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

Contact: chong@cs.ucdavis.edu

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.

Keywords: shared-memory, message-passing, bandwidth, latency, multiprocessor

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 [43] supports both shared memory and messaging styles of communication, the Stanford Dash [28] supports shared memory and prefetching, MIT Alewife [1], Fugu [30], and the Wisconsin Typhoon [38] 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 [8] [15] [44] [46] [21] [10]. 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 [8] 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 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

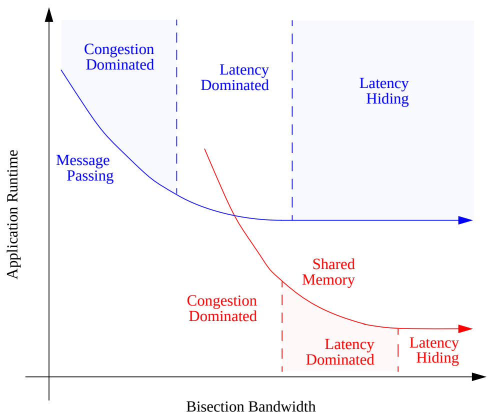


Figure 1: Regions of performance in processor cycles as bisection bandwidth varies

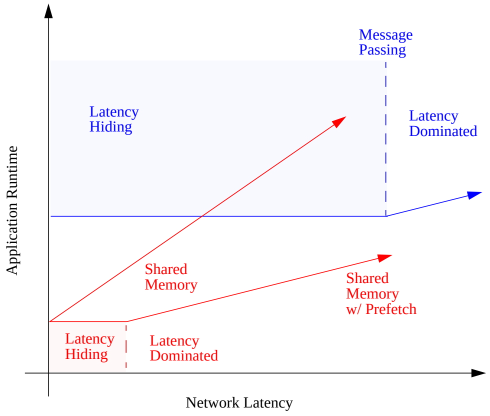


Figure 2: Regions of performance in processor cycles as network latency varies

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 The Impact of Programming Model on Performance

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 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.

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.

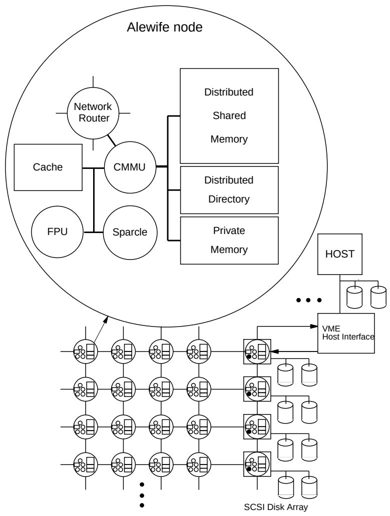
3 Experimental Platform

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



Miss	Home	# Inv.	hw/	Miss Penalty	
Type	Location	Msgs	sw	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

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 [14] 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 [45], 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 Remote Queues abstraction [5], 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 hardware-based, 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:

1. *Synchronization* time, which includes time spent in barriers, acquiring locks, and spin-waiting on synchronization variables.

2. *Message Overhead*, which includes processor overhead to send messages and receive messages, either via interrupts or polling. For

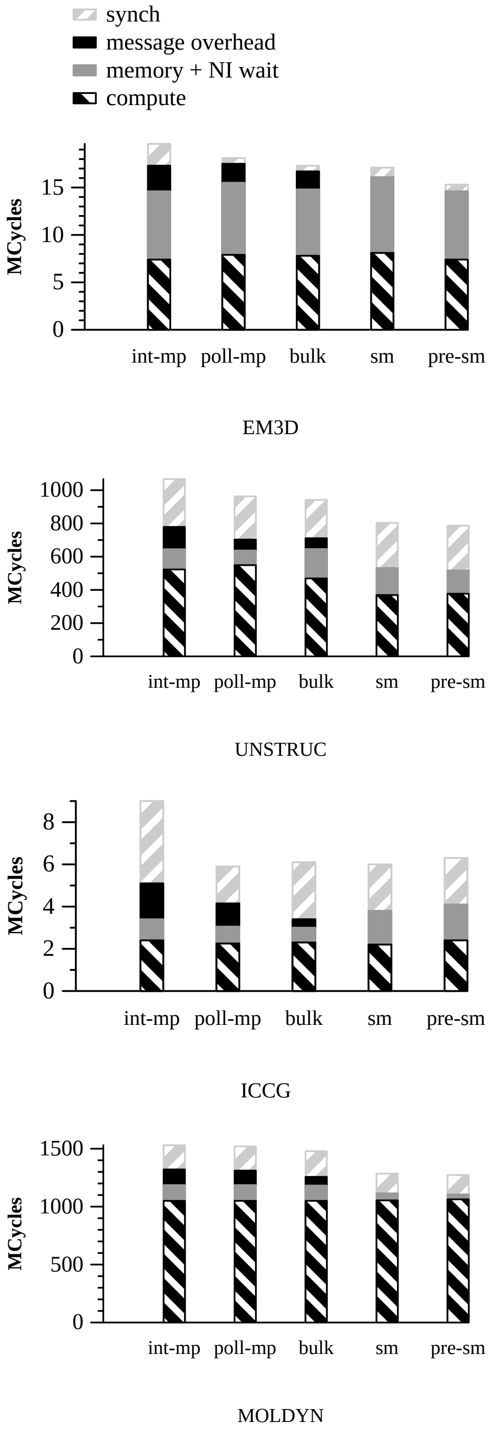


Figure 4: Summary of Performance on Alewife

bulk transfer, this time also includes gather-scatter copying time.

3. *Memory + NI Wait* time, which includes all time the processor is stalled waiting for cache misses and network interface resources. For shared memory, this time includes remote misses. For message passing, this time includes waiting for space in network input queues.

4. *Compute* time, the time the processor spends computing. Note that "useful work" would include both compute time and a portion of memory time.

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 (recall Figure 3), which makes shared memory a data-transfer mechanism competitive with message passing.

Prefetching, however, only achieves significant gains on one of the four applications, EM3D. This is because EM3D has a lower computation to communication ratio than the others, manifested as a small inner loop with communication in every iteration. While UNSTRUC and MOLDYN also show benefits on prefetching, the benefits are small, as they have higher computation to communication ratios. 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. It may be noted that programs in which we statically can predict which references are remote perform better under prefetching, as useless local prefetches are eliminated. A runtime check for remoteness is just as expensive as a useless prefetch, and hence is not performed.

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 is a small benchmark code originally developed at UC Berkeley to exercise the Split-C parallel language [11]. It models the propagation of electromagnetic waves through three-dimensional objects using algorithms described in [31].

EM3D operates on an irregular bipartite graph which consists of E nodes on one side, representing electric field value at that point, and H nodes on the other, representing magnetic field value at that point. Each iteration consists of two phases, representing updates on the E nodes, and then the H nodes. In each phase, the nodes need the values of their neighboring nodes to calculate new values. This involves propagating old values along edges of the graph and calculating new values at each node using those values. When edges are between nodes on two different processors, communication must occur. In contrast to ICCG, described later, the graph is undirected and updates are independent within each of the two phases. The code is barrier-synchronized between iterations and phases.

We started with shared-memory and CM-5 bulk-transfer codes from the University of Wisconsin at Madison [8]. The interrupt and polling message-passing versions were developed from the bulk-transfer code, which was itself first adapted to Alewife. Program parameters were: 10000 nodes, degree 10, 20 percent non-local edges, span of 3, and 50 iterations.

4.1.1 EM3D with Message Passing

For simplicity and efficiency, our message passing implementations perform a communications step before computing in each phase. This step makes sure that all non-local data necessary for the subsequent computation is available before any computation starts.

Pre-communicating all the data simplifies the computation step. It requires, however, that non-local data be stored in buffers until they are needed in the computation. Berkeley Split-C study [11] calls these buffered copies "ghost nodes," and they are equivalent to software managed cache.

Our fine-grained message-passing implementations communicate values of these ghost nodes five double-words at a time. Each double-word represents the value of one remote node to be copied into a ghost node. Values are sent with an active message indicating which specific node handler that will take care of the values at the destination node.

Calls to active message sends perform indirect references to irregular data, effectively performing a gather into the network send queue. Upon reception, the message causes the node handler to be executed and write the values into the ghost nodes, and the computation is preprocessed to use the values in that block in-place.

Our bulk-transfer implementation explicitly copies, or gathers the nodes into a contiguous buffer for subsequent DMA transfer. This copying cost can degrade performance. The undirected graph of EM3D, however, allows for large enough DMA transfers to cover this cost. Once the data arrives, preprocessing of the graph allows it to be used in-place. This preprocessing code, however, is extremely complex and represents high coding effort.

The computation for updating each node involves 2 double-precision floating point operations (FLOPs) for each edge incident to the node: a multiply of a coefficient and a value, and a addition to accumulate the result with the node's value.

4.1.2 EM3D with Shared Memory

The shared memory implementation of EM3D is much simpler because no pre-communication step is required. This eliminates communication and buffer-management code. More importantly, significant graph preprocessing code is eliminated. The remaining code just computes each phase, using simple memory references to graph structures, and all non-local data is communicated via the shared memory protocol. Barriers provide synchronization between phases and iterations.

Prefetching was inserted as follows. A write-prefetch is issued to get write-ownership of a node just before the computation for that node begins. This overlaps the computation to update a node with writing the result to the node. The write requires invalidating the copies cached in processors containing neighbors to that node.

Read prefetches were inserted to fetch 2 edge-values two edge-computations (4 FLOPs) ahead. That is, we fetch values for the $(i+2)$-the and $(i+1)$-th edges while updating a node with edge i. Inserting these prefetches was straightforward, requiring only 3 lines of code.

4.1.3 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 its communication time gains more significant.

4.2 UNSTRUC

UNSTRUC [35] simulates fluid flows over three-dimensional physical objects, represented by an unstructured mesh. 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.

Similar to EM3D, UNSTRUC computes upon an undirected graph. EM3D, however, uses an implicit red-black computation on a bipartite graph, so buffering between iterations is not necessary for shared-memory implementations. UNSTRUC, more typical of graph computations, computes upon every node of the graph every iteration. This means that old values must be buffered so that the current iteration does not modify them before they are no longer needed. This buffering is necessary in all our implementations and thus makes implementation effort more even between shared-memory and message-passing versions.

UNSTRUC performs significantly more computation per edge of its graph than either EM3D or ICCG. Every edge of the graph results in 75 single-precision FLOPs of computation, computing, 3 single-precision results for each node. The high FLOPs per edge makes good performance likely, because of a high computation-to-communication ratio.

4.2.1 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. A remote read is needed to fetch a non-local node value to compute results associated with an edge. Reads are done prior to the computation phase rather than on request, leveraging on the known communication patterns, thus avoiding round trips needed for a read-on-request model. Remote writes are used during the computation phase, to write the results back to remote nodes as soon as produced.

The bulk transfer implementation of UNSTRUC also uses remote reads and writes, except of entire arrays of node values instead of individual node values. Gather and scatter steps are necessary to copy node values into the contiguous arrays used for bulk transfer. The active messages are the same as for the fine-grained implementation, except that, instead of individual node values, an array of values is transfered via DMA. The values are used in-place once the data arrives at its destination buffer.

4.2.2 UNSTRUC with Shared Memory

The shared-memory implementation is essentially the same as the fine-grained message-passing implementation, which basically simulates shared memory. As mentioned before, buffering node values was necessary to isolate values between iterations. However, the message-passing codes required buffering the receive message data. This extra buffering is not needed in the shared memory implementation, since remote values are stored in cache. This results in moderate 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. Once again, prefetching required only minor code modifications.

4.2.3 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. When run on irregular datasets, ICCG is one of the most challenging and fine-grained applications in the literature. Similar to EM3D and UNSTRUC, ICCG performs a graph computation. Each node of the graph represents the solution (by substitution) for an unknown in a sparse linear system. ICCG is different, however, in that its computation graph is a directed acyclic graph rather than an undirected graph. This means that communication and computation can not be easily broken up into separate phases. Instead, each graph node must wait for all of its incoming edges to be communicated, perform a 2 FLOP computation (subtract and multiply) 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 [12].

We started from existing parallel ICCG algorithms [20] [41] and implemented an interrupt message-passing version. The remaining four versions were all derived from the initial message-passing code. Implementation details are available in [9].

4.3.1 ICCG with Message Passing

The ICCG computation graph is essentially a dataflow computation [3] and is easily implemented via active messages. Non-local edges, i.e. edges between nodes on different processors, are communicated with active messages. Each processor keeps a presence counter per local node to keep track of how many incoming edges have been satisfied for each node in its local memory. Once all incoming edges for a node have been satisfied, 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.

4.3.2 ICCG with Shared Memory

The use of shared memory is somewhat less obvious for this computation. The problem with shared memory is that processor 0 can perform a remote write to shared memory on processor 4 to communicate value v, but there is no synchronization event that tells processor 4 to subtract v from x_4. We can avoid this problem by adopting a *producer-computes* model, which specifies that the producer of the edge value, processor 0, compute the subtraction via a remote read-modify-write using shared memory. In fact, we keep a node's presence counter in the same cache line with its value, so that a single remote read-modify-write can be used to compute the node's value and decrement its counter. We also use a spin-lock per node to enforce atomicity. Generally, up to four messages are required for every non-local edge: a write ownership request to the home node, an invalidate to the previous writer, a cache-line transfer from the previous writer to the home node, and a cache-line transfer from the home node to the current writer. 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.

4.3.3 ICCG Performance

ICCG shows the largest improvement from message passing with interrupts to polling. 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. Asynchronous interrupts can cause some processors to fall behind in the computation, causing long idle times in others [6]. Progress is more important to ICCG than the other applications because of the data dependencies in its DAG computation. Polling provides greater control of message reception and computation progress, which allows for more balance processor progress in the computation [5]. The other mechanisms also avoid 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, and like UNSTRUC, it was developed by the University of Maryland and the University of Wisconsin at Madison [35]. The molecules are uniformly distributed over a cuboidal region with a Maxwellian distribution of initial velocities. 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 partitioned into groups to minimize the communication between the groups, which is done with the RCB algorithm from [4]. These groups of molecules are then allocated on the nodes.

Instead of relating each molecule to every other at each iteration of the application, a list of potentially interacting molecule-pairs is created every 20 iterations based on twice the cut-off radius. The most important data structures are the list of interactions and the lists of coordinates, forces, and velocities of each molecule in three dimensions. The list of interactions is built every 20 iterations and is local to the nodes. The velocities are local to each processor, the coordinations are written by the local processor and read by other processors, and the forces are updated by both local and remote processors. These data structures are distributed among the nodes according to the result of the RCB algorithm.

4.4.1 MOLDYN with Message Passing

The bulk-transfer implementation sends all the local molecules to the remote node. The remote node thereafter does the calculations of all interactions between the two nodes, collects force-deltas to each molecule for all interactions, and then returns them in a bulk transfer. We tried a fine-grained implementation 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.

4.4.2 MOLDYN with Shared Memory

The shared-memory version is a straight-forward implementation, where each node reads all interacting molecules' coordinations, and calculates force-deltas. Thereafter, the local node 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.

4.4.3 MOLDYN Performance

The high computation-to-communication ratio of MOLDYN tends to mask differences in our implementations. An interesting point, however, is that Alewife's message-passing mechanisms had some difficulty on the application. Interrupts had trouble receiving messages quickly enough to avoid network congestion. Polling was also difficult to implement. 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 just like UNSTRUC. However, the locks performed much better here, because of lower contention. The prefetching version did only about one percent better, despite static knowledge of remote data, because computation was dominant in MOLDYN.

5 Parametric Experiments

The previous section has 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 centered around Figures 1 and 2. As a goal of the current section, we would like to explore the space of these figures by actually measuring the variation in performance for different versions of our applications as a function of network parameters.

To this end, 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.

Machine (32 Processors)	Proc MHz	Topology	Bisection Bandwidth Mbytes/s	 bytes/ cycle	Network Latency	Remote Miss Latency	Local Miss Latency	Refs
MIT Alewife	20.0	4 × 8 Mesh	360	18.0	15	50	11	[1]
TMC CM5	33.0	4-ary Fat-Tree	640	19.4	50	N/A	16	[45][27]
KSR-2	20.0	Ring	1000	50.0	?	126	18	[7]
MIT J-Machine	12.5	4 × 4 × 2 Mesh	3200	256.0	7	N/A	7	[36]
MIT M-Machine	#100.0	4 × 4 × 2 Mesh	12800	128.0	10	154	21	[16][23]
Intel Delta	40.0	4 × 8 Mesh	216	5.4	15	N/A	10	[13] [34]
Intel Paragon	50.0	4 × 8 Mesh	2800	56.0	12	N/A	10	[22] [34]
Stanford DASH	33.0	2 × 4 4-proc clusters	480	14.5	31	120	30	[28]
Stanford FLASH	*200.0	4 × 8 Mesh	3200	16.0	62	352	40	[19]
Wisconsin T0	#200.0	none simulated	N/A	N/A	200	1461	40	[39][37]
Wisconsin T1	#200.0	none simulated	N/A	N/A	200	401	40	[39][37]
Cray T3D	150.0	4 × 2 × 2 Torus 2-proc clusters	4800	32.0	15	100	23	[40] [2]
Cray T3E	300.0	4 × 4 × 2 Torus	19200	64.0	110	300-600	80	[43][42]
SGI Origin	200.0	Hypercube 4-proc clusters	10800	54.0	60	150	61	[25][17]

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

Table 1: Parameter estimates for various 32-processor multiprocessors. *Network Latency* is for one-way network transit time of a 24-byte packet. *Remote Miss Latency* is an average of best- and worst-case write misses.

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 network congestion to the point of a network overflow software trap (see Section 3.1), 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 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

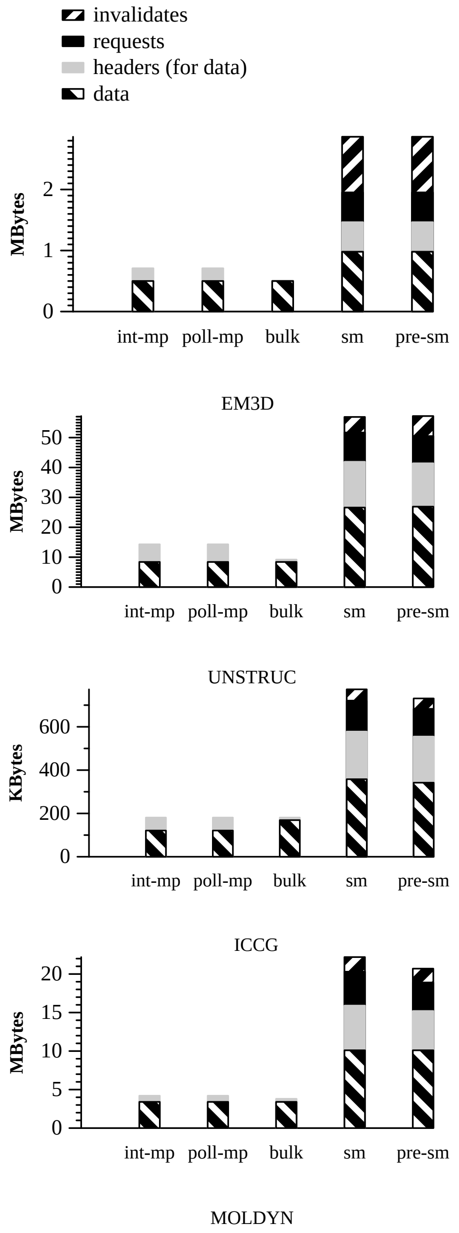


Figure 5: Breakdowns of communication volume for each communication mechanism.

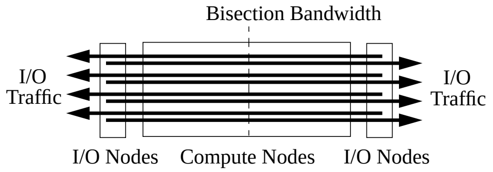


Figure 6: I/O Traffic in Cross-Traffic Experiment

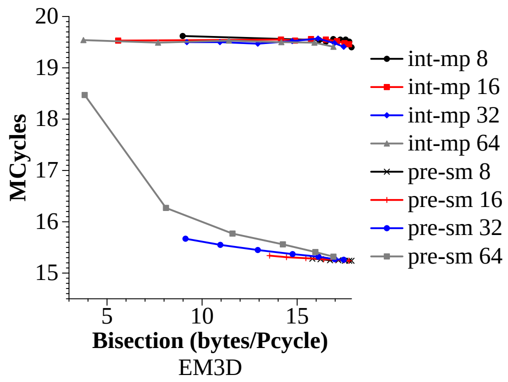


Figure 7: Sensitivity to IO-traffic message length.

bytes/Pcycle) 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 while adequate network performance is available, we can see a performance crossover 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.

[1] Note that FLASH has been redesigned to use the Origin network since [19].

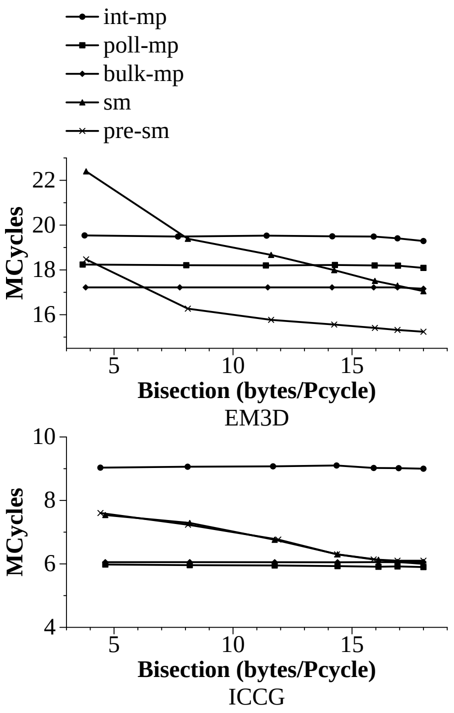


Figure 8: Execution time (in cycles) versus bisection bandwidth. Alewife is at 18 bytes/cycle (See Table 1).

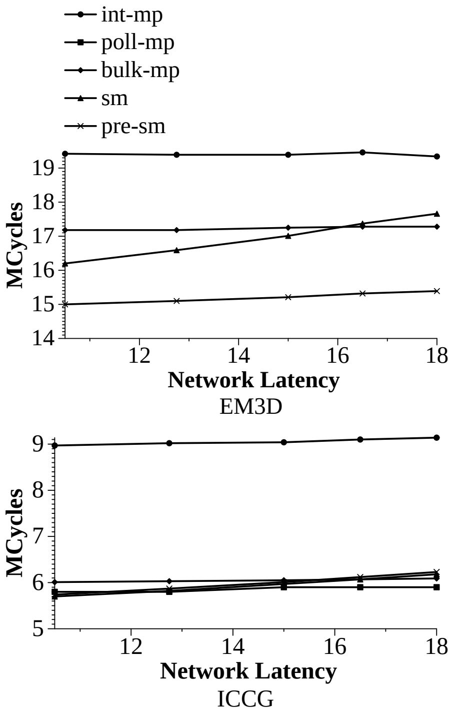


Figure 9: Network latencies emulated by varying node clock. Latency is for 1-way delivery of 24-bytes (see Table 1). Alewife is at 15.

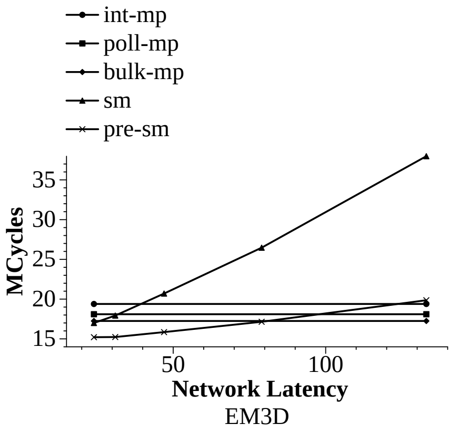


Figure 10: Network Latencies Emulated with Context-Switching.

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. Fig-

Machine	Bsctn BW	Net Lat in
(32 Processors)	bytes/lcl-miss	lcl-miss times
MIT Alewife	198	1.3
TMC CM5	310	3.1
KSR-2	900	?
MIT J-Machine	1792	1.0
MIT M-Machine	2688	0.5
Intel Delta	54	1.5
Intel Paragon	560	1.2
Stanford DASH	435	1.0
Stanford FLASH	1248	0.5
Wisconsin T0	N/A	5.0
Wisconsin T1	N/A	5.0
Cray T3D	736	0.7
Cray T3E	5120	1.4
SGI Origin	2700	1.2

Table 2: Multiprocessor parameter estimates recalculated in terms of local cache-miss latency.

ure 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 [32] have found that asynchronous implementations of applications such as EM3D are relatively insensitive to microsecond-latencies on networks of workstations.

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.

5.4 Compute- vs. Memory-bound

Processor cycles, however, are only a useful frame of reference for compute-bound applications. For many applications with irregular and large datasets, local cache-miss latency is the limiting factor to sequential performance. For these memory-bound applications, we should use local cache-miss latency as our basis for comparison. Recalculating our numbers from Table 1, we see in Table 2 that network latencies are much more comparable.

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 [8], 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 [21], 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 [38].

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 [15] increasingly important as processor speeds increase.

Woo et al. [46] compared bulk transfer with shared memory on simulations of the FLASH multiprocessor [19] running the SPLASH [18] 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 [32] 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.[35]. 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 [29], Martonosi and Gupta [33], and LeBlanc and Markatos [26]. These studies however do not compare machine implementations, as all programs ultimately used shared memory only.

Klaiber and Levy [24] 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, Ricardo Bianchini, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proc. 22nd Annual International Symposium on Computer Architecture*, June 1995.

[2] R. Arpaci et al. Empirical evaluation of the Cray T3D: A compiler perspective. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, 1995.

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

[4] M. J. Berger and S. H. Bokhari. A partitioning strategy for PDEs across multiprocessors. In *International Conference on Parallel Processing*, August 1985.

[5] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz. Remote queues: Exposing message queues for optimization and atomicity. In *1995 Symposium on Parallel Architectures and Algorithms*, Santa Barbara, California, July 1995.

[6] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In *International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[7] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.

[8] S. Chandra, J.R. Larus, and A. Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs ? In *Sixth International Conference on Architectural Support for Programming Languages*

and Operating Systems (ASPLOS VI). ACM, October 1994.

[9] Frederic T. Chong and Anant Agarwal. Shared memory versus message passing for iterative solution of sparse, irregular problems. Technical report, mit-lcs-tr-697, MIT Laboratory for Computer Science, Cambridge, MA, October 1996.

[10] Frederic T. Chong, Beng-Hong Lim, Ricardo Bianchini, John Kubiatowicz, and Anant Agarwal. Application performance on the mit alewife multiprocessor. *IEEE Computer: Special Issue on Emerging Applications for Shared-Memory Multiprocessors*, December 1996.

[11] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing*, November 1993.

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

[13] Thomas H. Dunigan. Communication performance of the Intel Tochston Delta mesh. ORNL Report ORNL/TM-11983, Oak Ridge National Laboratory, January 1992.

[14] Thorsten von Eicken et al. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, Queensland, Australia, May 1992.

[15] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill James R. Larus, Anne Rogers, and David A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing 94*, 1994.

[16] Marco Fillo, Stephen W. Keckler, W.J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, Ann Arbor, MI, November 1995. IEEE Computer Society.

[17] Mike Galles. The SGI SPIDER chip. Available from http://www.sgi.com/Technology/spider_paper/, 1996.

[18] Maya Gokhale, William Holmes, Andrew Kopser, Sara Lucas, Ronald Minnich, Douglas Sweeney, and Daniel Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1), January 1991.

[19] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *ASPLOS VI*, pages 274–285, San Jose, California, 1994.

[20] Bruce Hendrickson and Robert Leland. The Chaco user's guide. Technical Report SAND94-2692, Sandia National Laboratories, July 1995.

[21] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The effects of latency, occupancy and bandwidth on the performance of cache-coherent multprocessors. Technical report, Stanford University, Stanford, California, January 1995.

[22] Paragon XP/S product overview. Intel Corporation, 1991.

[23] Steve Keckler. Personal communication, November 1996.

[24] A. Klaiber and H. Levy. A comparison of message passing and shared memory for data-parallel programs. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, April 1994.

[25] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the International Symposium on Computer Architecture*, pages 241–251, 1997.

[26] T. LeBlanc and E. Markatos. Shared memory vs. message passing in shared-memory multiprocesors. In *Fourth IEEE Symposium on Parallel and Distributed Processing*, 1992.

[27] Charles E. Leiserson et al. The network architecture of the connection machine CM-5. In *Symposium on Parallel Architectures and Algorithms*, pages 272–285, San Diego, California, June 1992. ACM.

[28] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–80, March 1992.

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

[30] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and M. Frans Kaashoek. FUGU: Implementing Protection and Virtual Memory in a Multiuser, Multimodel Multiprocessor. Technical Memo MIT/LCS/TM-503, October 1994.

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

[32] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 85–97, 1997.

[33] M. Martonosi and A. Gupta. Tradeoffs in Message Passing and Shared Memory Implementations of a Standard Cell Router. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages III 88–96, 1989.

[34] Steven A. Moyer. Performance of the iPSC/860 node architecture. Technical Report IPC-TR-91-007, Institute for Parallel Computation, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA 22903, May 1991.

[35] S.S. Mukherjee, S.D.Sharma, M.D.Hill, J.R.Larus, A.Rogers, and J.Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Principles and Practice of Parallel Programming (PPoPP) 1995*, pages 68–79, Santa Clara, CA, June 1995. ACM.

[36] M.D. Noakes, D.A.Wallach, and W.J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *In Proceedings of the 20th Annual International Symposium on Computer Architecture 1993*, pages 224–235, San Diego, CA, May 1993. ACM.

[37] Steven K. Reinhardt. Personal communication, November 1996.

[38] Steven K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the International Symposium on Computer Architecture*, 1994.

[39] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, 1996.

[40] Steven P. Reinhardt and Winnie Williams. Cray T3D Software: Delivering the performance. 55 min. video, 1994.

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

[42] Steve Scott. Personal communication, November 1996.

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

[44] 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. *Journal of Parallel and Distributed Computing*, 27(2), June 1995.

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

[46] S. C. Woo, J. P. Singh, and J. L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, October 1994.