Integrating Message-Passing and Shared-Memory: Early Experience

David Kranz, Kirk Johnson, Anant Agarwal,
John Kubiatowicz, and Beng-Hong Lim
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper discusses some of the issues involved in implementing a shared-address space programming model on large-scale, distributed-memory multiprocessors. While such a programming model can be implemented on both shared-memory and message-passing architectures, we argue that the transparent, coherent caching of global data provided by many shared-memory architectures is of crucial importance. Because message-passing mechanisms are much more efficient than shared-memory loads and stores for certain types of interprocessor communication and synchronization operations, however, we argue for building multiprocessors that efficiently support both shared-memory and message-passing mechanisms. We describe an architecture, Alewife, that integrates support for shared-memory and message-passing through a simple interface; we expect the compiler and runtime system to cooperate in using appropriate hardware mechanisms that are most efficient for specific operations. We report on both integrated and exclusively shared-memory implementations of our runtime system and two applications. The integrated runtime system drastically cuts down the cost of communication incurred by the scheduling, load balancing, and certain synchronization operations. We also present preliminary performance results comparing the two systems.

1 Introduction

Researchers in parallel computing generally agree that it is important to support a *shared-address space* or *shared-memory* programming model—programmers should not bear the responsibility for orchestrating all interprocessor communication through explicit messages. Implementations of this programming model can be divided into two broad areas depending on the target architecture. With message-passing architectures, the shared-address space is typically synthesized by low-level system software, while traditional shared-memory architectures offload this functionality into specialized hardware.

In the following section, we discuss some of the issues involved in implementing a shared-address space programming model on large-scale, distributed-memory multiprocessors. We provide a brief overview and comparison of conventional implementation strategies for message-passing and shared-memory architectures. We argue that the transparent, coherent caching of global data provided by many shared-memory architectures is of fundamental importance, because it affords low-overhead, fine-grained data sharing. We point out, however, that some types of communication are handled less efficiently via shared-memory then they might be via message-passing. Building upon this, we suggest that it is reasonable to expect that future large-scale multiprocessors will be built upon what are essentially message-passing communication substrates. In turn, this suggests that multiprocessor architectures might be designed such that processors are able to communicate with one another via either shared-memory or message-passing interfaces, using whichever is likely to be most efficient for the communication in question.

In fact, the MIT Alewife machine [1] does exactly that—interprocessor communication can be effected either through a sequentially-consistent shared-memory interface or by way of a messaging mechanism as efficient as those found in many present day message-passing architectures. The bulk of this paper relates early experience we've had integrating shared-memory and message-passing in the Alewife machine. Section 2 overviews implementation schemes for shared-memory. Section 3 describes the basic Alewife architecture, paying special attention to the integration of the message-passing and shared-memory interfaces. Section 4 provides results comparing purely shared-memory and message-passing implementations of several components of the Alewife runtime system and two applications. We show that a hybrid thread scheduler utilizing both shared-memory and message-based communication performs up to a factor of two better than an implementation using only shared-memory for communication. Section 5 discusses related work. Finally, Section 6 summarizes the major points of the paper and outlines directions for future research.

1.1 Contributions of this Paper

In this paper, we argue for building multiprocessors that efficiently support both shared-memory and message-passing mechanisms for interprocessor communication. We argue that efficient support for fine-grained data sharing is fundamental, pointing out that traditional message-passing architectures are unable to provide such support, especially for applications with irregular and dynamic communication behavior. Similarly, we point out that while shared-memory hardware can support such communication, it is not a panacea either—we identify several other communication styles which are handled less efficiently by shared-memory than by message-passing architectures. Finally, we (i) describe how shared-memory and message-passing communication is integrated in the Alewife hardware and software systems; (ii) present some preliminary performance data obtained with a simulator of Alewife comparing the performance of software systems using both message-passing and shared-memory against implementations using exclusively shared-memory; and (iii) provide analysis and early conclusions based upon the performance data.

2 Implementing a Shared-Address Space

Implementations of a shared-address programming model can be divided into two broad areas depending on the underlying system architecture. On message-passing architectures, a shared-address space is typically synthesized through some combination of compilers and low-level system software. In traditional shared-memory architectures, this shared-address space functionality is provided by specialized hardware. This section gives a brief comparison of these implementation schemes and their impact on application performance. We conclude that the efficient fine-grained data sharing afforded by hardware implementations is of fundamental importance for some applications, but also point out that some types of communication are handled less efficiently via shared-memory hardware then they could be via explicit message-passing. In either case, we assume that the physical memory is distributed across the processors.

```
shared-address-space-reference(location)
    if currently-cached?(location) then
        // satisfy request from cache
        load-from-cache(location)
    elsif is-local-address?(location) then
        // satisfy request from local memory
        load-from-local-memory(location)
    else
        // must load from remote memory; send remote
        // read request message and invoke any actions
        // required to maintain cache coherency
        load-from-remote-memory(location)
```

Figure 1: Anatomy of a shared-address space reference.

2.1 Anatomy of a Memory Reference

When executing a multiprocessor application written assuming a shared-address space programming model, the actions indicated by the pseudocode in Figure 1 must be taken for *every* shared-address space reference. The pseudocode assumes that locally cached copies of locations can exist. (If shared-data caching is not supported, the first test is eliminated.) The most important parts of the code in Figure 1 are the two "local/remote" checks.

The local/remote check is the essence of the distinction between shared-memory and message-passing architectures. In the former, the instruction to reference memory is the same whether the object referenced happens to be in local or remote memory; the local/remote checks are facilitated by hardware support to determine whether a location has been cached (cache tags and comparison logic) or, if not, whether it resides in local or remote memory (local/remote address resolution logic). If the data is remote and not cached, a message will be sent to the remote node to access the data. Although the first test is unnecessary if local caching is disallowed, the second test is still required to implement a shared-address space programming model. Because shared-memory systems provide hardware support for detecting non-local requests and sending a message to fetch the data from the location, a single instruction can be used to access *any* shared-address space location, regardless of whether it is already cached, resident in local memory, or resident in remote memory.

Message-passing architectures, on the other hand, do not provide hardware support for these local/remote checks. Hence, they cannot use a single instruction to access locations in the shared-address space. Rather, they must implement the pseudocode of Figure 1 entirely in software.

The actions specified in Figure 1 may be performed by the programmer, compiler, runtime system, or hardware. For simplicity, assume that there are two types of applications: static and dynamic. In static applications, the control flow of the program is essentially independent of the values of the data being manipulated. Many scientific programs fit into this category. In dynamic applications, on the other hand, control flow is strongly dependent on the data being manipulated. Most real programs will lie somewhere between these two extremes, having some parts that are dynamic and others that are static. Data sharing in programs can also be fine- or coarse-grained.

Through a great deal of hard work, a programmer can often eliminate the overhead incurred by local/remote checks shown in Figure 1 by sending explicit messages. The fixed costs of such operations are typically high enough, however, that for efficient execution, the messages must be made fairly large. The result is that the granularity at which data sharing can be exploited efficiently is quite coarse. For static programs with sufficiently large-grained data sharing, these explicit messages are like assembly language: if done well, they always provide the best performance. However, as with assembly language, it is not at all clear that most programmers are better off if forced to orchestrate all communication through explicit messages. For static applications which share data at too fine a grain, even if overhead related to local/remote checks can be eliminated, performance suffers because messages typically aren't large enough to amortize any fixed overheads inherent in message-based communication. It is worth noting, however, that for architectures like iWarp [4] which in effect support extremely low overhead "messaging" for static communication patterns, this is not a problem.

Optimizations similar to those a programmer might use to eliminate the overhead of local/remote checks can be applied by a compiler. There has been a great deal of work in this area for scientific programs written in various dialects of FORTRAN and targeted at message-passing machines [5, 11, 14, 21, 24]. Of course, the constraints are the same as when the optimizations are programmer applied: applications must be primarily static and only coarse-grained data sharing can be exploited efficiently.

For dynamic applications, the compiler can't help much. It is usually not possible to know *a priori* (at compile time) whether or not a particular reference will be to local or remote memory. In such a case, the only way to achieve a shared-address space paradigm on message-passing architectures is essentially by executing something like the code in Figure 1 for every shared-address space reference in an application. This is typically done by having a low-level software layer that synthesizes a global address space, possibly also maintaining a cache of some sort. This software layer adds significant overhead to every shared-address space reference, *even when no communication is necessary*. Herein lies the primary advantage of shared-memory architectures: unlike shared-address space implementations on message-passing architectures, a shared-memory architecture will perform well on dynamic programs that re-use data because all data is automatically cached. If the data is not in the cache, but is in local memory, it will be automatically fetched into the cache. To get good performance on a message-passing architecture, the programmer or compiler must handle caching explicitly in order to generate the proper code, *i.e.* the program must be static. It is for these reasons that we believe that some hardware support for shared-memory programs is important; the fundamental hardware mechanism is coherent caching of global addresses.

Note that from this perspective, systems like [22] and Active Messages [23] provide just an efficient messaging interface—an implementation of a shared-address space programming model on top of such systems will face the same problems as those described above for implementations on message-passing architectures.

2.2 Defects of Shared-memory

The previous section discussed the aspects of application behavior that benefit from hardware support for shared-memory: dynamic memory reference behavior and fine-grain data sharing. The support for these behaviors have a cost, however, when the data sharing is coarse-grained or something is known statically about how shared data will be accessed. We enumerate below three scenarios in which we expect shared-memory to be less effective than message based communication:

Coarse granularity When a large chunk of data is to be shared, the cost of transferring the data from one processor to another must be incurred. The most efficient mechanism is a block transfer where all of the data is sent in a single message. If the copy is done with loads and stores of shared-memory, more network and memory bandwidth is required because of the fixed overhead associated with each shared-memory transaction. Section 4 contains several examples and suggests why it might be better sometimes to use shared-memory for copying in spite of these costs.

Known communication patterns Coherent caches on a multiprocessor work well when data is either read-only or is accessed many times on a single processor before being accessed by other processors. Frequent writes on different processors can cause poor performance because they make caching ineffective and lead to additional overhead due to invalidations or updates that must be sent over the network. Furthermore, in many cache coherence protocols, in order to acquire a cache line that is dirty in another processor's cache, that data must be communicated through a home or intermediate node instead of being passed directly to the requester. This is less efficient than communicating via direct point-to-point messages, as is possible when communication patterns are well known.

Combining Synchronization with Data Transfer A purely shared-memory implementation typically generates separate messages for communicating synchronization events and for transferring any associated data. Such transfers often occur during producer-consumer communication. While the latency associated with data transfer can often be tolerated through mechanisms like weak ordering and prefetching, the latency associated with the synchronization signal itself is hard to overlap. The central problem in overlapping the latency of synchronization signals lies in the inability of the consumer to predict exactly when to request a synchronization object; premature prefetching of the synchronization object can lead to even worse performance. A message from the producer to the consumer bundling both synchronization and data and informing the consumer of the availability of data is the natural mechanism in such situations. Section 4.3 provides an example of this.

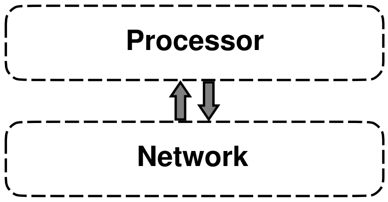

Figure 2: A message-passing interface.

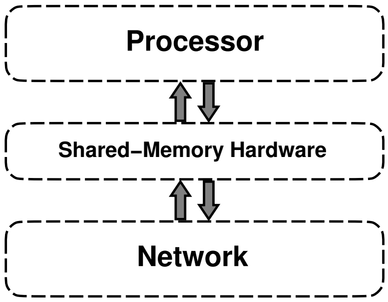

Figure 3: A shared-memory interface.

We argue that to get good performance with a tractable programming model on a wide variety of applications, an architecture must integrate hardware support for both shared-memory *and* message-passing. The software system and/or programmer can then choose the appropriate mechanism based on cost. Such an architecture has been designed at MIT. The next section describes how shared-memory and message-passing are integrated in the Alewife machine. We then investigate the extent to which we can use Alewife's message-passing functionality to address the defects of shared-memory described above.

3 Architectural Interfaces for Messages

Most distributed shared-memory machines are built on top of an underlying message-passing substrate [1, 9, 13]. Therefore any additional cost to support both message-passing in addition to shared-memory must only be paid at the processor-network interface. In this section, we describe the manner in which Alewife integrates shared-memory and message-passing communication facilities; we provide this detail in part to demonstrate that the complexity of integrating both message-passing and shared-memory is not unreasonable. See [12] for additional details.

Figures 2 and 3 show typical interfaces provided by message-passing multicomputers and shared-memory machines, respectively.

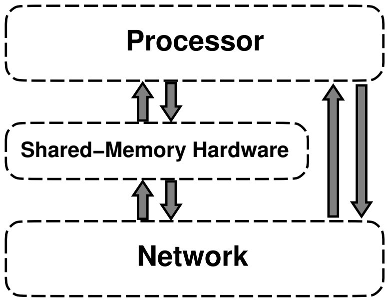

Figure 4: Alewife's integrated interface.

Operand 0
Operand 1
⋮
Operand m-1
Address 0
Length 0
Address 1
Length 1
⋮
Address n-1
Length n-1

Figure 5: Packet descriptor format.

In the message-passing version, the processor has direct access to the network and can send (or receive) packets directly into (or from) the network. Messages from the network are placed into queues accessible by the processor; the contents of these queues can then be accessed via conventional loads and stores or register ops, depending on whether the queues are memory- or register-mapped.

Shared-memory interfaces, on the other hand, provide a hardware layer between the network and the processor. This layer typically translates certain loads and stores into messages to other nodes; it models the shared-memory programming paradigm by abstracting away the communication layer. The presence of this layer prevents the processor from directly accessing the network.

Recognizing that shared-memory machines are built on top of a message-passing substrate, Alewife integrates direct network access with the shared-memory framework as depicted in Figure 4. In a sense, the machine allows controlled *breaking of the shared-memory abstraction* by the programmer, runtime system, or compiler when doing so can yield higher performance. In fact, Alewife's coherence protocol, LimitLESS directories [6], relies on the processor's ability to directly send and receive coherence packets into the network.

With Alewife's integrated interface, a message can be sent with just a few user-level instructions. A processor receiving such a message will trap and respond either by rapidly executing a message handler or queuing the message for later consideration when an appropriate message handler gets scheduled. Scheduling and queuing decisions are made entirely in software.

The Alewife hardware provides the following high-level interface for integrating messages and coherent shared-memory to the software system. The runtime system provides other higher-level abstractions built upon this interface (*e.g.* remote thread invocation and barrier synchronization) to compiler writers and application programmers.

1. *Coherent shared-memory loads and stores:* These operations, implemented as single processor instructions, comprise the traditional shared-memory operations. They invoke suitable coherence actions when multiple cached copies of a memory location exist.

2. *Processor-to-processor message:* On the source side, this operation allows a processor to write packets directly into a network queue; it is implemented using a sequence of processor instructions described below. At the destination, message arrival interrupts the processor and invokes a handler which can examine the data in the packet. It takes 5 cycles to get into the message handler. These messages can be used for efficient register-to-register communication between processors.

3. *Source-and-destination-coherent data transfer:* A processor executing this operation describes a region of memory on the local (source) node to be sent in a single message to some region of memory on a remote (destination) node. The data transfer is achieved in a way that leaves the source and destination caches consistent with their respective local memories. Such a transfer takes no action on copies of the source or destination data present in other caches. These messages can be used for coarse-grained bulk (memory-to-memory) data transfer and invoke DMA facilities at the source and destination.

The processor-to-processor message and the data-transfer mechanisms are both implemented using a single low-overhead interface to the network. In fact, the interface describes a generic message with the combined characteristics of processor-to-processor messages and bulk data transfer.

The network interface includes specifications at the source and at the destination. At the source, the network interface permits messages to be sent through a two phase process: *describe* then *launch*. Description proceeds by writing directly to registers on Alewife's network coprocessor, or *Communications and Memory-Management Unit* (CMMU). These writes proceed at the same speed as cached writes. The resulting descriptor, depicted in Figure 5, can be up to 16 words long. The descriptor consists of a variable number of explicit *operands* that will be placed at the head of the message followed by a number of address-length pairs describing data to be taken directly from memory and concatenated to the end of the packet. The first operand must specify the destination and a message type, while the remaining words are software defined.

Once a packet has been described, it is launched via an *atomic*, single-cycle instruction. The encoding of the launch instruction specifies both the number of explicit operands and the number of address-length pairs. Processor-to-processor messages specify only explicit operands, while bulk data transfer messages utilize only address-length pairs. The multiple address-length pairs allow multiple source regions of memory to be sent in a single packet. A single message with both explicit operands and multiple address-length pairs can also be sent. Both the user and supervisor are permitted to send messages.

On the destination side, for efficient reception of messages, the Alewife interface provides a 16-word, sliding window into the network input queue. On reception of a message, the CMMU interrupts the processor (unless the processor has masked message interrupts) while making the first 16 words of the packet visible in the reception window. The processor can examine words within this window by reading coprocessor registers; as with the output interface, these reads complete at the speed of a cached memory access.

Once the processor has examined the packet, it can execute a special coprocessor storeback instruction to remove data from the window. User code may dispose of user-generated messages. Two separate fields are encoded directly in the storeback instruction. First is the number of words to be discarded from the head of the window. Second is the number of words (following those discarded) to be stored to memory via DMA. If this option is chosen, the processor must write the starting address for DMA to a special controller register before issuing the storeback instruction. Multiple storeback instructions can be issued for a single packet to scatter it to memory. Either of these storeback fields can contain a special "infinity" value which denotes "until the end of the packet".

The Alewife CMMU has been completely implemented and is in the final stages of testing. We expect to have small prototype Alewife systems (four to 16 nodes) operational sometime this year. These systems will utilize nodes clocked at 33 MHz, coherent caches, and a two-dimensional mesh interconnect.

4 Results

This section provides results comparing the performance of the shared-memory and message-passing implementations of several library routines, our runtime system, and one application.

4.1 Experimental Environment

The results presented in this paper were obtained through the use of a detailed machine simulator. This simulator provides cycle-by-cycle simulation of all components of Alewife. Our original simulator implementation targeted desktop SPARC- and MIPS-based workstations; we have also developed a version of the simulator which runs on Thinking Machines' CM-5 multiprocessors. In the latter implementation, each CM-5 node simulates the processor, memory, and network hardware of one or more Alewife nodes. The CM-5 port of our simulator has proved invaluable, especially for running simulations of large Alewife systems (64 to 512 nodes).

4.2 Combining Tree Barriers

In order to implement barrier synchronization on Alewife, we use a *combining tree* scheme [16]. In such a scheme, a k-ary tree with n leaves is laid out across the n processors participating in the barrier such that exactly one tree leaf resides on each processor. Upon entering the barrier, a processor activates the leaf node of the tree, which in turn sends an *arrival* signal to its parent. Each internal node issues an arrival signal to its parent after arrival signals have been received from all its children. When the root of the tree has received arrival signals from all its children, it then issues *wakeup* signals to all its children. Internal nodes pass such wakeup signals on to all of their children. Once a wakeup signal arrives at a leaf node, the user thread which had been running on that processor before the barrier is allowed to continue execution.

In principle, only a single message need be sent to signal each combining tree arrival or wakeup. In a cache-coherent shared-memory implementation, arrivals and wake-ups are signaled via memory writes; even in a carefully tuned implementation, these writes require several messages each. By utilizing explicit messaging, the ideal of a single message per event is readily achieved. The benefit of doing so is substantial. On a 64-processor machine, our best shared-memory barrier implementation (utilizing a six-level binary tree carefully crafted to minimize the total number of message exchanges) executes in about 1650 cycles (50 μsec); a direct message-based implementation (utilizing a two-level eight-ary tree) takes only 660 cycles (20 μsec). By comparison, typical software implementations (e.g. Intel DELTA and iPSC/860, Kendall Square KSR1) take well over 400 μsec [9].

4.3 Remote Thread Invocation

To invoke a thread on a remote processor, a pointer to the thread's code and any arguments must be placed atomically on the task queue of another processor. To do so via shared-memory, the invoking processor must first acquire the remote task queue lock (which requires at least one network round-trip, if not more) and then modify and unlock the queue using shared-memory reads and writes, each of which can require multiple network messages. A message-based implementation is substantially simpler: all of the information necessary to invoke the thread is marshaled into a single message which is unpacked and queued atomically by the receiving processor. In this manner, we combine synchronization and data transfer in a single message, as suggested in Section 2.2.

We characterize the performance of these two implementation schemes by measuring two intervals: $T_{invoker}$, the time from when the invoking processor starts the operation until it is free to proceed with other work, and $T_{invokee}$, the time from when the invoking processor starts the operation until the invoked thread begins running (see Figure 6). With our best shared-memory implementation, these times are 353 and 805 cycles, respectively (10.7 and 24.4 μsec). With the message-based implementation, both times are reduced drastically, to 17 and 244 cycles, respectively (0.5 and 7.4 μsec). Note that for both implementations, these times were measured in the context of a complete thread scheduling and migration system; $T_{invokee}$ would be substantially smaller in a minimal system implementation.

4.4 Bulk Data Transfer

Memory-to-memory copy of data blocks between processors, as might be used to perform buffered disk I/O or when relocating large data objects, can be implemented using either shared-memory or message-passing. Figure 7 compares the performance of three different implementations of memory-to-memory copy. The first two implementations (*no-prefetching* and *prefetching*) utilize hand-coded inner loops to copy data between buffers using doubleword

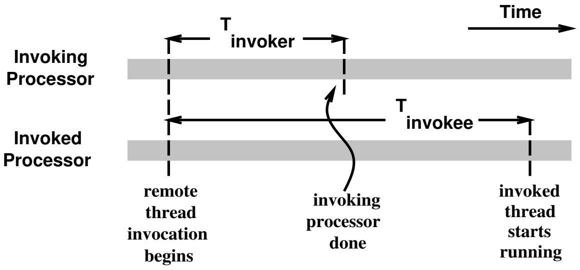

Figure 6: Remote thread invocation.

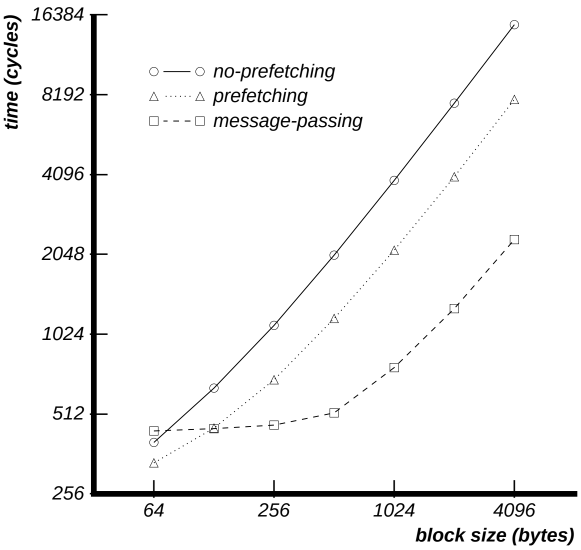

Figure 7: Memory-to-memory copy performance; see Section 4.4 for details.

(eight byte) loads and stores through the shared-memory interface (*no-prefetching* uses a simple copy loop; *prefetching* uses essentially the same loop but also prefetches one cache block (16 bytes) ahead). The third implementation (*message-passing*) copies data between buffers using a single message and the CMMU's DMA facilities.

As expected, *message-passing* performs substantially better than *no-prefetching* or *prefetching*, even for relatively small block sizes. With 256-byte blocks, *message-passing* is roughly 1.5 and 2.4 times faster than *no-prefetching* and *prefetching*, respectively (17.3 vs. 11.7 and 7.3 Mbyte/sec). The benefit of message-passing grows with larger block sizes, reaching a peak rate with four-kilobyte blocks more than 3 and 6 times faster than *no-prefetching* and *prefetching*, respectively (55.4 vs. 16.4 and 8.6 Mbyte/sec).

In some situations, when an application fetches a large block of data from a remote node, it will immediately "consume" that data. A particularly simple example of this is accum, a toy application which computes the sum of a linear array of integers which resides on a remote node. Figure 8 compares the performance of shared-memory and message-passing implementations of accum. The shared-memory implementation uses a straightforward inner loop which prefetches one cache block ahead. The message-passing implementation first transfers the entire array into local memory (using the memory-to-memory copy mechanism described above)

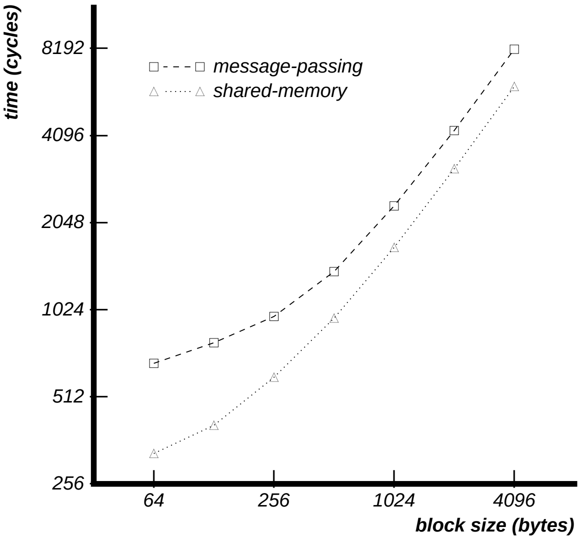

Figure 8: accum performance; see Section 4.4 for details.

and then performs the necessary computation entirely out of local memory.

In this case, the message-passing implementation runs substantially slower than the shared-memory implementation—ranging from roughly twice as slow for small blocks to about 1.3 times slower for large blocks. Given that the message-passing implementation serializes communication and computation, this is hardly surprising. To be fair, it might be reasonable to discount the time spent transferring data from the remote node, for during that time the local processor is idle and could be doing other useful work. Discounting this time amounts to subtracting the message-passing curve from Figure 7 from that in Figure 8; doing so provides a curve which rides just below the shared-memory curve in Figure 8. This is also not surprising, given that the "compute" phase of the message-passing implementation uses almost exactly the same inner loop as the shared-memory implementation—the only difference being that the shared-memory implementation has one additional instruction (a prefetch) per loop iteration.

From this last observation we can conclude that even if we were able to utilize the processor idle time during message transfer (perhaps by breaking large data blocks into smaller blocks and pipelining the transfer of and computation on those data blocks), the message-passing implementation might perform better than the shared-memory implementation, but only by a very small amount. We observe this effect with accum because:

- accum doesn't store the remote data it fetches for later use—array elements are only fetched such that they can be accumulated into the global sum.
- accum is a "static" application which accesses data in a particularly regular fashion—this allows data to be prefetched into the cache such that virtually all actual accesses to remote data hit in the cache, effectively hiding all communication latency.

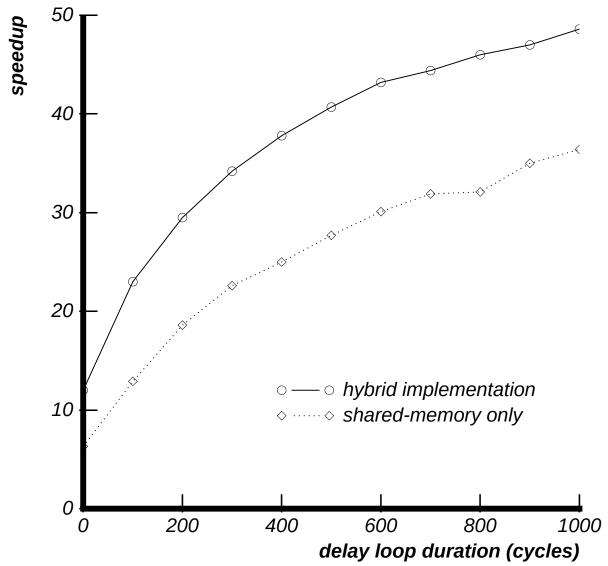

Figure 9: grain performance on 64 processors; see Section 4.5 for details.

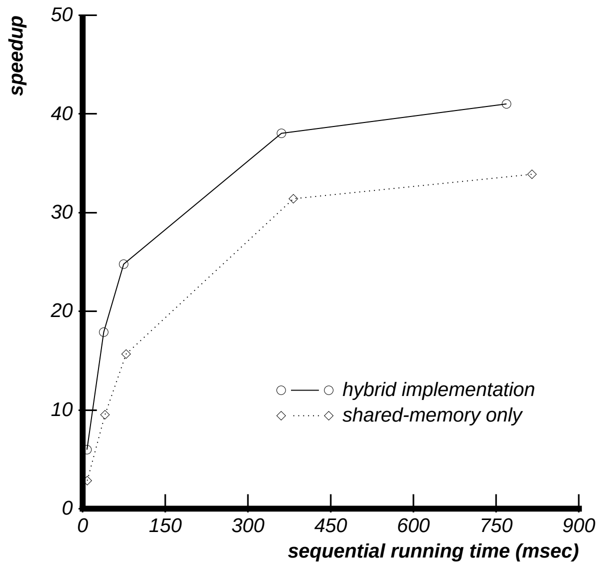

Figure 10: aq performance on 64 processors; see Section 4.5 for details.

4.5 Thread Scheduler

The Alewife runtime system is based on *lazy task creation* [17]. This system performs the tasks of load-balancing and dynamic partitioning of programs that cannot be statically partitioned effectively. As part of this research, we have implemented a version of this runtime system that uses message-based communication in both searching for work and thread migration. When compared to the original (shared-memory only) implementation, this hybrid (shared-memory + message-passing) implementation drastically cuts down on the communication overheads incurred by thread scheduling and load balancing operations. We benchmarked both the original shared-memory and the hybrid implementations using one synthetic application (grain) and an adaptive numerical integration code (aq).

As described in [17], the grain application enumerates a complete binary tree of depth n and sums the values found at the leaves using a recursive divide-and-conquer structure. Before obtaining the value found at each leaf, a delay loop of l cycles in duration is executed. Figure 9 compares the speedup for grain obtained with the two scheduler implementations running on 64 processors for $n = 12$ and a wide range of l. By choosing $n = 12$, we assure a sufficient amount of parallelism (4,096 leaf tasks for 64 processors); by varying l we can vary the "grain size" of that parallelism.

For very small grain size ($l = 0$, corresponding to a sequential running time[1] of 7.1 milliseconds), the lower searching and thread migration overheads afforded by the hybrid implementation scheme yield considerable benefit—grain runs almost twice as fast under the hybrid implementation (speedups of 12.0 vs. 6.3 for the hybrid and shared-memory implementations, respectively). As grain size l is increased, the relative benefit of the hybrid implementation decreases. This is as one would expect, since the fraction of time spent in "overhead" decreases with increased grain size. Even so, for relatively large grain size ($l = 1000$ cycles, corresponding to a sequential running time of 131.2 milliseconds) grain running under the hybrid scheme still runs around 33 percent faster than under the original shared-memory implementation (speedups of 48.6 and 36.4, respectively).

The aq application computes a numerical integration of a bivariate function over a rectangular domain. Like grain, aq utilizes a recursive divide-and-conquer structure, dividing space and recursing more deeply in those regions that are not sufficiently smooth at the current scale. Since the scale at which many integrands become sufficiently smooth can vary significantly across the domain of integration, the call tree resulting from this recursion is often relatively irregular. In the results presented here, we hold the integrand and domain of integration fixed; problem size is increased by changing the threshold for what is to be considered sufficiently smooth. Figure 10 compares the speedup for aq obtained with the two scheduler implementations running on 64 processors; sequential running time is used on the x-axis as a measure of problem size.

Once again, the hybrid scheduler implementation outperforms the shared-memory implementation by roughly a factor of two for small problem sizes. As problem size (and thus grain size) is increased, the relative advantage of the hybrid implementation decreases, although for the largest problem size shown in Figure 10 (sequential running time of around 800 milliseconds), the hybrid scheduler implementation still yields over 20 percent better performance than the original shared-memory implementation.

4.6 Jacobi SOR

Finally, we have developed both shared-memory and message-passing implementations of jacobi, a simple block-partitioned Jacobi SOR solver. In this application, processors only communicate with one another to exchange new values for their border elements. In the shared-memory implementation, this communication is ef-

[1] In this paper, the sequential running time of an application is taken to be the running time of that application when compiled for and run on a single node. Thus sequential running time doesn't include any scheduler or runtime system overhead. The speedup results shown in Figure 9 and Figure 10 are computed with respect to sequential running time.

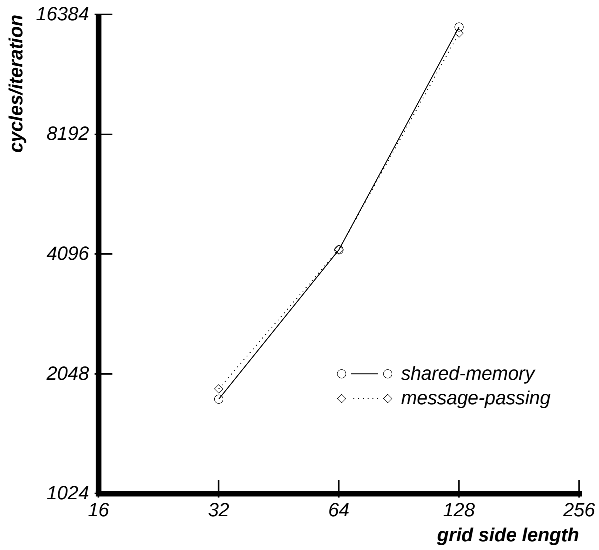

Figure 11: Jacobi SOR performance on 64 processors; see Section 4.6 for details.

fected through conventional shared-memory operations (without prefetching); in the message-passing implementation, all communication is accomplished through the message-based memory-to-memory copy mechanism described in Section 4.4. Figure 11 compares the performance obtained with both implementations for grid sizes of 32x32, 64x64, and 128x128 running on 64 processors.

In jacobi, the amount of data communicated with each neighboring processor on each iteration is proportional to the side length of the local grid block. Given the memory-to-memory copy and accum results from Section 4.4, it is not surprising that the performance of the shared-memory and message-passing implementations differs by such a small amount. With small grid sizes, the shared-memory implementation of jacobi performs slightly better than the message-passing implementation. Since only a small amount of data is transferred between neighboring processors when grid size is small, this follows directly from Figure 7, where we see that using shared-memory to copy small blocks of data between nodes is more efficient than using message-passing. For large grid sizes, the message-passing implementation of jacobi wins out by a small amount. In this case, much of the benefit of using messages to copy large blocks of data is masked by increasing computation-to-communication ratio—because computation-to-communication ratio in jacobi scales with problem size, efficient communication becomes less important for large problems.

5 Related Work

There have been several papers on comparing the performance of message-passing and shared-memory for particular applications. Martonosi and Gupta [15] compare the performance of a message-passing and a shared-memory implementation of a standard cell router. However, their comparison focused on the two *programming styles*, and not on the *architectural mechanisms* provided by the two styles. In contrast, our work focuses on mechanism, not programming style. We argue for the use of a shared-memory programming model, but consider a machine which integrates shared-memory communication mechanisms with a fast message mechanism. Our performance results compare the sole use of shared-memory mechanism with an integrated approach that allows the use of both shared-memory and message-passing mechanisms.

To our knowledge, there are no existing machines that support both a shared-address space and a general fine-grain messaging interface in hardware. In some cases where we argue messages are better that shared-memory, such as the barrier example, a similar effect could be achieved by using shared-memory with a weaker consistency model. For example, the Dash multiprocessor [13, 10] has a mechanism to deposit a value from one processor's cache directly into the cache of another processor, avoiding cache coherence overhead. This mechanism might actually be faster than using a message because no interrupt occurs, but a message is much more general.

Some shared-memory machines have implemented message-like primitives in hardware. For example, Beck, Kasten, and Thakkar [2] describe the implementation of SLIC—a system link and interrupt controller chip—for use with the Sequent Balance system. Each SLIC chip is coupled with a processing node and communicates with the other SLIC chips on a special SLIC bus that is separate from the memory system bus. The SLIC chips help distribute signals such as interrupts and synchronization information to all processors in the system. Although similar in flavor to this kind of interface, the Alewife messaging interface is built to allow direct access into the same scalable interconnection network used by the shared-memory operations.

Another example of a shared-memory machine that also supports a message-like primitive is the BBN Butterfly, which provides hardware support for block transfers. In an implementation of distributed shared-memory on this machine, Cox and Fowler [7] conclude that an effective block transfer mechanism was critical to performance. They argue that a mechanism that allows more concurrency between processing and block transfer would make a bigger impact. It turns out that Alewife's messages are implemented in a way that allows such concurrency when transferring large blocks of data. Furthermore, the Butterfly's block transfer mechanism is not suited for more general uses of fine-grain messaging because there is no support in the processor for fast message handling.

Finally, it is worth noting that some message-passing machines also provide limited support for the shared-address space programming model. For example, the J-machine [8] provides a global name space for objects and an object name cache which speeds up (but does not eliminate) the local/remote checks suggested in Figure 1.

6 Conclusions and Future Work

In this paper, we argue that multiprocessors should provide efficient support for both shared-memory and message-passing communication styles. We contend that implementations of a shared-address space programming model on traditional message-passing architectures can only efficiently execute static applications that exhibit coarse-grained data sharing. Shared-memory architectures with hardware support for coherent caching of global data circumvent these problems, allowing efficient execution of dynamic applications and applications that share data at a fine grain. Unfortunately, in traditional shared-memory architectures, shared-memory is the *only* communication mechanism available, even if a compiler, runtime system, or programmer knows that an operation could be accomplished more efficiently via explicit messaging; we identify several such scenarios.

We describe how shared-memory and message-passing communication are integrated in the Alewife hardware and software systems. We present preliminary results comparing the performance of various runtime system primitives and one complete application when implemented using exclusively shared-memory or a hybrid approach in which communication is effected through message-passing when appropriate. These results show performance gains on the order of two- to ten-fold for some of the runtime system primitives and up to a factor of two for applications running under the hybrid thread scheduler. We also show that in some cases message-based communication is not as efficient as that through shared-memory. From these observations, we draw the following general conclusions about the suggested "defects of shared-memory" of Section 2.2:

- When transferring large blocks of data between two nodes, using message-based communication to do so can yield a factor of three or more performance improvement over doing so via shared-memory, even if prefetching is used. If transferred data is consumed immediately in a regular fashion and not stored for later use (*e.g.* accum), judicious use of prefetching can eliminate any advantage of using messaging instead of shared-memory for data transfer.
- Message-based communication is only significantly more effective than shared-memory for applications with known communication patterns when (i) problem size can be scaled such that messages are large enough to amortize any (even modest) fixed messaging overhead and (ii) scaling problem size in such a manner does not increase the computation-to-communication ratio to the point where communication costs become insignificant (*e.g.* jacobi).
- When message-based communication can be used to bundle synchronization operations with data which must be operated on after successful synchronization, doing so is likely to yield substantial performance gains over shared-memory implementations of the same functionality (*e.g.* remote thread invocation).

As of this writing, we have only used explicit messaging in parts of the Alewife runtime system and software libraries. We plan to continue investigating further integration, including compile-time "communication optimizations" and programming systems which provide limited programmer access to both the shared-memory and message-passing interfaces. In addition, we note that a shared-object space with messages is the basis for implementing a parallel object-oriented language. In this sense shared-memory and message-passing might be integrated at the language level by integrating object-oriented and procedural programming as in T [18, 19], CLOS [3] and, more recently, Dylan [20].

7 Acknowledgments

The research reported on herein was funded in part by NSF grant # MIP-9012773, in part by DARPA contract # N00014-87-K-0825, and in part by a NSF Presidential Young Investigator Award. We would also like to thank Alan Mainwaring, Dave Douglas, and Thinking Machines Corporation for their generosity and assistance in porting our simulation system to the CM-5.

References

[1] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[2] Bob Beck, Bob Kasten, and Shreekant Thakkar. VLSI Assist for a Multiprocessor. In *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Washington, DC, October 1987. IEEE.

[3] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. *ACM SIGPLAN Notices*, 23, September 1988.

[4] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 70–81, June 1990.

[5] David Callahan and Ken Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Journal of Supercomputing*, 2(151-169), October 1988.

[6] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[7] A. Cox and R. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989. Also as a Univ. Rochester TR-263, May 1989.

[8] William J. Dally et al. The J-Machine: A Fine-Grain Concurrent Computer. In *IFIP Congress*, 1989.

[9] Thomas H. Dunigan. Kendall Square Multiprocessor: Early Experiences and Performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, March 1992.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990. IEEE.

[11] K. Knobe, J. Lukas, and G. Steele Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[12] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. Submitted for publication. Also available as MIT Laboratory for Computer Science Tech Memo, 1993.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[14] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2:361–376, July 1991.

[15] 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.

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

[17] E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[18] J. Rees and N. Adams. T: A Dialect of LISP. In *Proceedings of Symposium on Lisp and Functional Programming*, August 1982.

[19] J. Rees, N. Adams, and J. Meehan. The T Manual, Fourth Edition. Technical report, Yale University, Computer Science Department, January 1984.

[20] Apple Computer Eastern Research and Technology. *Dylan: an object-oriented dynamic language*. Apple Computer, Inc., 1992.

[21] Anne Rogers and Keshav Pingali. Process Decomposition through Locality of Reference. In *SIGPLAN '89, Conference on Programming Language Design and Implementation*, June 1989.

[22] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4), April 1982. Pages 246–260.

[23] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, May 1992.

[24] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(1), 1988.