# Scalable Inter-Cluster Communication Systems for Clustered Multiprocessors

Xiaohu Jiang and Donald Yeung
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

November 1997

**Abstract**

As workstation clusters move away from uniprocessors in favor of multiprocessors to support the increasing computational needs of distributed applications, greater demands are placed on the communication interfaces that couple individual workstations. This paper investigates scalable, efficient, and reliable communication systems for multiprocessor clusters that use commodity local area networks (LANs). Our design provides reliable message delivery using software protocol stacks. High performance is achieved via an efficient protocol stack design, parallel execution of protocol processing on multiple proxy nodes, and transmission of messages through multiple network interfaces in each multiprocessor. Our communication system is scalable because the parallel stack design delivers higher throughput as the physical communication processing and network interface resources in each multiprocessor are increased.

A prototype of the design is built on the Alewife multiprocessor. The prototype inter-cluster communication system is integrated with the Alewife Multigrain Shared-Memory System, and the performance of 8 parallel applications are studied on the platform. Our results show that the performance of communication-intensive applications can be severely impacted by inter-multiprocessor messaging delay due to software communication processing, and contention at communication proxies. Contention at proxies can be particularly severe when the number of processors in each multiprocessor is scaled. Scaling proxies along with compute processors can greatly relieve contention in the inter-cluster communication system.

# 1 Introduction

While traditional massively parallel processors (MPPs) can achieve good performance on a variety of important applications, their prohibitively high cost prevents them from becoming widely available. In recent years, parallel workstations, such as Symmetric Multiprocessors (SMPs), are quickly emerging. This class of machines can exploit parallelism in applications to achieve high performance, while benefit from the economy of high volume because their small-scale nature allows them to be commodity components. Not only commodity systems offer low cost, they often provide better performance in the long run since they track the rapid advances in hardware and software technologies well. Many researchers believe that by using these commodity multiprocessors as building blocks, high performance MPPs can be built in a cost-effective way. In this paper, we call these small- to medium-scale multiprocessors as clusters, and the MPPs built by assembling these clusters together as clustered multiprocessors.

Though people have proposed to construct scalable multiprocessors by integrating commodity computer clusters, many chose to build custom hardware ([4], [11], and [7]) to provide efficient, protected inter-cluster communication. While these systems achieve impressive performance, their inclusion of complicated custom hardware dramatically increases the cost and design time of the system, which is undesirable.

This paper studies the problem of how to build scalable, efficient, and reliable Inter-Cluster Communication Systems (ICCSs) using unreliable commodity Local Area Network (LAN) technologies. Commodity LAN technologies, such as Fast Ethernet, ATM, and HIPPI, are proposed as the hardware substrate for inter-cluster communication systems, while provide high level network services to applications via software protocol stack. By not introducing custom hardware, supporting standard communication protocols, and configuring communication processing resources in software, our design provides a flexible and cost-effective approach to build scalable clustered multiprocessors.

Building reliability on top of unreliable LANs using software protocol stack involves complicated protocol processing, which dramatically increases the processing overhead to send and receive messages. Contention for protocol processing resources can also severely delay message delivery. To provide high throughput and low latency inter-cluster communication which is absolutely necessary for parallel applications to achieve high performance on clustered multiprocessors, we propose a design which exploits parallelism in ICCSs.

Scalability is important in ICCS design since when cluster size scales up, higher communication load will be placed on ICCSs, and the ICCS performance needs to increase accordingly. A scalable design allows us to dedicate more resources to protocol processing and more Network Interfaces (NIs) to larger clusters to meet the increased requirement.

In this paper, we provide an qualitative, in-depth study of the impact of protocol stack processing overhead on Distributed Software Shared Memory (DSSM) applications. Our design features an efficient, scalable implementation of standard protocol stacks in each cluster. By choosing to implement standard protocols, such as TCP/IP or UDP/IP, many nice features of conventional network services, such as error detection and lost packets retransmission, end to end flow control, and in-order delivery, are supported and the implementation issues are well understood. To achieve high performance, we choose to only implement features required by the DSSM protocol. Our design is scalable, since multiple in-kernel protocol processing processes can run on multiple processors and multiple NIs can be implemented in each cluster, which greatly relieves contention in ICCSs. Also since our design allocates processing resources between computation and communication in software, it can potentially change its resource allocation for different applications.

A prototype implementation of the design is built on the MIT Alewife multiprocessor [3]. We integrate our prototype with the MIT Multi-Grain Shared-memory system (MGS) [15], which is a DSSM system also implemented on Alewife. The prototype provides a flexible platform for studying the following problems:

- How much overhead needs to be paid for inter-cluster protocol processing?

- By how much does contention in ICCSs impact application performance? To what extent can this impact be reduced by exploring parallelism in protocol processing?

- How should an ICCSs scale with their cluster size to deliver the best application performance.

A suite of eight parallel applications have been studied on our prototype. For computation intensive applications, the impact of protocol processing overhead on their performance is marginal. For communication intensive applications, the run time of their unoptimized versions can be more than doubled due to protocol processing overhead. But the slowdown can be greatly reduced by dedicating more processing nodes in each cluster to protocol processing for clusters which have two or more compute nodes.

The next section presents the system architecture of our ICCS design. Section 3 describes the the prototype system implemented on Alewife. Section 4 presents application performance results on our implementation. Section 5 discusses related works. Finally, section 6 summarizes the paper.

# 2 System Architecture

Parallel applications, such as some of the shared memory applications we test in later sections, generate large amount of inter-cluster messages in very short period of time. Efficient, reliable ICCSs are essential for these communication intensive applications to achieve good performance on clustered multiprocessors.

## 2.1 Why parallelism and scalability are important in ICCS design?

To understand the inter-cluster communication requirement of parallel applications, we did some measurements on MGS. Our results show that on a machine with two clusters, 8 compute node each, average inter-cluster communication load can reach to 500 messages per million machine cycles, with average message size at a few hundred bytes. On a

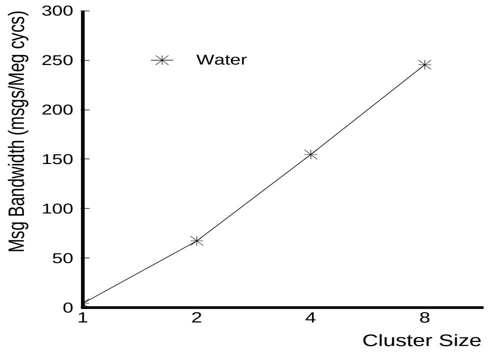


Figure 1: Application inter-cluster communication load on a 2 cluster machine.

200MHz commodity processor, these translates to a throughput of 100,000 messages per second and hundreds of Megabits per second. On a larger configuration, the aggregated ICCS load will further increase.

For ICCSs built using commodity components to achieve such a high throughput, while maintain contention at a low level to ensure short delay, parallelism has to be exploited in both protocol processing and physical network connections.

Figure 1 shows the inter-cluster communication throughput requirement of Water, a shared memory application, also on MGS. The figure shows clearly that as the cluster size scales up, inter-cluster messages are sent more and more frequently.

We propose a scalable ICCS design based on three observations. First, when cluster size increases, more processes run concurrently in each cluster, each processes send and receive messages at a high rate through the ICCS, creating an enormous aggregated communication load, a scalable ICCS will be able to meet this increasing demand by dedicating more processing resources to inter-cluster protocol processing, and adding more NIs to each cluster. Second, occasionally, a process might send many inter-cluster messages at a burst, which can cause contention and long delays in ICCS. A parallel ICCS can greatly reduce the contention by distributing the messages among multiple communication channels and deliver them in parallel. Third, when a process needs to send out or receive a very long message, such as a large file, the ICCS can break the message into

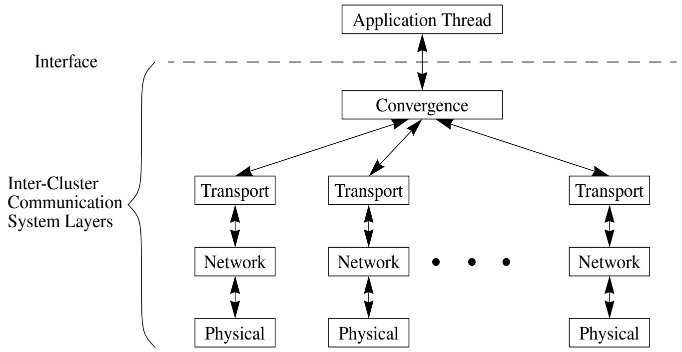


Figure 2: A model for parallel protocol stacks.

many small pieces, and send them through many channels in parallel, and reassemble them back together at the destination. The issue here is how to break, distribute, and reassemble the message pieces together efficiently. We will discuss techniques to solve these problems in the following sub- sections.

## 2.2 Parallel Protocol Stack

Figure 2 shows a parallel version of the layered protocol stack model. Standard protocol stacks, which include transport, network, and physical layers, are replicated, each runs as an independent protocol processing thread. A convergence layer is added between the application thread and the protocol processing threads of the cluster. One convergence module (either as a library function linked with the application thread or an independent thread) is assigned to each application thread. The convergence module distributes outgoing application messages to multiple replicated protocol stacks, and serializes the incoming messages before submiting them to the application thread. The convergence module's task is to hide the underlying multiple message paths from the application thread, which "thinks" that it has a single logical connection with the application thread in another cluster. All application thread/convergence module pairs share the same set of protocol stack threads in a cluster.

As Figure 3 illustrates, an inter-cluster connection between a pair of application processes has two levels. Each proxy node has a protocol stack thread running on it. Low

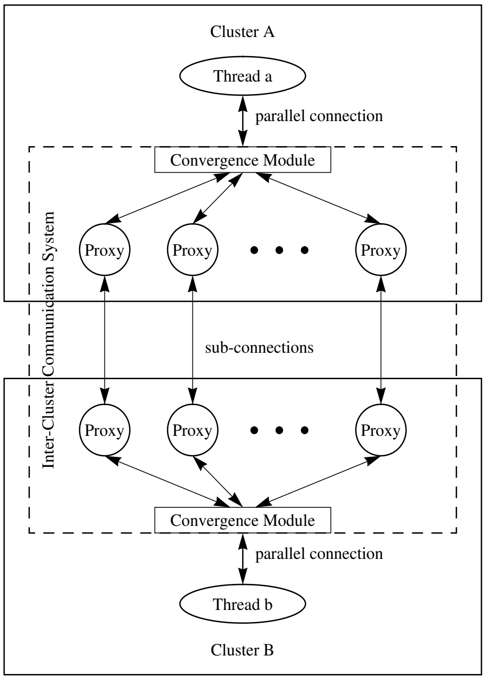


Figure 3: A schematic of parallel connections.

level sub-connections are established between transport layers of protocol stacks in different clusters. A high level logical connection, which we refer to as a parallel connection, connects two processes in different clusters together. Note sub-connections can be established independently of higher level parallel connections and shared by multiple parallel connections.

Reliability and in order delivery can be easily supported in this layered design. Each sub-connection guarantees delivery of every inter-cluster message distributed to it in order to their destined transport layer. The fact that sub-connections are reliable greatly simplifies the implementation of the convergence layer. Error detection and re-transmission are completely handled in the transport layer of each replicated protocol stack. Each sub-connection message contains a sender port id and a receiver port id, and a parallel connection sequence number. Together, the sender and receiver's port id identify the message's parallel connection. The sequence number indicates the position of the message in the whole message sequence of this parallel connection, which is used by the convergence layer to maintain the order of which it presents the received messages to the destination process. To support in order delivery, received messages with larger sequence numbers need to be queued in the convergence module until all messages with smaller sequence numbers are passed to the application thread. Some applications, such as software shared memory systems, do not require in-order delivery, which makes the convergence layer even simpler.

End-to-end flow control and buffering are also handled at two levels. At the sub-connection level, traditional techniques are well studied and can be implemented in straightforward manner. At the parallel connection level, no flow control is needed if in-order delivery is not required, otherwise a simple windowing protocol can solve the problem. Only very simple operations need to be carried out for the windowing protocol, which is not likely to become a bottolneck.

Balanced load on proxies is preferred since contention in a heavily loaded sub-connection can severely delay its message delivery. If another lightly loaded sub-connection between the two clusters is available, by sending new messages through this alternative path, much communication time can be saved.

## 2.3 Connection Establishment and Application Interface

A standard, simple interface is preferred between application processes and inter-cluster communication. We propose two ways to establish a parallel connection.

The first way is used to establish parallel connections within a single logical machine. We refer to a group of connected clusters as a single logical machine if each cluster in the group can be addressed by a unique cluster id. Machine wise sub-connections are established at the machine's initialization phase. Similar to a UNIX socket, to establish a parallel connection, a server process first registers itself to the system by binding itself to a port, and then wait for other processes to connect to it. A connect system call at the client side, which addresses the destination process by its cluster id and port number, establishes a parallel connection between itself and the server process.

The second way is used to establish parallel connections between processes in different machines (a single cluster can also be a logical machine), i.e., the clusters can not address each other by their cluster ids. All steps are similar to establishing parallel connections within a machine. The only exception is when the client process calls the connect system call, instead of addressing the destination cluster by its cluster id, it specifies all NI addresses of the cluster. The sub-connections will be established before the requested parallel connection establishes, assuming the destination cluster is ready to accept the sub-connection requests.

# 3 A Scalable ICCS Implementation on Alewife

As a demonstration, we implemented a scalable ICCS on the MIT Alewife multiprocessor. The system is integrated with the MGS system to provide inter-cluster shared memory support. This prototype provides a flexible environment to evaluate application performance on a physical implementation of clustered multiprocessors, as well as the impact of various inter-cluster communication system configurations on end-application performance. In this section, we first give a brief overview of Alewife and MGS, and then discuss the implementation in detail.

## 3.1 The Alewife Multiprocessor

Alewife is a CC-NUMA machine with efficient support for Active Messages. The machine consists of a number of homogeneous nodes connected by a 2-D mesh network. Each node contains of a modified SPARC processor called Sparcle, a floating point unit, a 64K-byte direct-mapped cache, 8M-bytes DRAM, a 2-D mesh routing chip, and a communications and memory management unit. Alewife supports DMA, which relieves the processor of bulk data transfer overhead. There are four hardware contexts in each Sparcle processor. The hardware contexts are used for fast context switching, which enables fast active message invocation.

## 3.2 The Alewife MGS System

The Alewife MGS System is a system built on Alewife to study shared memory application performance on clustered multiprocessors. It logically partitions an Alewife machine into multiple clusters, and supports memory sharing at two level of granularities. Within each cluster, MGS relies on Alewife hardware to support sharing at cache-line grain. Sharing between clusters is supported at page grain by using a software layer that sends messages through the inter-cluster communication system, which is designed in this thesis.

## 3.3 Protocol Selection

Our parallel protocol stacks each implements the Internet Protocol (IP) [2] in the network layer, and the User Datagram Protocol (UDP) [1] augmented with a simple sliding window protocol in the transport layer. The sliding window protocol provides end-to-end flow control, buffer management, and lost message re-transmission. UDP provides a checksum for message payload integrity, and IP performs fragmentation and message routing.

## 3.4 Proxy Protocol Processing Thread

Figure 4 shows the implementation of protocol processing on a proxy. Active Messages [13] are used for message passing through both intra- and inter-cluster networks. The design can be easily modified to handle message passing model. Inter-cluster messages are sent from a source compute node to a proxy in its cluster, processed and routed to a proxy of the destination cluster, and eventually delivered to the destination compute node.

When an out-going data message arrives at a proxy through the intra-cluster network, an interrupt handler is invoked on the proxy which buffers the message into a message buffer, and queues the buffer to the slid_win_send (sliding window send) module. When the slid_win_send module dequeues the message from its message processing queue, it finds the message's destination cluster from its destination descriptor, and dispatch the message to the corresponding connection. The slid_win_send module appends its header

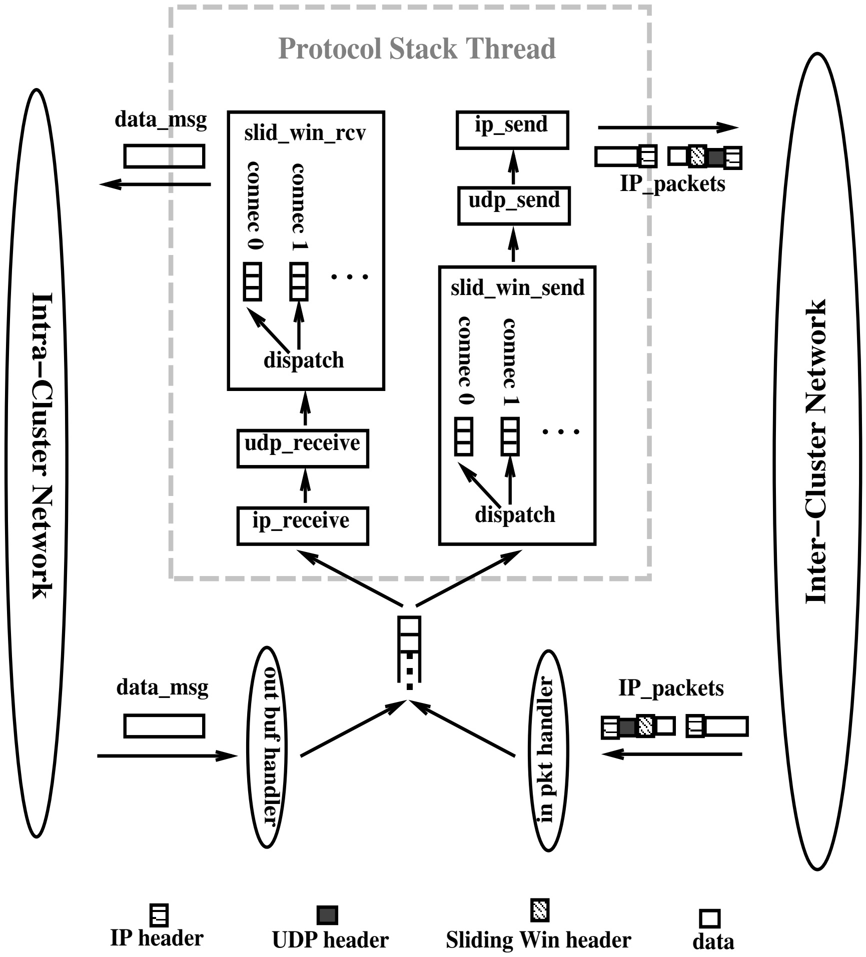


Figure 4: The protocol stack implementation on a proxy.

to the message, starts a SEND timer for the message, and passes the message to the udp_send module. A UDP header will then be appended to the message, which contains a checksum of the message contents and a port number to identify the receiving thread in the destination cluster. The ip_send module fragments the message, if necessary, and adds an IP header to each fragment. Finally, the individual fragments are sent out through the physical network interface.

To receive messages, IP packets arrive at a proxy from the Inter-cluster network. The in-buffer handler queues them to the ip_receive module through the same queue used by the out-buffer handler. IP fragments are reassembled in the ip_receive module, and passed to the udp_receive module, which validates the checksums, dropping those messages with

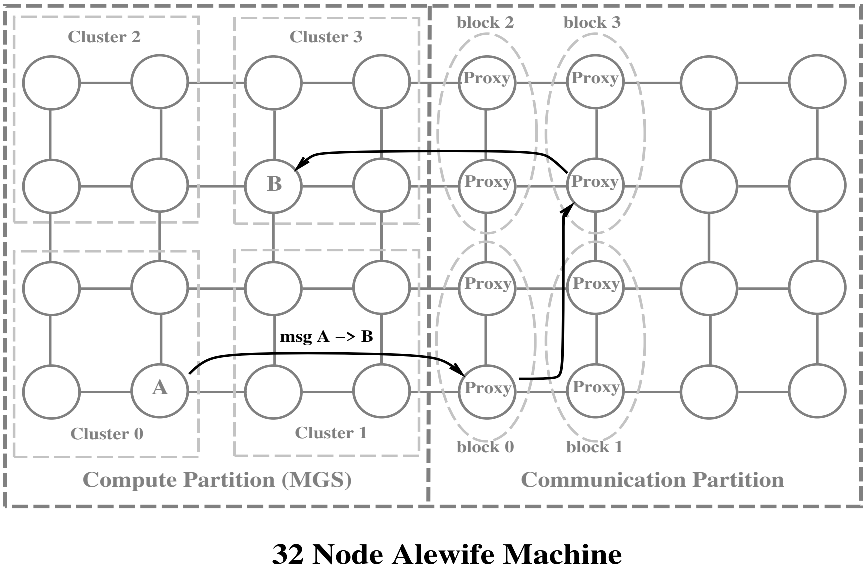


Figure 5: Configuration on a 32-node Alewife Machine.

inconsistent checksums. Messages with valid checksums are passed to the slide_win_rcv module. The slid_win_rcv dispatches the messages to the corresponding connections and eventually delivers them to their destination compute nodes via the intra-cluster network.

## 3.5 Scalable ICCS Testbed

Figure 5 shows the testbed used to evaluate our scalable ICCS design on a 32-node Alewife machine, divided into two partitions. The compute partition runs MGS on 16 nodes; all application computation is performed in this partition, and MGS further partitions it into equal sized cluster. The communication partition consists of 2 to 16 nodes (proxies), depending on the desired configuration.

Since the Alewife mesh network has very high performance relative to its processor speed, we ignore its message transmission delay and the interference between intra- and inter-cluster messages. This allows us to place a cluster's compute nodes and proxies in separate partitions, which leads to a simpler implementation.

We further divide the proxies in the communication partition into blocks, and dedicate each block to a single cluster for running its protocol stacks. As shown in Figure 5, to send a message from node A in one cluster to node B in another cluster, node A first chooses a proxy in its cluster's block, then it DMAs the message to the proxy. This proxy will then pick a connection to a destination proxy of node B's cluster, package the message in one or more IP packets, and send them through the Alewife network to the destination proxy. The destination proxy will examine the correctness of the received packets, extract the correctly delivered message data from them, and DMA the message to its destination node B.

### 3.6 Proxy Load Balancing

When each cluster is equipped with more than one proxy, load balancing among the proxies is an important issue. By sending messages to the most lightly-loaded proxies in in the cluster, contention can be reduced.

Two different approaches are implemented to distribute inter-cluster messages to proxies. The *fixed* algorithm always dispatches messages from one compute node to the same proxy, while the *round-robin* algorithm routes out-going messages from each compute node to the proxies in its cluster in a round-robin fashion. Other more intelligent algorithms which try to dynamically determine the most lightly loaded proxy are possible. Such algorithms route messages to proxies based on real-time load information. However, such algorithms are complex and difficult to justify. As our experiment results turn out, simple algorithms such as the *round-robin* algorithm provide adequate proxy load balance on the applications we studied.

## 4 Experimental Results

In this section, we present the experimental results obtained on our ICCS testbed. A total of eight parallel applications are studied.

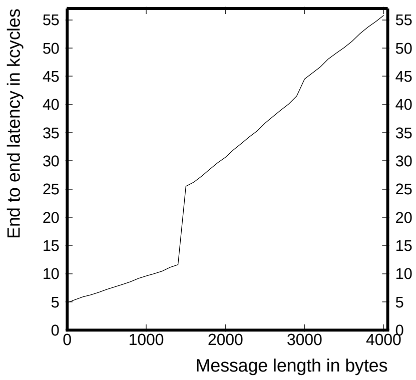


Figure 6: End to end latency vs. message size.

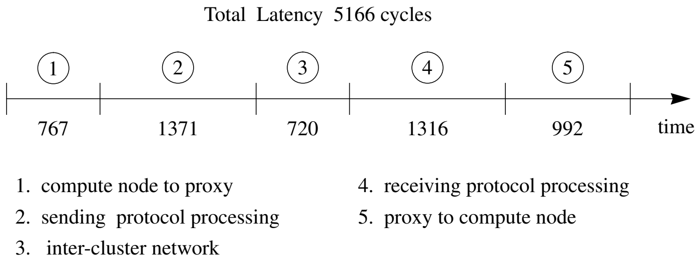


Figure 7: End to end timeline for a 100-byte message.

## 4.1 End-to-End Latency

Figure 6 depicts the end-to-end message latency as a function of message size. Message size ranges from 0 to 4000 bytes. We choose the network layer Maximum Transmit Unit (MTU) as 1500 bytes, which is the MTU for Ethernet. The latency curve jumps at 1500 bytes and 3000 bytes, because of fragmentation in the IP layer.

The breakdown of message transmission cost for a 100-byte message that we measured on our scalable ICCS is shown in Figure 7. The inter-cluster network latency, 720 machine cycles or 36 $\mu s$ (at a 20 MHz Alewife clock frequency), closely matches the reported latency on Fast Ethernet and ATM networks [14].

## 4.2 Parallel Applications Performance

Table 1 lists eight shared memory applications studied on our ICCS testbed. Jacobi is a 2-D grid relaxation program. Matrix Multiply multiplies two square matrices. Gauss performs Gaussian elimination on a matrix. FFT computes a one-dimensional fast Fourier transform. Traveling Salesman Problem (TSP) computes the solution to a 10-city traveling salesman problem using a branch and bound algorithm, and a centralized work queue to distribute work. Water is a molecular dynamics application taken from the

| Application | Problem Size |
|---|---|
| Jacobi | 2048 x 2048 Grid, 3 Iterations |
| Matrix Multiply | 384 x 384 Matrices |
| Gauss | 512 x 512 Matrix |
| FFT | 32K Elements |
| TSP | 10-City Tour |
| Water | 343 Molecules, 2 Iterations |
| Barnes-Hut | 2K Bodies, 3 Iterations |
| Unstructured | 2800 Nodes, 17377 Edges, 1 Iteration |

Table 1: List of applications and their problem sizes.

SPLASH [12] benchmark suite. Barnes-Hut is also taken from the SPLASH benchmark suite. Finally, Unstructured is a computation over an unstructured mesh from the University of Wisconsin, Madison, and the University of Maryland, College Park [10].

Figure 8 through Figure 15 present performance results of the eight applications measured on our platform, all using the round-robin algorithm for proxy load balancing. The horizontal axes of these figures are cluster size, which is explained in Figure 5. The total number of nodes dedicate to computing are fixed to 16 for all configurations.

Results for six sets of ICCS configurations are reported in Figure 8 through Figure 15. The first set of configurations are marked as “no delay”. In this set of configurations MGS are configured to send inter-cluster messages directly through the fast Alewife network, thereby bypassing all proxies and protocol stack processing overhead. This is equivalent to having an extremely high-performance ICCS which delivers inter-cluster messages instantly. Not only MGS allows us to configure it to transmit inter-cluster messages directly through the Alewife network, it also allows us to insert an arbitrary delay before it launches every inter-cluster message into the network. In the set of configurations marked as “no contention”, for each application, we first measure its average inter-cluster message length, then send a message of the same length through an idle ICCS, measure its latency, configure the MGS to use this latency as the inserted delay, run the application on this MGS, and record its execution time. In some sense this execution time represents the ap-

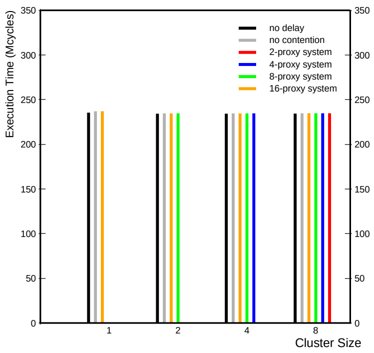


Figure 8: Jacobi

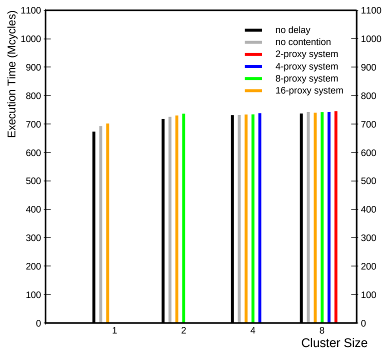


Figure 9: Matrix Multiply

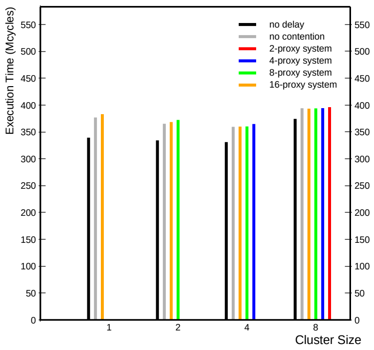


Figure 10: Gauss

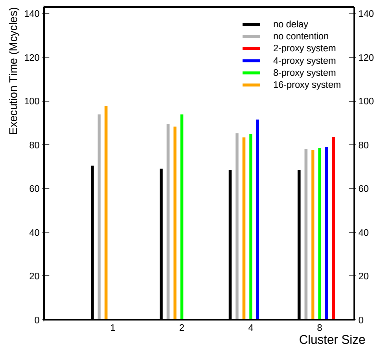


Figure 11: FFT

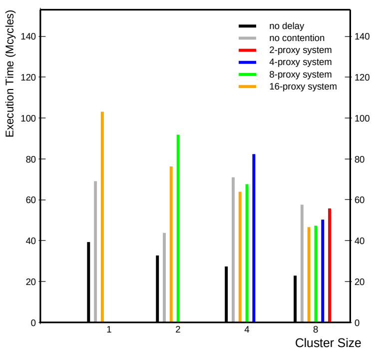


Figure 12: TSP

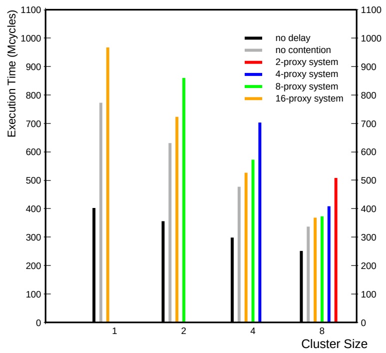


Figure 13: Water

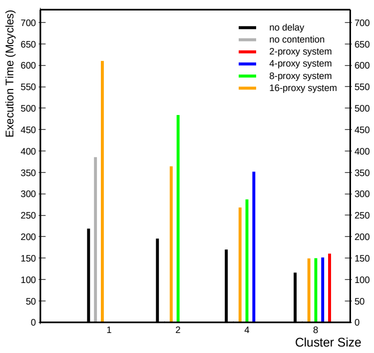


Figure 14: Barnes-Hut

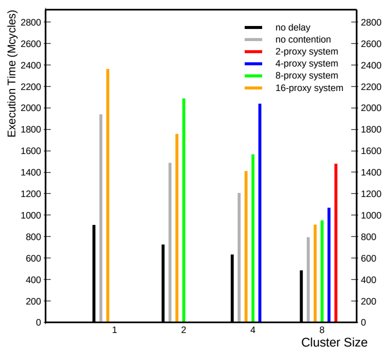


Figure 15: Unstructured

plication performance on an ICCS with zero contention, though in general this approach over delays short messages, and under delays long messages. All other four sets of configurations send inter-cluster messages through the ICCS, with the total number of proxies available as 2, 4, 8, 16. The total number of proxies per cluster in all configurations varies from 1 to the cluster size. When fewer proxies are available, more messages have to be sent through the same proxy. Contention in the ICCS will increase inter-cluster messaging latency, thus negatively impacting application performance.

Three out of the eight applications, Jacobi, Matrix Multiply, and Gauss, are computation-bound. Their performance is insensitive to the latency in the ICCS and therefor to ICCS contention. For communication intensive applications, TSP, Water, Barnes-Hut, and Unstructured, a much higher inter-cluster messaging frequency is observed. These applications are much more sensitive to ICCS latency. They put heavy load on the ICCS, increasing contention and decreasing application performance. Figure 11 to Figure 15 clearly show that by increasing the number of proxies per cluster, application performance improves dramatically since the increased throughput greatly reduces contention in the ICCS. By comparing application execution time between the "no contention" configurations and that with various number of proxies, the impact of contention in the ICCS on application performance is obvious. This supports our argument made earlier in the paper that scalable ICCSs are important for the success of scalable clustered multiprocessors.

## 4.3 Scalability of Intra-cluster communication system

An important question we intend to answer in this paper is when building a balanced clustered multiprocessor, how should an inter-cluster communication system scale with its machine and cluster size? This section presents some early intuition about the answer to this question, and some data which supports our intuition.

Our intuition suggests that while cluster size increases, the number of proxies needed

[2] Part of the data presented in these tables are not measured. Instead they are computed by linear interpolating values from the nearest measured points.

| clu size | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| clu proxies | 1 | 1.41 | 2 | 2.83 |
| msg time | 8920 | 9232 | 8017 | 8243 |
| que len | 0.534 | 0.636 | 0.502 | 0.556 |
| proxy util | 0.352 | 0.387 | 0.362 | 0.382 |
| slowdown | 2.403 | 2.256 | 1.918 | 1.567 |

Table 2: Scaling inter-cluster communication size with cluster size [2] (Water).

per cluster should scale up as well, but not as fast. Communication between clusters in a clustered multiprocessor occurs via message passing through an inter-cluster communication system, which is always more costly compared to local access of data within the same cluster. To achieve good performance, applications should always carefully place data to exploit locality, either manually or automatically using compiler technology. Due to communication locality, when cluster size increases, the average messaging rate between clusters decreases, which means when cluster size increases, the aggregate inter-cluster messaging rate will also increase, but less rapidly. Furthermore, the increase of the average interval between two inter-cluster messages at compute nodes reduces the sensitivity of application performance to inter-cluster communication delay.

Results in Table 2 show for Water, the average message delay in the ICCS, average queue length in proxies, average proxy utilization, and run time slow down comparing with "no delay" measurments when cluster size and number of proxies per cluster scales. When we scale the number of proxies per cluster as the square root of cluster size, the message processing time, queue length, and proxy utilization remain relatively the same. The application slowdown even improves with the scaling. This result consistent with our discussion above.

We also carried out similar analysis for the other three communication intensive applications. Those results support our intuition as well.

## 5 Related Work

This paper focuses on building scalable inter-cluster communication system using unreliable commodity LAN technologies. Reliability and flow control are built in software protocol layers. Three main issues need to be addressed to build a highly efficient system this way, namely, low cost user kernel boundary crossing, parallel protocol stack processing, and low network latency.

Traditional user kernel boundary crossing are very expensive since expensive context switch are used. Lim[9] and el. suggested a zero context switching design which dedicates a proxy node in each SMP cluster to handle inter-cluster network services. Their design achieves good performance for small clusters. When cluster size scales up, contention in proxies quickly degrades the overall application performance.

Parallel protocol stack processing has not been carefully studied for clustered multiprocessors that target parallel applications. Results from researchers often assume that networks are reliable. Related work ([6], [8], and [5]) have concentrated on increasing throughput rather than reducing contention to reduce latency.

Commodity LAN technologies are getting faster. Welsh *et. al.* [14] have studied how to provide fast user-level communication on Fast Ethernet and ATM. Their results are impressive, but to provide efficient inter-cluster communication for parallel applications, parallelism still has to be exploited.

Much work ([4], [11], and [7]) has been done to provide efficient, protected inter-cluster communication by building custom hardware. We focus on commodity hardware given the enormous cost advantage of leveraging commodity components.

## 6 Summary

This paper describes issues of how to design scalable, efficient, reliable inter-cluster communication systems using commodity components. The design achieves high performance by parallel execution of protocol stack processing on multiple proxy nodes, and by providing multiple physical network connections from each cluster to the inter-cluster network. We addresses several issues necessary in a scalable design, such as application interface, reliability, in-order delivery, end to end flow control, and proxy load balancing. A prototype of the design is implemented and integrated with the MGS software shared memory system. Performance of numerous shared memory applications are studied on the platform. We draw the following conclusions from this study:

- On clustered multiprocessors with small cluster size, for communication intensive applications to achieve good performance, proxy performance should be comparable with those of the compute nodes.
- When cluster size scales up, contention at proxies can be significant, making the exploitation of parallelism in inter-cluster communication systems crucial.
- For fixed machine size, when cluster size scales up, to deliver the speed up potential provided by the larger cluster size, the number of proxies in each cluster also needs to increase. However, a sub-linear increase is likely to be sufficient. For the applications and configurations we tested, an increase of the number of proxies per cluster as the square root of the cluster size stables the application slow down factor.
- Simple round-robin message dispatching algorithms achieve good results for proxy load balancing within each cluster. More complicated algorithms will not likely increase application performance much. (measurement results are omitted to keep the paper from getting too long).

In conclusion, our results suggest that scalable, efficient, reliable inter-cluster communication systems built using commodity components are possible.

# References

[1] RFC 768, User Datagram Protocol.

[2] RFC 791, Internet Protocol.

[3] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

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

[5] Schmidt D.C. and Suda T. Measuring the Performance of Parallel Message-based Process Architectures. In *IEEE INFOCOM*, pages 624–633, Boston, MA, April 1995.

[6] Yates D.J., Nahum E.M., Kurose J.F., and Towsley D. Networking Support for Large Lcale Multiprocessor Servers. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurment and Modeling of Computer Systems*, pages 116–125, Philadelphia, PA, May 1996.

[7] Richard Gillett, Michael Collins, and David Pimm. Overview of Memory Channel Network for PCI. In *Proceedings of the 41st Annual IEEE Computer Society Computer Conference*, pages 244–249, Santa Clara, CA, February 1996.

[8] N. Jain, M. Schwartz, and T.R. Bashkow. Transport Protocol Processing at Gbps Rates. In *SIGCOMM*, pages 188–199, Philadelphia, PA, September 1990.

[9] Beng-Hong Lim, Philip Heldelberger, Pratap Pattnaik, and Marc Snir. Message Proxies for Efficient, Protected Communication on SMP Clusters. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 116–127, February 1997.

[10] Shubu Mukherjee, Shamik Sharma, Mark Hill, Jim Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines.

In *Proceedings of the 5th Annual Symposium on Principles and Practice of Parallel Programming*, pages 68–79, July 1995.

[11] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, May 1996.

[12] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[13] 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.

[14] Matt Welsh, Anidya Basu, and Thorsten von Eicken. ATM and Fast Ethernet Network Interfaces for User-Level Communication. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 332–342, February 1997.

[15] Donald Yeung, John Kubiatowicz, and Anant Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 44–55, May 1996.