereby overlapping computation and communication. Another technique for tolerating network latency is to use a relaxed memory consistency model such as release consistency, which allows a node to have multiple pending memory accesses and to overlap the memory accesses with computation. That is generally not allowed by sequential consistency, which is the most intuitive memory model for the programmer. Many shared-memory implementations also permit *caching* of shared data; this has the effect of depressing the overall volume of communication for applications which exhibit sufficient locality.

"Message passing" refers to communication which is *asynchronous* and *unacknowledged*. Message-passing applications communicate by interactions with the network interface (or operating system). To send a message, these applications must first construct, then launch the message. At the receiver, messages are extracted from the network interface either by a polling thread or an interrupt handler. Note that, in contrast to shared memory, message-passing communication requires only a single pass through the network. Also, in contrast to shared memory, message passing exhibits higher communication overhead, since messages must be constructed explicitly in software.

2.1 Variations of Bisection Bandwidth

Figure 1 illustrates how we expect the performance of communication mechanisms to scale on our applications as bisection bandwidth varies with respect to processor speed. The *Shared Memory* curve is indicative of performance for applications either with or without prefetching. The *Message Passing* curve is indicative of applications using interrupts, polling, or bulk transfer. Because shared memory consumes more bandwidth than message passing, we expect its performance to degrade more quickly. We see three possible regions of performance:

Latency Hiding: In this region, decreases in bisection bandwidth are hidden by low communication volume or parallel *slackness* in the computation. That is, the application always does useful computation while waiting for communication to travel through the network. Consequently, application performance is unaffected by increasing network latencies in this region.

Latency Dominated: In this region, decreases in bisection bandwidth results in higher communication latency which can not be hidden with useful computation. In message-passing communication, this may happen because there is too much latency and not enough parallel work. In shared-memory based on sequential consistency, latency can often not be hidden because the processor is stalled upon a reference to data that is not in the cache. In an invalidation protocol, the processor must wait for at least one roundtrip set of messages for any reference.

Congestion Dominated: In this region, congestion in the network accounts for more of the degradation in performance than the linear decrease in bisection bandwidth on the X-axis. As bandwidth decreases, messages stay in the network longer and congestion increases non-linearly. We expect the Congestion Dominated region to occur earlier in shared memory because, as we shall see in the next section, it requires up to *six* times as much communication volume as message-passing on the same application.

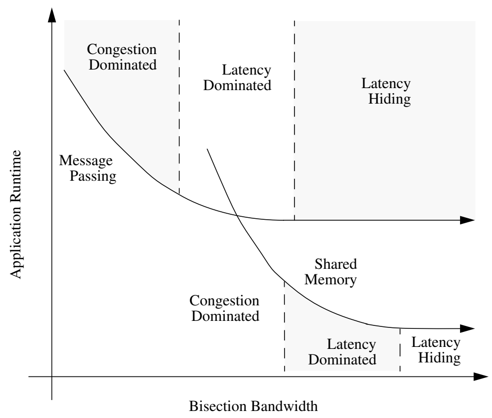


Figure 1: Regions of performance in processor cycles as bisection bandwidth varies

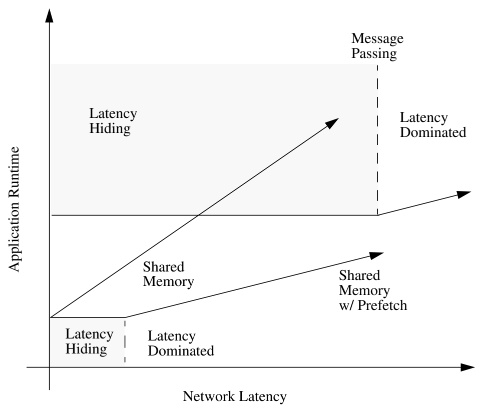


Figure 2: Regions of performance in processor cycles as network latency varies

2.2 Variations in Latency

We also expect our mechanisms to tolerate network latency differently. This is illustrated in Figure 2. Once again, we see regions where latency is hidden and where it is not. The one-way nature of message passing allows for the best latency hiding. Latency only becomes an issue when lack of parallelism in the application causes waiting for message results. Under sequential consistency, shared memory does not hide latency without prefetching. Furthermore, prefetching hides latency less well than message passing, depending upon the user or compiler to predict future references. Because message passing and prefetching can have some number of outstanding requests, the slope of their performance degradation is shallower than that for shared memory without prefetching.

3 Experimental Platform

In this study, we made use of the MIT Alewife machine[1]. Alewife provides a unique opportunity to explore the behavior of a number of different communication mechanisms in a single hardware environment. In the following, we first discuss the Alewife architecture, then proceed to describe the communication mechanisms that we used in the rest of the paper.

3.1 The Alewife Multiprocessor

Figure 3 shows an overview of the architecture. The nodes in an Alewife machine can communicate via either shared memory or message passing. Each node consists of a Sparcle processor (a modified SPARC), a floating point unit, 64K bytes of direct-mapped cache with a line size of 16 bytes, 8M bytes of DRAM, an Elko-series 2D-mesh routing chip (EMRC) from Caltech, and a custom-designed Communication and Memory Management Unit (CMMU). This paper uses a 32-node version of the Alewife machine with a 20MHz Sparcle processor and EMRC network routers operating with a per-link bandwidth of 40M bytes/second.

As shown in Figure 3, the single-chip CMMU is the heart of an Alewife node. It is responsible for coordinating message passing and shared memory communication as well as handling more mundane tasks such as DRAM refresh and control. It implements Alewife's scalable LimitLESS cache coherence protocol under sequential consistency, and provides Sparcle with a low-latency interface to the network. To communicate via shared-memory, users simply read/write from the shared address space; the CMMU takes care of the details of acquiring remote data and caching the results locally. Similarly, users send and receive messages by accessing hardware network queues directly through the CMMU.

To aid experiments, the CMMU contains hardware statistics counters that allow non-intrusive monitoring of a wide-array of machine parameters and application performance characteristics. Sample statistics include parameters such as network bandwidth consumption, cache hit ratios, memory wait time, and breakdowns of network packet types. Also supported are profiling statistics such as number of cycles spent synchronizing. Access to statistics is provided through simple, command-line facilities which are integrated with the normal Alewife execution environment.

3.2 Communication Mechanisms

Alewife provides an integration of both shared-memory and message-passing communication mechanisms. In this study, we employ the following communication mechanisms: message passing with interrupts and polling, bulk transfer via DMA, and shared memory with and without prefetching. We will briefly describe how each of these operates on the Alewife machine.

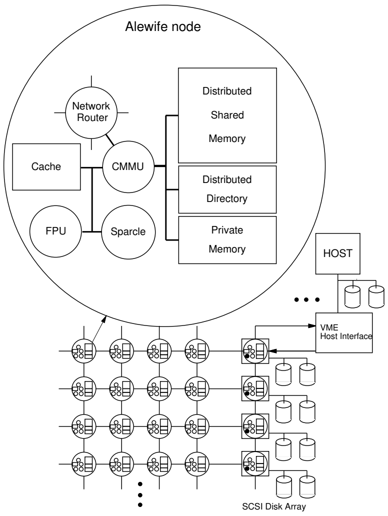
Message passing with interrupts: For message passing, Alewife supports active messages [10] of the form:

send_am(proc, handler, args...)

which causes a message to be sent to processor *proc*, interrupt the processor, and invoke *handler* with *args*. An active message with a null handler, no body and no arguments, only takes 102 cycles plus .8 cycles per hop. The Alewife network interface (within the CMMU memory controller) can hold up to fourteen 32-bit arguments for an active message.

When a message arrives, the receiving processor is interrupted and it runs the handler associated with the message. This interrupt-driven approach is the most intuitive notion of active messages, but processor interrupts can be very expensive.

Message passing with polling: Active messages come in two flavors, those received via interrupt and those received via polling. In fact, on systems such as the Thinking Machines CM5 [29], the expense of interrupts led to the predominant use of polling. For active-message reception via polling, the receiving processor is computing along its main thread of computation, and if messages arrive they are deferred until the computation reaches a point where the user or compiler has explicitly inserted a polling call in the code. We use the



Miss Type	Home Location	# Inv. Msgs	hw/ sw	Miss Penalty	
				Cycles	μsec
Load	local	0	hw	11	0.55
	remote	0	hw	38	1.90
	remote (2-party)	1	hw	42	2.10
	remote (3-party)	1	hw	63	3.15
	remote	–	sw†	425	21.25
Store	local	0	hw	12	0.60
	local	1	hw	40	2.00
	remote	0	hw	38	1.90
	remote (2-party)	1	hw	43	2.15
	remote (3-party)	1	hw	66	3.30
	remote	5	hw	84	4.20
	remote	6	sw	707	35.35

† This sw read time represents the throughput seen by a single node that invokes LimitLESS handling at a sw-limited rate.

Typical shared memory miss penalties

Figure 3: The MIT Alewife multiprocessor

Remote Queues abstraction [4], which supports polling with selective interrupts for system messages.

Bulk transfer: Bulk transfer is accomplished in Alewife by adding *(address, length)* pairs that describe blocks of data to the end of an active message. The CMMU uses a DMA mechanism to append this data to the outgoing message, after the handler arguments. On the receive side, the handler is invoked with its arguments and may either direct the CMMU to store the data to memory via DMA or consume the data directly from the network interface.

Shared Memory: Alewife provides sequentially-consistent shared memory using the LimitLESS cache-coherence protocol. The LimitLESS hardware directly tracks up to five copies of data, trapping into software for data items that are more widely shared. A shared-memory read miss handled in hardware takes 42 or 63 processor cycles depending on whether the block is dirty or clean, plus 1.6 cycles per hop in the network to the processor where the data resides. The table in Figure 3 summarizes shared memory costs on Alewife.

Shared Memory with Prefetching: As a means to tolerate memory latency, Alewife supports non-binding software prefetch of both read-shared and read-exclusive data through special prefetch instructions. These instructions take an address and check to see if data for this address is present on the local node; if not, they *initiate* a transaction to fetch this data into a local prefetch buffer but do not wait for data to return. Later references to the data will transfer it from the prefetch buffer into the cache.

4 Alewife-Specific Results

In this section, we describe our applications and their performance on Alewife. This will give us a starting point from which to vary bandwidth and latency in Section 5. Figure 4 summarizes the performance of each communication mechanism on each of our four applications. Execution time is broken down into four components (from bottom); (i) *compute* time including cache hits, (ii) *memory + NI wait* time which includes all time the processor is stalled waiting for cache misses and network interface resources, (iii) *message overhead*, which is the processor overhead to send and receive messages, or the gather-scatter copying time at bulk transfers, and finally (iv) *synchronization* time, including barriers, locks, and spinning on synchronization variables.

Overall, we see that shared memory mechanisms performed well, even though our irregular computations have little data re-use and are data-driven. This is primarily because of the low remote-miss penalty on Alewife, which makes shared memory a data-transfer mechanism competitive with message passing.

The impact of prefetching depends heavily on the computation to communication ratio, and while it is low in EM3D leading to a significant performance boost, it is much higher for the other applications. In the case of ICCG, the low ratio of remote data causes most prefetches to be useless, and add overhead, thus slowing down the prefetching version.

Bulk transfer fails to achieve a significant advantage on any application. The irregularity of the applications makes gather-scatter copying costs and idle time a significant factor. Gather-scatter costs can be as high as 60 cycles per 16-byte cache line of data. With shared-memory and message-passing overheads as low as 100 cycles, bulk transfer shows little gain on Alewife.

On applications with high messaging overhead, polling worked well versus message-passing with interrupts. On ICCG, in particular, frequent asynchronous message interrupts produced uneven processor progress and high synchronization times. We also observe this effect, to a lesser degree, on EM3D and UNSTRUC.

The remainder of this section provides brief application descriptions and more detail on performance results.

4.1 EM3D

EM3D from Berkeley models the propagation of electromagnetic waves through three-dimensional objects [21]. It uses an implicit red-black computation on an irregular bipartite graph. EM3D is iterative, and is barrier-synchronized between two phases. Communication is taking place because of data being updated within one phase that will be used by other nodes in the subsequent phase.

Our versions are based on the code from the University of Wisconsin [6]. Program parameters are 10000 nodes, degree 10, 20 percent non-local edges, span of 3, and 50 iterations.

EM3D with Message Passing

Our message passing implementations perform a communication step in each phase, making sure that all non-local data needed in the phase is available before any computation starts.

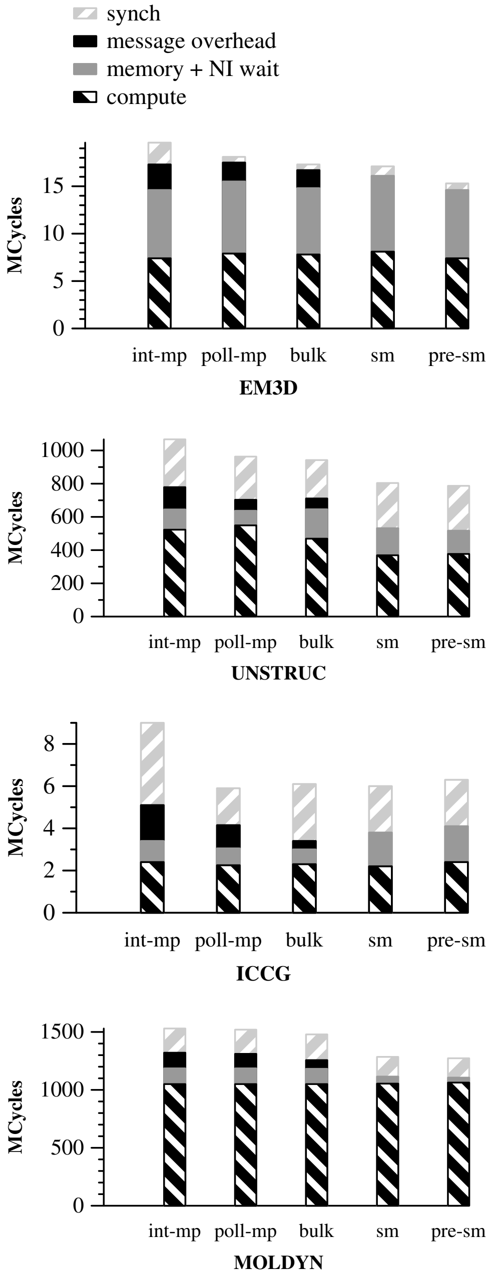


Figure 4: Summary of performance on Alewife

Our fine-grained message-passing implementations communicate five double-words at a time with an active message.

Our bulk-transfer implementation explicitly gathers communication data into a contiguous buffer for subsequent DMA transfer. EM3D allows for large enough DMA transfers to cover for the gathering overhead. Once the data arrives, preprocessing of the data structures allows it to be used in-place. This preprocessing code, however, is extremely complex and represents high coding effort.

EM3D with Shared Memory

The shared memory implementation of EM3D is much simpler because neither preprocessing code nor any pre-communication step are required. Barriers provide synchronization between phases and iterations.

Prefetching was inserted as follows. A write-prefetch is issued to get write-ownership of data structures before the computation for that data begins. In addition, read prefetches were used to fetch two values two computations ahead of use. Inserting these prefetches was straightforward, requiring only 3 lines of code.

EM3D Performance

For EM3D, the results in figure 4 show that for Alewife, shared-memory performs competitively to message-passing. Bulk-transfer gains from lower overheads of DMA transfer, but pays the costs of message aggregation and software caching. The fine grained versions suffer from higher message overhead, which offset the benefits from avoiding copying for message aggregation.

EM3D is our only application which benefits significantly from prefetching. As explained earlier in this section, this stems from its low ratio of computation to communication, making gains in communication time more significant.

4.2 UNSTRUC

UNSTRUC simulates fluid flows over three-dimensional physical objects, represented by an unstructured mesh [24]. The code operates upon *nodes*, *edges* between nodes, and *faces* that connect three or four nodes. We used MESH2K as an input dataset, a 2000 node irregular mesh provided with the code.

The shared-memory versions obtained were optimized for data distribution and privatization. The message-passing versions were developed from the shared-memory versions.

Every edge of the graph results in 75 single-precision FLOPs of computation, with 3 single-precision results for each node. The high computation-to-communication ratio, which is significantly higher than both EM3D and ICCG, makes good performance likely.

UNSTRUC with Message Passing

The fine-grained message-passing implementation uses active messages to both read and write remote values while computing values for an edge. Remote reads are done prior to the computation phase rather than on request, thus avoiding round trips needed for a read-on-request model. Remote writes are used to write the results back to remote nodes as soon as produced.

In the bulk transfer implementation of UNSTRUC, gather and scatter steps are necessary to copy node values into the contiguous arrays used for bulk transfer. Active messages are used, and an entire array of values is transferred via DMA. The values are used in-place once the data arrives at its destination buffer.

UNSTRUC with Shared Memory

The shared-memory implementation is similar to the fine-grained message-passing implementation, but without the code for buffering the receive message data. This results in savings in buffer management code and memory usage.

Our prefetching implementation inserts two write prefetches, two edge-computations ahead, to get write ownership of upcoming node values. Prefetching required only minor code modifications.

UNSTRUC **Performance**

We note that the shared memory implementations do not perform better than message-passing despite their avoidance of message aggregation and extra buffering. This is primarily because they incur locking overhead while protecting updates to shared node data. Message passing avoids locking as the non-interruptible nature of handlers automatically provides mutual exclusion of writes.

Compared to bulk-transfer, the fine-grained versions avoid message aggregation overhead, but have higher message handling overheads. The lower per-message overhead of the polling version allows it to outperform the interrupt based version.

4.3 ICCG

ICCG is a general iterative sparse matrix solver using conjugate gradient preconditioned with an incomplete Cholesky factorization. It performs a graph computation, where each node of the graph represents the solution (by substitution) for an unknown in a sparse linear system. Each graph node must wait for all of its incoming edges to be communicated, perform a 2 FLOP computation for each edge, and then communicate data along its outgoing edges.

We measure the performance of the ICCG sparse triangular solve kernel running on a large structural finite-element matrix, the BCSSTK32 2-million element automobile chassis, obtained from the Harwell-Boeing benchmark suite [9].

We started from existing parallel ICCG algorithms [14] [26]. Implementation details are available in [7].

ICCG **with Message Passing**

The ICCG computation graph is essentially a dataflow computation [2] and is easily implemented via active messages. For each node in its local memory, a processor keeps track of when all incoming edges have been satisfied, whereafter the outgoing edges can be processed.

Bulk transfer is also straightforward. We buffer up multiple non-local edges into buffers in memory, one buffer for each processor that edges are destined for. Unfortunately, this buffering incurs significant cost in memory operations and idle time.

ICCG **with Shared Memory**

In the shared-memory model, the program must be extended with synchronizations to coordinate multiple processors providing updated edges to the same node. We use a *producer-computes* model, in which the producer of the edge value also makes the corresponding updates to the node value using shared memory. We use a spin-lock per node to enforce atomicity. In our implementation, up to four messages are required for every non-local edge and additional messages may also be required to successfully acquire the spin lock. On Alewife, the lock request can be piggy-backed on the write ownership request. For prefetching, two write prefetches were inserted two nodes ahead of our computation loop.

ICCG **Performance**

ICCG shows the largest improvement from interrupts to polling for message passing. The low computation-to-communication ratio of ICCG results in a large number of messages, which makes interrupt overhead significant. Polling cuts this overhead by about 35 percent. More importantly, interrupts cause dramatically more synchronization time than polling, because of the large load imbalance caused by asynchronous interrupts [5]. Polling provides greater control of message reception, which allows for a more balanced computation [4]. The other mechanisms also avoid load imbalance. Shared memory mechanisms do not use interrupts and are similar to polling. Bulk transfer uses fewer messages than finer-grained message-passing and thus few interrupts.

4.4 MOLDYN

MOLDYN is a molecular dynamics application developed by the University of Maryland and the University of Wisconsin [24]. A molecule's position is determined by its own velocity and the force employed by other molecules within a certain cut-off radius. The molecules are distributed between the processing nodes using the RCB algorithm [3], minimizing the communication.

The force of a molecule is affected by all its interactions with other molecules, and can therefore be updated by both local and remote processors, while the coordinates are written by one processor but potentially read by many.

MOLDYN **with Message Passing**

The bulk-transfer implementation sends the coordinates of local molecules to a remote node that calculates all interactions between molecules from the two nodes. The remote node collects force-deltas to each molecule, and then returns them in a bulk transfer.

In the fine-grained implementation, we tried a version which interleaves communication and computation, but we found that the message handlers tied up network resources for too long and caused network congestion. Instead, we implemented a communication phase similar to bulk transfer, attempting to overlap sending and receiving of messages.

MOLDYN **with Shared Memory**

The shared-memory version is a straight-forward implementation, where each node reads all interacting molecules' coordinations, calculates force-deltas in one phase, and in another phase updates the forces of its own molecules based on force-deltas from other nodes.

Prefetching was inserted as follows. For writes to remote force-delta locations, those locations were write-prefetched to gain exclusive write ownership, one iteration prior to when they were actually written. Remote coordinates were similarly read-prefetched one iteration prior to use.

MOLDYN **Performance**

The high computation-to-communication ratio of MOLDYN tends to mask differences in our implementations. On Alewife, however, interrupts had trouble receiving messages quickly enough to avoid network congestion. Although polling often worked well, bursty traffic forces us to be conservative when inserting polling calls.

The shared-memory versions used locks to protect writes to shared data, which worked well because of low lock-contention. Because computation is dominant, the prefetching version did only about one percent better despite static knowledge of remote data.

5 Parametric Experiments

The previous section presented detailed comparisons of communication mechanisms on the Alewife multiprocessor. To the extent that Alewife is a "balanced" architecture indicative of future trends, our comparisons are interesting. In general, however, the relative performance of shared memory and message passing are dependent upon the relative speeds of processors, memory systems, network interfaces, and network switches in a particular multiprocessor design. In Section 2, we developed an intuition about how multiprocessor communication mechanism scale with network bandwidth and latency. This intuition focused upon Figures 1 and 2. In this section, we would like to explore the space of these figures by measuring the variation

Machine Type (32 Processors)	Proc clock (MHz)	Network Interconect Topology	Bisection Bandwidth Mbytes/ second	Bisection Bandwidth bytes/ cycle	Network Latency (cycles)	Remote Miss Penalty	Local Miss Penalty
MIT Alewife	20.0	4×8 Mesh	360	18.0	15	50	11
TMC CM5	33.0	4-ary Fat-Tree	640	19.4	50	N/A	16
KSR-2	20.0	Ring	1000	50.0	?	126	18
MIT J-Machine	12.5	$4 \times 4 \times 2$ Mesh	3200	256.0	7	N/A	7
MIT M-Machine	#100.0	$4 \times 4 \times 2$ Mesh	12800	128.0	10	154	21
Intel Delta	40.0	4×8 Mesh	216	5.4	15	N/A	10
Intel Paragon	50.0	4×8 Mesh	2800	56.0	12	N/A	10
Stanford DASH	33.0	2×4 4-proc clusters	480	14.5	31	120	30
Stanford FLASH	*200.0	4×8 Mesh	3200	16.0	62	352	40
Wisconsin T0	#200.0	none simulated	N/A	N/A	200	1461	40
sWisconsin T1	#200.0	none simulated	N/A	N/A	200	401	40
Cray T3D	150.0	$4 \times 2 \times 2$ Torus 2-proc clusters	4800	32.0	15	100	23
Cray T3E	300.0	$4 \times 4 \times 2$ Torus	19200	64.0	110	300-600	80
SGI Origin	200.0	Hypercube 4-proc clusters	10800	54.0	60	150	61

* projected, # simulated, latencies given in processor cycles.

Network Latency is for one-way network transit time of a 24-byte packet. *Remote Miss Penalty* is an average of best- and worst-case write misses. Full references can be found in MIT technical memo MIT-LCS-TM-562.

Table 1: Parameter estimates for various 32-processor multiprocessors

in performance for different versions of our applications as a function of network parameters.

We present three sets of experiments on the Alewife machine to investigate scaling of communication performance. First, we examine the variations in *communication volume* produced by different communication mechanisms. Second, we vary *network bandwidth* by introducing cross traffic to simulate reduced network bandwidth and increased congestion. Finally, we vary *network latency* by altering the clock speed of the processing nodes to change the relative latency of the network.

Before we begin, however, we might ask the following question: Why do different research studies report contradictory results when comparing communication mechanisms? As shown in Table 1, the answer is that machines have widely differing parameters, thereby inhabiting vastly different regions of the communications performance space.

5.1 Communication Volume

As a first step toward exploring the performance space, we measure a key statistic: communication volume. Communication volume is the amount of data injected into the network over the course of an execution. Figure 5 shows the average communication volume for each version of each of our applications. Although shared memory provides the most performance for the coding effort, it also causes a significantly higher strain on network bandwidth than message passing or bulk transfer. That increase in communication volume, however, would be lower for systems with a larger cache line size for most applications. Figure 5 further breaks the communication volume into the following four components:

1. *Invalidates* – all traffic associated with invalidating cached copies of remote data.
2. *Requests* – read, write, and modify requests.
3. *Headers (for data)* – all message headers for message passing; message headers for cache-line transfers for shared memory.
4. *Data* – message-passing payload and shared-memory cache lines.

Message interrupts and polling produce the same message volume, since they send the same messages, but only receive them differently. Bulk transfer saves on message headers. Note, however, that bulk transfer for ICCG loses the saving in header traffic to padding in the payload. DMA on Alewife requires double-word alignment, which ends up producing a significant effect on ICCG's small bulk transfers.

Where message passing uses a single message to communicate a value along each edge of a graph problem, shared memory (using an invalidation protocol) must use at least four: the writer must invalidate the reader's copy, the reader acknowledges the invalidate, the reader later requests a valid copy, and the write responds with valid copy. Additional messages may be required if the writer must invalidate cached copies on more than one reader. Additional traffic is generated when spin-locks are necessary to enforce atomic read-modify-writes.

Interestingly, this increased volume does not impact performance on Alewife. Even when message-passing traffic causes enough network congestion to cause message queue overflows, the corresponding shared memory traffic does not cause congestion. The key factor is *occupancy* at the endpoints. Shared memory pulls messages out of the network much faster than message passing, resulting in a clearer network even at higher volume. As we shall see in the next section, however, the effect of low occupancy can only compensate for network bandwidth up to a point.

5.2 Bisection Bandwidth Emulation

In this section, we show that decreasing bisection bandwidth causes shared-memory performance to degrade more quickly than message-passing performance, resulting in a cross-over point when other Alewife parameters are held constant.

Using background cross-traffic, we emulate the performance of systems with lower bisection bandwidth (per processor cycle) than Alewife. In addition to compute nodes, Alewife has I/O nodes which can be added in columns at either side of the 2-dimensional mesh. We use these I/O nodes to send messages from the edges of the 2D mesh across the bisection in both directions (see Figure 6). On a 32-node machine, the network has 8 nodes in the X-direction and 4

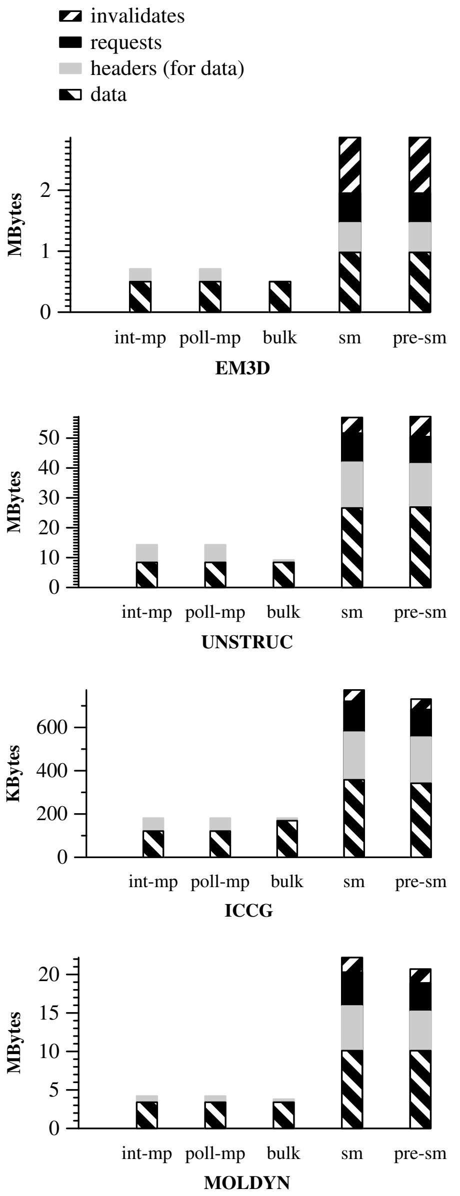


Figure 5: Communication volume for each mechanism

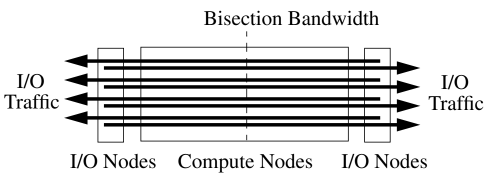


Figure 6: I/O cross-traffic experiment

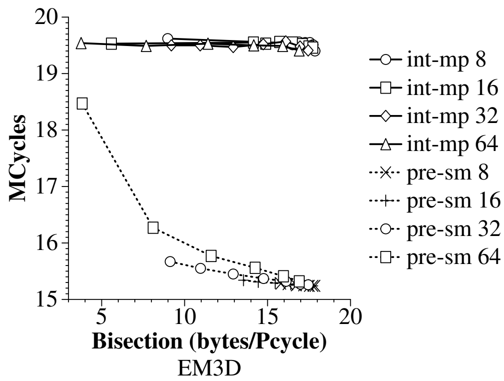


Figure 7: Sensitivity to IO-traffic message length.

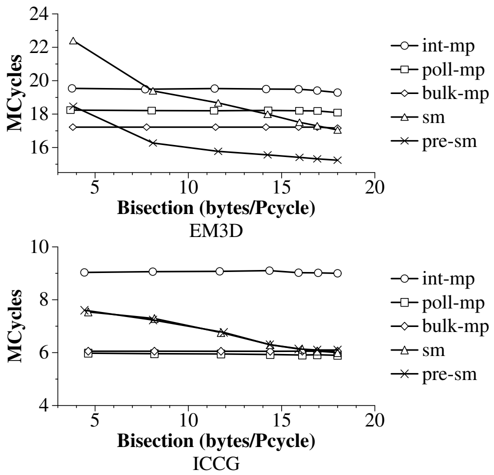


Figure 8: Execution time (in cycles) versus bisection bandwidth. Alewife bisection is 18 bytes/processor-cycle (See Table 1).

nodes in the Y-direction. We use 4 I/O nodes on each edge to send messages off the opposite edge of the mesh. The messages travel off the edge of the network without disturbing our applications on any of the compute nodes.

The bisection of the emulated system is calculated by taking Alewife's bisection (18 bytes/processor-cycle) and subtracting the amount of cross traffic sent. The smaller the cross-traffic messages used, the more accurate our emulation. Small messages, however, limit the rate at which the I/O nodes can send cross-traffic, preventing the emulation of systems with lower bisection. Figure 7 shows the sensitivity of our experiments to cross-traffic message length. For the remainder of our experiments, we chose 64-byte cross-traffic messages, a relatively small size which still allows a wide range of bisection emulation.

Figure 8 plots application performance as the amount of I/O cross-traffic varies. The X-axis plots bisection bandwidth in bytes per processor cycle. The Y-axis plots application runtime in processor cycles. We can see that the high communication volume of shared-memory mechanisms cause application performance to degrade dramatically faster than message-passing mechanisms as bisection bandwidth decreases. While our results have shown that shared-memory mechanisms perform very well when adequate network performance is available, we can see a performance crossover

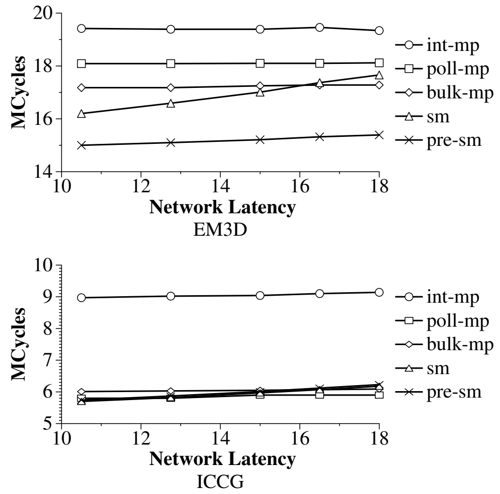


Figure 9: Network latencies emulated by varying node clock. The latency gives 1-way delivery of 24 bytes (see Table 1). The Alewife machine is at 15 processor-cycles.

with message-passing mechanisms as bisection bandwidth decreases. Referring back to Table 1, we see that most machines have much higher bisection bandwidth per processor cycle than our cross-over point. However, we notice that low-dimensional mesh architectures such as DASH and FLASH [1] approach the cross-over points. As processor speed increase, providing adequate bisection bandwidth will become increasingly expensive.

5.3 Network Latency Emulation

We also perform an experiment which demonstrates that shared memory is less tolerant of network latency than message passing. The Alewife machine has a programmable clock generator which can vary the clock speed of the processing nodes from 14 MHz to 20 MHz. The Alewife network is asynchronous and communication latency through the network is unaffected by the change in processor clock. Consequently, we can incrementally slow down the Alewife processing nodes from their normal 20 MHz to 14 MHz, giving the nodes the appearance of a faster and faster network. If we plot application performance in processor cycles, we can see the performance trend as relative network latency varies.

Figure 9 gives such a plot. The X-axis plots network latency (in processor cycles) to deliver a 24-byte message, as used in Table 1. The Y-axis plots application runtime in processor cycles. The points were obtained by varying processor clock speed as just described. We can see that both shared memory implementations are more susceptible to increased network latency than message passing implementations. This is because network latency shows up as processor stall time when the processor blocks on a shared memory operation. Prefetching hides this latency somewhat, but not as well as message passing.

To emulate higher network latencies, we use Alewife's fast context-switching mechanism. On every remote miss, we perform a context switch to a thread running a delay loop. The resulting execution emulates an ideal network with uniform access times and infinite bandwidth. Figure 10 shows our results. Note that prefetching is not precisely modeled, since the success of a prefetch is dependent upon Alewife's original network latency rather than the emulated latency. The message-passing and bulk transfer curves are plotted for reference only. Their network latencies are not varied and are based upon Alewife network latencies. However, since our message-passing applications use asynchronous, unacknowledged communication, we expect that message-passing performance will remain relatively constant. Other studies [22] have found that asynchronous implementations of applications such as EM3D are relatively insensitive to microsecond-latencies on networks of workstations.

[1] Note that FLASH has been redesigned to use the Origin network since [13].

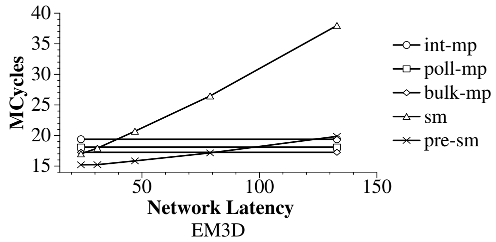


Figure 10: Network latencies emulated with context-switching.

Referring back to Table 1, we see that network latency is a serious issue for shared memory that will worsen as processor speeds increase. All modern machines have considerably higher network latencies than Alewife.

6 Related Work

Not only has our work has been strongly influenced by studies from Wisconsin, Stanford, and Maryland, but these previous results help confirm the general context established by our emulations. Our comparison of communication mechanisms is similar to Chandra, Larus and Rogers [6], but we have available a larger set of mechanisms and we generalize to a range of system parameters. This generalization is similar to the study of latency, occupancy, and bandwidth by Holt et. al [15], which focuses exclusively upon shared-memory mechanisms. Although the Alewife machine provides an excellent starting point for the comparison of a large number of communication mechanisms, our results are greatly enhanced by our use of emulation, an approach inspired by the work at Wisconsin [25].

Chandra, Larus and Rogers compare four applications on a simulation of a message-passing machine similar to a CM-5 multiprocessor against a simulation of a hypothetical machine also similar to a CM-5, but extended by shared-memory hardware. Their results are a good point of comparison for our emulation results, since both Alewife and the CM-5 are SPARC-based architectures with very similar parameters. They found that message passing performed approximately a factor of two better than shared memory while simulating a network latency of 100 cycles. Referring back to Figure 10, assuming that message-passing continues to hide latency, our network latency emulation shows the same result.

Our results also agree well with studies from Stanford. Holt et al. found latency to be critical to shared-memory performance, as we did. They also found that node-to-network bandwidth was not critical in modern multiprocessors. Our study shows, however, that bandwidth across the *bisection* of the machine may become a critical cost in supporting shared memory on modern machines. Such costs will make message passing and specialized user-level protocols [11] increasingly important as processor speeds increase.

Woo et al. [30] compared bulk transfer with shared memory on simulations of the FLASH multiprocessor [13] running the SPLASH [12] suite. They found bulk transfer performance to be disappointing due to the high cost of initiating transfer and the difficulty in finding computation to overlap with the transfer. Although, Alewife's DMA mechanism is cheaper to initiate than theirs, we also found bulk transfer to have performance problems. Our problems arose from the irregularity of our application suite, which caused high scatter/gather copying costs and limited data transfer size.

Concurrent work at Berkeley [22] explores the effect of message-passing latency, overhead and bandwidth on networks of workstations. They measured performance of several programs written in Split-C and compared their results with predictions from the LogP model. The effects of overhead and gap on applications were predicted well by LogP. The effects of latency and bandwidth, however, were too complex for a simple model. Our study focuses on these more complex effects and how they differ for a variety of communication mechanisms. As we have seen, their empirical results on latency are consistent with ours.

Several of our applications were borrowed from a Maryland-Wisconsin study, Mukherjee et. al.[24]. We extended their codes by developing optimized implementations for each of our communication mechanisms. They studied a different problem, in that results on a software shared-memory interface run on a message-passing machine were compared to those directly obtained on the message-passing machine.

Another group of studies simulated message passing on a shared memory machine, and compared the performance of message passing programs using this simulation, against programs using shared memory directly. This class includes papers by Lin and Snyder [19], Martonosi and Gupta [23], and LeBlanc and Markatos [17]. These studies however do not compare machine implementations, as all programs ultimately used shared memory only.

Klaiber and Levy [16] study the performance of programs which accesses shared memory or message passing runtime libraries. These libraries generated traces for shared memory and message passing simulators, to generate statistics on message traffic. However, their programs were not fined tuned for any particular architecture, and hence not fair to either. Our programs are highly optimized for the mechanisms and performance parameters of a machine supporting both communication methods. Further, their machine independant libraries tend to introduce unnecessary overheads for both methods, leading to additional loss of comparison accuracy. Finally they report only message traffic, not execution time numbers. They do not show how message traffic impacts runtime.

7 Conclusion

Our results provide a framework from which to evaluate mechanisms on systems with differing bisection bandwidth and network latency. We find that shared memory provides good general performance with minimal coding effort. Our experiments, however, each show performance cross-overs between shared memory and message passing as system parameters change. We evaluate these crossover points with respect to real systems. Although most existing multiprocessors provide adequate bisection bandwidth to support shared memory, providing this bandwidth in future systems will be at least as important. Network latency is an even more severe problem, but depends heavily upon whether an application is compute- or memory-limited.

References

[1] Anant Agarwal et al. The MIT Alewife machine: Architecture and performance. In *ISCA '22*, 1995.

[2] Arvind, David E. Culler, and Gino K. Maa. Assessing the benefits of fine-grained parallelism in dataflow programs. In *Supercomputing '88*. IEEE, 1988.

[3] M. J. Berger and S. H. Bokhari. A partitioning strategy for PDEs across multiprocessors. In *ICPP*, 1985.

[4] Eric A. Brewer et al. Remote queues: Exposing message queues for optimization and atomicity. In *SPAA '95*, 1995.

[5] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In *IPPS*, 1994.

[6] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs. In *ASPLOS VI*, pages 61–73, 1994.

[7] Frederic T. Chong and Anant Agarwal. Shared memory versus message passing for iterative solution of sparse, irregular problems. Tech rpt, mit-lcs-tr-697, MIT Lab for Comp Sci, Cambridge, MA, 1996.

[8] Frederic T. Chong et al. Application performance on the mit alewife multiprocessor. *IEEE Computer*, Dec 1996.

[9] Ian S. Duff, Roger G. Grimes, and John G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection. Tech Rpt TR/PA/92/86, CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France, 1992.

[10] Thorsten von Eicken et al. Active messages: a mechanism for integrated communication and computation. In *ISCA '19*, 1992.

[11] Babak Falsafi et al. Application-specific protocols for user-level shared memory. In *Supercomputing 94*, 1994.

[12] Maya Gokhale et al. Building and using a highly parallel programmable logic array. *Computer*, 24(1), January 1991.

[13] Mark Heinrich et al. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *ASPLOS VI*, pages 274–285, 1994.

[14] Bruce Hendrickson and Robert Leland. The Chaco user's guide. Tech Rpt SAND94-2692, Sandia National Labs, 1995.

[15] C. Holt et al. The effects of latency, occupancy and bandwidth on the performance of cache-coherent multprocessors. Tech rpt, Stanford Univ, Stanford, CA, Jan 1995.

[16] A. Klaiber and H. Levy. A comparison of message passing and shared memory for data-parallel programs. In *ISCA '21*, 1994.

[17] T. LeBlanc and E. Markatos. Shared memory vs. message passing in shared-memory multiprocesors. In *4th SPDP*, 1992.

[18] Daniel Lenoski and others. The Stanford Dash multiprocessor. *Computer*, 25(3):63–80, 1992.

[19] C. Lin and L. Snyder. A comparison of programming models for shared-memory multiprocessors. In *ICPP*, 1990.

[20] K. Mackenzie et al. Exploiting two-case delivery for fast protected messaging. In *HPCA-4*, 1998.

[21] N. K. Madsen. Divergence preserving discrete surface integral methods for Maxwell's curl equations using non-orthogonal unstructured grids. Tech Rpt 92.04, RIACS, 1992.

[22] Richard P. Martin et al. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *ISCA '24*, pages 85–97, 1997.

[23] M. Martonosi and A. Gupta. Tradeoffs in message passing and shared memory implementations of a standard cell router. In *ICPP*, 1989.

[24] Shubhendu S. Mukherjee et al. Efficient support for irregular applications on distributed-memory machines. In *PPoPP'95*, pages 68–79, 1995.

[25] S. K. Reinhart, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *ISCA '21*, 1994.

[26] R. Schreiber and W. Tang. Vectorizing the conjugate gradient method. In *Proceedings Symposium CYBER 205 Applications*, 1982.

[27] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS VII*, 1996.

[28] Jaswinder Pal Singh, Chris Holt, and John Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-hut, fast multipole, and radiosity. *JPDC*, 27(2), 1995.

[29] Thinking Machines Corporation, Cambridge, MA. *CM-5 Technical Summary*, November 1993.

[30] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Asplos VI*, pages 219–229, 1994.