Multi-Protocol Active Messages on a Cluster of SMP's

(to appear in the Proceedings of SC97)

Steven S. Lumetta, Alan M. Mainwaring, and David E. Culler
Computer Science Division
University of California at Berkeley
{stevel,alanm,culler}@CS.Berkeley.EDU

August 25, 1997

Abstract

Clusters of multiprocessors, or Clumps, promise to be the supercomputers of the future, but obtaining high performance on these architectures requires an understanding of interactions between the multiple levels of interconnection. In this paper, we present the first multi-protocol implementation of a lightweight message layer—a version of Active Messages-II running on a cluster of Sun Enterprise 5000 servers connected with Myrinet. This research brings together several pieces of high-performance interconnection technology: bus backplanes for symmetric multiprocessors, low-latency networks for connections between machines, and simple, user-level primitives for communication. The paper describes the shared memory message-passing protocol and analyzes the multi-protocol implementation with both microbenchmarks and Split-C applications. Three aspects of the communication layer are critical to performance: the overhead of cache-coherence mechanisms, the method of managing concurrent access, and the cost of accessing state with the slower protocol. Through the use of an adaptive polling strategy, the multi-protocol implementation limits performance interactions between the protocols, delivering up to 160 MB/s of bandwidth with 3.6 microsecond end-to-end latency. Applications within an SMP benefit from this fast communication, running up to 75% faster than on a network of uniprocessor workstations. Applications running on the entire Clump are limited by the balance of NIC's to processors in our system, and are typically slower than on the NOW. These results illustrate several potential pitfalls for the Clumps architecture.

1 Introduction

Clusters of multiprocessors, or "Clumps," promise to be the supercomputers of the future [2, 31], but obtaining high performance on these architectures requires an understanding of interactions between the multiple levels of interconnection. In this paper, we develop and measure a multi-protocol Active Message layer using the Sun Gigaplane memory system [27] and the Myricom network [4]. Our system is the first implementation of a lightweight message layer to transparently handle multiple communication protocols. The uniform, user-level communication abstraction that results serves as a powerful building block for applications on Clumps. This research brings together several pieces of high-performance interconnection technology: bus backplanes for symmetric multiprocessors, low-latency networks for connections between machines, and simple, user-level primitives for communication.

Several groups have studied the problem of programming Clumps [3, 6, 11, 13, 15, 16]. Some of these efforts focus on issues related to shared virtual memory [17, 32], but most relate to high-level message libraries such as MPI. The software overheads associated with memory allocation and tag matching in traditional message-passing libraries often obscure the machine-level performance interactions and design issues associated with the actual communication. Very little work has addressed the complicated set of tradeoffs involved in implementing a fast communication layer on the combination of cache-coherent memory and a low-latency network.

Three aspects of the communication layer are critical to performance on Clumps: the arrangement of data to reduce cache-coherence transactions, the management of concurrent access to communication data structures, and the confinement of adverse interactions between communication protocols. To quantify these dimensions, we have built a multi-protocol implementation of Active Messages-II [21] that transparently directs message traffic through the appropriate medium, either shared memory or a high-speed network. The implementation operates on a cluster of four Sun Enterprise 5000 servers running the Solaris 2.5 operating system and interconnected by a Myrinet with multiple NIC's per SMP.

This paper describes the shared memory message-passing protocol and analyzes the effects of the hardware and software architectures on communication performance. Using both microbenchmarks and a range of applications, we illuminate the important design tradeoffs for a multi-protocol communication layer. While Clumps have a clear engineering advantage over networks of uniprocessor workstations as the base architecture for large-scale systems, several obstacles limit their performance advantages at the application level. The paper brings several of these problems to light and discusses their implications.

The remainder of the paper is organized as follows: in the next section, we describe the hardware and software architectures of the system; Section 3 then details the data structures and operations used for communication through shared memory; in Section 4, we apply microbenchmarks to evaluate base performance characteristics; Section 5 compares the performance of several applications running on a Clump and on a network of workstations; Section 6 provides information about related work; and Section 7 presents our conclusions.

2 Background

In this section, we describe the relevant features of the hardware and software architectures on which the multi-protocol AM-II implementation operates.

2.1 Hardware Architecture

The experimental platform appears in Figure 1. Each Enterprise 5000 server contains eight 167 MHz UltraSPARC processors with 512 kB of L2 cache per processor and a total of 512 MB of main memory. Three Myricom M2F network interface cards (NIC's) on independent SBUS's provide access to a high-speed network connecting the servers. Each NIC contains 256 kB of SRAM and employs a 37.5 MHz LANai processor to manage transfers between the host and the network. Links in the network provide 160 MB/s of bidirectional bandwidth; bandwidth between the host and the NIC is limited by the SBUS to an observed maximum of 38 MB/s.

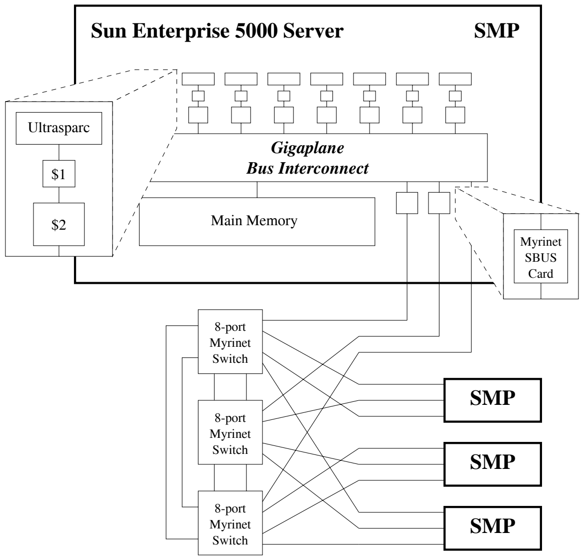

Figure 1: Target architecture for the multi-protocol AM-II implementation—a cluster of symmetric multi-processors, or Clump. Processors inside an SMP are connected via Sun's Gigaplane Interconnect, which provides cache-coherency and delivers up to 2.7 GB/s of bandwidth [27]. Communication between SMP's utilizes multiple, independent SBUS connections to a Myrinet high-speed network with internal link bandwidths of 160 MB/s [4].

The critical components of hardware performance are the memory hierarchy and the network, as these characteristics have direct impact on the speed at which data moves from one processor to another. The cost of synchronization primitives is also pertinent when managing simultaneous access by multiple processes. Using microbenchmarks based upon those of Saavedra-Barrera [25], we are able to measure these values for our system, as shown in Table 1. For comparison, the table also gives the parameters for an UltraSPARC Model 170 workstation, which uses the same processor as the Enterprise 5000. For message-passing via shared memory, the latency of accesses to data within another processor's L2 cache is of particular interest, as it represents the minimum cost to transfer data between two processors. On the Enterprise 5000, such a transfer requires 80 cycles, significantly more than the base memory latency of 50 cycles.

2.2 Software Architecture

Active Messages are a well-known model of communication in the parallel programming community and are typically among the fastest methods of commu-

	Enterprise 5000 server	UltraSPARC 170 workstation
L2 size	512 kB	256 kB
L2 miss (memory)	50 cycles	42 cycles
L2 miss (other L2)	80 cycles	N/A
memcpy bandwidth	200 MB/s	160 MB/s
NIC 32-bit read	152 cycles	114 cycles
NIC 32-bit write	54 cycles	34 cycles
SBUS bandwidth	3x38 MB/s	38 MB/s
compare-and-swap	15 cycles	15 cycles

Table 1: Selected memory, network, and synchronization primitive parameters for our Enterprise 5000 servers and UltraSPARC Model 170 workstations. Both use 167 MHz processors. The uniprocessor boasts lower memory latency but provides lower memory bandwidth as well. Note that write latency for NIC memory is usually hidden by the write buffer.

nication available [22, 26, 28, 29, 30]. Each active message contains a reference to a handler routine. When a message is received, the communication layer passes the data to the handler referenced by the message, typically as formal parameters. The association between message arrival and the execution of a particular block of code is the origin of the term "active message."

Most Active Message implementations assume the use of a SPMD model of programming and the availability of a reliable network that is space-shared between users or is time-shared with long, globally-scheduled time slices. The AM-II specification [21] defines a uniform communication interface that provides the functionality required for general-purpose distributed programming yet permits implementations yielding performance close to that available at the hardware level. AM-II abstracts communication into point-to-point messages between communication *endpoints*. A group of communicating endpoints form a *virtual network* with a unique protection domain. Traffic in one virtual network is never visible to a second virtual network, yet each virtual network retains the direct, user-level network access necessary for high performance. When distinct virtual networks share the same physical network resources, each continues to perceive private resources, albeit with potentially reduced performance. This communication multiplexing is critical to high-performance message-passing with Clumps, since many processes are expected to be communicating at once. Also, the ratio of NIC's to processors in an SMP might, in general, differ from one.

The Active Message layer assigns each endpoint a unique, global name. To create a virtual network, endpoints map a set of such names into a table of message destinations. Destinations in the virtual network are then indexed using a small integer. Access rights in a virtual network take the form of a 64-bit

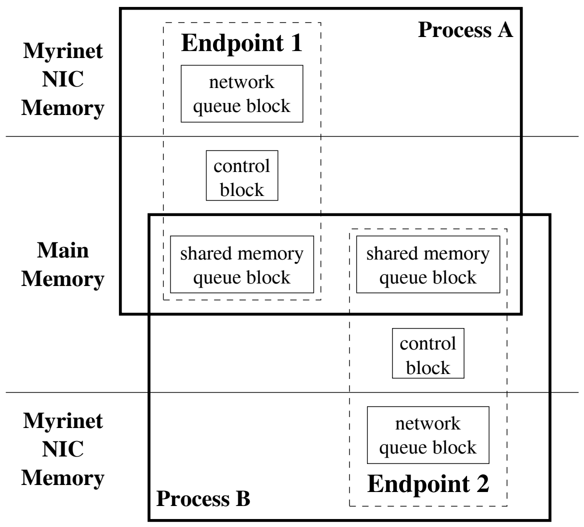

Figure 2: Data layout for an AM-II endpoint. The control block resides in main memory, the network queue block resides on the network interface card (NIC), and the shared memory queue block resides in a shared memory segment. Only the shared memory queue block is accessible to other processes.

tag specified by each destination endpoint. A sender must know the value of an endpoint's tag before sending any messages to that endpoint. Tags provide a reasonable level of protection against both inadvertently misdirected and malicious messages.

Communication in AM-II uses a request-reply paradigm. Messages originating outside the Active Messages layer are called *requests*, and all request handler routines must issue a *reply* using an opaque message token which holds information about the requesting endpoint's name and tag. A message references a handler routine with a small integer, which is used by the recipient as an index into an endpoint-specific table of such routines. The Active Message layer reserves index 0 of the table for the user-defined handler to which messages are returned in the case of network failure or other faults, *e.g.*, denial of access.

Two types of message are relevant to this work: *short messages* carry up to eight 32-bit arguments; a *bulk data transfer* extends a short message with a block of up to 8 kB of data.

3 Data Structures and Operations

In this section, we describe the data structures and operations used to manage *local messages*, *i.e.*, messages passed through shared memory. The design

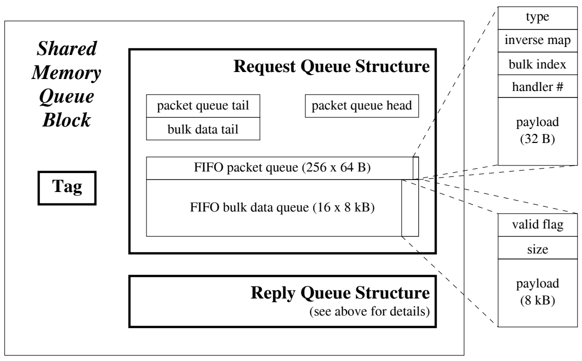

Figure 3: Block diagram of a shared memory queue block. Short messages use only the packet queue. Bulk data transfers are written into the bulk data queue as well.

space of shared memory message-passing layers has four dimensions: data layout, access control, concurrency management, and polling strategy. The section progresses through each of these dimensions, touching on the issues related to each and explaining the position taken in this work.

3.1 Data Layout

The multi-protocol representation of an endpoint is a natural extension of the representation developed for the Myrinet AM-II implementation [7]. As shown in Figure 2, an endpoint breaks into three blocks: the control block holds information such as the table of handler routines and the table of message destinations, the network queue block holds message queues for the network, and the shared memory queue block holds message queues for shared memory. The control block is used primarily by AM-II library functions and is stored in main memory. The network queue block is handled by the LANai processor and hence resides in memory on the NIC. The shared memory queue block is placed in a System V shared memory segment to allow access by multiple processes within the SMP. During idle periods, each block can be migrated to backing storage in the conventional fashion. Data in NIC memory back into main memory, and data in main memory back onto disk.

A diagram of the shared memory queue block appears in Figure 3. A copy of the endpoint tag is used for access control, while two queue structures hold request and reply messages received by the endpoint. Each queue structure further divides into three sections: queue tail information, accessed only by senders; queue head information, accessed only by recipients; and two FIFO data queues, accessed by both senders and recipients. The queues are the packet queue, which contains the handler index and arguments, and the bulk data queue, which holds data for bulk data transfers. Short messages use only the packet queue, while bulk data transfers use both queues.

The shared memory queue block has been carefully tuned for performance. Data are laid out so as to eliminate false sharing and thereby to reduce bus transactions. Each packet, for example, occupies a distinct L2 cache line, and bulk data blocks begin on cache boundaries to increase copying speed.

In addition to the handler index and arguments, entries in the packet queue contain three other fields: a packet type, an inverse queue block mapping, and a bulk data index. The first of these, the packet type, differentiates between short messages and bulk data transfers. It also serves as the handshake state in transferring data from a sender to a recipient. A valid flag serves the latter purpose for the bulk data queue. The inverse queue block mapping points to the shared memory queue block of the sending endpoint in the address space of the process that owns the receiving endpoint, enabling reply messages to avoid a potentially expensive lookup operation. The last field, the bulk index, records the association between a bulk data transfer packet and the data itself. The bulk data queue is significantly shorter than the packet queue, allowing the shared memory queue block to fit into a reasonable amount of memory (293 kB).

The shared memory queue block differs significantly from the network queue block in its lack of send queues. The absence arises from a fundamental difference between the methods used to transmit data over the network and within an SMP. In the network case, a sender cannot directly deposit data into memory located across the network, and must instead rely on a third party, such as the intelligent Myrinet NIC's, to move the data. Within an SMP, the situation is just the opposite: direct access is possible through shared memory, and no third party exists to perform the transfer.

3.2 Access Control

Access control for an endpoint has two components, accessibility and security. Accessibility determines when a process is allowed to map shared data into its address space. Security protects an endpoint against error, malice, and spying by processes that have already mapped its data.

In this work, we assume the use of multiple processes within an SMP rather than a number of threads. Communication between multiple processes matches the message-passing model more closely than does communication between threads sharing an address space. Interprocess communication in Unix typically utilizes the System V IPC layer, which provides a number of standard mechanisms for communication. By choosing to use System V shared memory segments as storage for the shared memory queue block, we implicitly tie access control decisions to the model supported by System V interprocess communication. The IPC model is quite similar to that used by traditional Unix filesystems. Each segment has distinct read and write access bits for the owner of the segment, for a Unix group associated with the segment, and for all other users. Although perhaps not impossible to build, a system that addresses security issues through the IPC access model requires multiple segments and significantly more complex operations than our performance requirements can tolerate. These considerations compel us to assume a high level of trust between endpoints communicating through shared memory.

Processes other than the one that owns the endpoint can access only the shared memory queue block. To obtain such access, a process must map the block into its address space as follows. The segment identifier for the block is used to extend the endpoint name—other processes obtain the identifier when they learn the name. The actual mapping into another process' address space occurs when an endpoint owned by that process adds the endpoint associated with the block to its table of message destinations. The inverse mapping is performed at the same time to guarantee that reply messages also travel through shared memory. In Figure 2, processes A and B have mapped the shared memory queue blocks for endpoints 1 and 2 into their address spaces. A hash table guarantees that no shared memory segment is mapped into multiple locations in a single address space.

3.3 Concurrency Management

Multiple sender processes may access a shared memory queue block concurrently, requiring atomic enqueue operations to prevent interference. The local message send operation attempts to minimize the cost of concurrent access and its impact on applications.

When sending a message, the Active Message layer first decides whether to use a shared memory protocol or a network protocol. For a local message, the layer next checks the tag in the destination queue block and returns any message that lacks access rights. After this check, the sender attempts to enqueue the message into the appropriate queue. To enqueue a short message, the sender first obtains a packet assignment by atomically incrementing the packet queue tail using the compare-and-swap instruction (CAS), then claims the assigned packet by changing its type from **free** to **claimed**, again using CAS. If the claim fails, the queue is full, and the sender backs off exponentially and polls for messages to prevent possible deadlock. Once the claim succeeds, the sender writes the data into the packet and completes the enqueue operation by changing the packet type to **ready**. For bulk data transfers, a sender claims a bulk data block before obtaining a packet assignment. After filling both packet and block, the sender marks the full packet with **ready-bulk**.

Given the effort made to achieve high performance, the use of two synchronization primitives, and in particular the CAS instruction, may seem peculiar. We have studied the performance of a range of mechanisms for managing concurrent access to the shared memory queue blocks, including the mutual exclusion techniques described in [23]. The communication regime is one of low resource contention. The time spent in the critical section of the send operation is small when compared with the total overhead of sending a message, and only intense all-to-one communication results in non-trivial contention for the shared queues. CAS is reasonably inexpensive on the Enterprise 5000, and the degree to which our send operation reduces the impact of senders being swapped out on the progress of other senders results in a level of robustness that proves quite advantageous in multiprogrammed systems. Furthermore, the method outlined above results in superior application performance even for a dedicated system. The interested reader is referred to [20] for further detail.

Although the AM-II library provides support for protected access to an endpoint using multiple receiver threads, we have assumed the use of a single thread per process in this work. The issues and costs for concurrent access by receivers are similar to those for senders. In the absence of concurrency, the local poll operation need only check the type of the packet at the head of each packet queue. When a message is available, the recipient advances the packet queue head and passes the arguments and, for bulk data transfers, the associated data block, to the appropriate handler routine. After this call returns, the packet is marked as **free** and the data block is marked as invalid.

3.4 Polling Strategy

Message polling operations are ubiquitous in Active Message layers. Responsiveness demands that a layer poll for incoming messages when sending a message [5]. In the case of the multi-protocol implementation, however, the interaction between the lightweight shared memory protocol and the more expensive network protocol can have significant impact on the performance of the former. The problem does not arise from the act of pulling in messages, but from the cost of checking repeatedly for messages when no messages are present. The empty packets at the head of shared memory packet queues remain cache-resident during periods of communication and cost only a handful of cycles to poll. Network endpoints, however, reside in uncacheable NIC memory. Reads from this memory incur an overhead of 152 cycles, leading to an order of magnitude difference between polling costs for the two substrates. Obtaining a high level of performance for programs that rely primarily on the shared memory protocol for messages requires a more sophisticated polling strategy.

We explored both fractional and adaptive strategies for polling. A fractional strategy performs the more heavyweight poll for only a fraction of all polling operations. To balance the protocols, a successful poll accepts a correspondingly larger number of messages when using the more expensive protocol. For example, a strategy that polls the network only once in every four calls to poll then accepts up to four times as many network messages in a single network poll as it does shared memory messages in a shared memory poll. An adaptive strategy adjusts polling rates dynamically in response to traffic patterns. The adaptive strategies that we investigated varied a fractional polling rate for the network between minimum and maximum values based on a history of recent network polling efforts. Within the boundaries, the strategy polls whenever the history predicts the presence of a message.

After investigating a wide selection of strategies, we settled on an adaptive strategy with a maximum network polling frequency of one in eight, slightly above the cost ratio of one to ten between shared memory and network polling. Allowing more frequent network polling generally decreased application performance. Our strategy ranges between frequencies of one in eight and one in thirty-two based on the number of network messages received by the last thirty-two network polls. Applications were not very sensitive to small differences in these parameters, but neither did all applications respond in the same way to changes. The minimum network polling frequency of one in thirty-two is small enough that applications running inside of an SMP run within 12% of their execution times using a single-protocol shared memory layer with no network accesses.

3.5 Summary

The data for an active endpoint are split between main memory, NIC memory, and shared memory segments. Messages between multiple processes within an SMP travel through the shared memory queue block, which must first be mapped into each process' address space. Any endpoint in an SMP can be mapped by any process in that SMP, but only the shared memory queue data become accessible through the mapping. We assume a high level of trust between endpoints communicating through shared memory.

The queue block holds a tag for access control and two queue structures for receiving local messages. No send queues are used for local messages—a sender deposits data directly into a receive queue at the destination endpoint. The local send operation uses CAS to minimize interference between senders. The operation takes five steps: checking the destination tag, obtaining a packet assignment, claiming the packet, writing the data, and marking the packet as ready for receipt. Bulk data transfers obtain a data block assignment and claim the block before obtaining a packet assignment. When a local message arrives, a recipient notices the change in the type of the packet at the head of the queue. After passing the data to the appropriate handler routine, the recipient frees the packet for reuse. The structure of the shared memory queue block keeps the number of bus transactions produced by these operations to a minimum.

Message polling operations are ubiquitous in Active Message layers, but polling the network typically costs an order of magnitude more than polling shared memory. The source of this difference lies in the storage used for the two queue blocks. Shared memory segments benefit from the Gigaplane's cache-coherence support. NIC memory does not, and is uncacheable. To retain the base performance of the shared memory protocol, we adopt an adaptive polling strategy. The strategy polls the network whenever a history of the last thirty-two network poll operations predicts the presence of a message, bounded by a minimum polling rate of one in thirty-two and a maximum rate of one in eight.

	Shared Memory	Multi-Protocol Shared Memory	Multi-Protocol Myrinet	Myrinet
Latency (L)	-0.3	0.1	15.5	13.8
Send Overhead (o_s)	1.8	2.1	3.3	5.6
Receive Overhead (o_r)	1.3	1.4	8.6	8.1
Gap (g)	3.1	3.6	17.6	17.6
Gap per Byte (G)	0.00602	0.00625	0.0305	0.0315
Bandwidth ($1/G$)	166 MB/s	160 MB/s	32.8 MB/s	31.7 MB/s
Half-power Point	5.1 kB	5.3 kB	8.6 kB	8.7 kB
Round Trip Time (RTT)	5.6	7.2	55	55

Table 2: LogP parameters and round trip times in microseconds. Values for both the multi-protocol and the single-protocol implementations are included for comparison. The polling strategy used in the multi-protocol version limits the impact of network polling on the shared memory protocol and actually improves the network parameters.

4 Microbenchmark Analysis

In this section, we investigate the performance of the multi-protocol implementation with microbenchmarks. After a brief explanation of the LogGP network model, we present parameters for both protocols used singly and in combination. We next illustrate the costs of message-passing relative to the cache-line transfer time with a detailed breakdown of the overhead involved in sending a local message. A summary of the issues and implications concludes the section.

4.1 Methodology

The LogP model of network performance [9] attempts to characterize communication networks as a set of four parameters: L, an upper bound on the network latency (wire time) between processors; o, the processor busy-time required to inject a message into the network or to pull one out; g, the minimum time between message injections for large numbers of messages; and P, the number of processors. The overhead o is often separated into send overhead, o_s, and receive overhead, o_r. LogP assumes a small, fixed message length for communication, neglecting common hardware support for large transfers. LogGP [1] extends the LogP model with the parameter G, the time per byte for long messages.

LogP parameters were measured using a microbenchmark from the suite described in [10]. To measure G, we constructed a second benchmark to fragment bulk data transfers of arbitrary length into 8 kB chunks and to pipeline those chunks through the Active Message layer. The sending process copies the data from a send buffer into the receiving endpoint, and the receiving process copies the data from the endpoint into a receive buffer.

4.2 Results

The LogGP parameters for the multi-protocol implementation appear in Table 2 alongside the values measured for the single-protocol implementations.[1] Round trip times for local messages are roughly an order of magnitude less than for network messages, and sustainable bandwidth is roughly five times greater, peaking at about 160 MB/s for the multi-protocol implementation. The negative latency for the single-protocol shared memory case indicates overlap in time between the send and receive overheads [19], in this instance due to the poll operation. The adaptive polling strategy limits the impact of network polling on local messages to an average of 0.2 microseconds for both send and receive overhead and an increase of roughly 30% in round trip time. For the network protocol, the adaptive polling strategy reduces total overhead and results in slightly improved performance. The upper bound on network polling reduces the send overhead by 2.3 microseconds, but the additional shared memory poll operations add 0.5 microseconds to the receive overhead. The smaller total overhead implies that less latency can be hidden, and that parameter consequently rises. The gap and round trip times remain unchanged.

4.3 Overhead Breakdown

Having explored the coarse measurements of the implementation, we now investigate the contributions of each component of the send operation towards the total overhead. Recall that after receiving a message, the recipient marks the message packet as **free**

[1]Recent efforts have reduced the round trip time for AM-II on Myrinet to 42 microseconds, and further optimization is planned. These improvements have limited impact on the issues discussed in this paper, however.

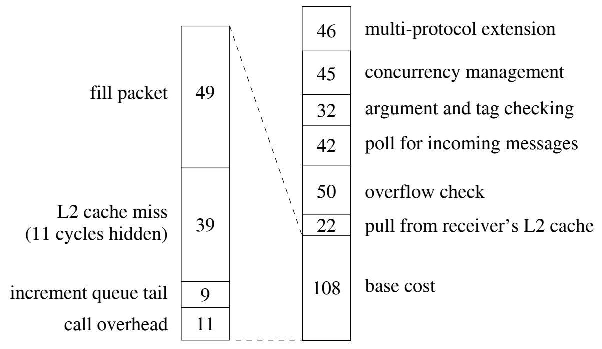

Figure 4: Breakdown of send overhead in cycles for the shared memory protocol. The left bar shows the costs of each component for the base case, which performs no error checking or concurrency management for the destination queue. The cost of the latter appears in the right bar. The send overhead totals 299 cycles (1.8 microseconds) for the shared memory protocol and 345 cycles (2.1 microseconds) for the multi-protocol layer.

to allow its reuse. On the Enterprise 5000, that change invalidates the cache line occupied by the packet on other processors. Hence, sending a short message generally incurs at least one L2 cache miss. We expect free packets to remain cache-resident on the receiving processor during periods of communication. In the LogP microbenchmark, for example, the processors access roughly 32 kB of communication data—quite a bit less than the size of the 512 kB L2 cache. The typical time spent servicing the L2 miss is hence closer to the 80 cycle latency to access another L2 cache than to the 50 cycle main memory latency.

A breakdown of the send overhead for short messages appears in Figure 4. The left bar illustrates the base cost of a short message in the absence of error checking and concurrency management for the destination queue. The total of 108 cycles (0.65 microseconds) also assumes that the message packet is not resident in the receiver's cache. To reach the base cost, the sender prepares eight arguments and calls the Active Message layer in a total of 11 cycles. Locating the destination endpoint and advancing the tail of the queue require another 9 cycles. Finally, the layer obtains and fills a packet, incurring an L2 cache miss in the process. Filling the packet takes another 49 cycles, but allows the processor to hide 11 cycles of the miss latency.

The right bar in the figure extends the base cost with measurements of the remaining components of send overhead. When queue packets are resident in the receiver's L2 cache, each message incurs an additional 22 cycle penalty. A check for destination queue overflow is responsible for the next 50 cycles, primarily due to an extra bus transaction. The check reads the packet type and makes immediate use of the result, incurring the full overhead of the first transaction. Filling the packet then results in an invalidation, the second transaction.[2] The local poll operation performed before each send adds another 42 cycles. Function argument and endpoint tag checking by the Active Message layer introduce another 25 cycles of overhead. Concurrency management using CAS adds 45 cycles, bringing the total to 299 cycles (1.8 microseconds) when using only the shared memory protocol. Finally, inclusion of the network protocol more than doubles the time spent in the poll operation, bringing the total for the multi-protocol implementation to 345 cycles (2.1 microseconds).

4.4 Summary

The numbers presented in this section help to illuminate the performance of message-passing across cache-coherent buses and the interactions between the two protocols. Until the most recent generation of machines and interconnection technology, network communication often provided greater bandwidth than that available from the memory system. With our system, the shared memory protocol provides five times the bandwidth available from the network, peaking at 166 MB/sec for the single-protocol implementation. The end-to-end latency of local messages, 2.8 microseconds, is an order of magnitude smaller than that of network messages. The adaptive polling strategy limits the impact of network polling on local message latency to an increase of roughly 30% and reduces peak bandwidth by 4%. For the network protocol, the polling strategy reduces total overhead but has a negligible effect on overall performance. In light of the breakdown of send overhead, we note that bus transactions make up only a third of the cost. Another third is spent on the basic mechanisms of the operation: call overhead, argument checks, queue advancement, and packet filling. The final third of the time is split between managing concurrent access between senders and polling for incoming local messages.

[2] One transaction can be eliminated if the sender prewrites an unused part of the packet before the overflow check and uses a memory barrier to prevent reordering. This approach reduces the send overhead by 40 cycles when measured in isolation, but has a negative impact on the full LogP parameters and application results, presumably due to cache line thrashing when a receiver polls during the send operation.

5 Application Analysis

In this section, we present execution time measurements for three applications running within a Clump and on uniprocessor UltraSPARC workstations communicating through a Myrinet network (a NOW). The applications are drawn from the Split-C application suite [8] and are written in a bulk synchronous style—processors proceed through a sequence of coarse-grained phases, performing a global synchronization between each phase. The results shed light on several performance issues for Clumps and illustrate the impact of the multi-protocol implementation.

5.1 Theory of Performance

The powerful interconnect within each SMP promises Clumps a significant performance advantage with respect to a NOW. Most applications can easily reap the benefits offered by fast communication. The full potential of Clumps may be difficult to achieve, however, as certain aspects of the system can degrade overall performance.

The most important of these issues is the balance of processors to NIC's inside each SMP. A Clumps application tuned to take advantage of the shared memory protocol makes subsequently less use of the network protocol, allowing processors that share network resources to operate at full potential. But for some applications, such as those with phases of all-to-all communication, tuning for Clumps may not be possible. In our system, each SMP uses three network interfaces to handle network traffic for eight processors, leading to three-way sharing for two of the NIC's.

A second complication arises for applications written in a bulk synchronous style, which implicitly assume a reasonably balanced load. The use of multi-protocol communication can violate that assumption, as performance benefits depend on the fraction of traffic routed through the shared memory protocol. In such a case, the improvement in application execution time reflects only the minimum of the per-processor improvements—other processors idle until the slowest processor has finished the phase. As a first step towards tuning a bulk synchronous application for a Clump, a programmer can arrange the virtual processor layout to reduce the amount of communication traffic that must travel through the network.

Finally, the additional complexity of cache-coherence support within an SMP results in longer memory latencies. On the Enterprise 5000, the difference is roughly 20% for memory not shared between processors. This memory latency penalty does not usually play a large role in performance, however, as it is mitigated by a number of factors, including a larger L2 cache, higher per-processor memory bandwidth, and locality of access in the application codes. A more restrictive constraint occurs in the form of aggregate memory bandwidth. Although the Gigaplane provides more bandwidth than the eight processors can use, the memory banks in each of our SMP's are not fully populated, limiting aggregate memory bandwidth to 400 MB/s. This artifact compounds the effect of the memory latency penalty and has significant impact on performance.

	Input Parameters	Memory
3-D FFT	256x256x256 values	14 MB
CON/comm	3D underlying lattice 512,000 nodes/processor 25% edges present	32 MB
CON/comp	2D underlying lattice 640,000 nodes/processor 40% edges present	40 MB
EM3D/naive	2,500 nodes/processor degree 20, 40% remote naive layout	8.3 MB
EM3D/good	2,500 nodes/processor degree 20, 40% remote good layout	8.3 MB

Table 3: Input parameters and per-processor memory usage for application runs on the Clump. The CON runs differ in the balance between communication and local computation. The EM3D runs differ in the layout of virtual processors.

5.2 Application Profile

We chose to run each of the three Split-C applications with one or two sets of input parameters to illustrate performance effects. For each run, Table 3 lists the input parameters and per-processor memory requirement when running on the Clump. Table 4 separates communication volume for each run into network and local traffic.

The first application, 3-D FFT, performs a fast Fourier transform in three dimensions and typifies regular applications that rely primarily on bulk communication. The all-to-all communication pattern used in 3-D FFT exposes the effect of the SMP's processor to NIC balance.

The second application, CON, finds the connected components of a distributed graph. CON performs a large amount of fine-grained communication in a statistically well-defined pattern. The balance between computation and communication in CON depends strongly on the input parameters. We selected a communication-bound run to highlight the benefits of the fast communication and a second, computation-bound run to demonstrate the effect of the SMP memory latency penalty. The input parameters for the first run result in a period of high contention and load imbalance near the end of the execution.

	Network Communication				Local Communication				
	Short Messages				Short Messages				
	Mean	Min.	Max.	Bulk Data	Mean	Min.	Max.	Bulk Data	% Local
3-D FFT	3,270	3,072	3,468	3,072 x 4 kB	954	896	1,292	896 x 4 kB	22.6
CON/comm	67,236	51,541	240,919	292 x 16 B	50,138	43,614	90,346	130 x 16 B	42.7
CON/comp	5,993	5,601	6,498	10 x 16 B	5,890	5,351	6,411	6 x 16 B	49.5
EM3D/naive	608,817	199,600	1,015,500	none	1,013,094	804,065	1,222,500	none	62.5
EM3D/good	407,369	0	818,500	none	1,216,549	1,198,407	1,223,500	none	74.9

Table 4: Per-processor communication volume for the Clump. Only 3-D FFT uses a significant number of bulk transfers. The communication-bound CON run suffers from a load imbalance in network traffic. Differences in virtual processor layout result in markedly different traffic distributions for the EM3D runs.

	8-way SMP		NOW (8 proc.)	
	Shared Mem.	M-P	M-P	Myri.
3-D FFT	8.7	9.0	7.0	6.76
CON/comm	1.96	2.1	4.01	4.01
CON/comp	1.40	1.44	1.18	1.19
EM3D	7.7	8.6	33.9	33.8

Table 5: Application execution times in seconds. Values for both the multi-protocol and the single-protocol implementations are included for comparison. EM3D and the communication-bound CON take advantage of the multi-protocol layer to obtain superior performance on the SMP. The computation-bound CON and 3-D FFT performance on the SMP exhibit inferior performance due to an artifact of our Clump that limits aggregate memory bandwidth.

	Clump (4 8-way SMP's)	NOW (32 proc.)
3-D FFT	6.8	2.2
CON/comm	15	7.7
CON/comp	2.7	1.3
EM3D/naive	54	48
EM3D/good	38	52
EM3D/naive bal.	28 to 52	25 to 47
EM3D/good bal.	5.5 to 37	28 to 51

Table 6: Application execution times and balance in seconds. The Clump's smaller per-processor network bandwidth restricts performance for both 3-D FFT and the CON runs. The EM3D run with poor virtual processor layout typifies performance for bulk synchronous programs. The improved layout obtains superior performance by reducing network traffic.

EM3D, the last application, propagates electromagnetic radiation in three dimensions on an irregular mesh and represents the class of applications that perform irregular, fine-grained communication. EM3D alternates between updates to the electric and magnetic fields in a bulk synchronous manner. We use two runs of EM3D to show the effect of the bulk synchronous style and the advantage of intelligent virtual processor layout. Both runs partition the underlying coordinate space on 32 processors into 4x4x2 blocks. The first run uses a naive layout for virtual processors, placing processors within an SMP into 4x2x1 blocks. In the second run, denoted EM3D/good in the tables, an SMP's processors instead occupy 2x2x2 blocks, reducing both the aggregate network traffic and the upper bound on per-processor network traffic.

5.3 Results

Table 5 presents execution times in seconds for the three applications running on one SMP and on an 8-processor NOW. For each platform, the table gives timings using both the multi-protocol implementation and the appropriate single-protocol implementation. Each entry represents the average of multiple executions and is reported to a precision no greater than that allowed by the variance between the executions. The application-level impact of network polling on the shared memory protocol ranges from 3 to 12%. Use of the multi-protocol implementation and associated polling strategy on the NOW has negligible impact on most applications, but reduces 3-D FFT performance by about 4%. In terms of the SMP to NOW comparison, the EM3D run, which partitions the problem space into 2x2x2 blocks, results in the most dramatic improvement, with the SMP finishing 75% faster than the NOW. The remaining runs, 3-D FFT and the two CON runs, step through large sets of data, requiring high memory bandwidth to support all eight processors. The memory latency penalty constrains performance for these runs. The worst case, 3-D FFT, takes 33% longer on the SMP. The CON runs require less memory bandwidth and use fast communication to greater advantage. For the computation-bound run, the SMP execution is 21% slower than that of the NOW. In the communication-bound run, the effect of fast communication dominates, allowing it to finish 48% faster on the SMP.

Application execution times in seconds on the Clump and a 32-processor NOW appear in Table 6.

The NOW uses the single-protocol Myrinet implementation. The bottom section of the table provides information on the communication load balance between processors for each EM3D run; each entry represents the range across processors of time spent in the communication phase. The effect of the processor to NIC balance in our system is apparent in the degradation of 3-D FFT performance, which takes 209% (a factor of three) longer on the Clump than on the NOW. The communication-bound CON run is also affected by the sharing of network resources, requiring 95% more time on the Clump. For the computation-bound CON run, the 108% slowdown on the Clump results from a combination of the memory latency penalty and the processor to NIC balance. The EM3D runs demonstrate the effect of the bulk synchronous style on execution time. In each case, execution time is limited by the slowest of the processors. The benefit of the improved layout in generating less network traffic is enhanced by the processor to NIC balance, leading to a 30% improvement in execution time. For the NOW, the naive layout proves superior due to bandwidth thinning in the upper regions of the tree network connecting the NOW. Machines in the same sub-branch of the NOW network enjoy greater aggregate bandwidth than do machines in different branches, and the naive layout takes better advantage of this arrangement. Comparing the best EM3D result on each platform, the Clump finishes in 21% less time.

5.4 Summary

Clumps promise significant performance gains to applications that can reap the benefits of fast communication, but their full potential may be difficult to achieve. In a NOW, bandwidth scales implicitly with the number of machines, but the balance of processors to NIC's in an SMP must be considered more carefully. The presence of multiple I/O buses is critical, and some applications may require a one-to-one processor to NIC relationship to achieve performance. Alternatively, an SMP may be equipped with a single, large interface, provided the cost of such a device is not prohibitive.

Also critical to performance is the aggregate memory bandwidth of each SMP. Underpopulation of the memory banks in our Clump results in an artificially low limit and severely restricts the performance of some applications. While the problem is not inherent in our system, it is one that future systems must remember to avoid: for large computational problems, caches do not always adequately buffer memory accesses.

The bulk synchronous style of programming has achieved some degree of popularity for parallel programming. When using a Clump, a programmer must put in a greater effort to balance the load between processors. Optimization of the virtual processor layout onto the Clump is a good first step.

6 Related Work

We have investigated issues related to efficient message-passing through both shared memory and the network within a Clump. The literature pertaining to Clumps is still fairly limited, but covers quite a wide range of topics.

One approach to Clumps that has received much attention over the years is the extension of shared memory between SMP's. Recent efforts on this front include SVM [17] and MGS [32]. These studies investigate a problem complementary to our own in that both seek to optimize common techniques in one medium to allow use of those techniques in both. Each view proves more natural and effective than the other for interesting classes of applications.

In the smaller body of message-passing work, Nexus comes closest to our own. Nexus is a portable programming system [13] that focuses primarily on portability and on support for heterogeneity. It supports arbitrary sets of machines, processes (or contexts, in Nexus terminology) and threads. Nexus generally builds on top of existing communication layers, resulting in somewhat higher overheads than those obtained with Active Messages. The communication abstractions are similar to those of AM-II, but the style of communication is different. Like AM-II, Nexus has endpoints that define tables of handler routines, but Nexus does not require that communication obey a request-reply paradigm. This flexibility allows Nexus to use endpoint names, or startpoints, to initiate messages. A startpoint can be bound to multiple endpoints, allowing for multicast communication.

Since Nexus platforms can support multiple communication protocols between a startpoint and an endpoint, Nexus has explored multi-protocol communication from a more general perspective than have we [14]. Although shared memory is mentioned in the work, numbers are provided only for more expensive underlying protocols, making a direct comparison impossible. The Nexus multi-protocol paper also notes the wide variance between polling costs for different protocols and presents data for fractional polling strategies. We explored more adaptive strategies to reduce the impact of network polling to a satisfactory level.

An interesting study by Lim et. al. [18] investigates the use of one processor in each SMP as a message proxy for the remaining processors. The work focuses on providing multiple users with protected access to a single network resource and evaluates the proxy approach in detail. AM-II sidesteps the question of protected access by taking advantage of an SMP's virtual memory system to grant direct access to a subset of network resources. An intelligent NIC plays an essential role in the AM-II approach.

The remaining message-passing work on Clumps pertains primarily to the problem of programming them for performance. This paper does not speak directly to that problem, although some insight can be gained from the section on applications. Such efforts often assume that a programmer is willing to rewrite most or all of an application to obtain performance.

The P4 programming system [6] was probably one of the first systems to recognize Clumps as a platform. P4 provides mechanisms start multiple threads on one or more machines and to communicate between such threads using either message-passing or shared memory constructs. The programmer must explicitly select the appropriate library call. The library also provides a number of useful reduction operations.

SIMPLE [3] provides functionality similar to P4, but extends the library with broadcast operations and a variety of tuned, many-processor communication methods. SIMPLE also attempts to lighten the programmer's burden by offering functions that involve all processors, all processors in an SMP, one processor in each SMP, and so forth.

A paper by Fink and Baden [11] attacks the problem of balance in bulk synchronous algorithms by rebalancing computation and communication for a regular problem within an SMP. Given a 2D domain partitioned in one dimension between SMP's in a Clump, the paper calculates a non-uniform partitioning of the domain within each SMP such that the time spent in a phase is roughly equal for each processor. Essentially, the analysis gives processors on boundaries less computation to balance the cost of communication.

KeLP, by the same authors, seeks to simplify the process of application development. Recent extensions to KeLP [12] add new functionality to support applications on Clumps. With KeLP, a programmer expresses data decomposition and motion in a block-structured style. The runtime system then employs inspector-executor analysis to overlap communication with computation. No global barriers are used; interprocessor synchronization occurs only through communication dependencies.

In work related less directly to Clumps, Mukherjee and Hill [24] have investigated the advantages of making NIC memory cacheable. For multi-protocol communication, the importance of cacheable NIC memory is the resulting reduction in the cost of polling the network. Such systems might not require a sophisticated polling strategy.

7 Conclusion

Obtaining the performance potential in the Clumps architecture requires an understanding of interactions between the multiple levels of interconnection. In this paper, we have addressed the tradeoffs involved in implementing a fast communication layer that uses both cache-coherent memory and a low-latency network to route messages. Our multi-protocol active message layer operates on a cluster of four 8-processor Sun Enterprise 5000 servers interconnected by a Myrinet with three NIC's per SMP. We evaluated communication performance with microbenchmarks and with applications, bringing light to a number of performance issues for the platform and illustrating the results of our decisions.

In the design of a multi-protocol layer for Clumps, three critical aspects must be considered: data layout, concurrent access, and polling strategy. For our implementation, the data for an endpoint are split between main memory, NIC memory, and shared memory. Local communication passes through the shared memory portion of the endpoint, which is structured to minimize bus transactions. Concurrent access by multiple senders is handled using the compare-and-swap instruction to reduce interference between senders. Accessing network message data in uncacheable NIC memory is expensive—typically an order of magnitude more costly than accessing shared memory data. To retain the base performance of the shared memory protocol, we developed an adaptive polling strategy that varies the rate of network polling between upper and lower bounds based on a history of recent network activity.

When measured with microbenchmarks, our implementation illustrates the performance tradeoffs for fast communication on Clumps. Use of a multi-protocol communication layer has little impact on network message performance, but both careful engineering and an adaptive polling strategy are necessary to retain high performance through shared memory. The shared memory protocol provides five times the bandwidth available from the network, peaking at 160 MB/sec. End-to-end latency for short local messages is 3.6 microseconds, a factor of eight less than the corresponding number for the network protocol. The abstractions necessary to support message-passing consume a significant fraction of this time. For example, the overhead involved in a local send operation is 2.1 microseconds. Bus transactions account for 32% of this number; basic mechanisms such as call overhead and packet filling make up another 29%; managing concurrent access leads to another 13%; and the remaining 26% is split nearly evenly between polls for each protocol.

We studied application-level performance issues using three Split-C applications with five sets of input parameters. The applications use a bulk synchronous style and perform significant amounts of communication. To establish a base case, we presented the same results for a NOW. Within an SMP, applications take advantage of the shared memory protocol to achieve improved performance, but some are constrained by aggregate memory bandwidth limitations. Inclusion of the network protocol slows these applications by no more than 12%. The use of the full Clump brings the balance between processors and NIC's to light. For 3-D FFT, which uses all-to-all communication, we observed a factor of three slowdown, as we might expect given the three-way sharing of NIC's by processors in our system. The Clump numbers also illustrate a drawback of the bulk synchronous programming style: although many processors might be able to take advantage of fast communication, a performance increase requires that all processors do so. By rearranging the virtual processors to increase the fraction of traffic sent through shared memory, we demonstrated improved performance. These obstacles to performance are not insurmountable, but they do illustrate several potential pitfalls for Clumps.

The Clumps architecture presents a wealth of interesting new tradeoffs and possibilities. With this paper, we have begun to explore these issues and have illustrated our findings with measurements at two levels. In the future, we plan to continue our investigation of the low-level aspects of these systems in order to build a solid foundation of understanding for exploring more abstract interactions.

Acknowledgements

The authors wish to thank Sun Microsystems, Inc. for donating the Enterprise 5000 servers on which this work is based. This work was also supported in part by funding from National Science Foundation Infrastructure Grant CDA 94-01156, Lawrence Livermore National Laboratory Intra-University Transaction Agreement B336568, and the Defense Advanced Research Projects Administration Grant F30602-95-C-0014. Brent Chun developed much of the Myrinet AM-II protocol, which plays a key role in enabling this work. Thanks are also due to the members of the Clumps and NOW groups at U. C. Berkeley for their encouragement and support. In particular, we are grateful for numerous conversations with Arvind Krishnamurthy and for valuable feedback on the presentation from Kathy Yelick. Rich Martin helped with the selection of applications and input parameters. Finally, we wish to acknowledge the efforts of David Gay and Randi Thomas in developing and tuning the 3-D FFT code.

References

[1] A. Alexandrov, M. Ionescu, K. E. Schauser, C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation," *7th Annual Symposium on Parallel Algorithms and Architectures*, July 1995.

[2] Accelerated Strategic Computing Initiative, a program of the Department of Energy. Information is available via *http://www.llnl.gov/asci-alliances/*.

[3] D. A. Bader, J. JáJá, "SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMP's)," preliminary version, May 1997, available via *http://www.umiacs.umd.edu/research/EXPAR.*

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, W. Su, "Myrinet—A Gigabit-per-Second Local-Area Network," *IEEE Micro*, Vol. 15, February 1995, pp. 29-38.

[5] E. A. Brewer, B. C. Kuszmaul, "How to Get Good Performance from the CM-5 Data Network," *Proceedings of the 8th International Parallel Processing Symposium*, April 1994.

[6] R. Butler, E. Lusk, "Monitors, Message, and Clusters: the p4 Parallel Programming System," available via *http://www.mcs.anl.gov/home/lusk/p4/p4-paper/paper.html.*

[7] B. N. Chun, A. M. Mainwaring, D. E. Culler, "A General-Purpose Protocol Architecture for a Low-Latency, Multi-gigabit System Area Network," *Proceedings of Hot Interconnects V*, Stanford, California, August 1997.

[8] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. S. Lumetta, T. von Eicken, K. Yelick, “Parallel Programming in Split-C,” *Proceedings of Supercomputing 1993*, Portland, Oregon, November 1993, pp. 262-73.

[9] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, “LogP: Towards a Realistic Model of Parallel Computation,” *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993.

[10] D. E. Culler, L. T. Liu, R. P. Martin, C. O. Yoshikawa, “Assessing Fast Network Interfaces”, *IEEE Micro*, Vol. 16, No. 1, February 1996, pp. 35-43.

[11] S. J. Fink, S. B. Baden, “Non-Uniform Partitioning of Finite Difference Methods Running on SMP Clusters,” submitted for publication, available via *http://www-cse.ucsd.edu/users/baden/MT.html.*

[12] S. J. Fink, S. B. Baden, “Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters,” submitted for publication, available via *http://www-cse.ucsd.edu/users/baden/MT.html.*

[13] I. Foster, C. Kesselman, S. Tuecke, “The Nexus Approach to Integrating Multithreading and Communication,” *Journal of Parallel and Distributed Computing*, Vol. 37, August 1996, pp. 70-82.

[14] I. Foster, J. Geisler, C. Kesselman, S. Tuecke, “Managing Multiple Communication Methods in High-Performance Networked Computing Systems,” *Journal of Parallel and Distributed Computing*, Vol. 40, January 1997, pp. 35-48.

[15] W. W. Gropp, E. L. Lusk, “A Taxonomy of Programming Models for Symmetric Multiprocessors and SMP clusters,” *Proceedings of Programming Models for Massively Parallel Computers 1995*, October 1995, pp. 2-7.

[16] M. Haines, D. Cronk, P. Mehrotra, “On the Design of Chant: A Talking Threads Package,” *Proceedings of Supercomputing 1994*, Washington, D.C., November 1994, pp. 350-9.

[17] D. Jiang, H. Shan, J. P. Singh, “Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors,” *Proceedings of Principles and Practice of Parallel Programming*, 1997, pp. 217-29.

[18] B.-H. Lim, P. Heidelberger, P. Pattnaik, M. Snir, “Message Proxies for Efficient, Protected Communication on SMP Clusters,” IBM Almaden Research Report #RC 20522 (90972), August 1996.

[19] L. T. Liu, D. E. Culler, “Evaluation of the Intel Paragon on Active Message Communication,” *Proceedings of Intel Supercomputer Users Group Conference*, June 1995, also available via *http://now.CS.Berkeley.EDU.*

[20] S. S. Lumetta, D. E. Culler, “Managing Concurrent Access for Shared Memory Active Messages,” U. C. Berkeley Technical Report in preparation.

[21] A. M. Mainwaring, D. E. Culler, “Active Message Applications Programming Interface and Communication Subsystem Organization,” U. C. Berkeley Technical Report #CSD-96-918, October 1996, also available via *http://now.CS.Berkeley.EDU.*

[22] R. Martin, “HPAM: an Active Message Layer for a Network of HP Workstations,” *Proceedings of Hot Interconnects II*, Stanford, California, August 1994, pp. 40-58.

[23] J. M. Mellor-Crummey, M. L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,” *ACM Transactions on Computer Systems*, Vol. 9, No. 1, February 1991, pp. 21-65.

[24] S. S. Mukherjee, M. D. Hill, “A Case for Making Network Interfaces Less Peripheral,” *Proceedings of Hot Interconnects V*, Stanford, California, August 1997.

[25] R. H. Saavedra, “Micro Benchmark Analysis of the KSR1,” *Proceedings of Supercomputing 1993*, Portland, Oregon, November 1993, pp. 202-13.

[26] K. E. Schauser, C. Scheiman, “Experiences with Active Messages on the Meiko CS-2,” *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[27] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, B. Liencres, “Gigaplane: A High Performance Bus for Large SMPs,” *Proceedings of Hot Interconnects IV*, Stanford, California, August 1996, pp. 41-52.

[28] L. Tucker, A. M. Mainwaring, "CMMD: Active Messages on the CM-5," *Parallel Computing*, Vol. 20, No. 4, August 1994, pp. 481-96.

[29] T. von Eicken, V. Avula, A. Basu, V. Buch, "Low-latency Communication over ATM Networks Using Active Messages," *Proceedings of Hot Interconnects II*, Stanford, California, August 1994, pp. 60-71.

[30] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Qld., Australia, May 1992, pp. 256-66.

[31] P. R. Woodward, "Perspectives on Supercomputing: Three Decades of Change," *IEEE Computer*, Vol. 29, October 1996, pp. 99-111.

[32] D. Yeung, J. Kubiatowicz, A. Agarwal, "MGS: A Multigrain Shared Memory System," *Proceedings of the 23rd International Symposium on Computer Architecture*, Philadelphia, Pennsylvania, May 1996, pp. 44-55.