A Scalable Parallel Inter-Cluster Communication System for Clustered Multiprocessors

by

Xiaohu Jiang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1997

Author .....
Department of Electrical Engineering and Computer Science
August 21, 1997

Certified by .....
Anant Agarwal
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

A Scalable Parallel Inter-Cluster Communication System for Clustered Multiprocessors

by

Xiaohu Jiang

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 1997, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Clustered multiprocessors have been proposed as a cost-effective way for building large-scale parallel computers. A reliable and highly efficient inter-cluster communication system is a key for the success of this approach. This thesis presents a design of a scalable parallel inter-cluster communication system. The system achieves high bandwidth and low latency by leveraging parallelism in protocol processing and network access within each cluster. Intelligent Network Interfaces (INIs), which are network interface cards equipped with protocol processors, are used as building blocks for the system. 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 performance of Water, a SPLASH benchmark, is studied in detail on the platform.

Our results show that the introduction of a software protocol stack in the inter-cluster communication system can increase application run time by as much as a factor of two. For applications with high inter-cluster communication requirements, contention at INIs can be severe when multiple compute nodes share a single INI. Our initial results also suggest that for a given application and machine size, when the size of clusters and their inter-cluster communication system are scaled proportionally, contention in inter-cluster communication levels off. For Water, we found that the impact of inter-cluster communication overhead on overall run time reduces even when the number of INIs assigned to each cluster scales with the square root of the cluster size. Again, this result assumes a fixed machine size.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

I am really grateful to Donald Yeung, who introduced me to the topic of this thesis. Through the long discussions we had during the past year or so, Donald has been consistently giving me guidance, as well as hands on help for my research. Many thanks should also go to John Kubiatowicz, who answered all my questions about the Alewife system. In addition, Anant Agarwal, My advisor, has guided me all the way through my research.

Many other people also have given me help. Among them, members of the Alewife group, Ken Mackenzie, Victor Lee, Walter Lee, Matt Frank provided me insight to various problems through discussions. My officemates Michael Taylor and Benjamin Greenwald have generated a friendly and amusing environment, which makes the whole experience enjoyable.

Finally, I want to thank my girl friend, Xiaowei Yang. Whose love and emotional support make me feel that every day of my life, busy or relaxed, is a wonderful day.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While traditional massively parallel processors (MPPs) can achieve good performance on a variety of important applications, the poor cost-performance of these systems prevent them from becoming widely available. In recent years, small- to medium-scale multiprocessors, such as bus-based Symmetric Multiprocessors (SMPs), are quickly emerging. This class of machines can exploit parallelism in applications to achieve high performance, and simultaneously benefit from the economy of high volume because their small-scale nature allows them to be commodity components. Many researchers believe that by using these smaller multiprocessors as building blocks, high performance MPPs can be built in a cost-effective way. In this thesis, we call these small- to medium-scale multiprocessors as clusters, and the MPPs built by assembling these clusters together as clustered multiprocessors. Furthermore, we define an inter-cluster communication system as the combination of a inter-cluster network, which is usually a commodity Local Area Network (LAN), some number of cluster network interface cards, and processor resources used to execute inter-cluster message related protocol processing in each cluster.

Low latency, high throughput, reliable inter-cluster communication is one of the keys to the success of clustered multiprocessors. Technologies such as block multi-threading have been used by shared memory multiprocessors to hide cache miss and remote cache line fetch latency, they can also be used for inter-cluster communication latency tolerance, but the potential is limited. When out of available contexts to switch to, compute nodes have to be blocked to wait for the arrival of the long delayed inter-cluster message replies.

Large amount of processor cycles can be wasted. Moreover, as required by most message passing interfaces and inter-cluster shared memory protocols, inter-cluster communication systems have to provide reliable communication between clusters.

Using commodity components as building blocks is the key reason for why clustered multiprocessors are cost effective. The economy also pushes designers to design inter-cluster network using commodity technology. Though high-end LAN technology promises high throughput and low latency, it does not guarantee reliability. To provide reliable communication between clusters, software protocol stacks have to be introduced, which dramatically increase the protocol processing overhead.

To perform complicated software protocol processing while still maintains high performance, we propose to execute protocol stacks in parallel using multiple protocol processors in each cluster. Parallelism is exploited by simultaneously processing multiple messages on separate protocol processors through all phases of the protocol stacks.

In this thesis we present a simple design of a scalable parallel inter-cluster communication system using Intelligent Network Interfaces (INIs). An INI is basically a network card equipped with a protocol processor. We also refer to an INI as an INI node. In a cluster, each INI node runs its own protocol stack, and interfaces to the inter-cluster network directly. Messages can be scheduled to different INI nodes of the cluster to balance load of the INI nodes. A prototype implementation of the design is built on the MIT Alewife machine [3].

We integrate our prototype with the MIT Multi-Grain Shared-memory system (MGS) [10], which is a Distributed Scalable Shared-memory Multiprocessor (DSSMP) implemented also on Alewife. The reasons that we choose to integrate our system with MGS are: first, the DSSMP is an important application for clustered multiprocessors; second, DSSMPs creates high inter-cluster communication traffic; third, each cluster in a DSSMP is scalable, thus requiring a scalable inter-cluster communication system.

The prototype we build will provide a platform for studying how applications will stress inter-cluster INI systems, and how much the INI performance will impact the end-application performance. Using this platform, the thesis conducts an in-depth study which intends to answer the following questions:

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

- How much parallelism exists in inter-cluster communication traffic that can be effectively exploited?

- By how much does contention impact inter-cluster communication system performance? How can contention be reduced?

The goal of this research is to find out for a given cluster and machine configuration, what is the performance requirement of the inter-cluster communication system? and how can an inter-cluster communication system be built in a cost-effective way?

The results of our study show that scalable parallel inter-cluster communication systems can be designed in a straight-forward way. From our results, we can draw three observations. First, when cluster size is one, for shared memory application with high inter-cluster communication requirement, the inter-cluster communication protocol stack processing overhead is comparable to the application computation overhead. Assuming an INI node is assigned to each one compute node cluster, this implies that for these applications to achieve reasonable performance, the processing power of an INI node and a compute node has to be comparable. Second, for a given application and machine size, when the size of clusters and their inter-cluster communication system are scaled proportionally, contention in inter-cluster communication levels off. This lead us to believe that to build a balanced clustered multiprocessor, when scale up its cluster size while keep the machine size unchanged, the clusters' inter-cluster communication systems need only scale up at a lower rate compare to their cluster size. One explanation for this phenomenon is that by only increasing cluster size but not machine size, clusters are simply merge into larger clusters, part of the original inter-cluster message are now delivered via intra-cluster network. Third, for shared memory applications, compute nodes in a cluster generally have balanced inter-cluster communication requirements, since complicated strategies which intend to balance protocol processing load on INI nodes within a cluster do not improve application performance much.

Following this introduction, the next chapter provides an overview of clustered multiprocessors. Chapter 3 presents the design of our parallel scalable inter-cluster communication system. Chapter 4 discusses the implementation of the prototype on Alewife platform.

Chapter 5 describes the DSSMP applications studied on our prototype. Chapter 6 presents an analytical model of the clustered multiprocessor system. Chapter 7 concludes by examining the implications of our results, and draw some insights for inter-cluster communication system design.

Chapter 2

Clustered Multiprocessors

As shown in Figure 2-1, the clustered multiprocessors we propose have a two-level architecture. Multiple Processing Elements (PEs) are connected by an intra-cluster network to form a cluster, while numerous clusters are linked together to form the clustered multiprocessor. As mentioned in the previous chapter, we add one or more INI nodes in each cluster to handle protocol stack processing for inter-cluster messages.

Within a cluster, an intra-cluster network provides high speed, reliable communication between PEs. A cluster is a small- to medium-scale hardware cache-coherent shared-memory multiprocessor. Its interconnect is built using a high performance VLSI network, which provides reliable message delivery.

Unlike the intra-cluster network, the desire to use commodity technology outside each cluster requires the inter-cluster network to use LANs which are unreliable. The unreliability of LANs necessitates building reliability protocol processing in software using expensive software. To leverage parallelism in the inter-cluster communication system and to achieve good modularity and scalability, we propose to build the inter-cluster communication system using INI nodes within each cluster. An INI node is a module contains a processor for protocol processing, some amount of memory, network interfaces to both intra- and inter-cluster network. In this thesis, we assume an INI is a single physical component, but our design can easily adapt to other configurations such as processing protocol stacks on SMP PEs, instead of on network cards.

In our design, each INI node runs its own protocol stack and interfaces directly to

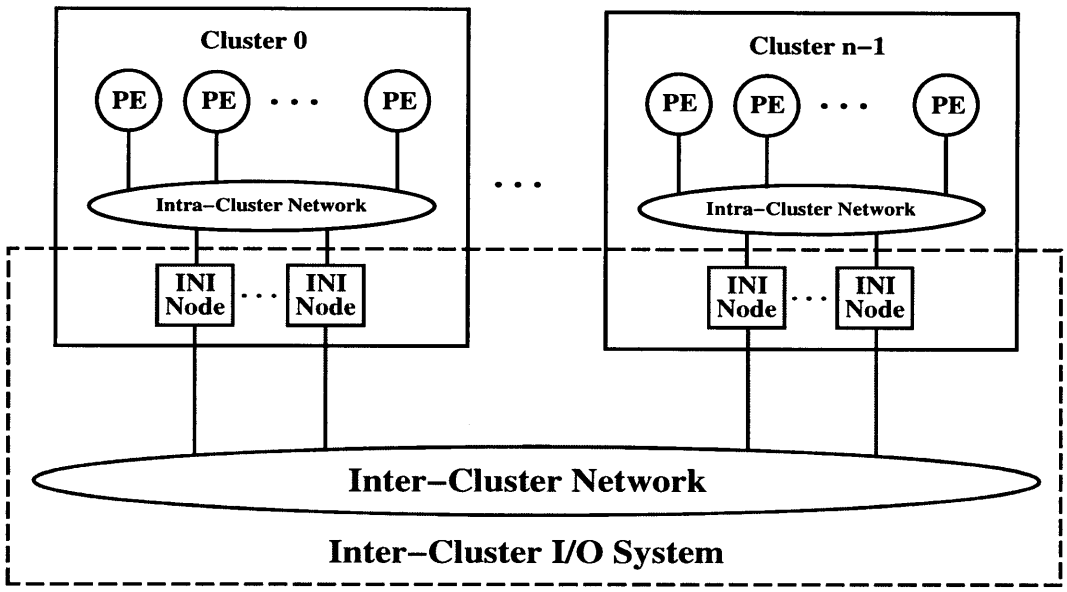

Figure 2-1: A clustered multiprocessor.

the inter-cluster network. Good performance is achieved by parallelizing both protocol processing and data movement. To send out a message, first the message data is placed in the INI nodes' memory, along with a destination descriptor. Based on the message's destination, the INI node will then choose a connection to an INI node of the destined cluster, push the message down through the protocol stack, fragment it into packets if necessary, and eventually deliver the packets to the destined cluster through the inter-cluster network. Part of the destination descriptor is also wrapped in the message packets. At the receiving INI node, the packets are reassembled back into a single message. The received message is then delivered to the appropriate PE based on the destination information which is received along with the message. The correctness checking and lost message re-transmission is provided in the transport layer of the protocol stack.

Chapter 3

System Design

This chapter presents the design of a scalable parallel inter-cluster communication system. Our design goal is to provide reliable inter-cluster communication with low protocol stack processing overhead. Many design decisions are based on the assumption that the inter-cluster network is high performance, low error rate network, which allows our design to be simple and efficient.

Our system uses INI nodes as building blocks. INI nodes combine protocol processing resources and network interfaces into one module, which permits a physically scalable implementation. The design can be trivially modified to fit architectures which distribute processing resources and network interface in different modules.

3.1 Protocol Stack

Figure 3-1 shows the design of the INI node protocol stack thread. Here we assume Active Messages [9] are used for message passing through both intra- and inter-cluster networks. The design can easily adapt to any other general message passing model. Inter-cluster messages are send from its source compute node to an INI node of its cluster, processed and routed to an INI node of the destination cluster, and eventually delivered to the destination INI node.

When an out-going data message arrives at an INI node through the intra-cluster network. An interrupt handler is invoked on on the INI node which buffers the message into a message

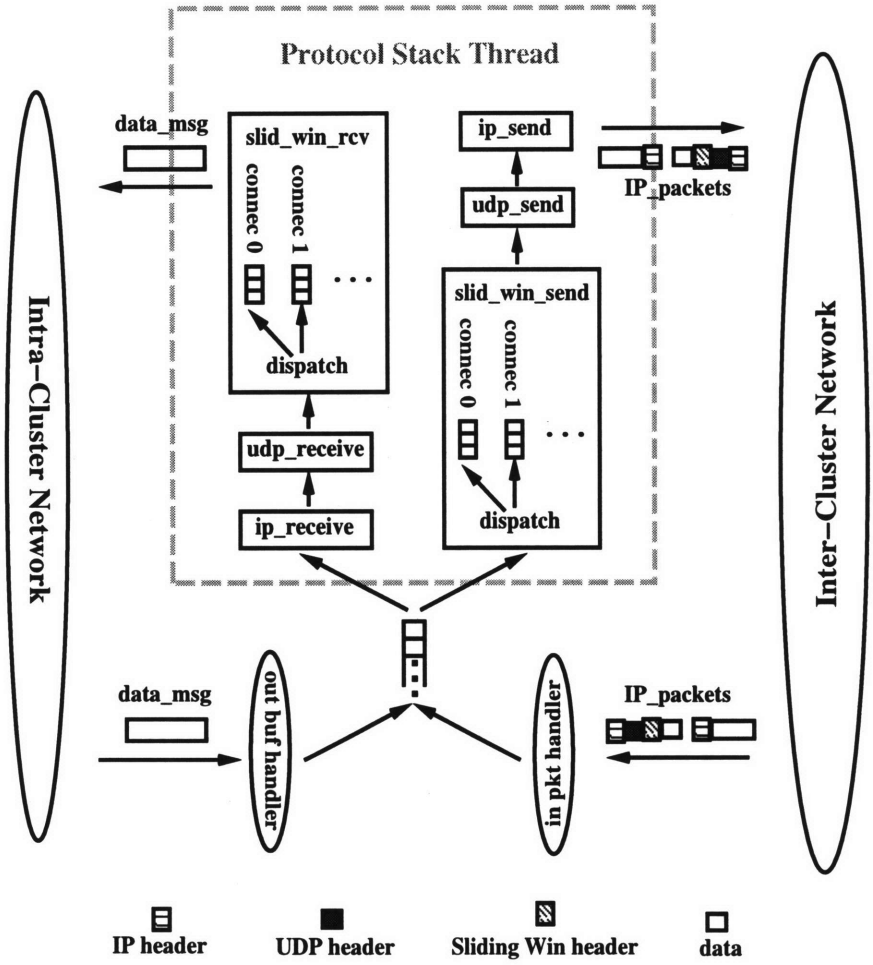

Figure 3-1: INI node protocol stack implementation.

buffer in the INI node's memory, and queues the buffer to the sliding window sending module. When the sliding window send module dequeues the message from its message processing queue, it finds the message's destination cluster from its destination descriptor, chooses a connection to an INI node on destined cluster, and dispatches the message to the connection. The sliding window send module will append its header to the message, start a SEND timer for the message, and pass the message to the UDP sending module. A UDP header will be appended to the message, which contains a checksum of the message contents. IP sending module will fragment the message if necessary, add an IP header to each fragment, and send the fragments out through the physical the physical network interface.

To receive messages, IP packets arrive an INI node from the Inter-cluster network. The

in-buffer handler queues them to the IP receiving module through the same queue used by the out-buffer handler. IP fragments are reassembled in the IP receiving module, and passed to the UDP receiving module, which validates the checksums, dropping those messages with inconsistent checksums. Messages with valid checksums are passed the sliding window receiving module. The sliding window protocol receiving module dispatches the messages to the corresponding connections and eventually delivers them to their destination compute nodes via intra-cluster network.

The queue used between handlers and the protocol stack thread is a First In First Out (FIFO) queue under most circumstances, except when the top message is an outgoing message, and the sending window of its connection is filled up since blocking the connection's outgoing path. In this situation the elements deeper in the queue will be processed before the top one.

3.2 End-to-end Reliability

The criteria of our protocol selection is to provide reliability with low overhead. We also favor commodity LAN protocols because of its wide availability. Our design chooses the Internet Protocol (IP) [2] as the network layer protocol, and the User Datagram Protocol (UDP) [1] augmented with a simple sliding window protocol as the transport layer protocol. Among the three protocols, the sliding window protocol provides lost message re-sending and duplication checking, UDP provides correctness checking, and IP does fragmentation and routes messages to their destination.

In our design, an INI node has a logical connection to every INI node it might communicate with. Connections at INI nodes are statically allocated. At booting time, machine configuration packets are broadcasted to every INI node. All INI nodes will then initialize its protocol stack, and set up a connection to each INI node of other clusters in the machine. After the initialization phase, each INI node runs its protocol stack independently from other INI nodes. This implies that in-order delivery can only be enforced for messages going through the same connection.

3.2.1 The Sliding Window Protocol

The sliding window protocol tags each message with a sequence number. Sequence numbers of different connections are independent of each other. At any instant in time, the sending side of a connection maintains a window which contains a set of consecutive sequence numbers it is permitted to send. The sender can continue to send out new frames until its window filled up without waiting for acknowledgments. The sending window moves forward every time the sender receives an acknowledgment of the message with the sequence number at the window bottom.

There is a logical SEND timer associated with each unacknowledged message. A retransmission will be triggered if an ACK for the message has not been received until the timer timeout. Acknowledgments are sent back to the sender either piggy-backed on incoming data messages or via special ACK messages. Similarly, the receiving side of the connection maintains a window of sequence numbers corresponding to messages it is allowed to receive. Every time the message with the sequence number at the window bottom is received, it is delivered to its compute node and the receiving window moves forward. Duplicated messages and messages whose sequence number fall out of the window are rejected. There is an ACK timer for each connection. The timer triggers a special ACK message to be sent if there is no reverse traffic on the connection for a certain amount of time.

Since we assume a low message loss rate, we choose not to do runtime timer adjustment; instead, we set a fixed SEND and ACK timeout for all messages. The SEND timeout is set several times longer than the average message Round Trip Time (RTT), and the ACK timeout is set relatively smaller. As an alternative to accurate SEND timers, we use NAKs to speed up lost message re-transmit. The sliding window protocol will optionally send out NAK messages if the received message sequence number is out of order. Each NAK message is a special message that requests the sender to retransmit a message identified by the NAK message.

Our sliding window protocol header contains four fields, an opcode (DATA, ACK, or NAK), a connection id, a SEND sequence number, and an ACK sequence number.

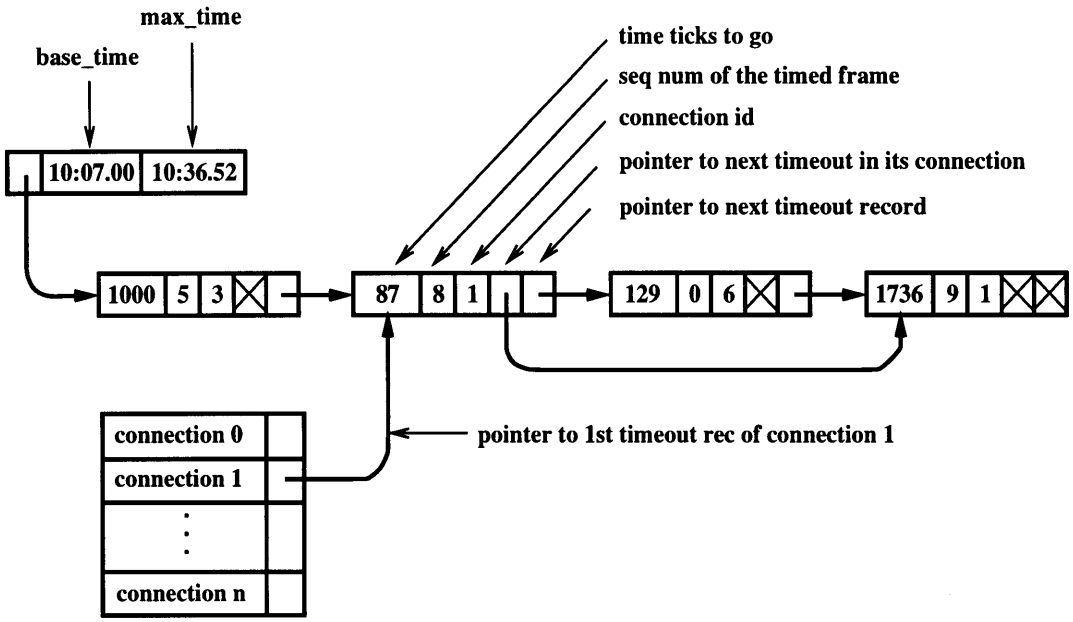

Figure 3-2: The send_timers structure.

3.2.2 Software Timers

Logically, our sliding window protocol allows multiple SEND timers and an ACK timer per connection. Each INI node can potentially need to maintain many logical timers simultaneously. Using a physical hardware timer for every logical timer will be costly or even impossible for certain architectures. To reduce timer checking cost, our design uses software timers. There are two software timer structures for each INI node, a send_timer structure and an ack_timer structure. Two hardware timers are used, one for each these software timer structures. They will be referred to as the hardware SEND timer and the hardware ACK timer.

As shown in Figure 3-2, the send_timers structure contains a base time field, a max_time field, and a doubling linked list which links all pending SEND timers together. All SEND timers of the same connection are also linked together. Each timer record has six fields. The time ticks to go field stores the number of time ticks the timer will timeout after its previous timer timeout. The second field records the sequence number of the unacknowledged message timed by the timer. The third field is the connection id of the message. The forth field is the pointer used to form the link list which links records for the same connection together. The last two fields are pre- and next-pointers used to form the doubly linked list.

We do not show the pre-pointers in the diagram to keep it simple.

If there is no pending SEND timer in an INI node when a SEND timer is about to start, the INI node records the current time in the base_time field of the send_timers structure, and sets the max_time field as the sum of current time and SEND time out. The node then allocates a timer record, sets its time ticks to go field using the SEND timeout, places it as the first record in the doubly linked list and the link list of its connection, and starts the hardware SEND timer. If there are pending SEND timers in the node when a new SEND timer needs to start, a timer record needs to be allocated. The node computes the record's time ticks to go by subtracting the max_time value from the sum of current time and the SEND timeout, links it into the link lists. Since all SEND timers have the same timeout, every new SEND timer will expire only after all currently pending SEND timers expire. This allows us to trivially compute the time ticks to go field value of the new record, and always place it at the end of the link lists. When a SEND timer times out, the node removes the first record from the link lists, and the hardware SEND timer restarts using the value in the time ticks to go field of the next record. Every time a piggy-backed ACK comes in, the node goes through all records linked on its connection's link list and delinks those records acknowledged by the ACK. The total number of active SEND timers are bounded by the aggregate window size of all connections of the INI node. Timer records are recycled.

The ack_timers structure is similar to the send_timers structure, except there is at most one active ACK timer per connection, which implies the connection link list used in send_timers structure can be omitted in the ack_timers structure.

3.3 Load Balancing

When each cluster is equipped with more than one INI node, load balancing among INI nodes is an important issue. By sending messages to lightly loaded INI nodes in the cluster, we can reduce the amount of contention at the INI nodes, and this decrease the latency of inter-cluster messages.

There are numerous algorithms to balance load on INI nodes in the same clusters. the *static* algorithm statically allocates each compute node to an INI node in its cluster, and

always routes messages from a compute node to its assigned INI node. The *round-robin* algorithm routes out-going messages from each compute node to the INI nodes of its cluster in a round-robin fashion. More general algorithms which intend to always direct a message to a lightly loaded INI node along its route requires those nodes which make routing decisions to monitor the load condition on other INI nodes at real time, which can be complicated and costly. Application performance increases by applying complicated load balancing algorithms only when the speed-up obtained out-weighs the cost of the algorithm.

In the following chapters, we will discuss how to implement various INI load balancing algorithms in our system. By comparing application performance and the amount of INI load inbalance measured during application runs, we discuss which algorithm delivers the best application performance, and how much performance gain can applications potentially get from implementing complicated INI load-balancing algorithms in the inter-cluster communication system.

3.4 Related Work

Much research has been performed in the area of parallelizing protocol stack processing on multiprocessor platforms. Among them, Yates et seq. [5] studied connection-level parallelism of TCP/IP and UDP/IP protocol stacks. Their work focuses on how throughput scales with the number of processors, as well as the number of connections, and the fairness of bandwidth distribution across connections. In contrast to our work, Yates' work targets on networking support for multiprocessor servers, to which throughput is the only major concern. Our work targets inter-cluster communication for single shared memory applications where low latency as well as high throughput are desired. Jain [7] presented implementations of parallelizing transport protocol processing at various granularities. Their implementations assume special hardware support for time consuming computations such as checksum calculation and framing. In our work, adding special-purpose hardware is undesirable since we are targeting inter-cluster communication system for commodity systems. Schmidt [4] compared performance between implementations of exploiting connectional parallelism and connection-oriented message parallelism. Connectional parallelism de-

multiplexes all messages bound for the same connection onto the same process, whereas message parallelism demultiplexes messages onto any available process. In his implementation, each node can simultaneously support multiple protocol processing processes, and protocol processing nodes synchronization are performed via shared memory. Schmidt concluded the connectional parallelism is more suitable than message parallelism as the number of connections is no less than the number of processor used for protocol processing. Our design exploits connectional parallelism. There is only one protocol processing thread on each INI node, and it has connections to all INI nodes of other clusters. The number of connections going out of a cluster is always no less than the number of INI nodes assigned to that cluster.

Chapter 4

Prototype Implementation

To demonstrate our design of a scalable parallel inter-cluster communication system can achieve good performance, a prototype system is implemented on the 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 the following sections of this chapter, we first give a brief overview of Alewife and MGS, and then discuss the implementation of our scalable parallel inter-cluster communication system in detail.

4.1 Alewife

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 consists of a modified SPARC processor called Sparcle, a floating point unit, a 64K-byte direct-mapped cache, 8M-bytes RAM, 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 Alewife node processor. This enables fast context switching which is critical for fast active message handler invocation.

On Alewife, each node can send messages to any other node connected with it by the mesh network. There's a two word header for each user- or kernel-level message. The first word contains a six bit opcode and a destination descriptor which is used for routing the message to its destination node. The second is a pointer to a handler function. When a message arrives at it's destination, it interrupts the running non-atomic thread. The destination processor automatically invoke the handler, and returns to the interrupted thread after the handler finishes. Message handlers run atomically on Alewife; if a message arrives at a node while the previous message handler is still running, it will be queued in the Alewife hardware FIFO queue. To prevent messages from blocking the network, it is important to keep message handlers simple so that they will terminate quickly (on the order of sever hundred cycles). On an unloaded Alewife machine, a system-level handler can be entered in approximately 35 cycles[3]. This time includes reading the message from the network, dispatching on an opcode in the header, and setting up for a procedure call to a handler routine written in C.

Alewife's interconnect uses the CalTech Mesh Routing Chip (MRC) as its network router. The MRC supports two dimensions of routing with two unidirectional 8-bit channels along each dimension. Each channel has a peak bandwidth of 48 Mbytes/sec. Currently the Sparcle processor of our machine configuration runs at 20 MHz. Compared to the processor throughput, the Alewife network provides very low latency and high bandwidth communication between Alewife nodes.

4.2 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.

4.3 Protocol Implementation

Our prototype system fully supports UDP and IP protocols. The sliding window protocol in our implementation has a four byte header. There are four fields in the header, all have a length of a byte, or eight bits. The first field is an opcode, currently only three opcodes (DATA, ACK, and NAK) are used. The second field is a connection id, which is used by the receiving node to dispatch the received message to the correct sliding window protocol connection. For data messages, the third field contains an eight-bit sequence number of the data frame. This field is unused for control messages (ACK or NAK). The last field is used to carry the piggy-backed acknowledgment sequence number.

With a limited sequence number range, windowing-based protocols have to reuse their sequence numbers. The protocols will fail if a sequence number conflict occurs. Suppose that for some reason a message is delayed in the network for a long time. The sender may time out and retransmit the message. If the retransmitted message arrives before the delayed message, the sliding window protocol will then move on and transmit the following frames, which will continue to consume sequence numbers. Due to sequence number reuse, after a certain period of time, the protocol window will wrap around. Thus, new messages will receive sequence numbers that alias with the sequence numbers of outstanding messages. A protocol failure will happen if the delayed message arrives at its destination at this time, and is falsely recognized as a valid new message.

To avoid unreasonablely long network delay from happening, IP bounds the maximum message life time in the network to no larger than 256 seconds[2]. This ensures that any windowing protocol which does not reuse any of its sequence numbers within 256 seconds can be guaranteed to avoid the sequence number aliasing problem. The maximum throughput of our implementation is about 10^4 messages/sec, which means our sliding window protocol needs a sequence number range of 22 bits. Currently, we choose our sliding window sequence number range as 8 bits, simply for protocol cleanness. A larger sequence number range can be trivially adjusted in our implementation, and we expect this change will not cause any noticeable change in our system.

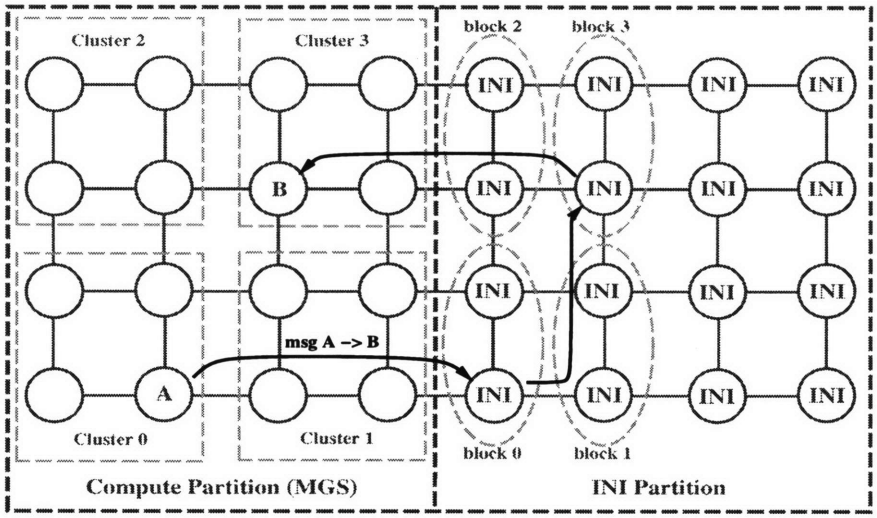

Figure 4-1: Configuration on a 32-node Alewife Machine.

4.4 System Configurability

Figure 4-1 shows the configuration of our prototype system built on a 32-node Alewife machine. The machine is first divided into two partitions. The compute partition runs MGS on 16 nodes; all application computation is performed in this partition. The INI partition consists of 2 to 16 nodes, depending on the desired configuration.

Since the Alewife mesh interconnect has very high performance relative to its processor speed, we can 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 INI nodes in separate partitions, which leads to a simpler implementation.

Our scalable parallel communication system is implemented in the INI partition. We further divide the nodes in the INI partition into blocks, and dedicate each block to a single cluster for running its protocol stacks. As shown in Figure 4-1, to send a message from node A in one cluster to node B in another cluster, node A first chooses an INI node in its cluster's INI block, then it DMAs the message to the INI node. This INI node will then pick an INI node in the INI block of node B's cluster, package the message in one or more IP packets, and send them through the Alewife network to the destination INI node. The destination INI node 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.

4.4.1 Performance Related Optimizations

To achieve high performance, we implement numerous optimizations in our prototype system. Some of them are listed below:

- **Avoid extra copying** whenever possible. In our system, messages are always passed through protocol stack layers by their buffer pointers instead of copying. A message stays in the same buffer throughout its lifetime in an INI node if it does not need to be fragmented. In the situation that fragmentation is unavoidable, a single copy is performed each time a message is sent or re-sent. Also, copying is done when the received IP fragments are reassembled back into a single message.

- **Prefetch and double word load** in UDP checksum computation. Checksum computation and copying are two major expensive operations in our protocol stack. We optimized our checksum computation code by hand-coding it in assembly, and further improved its performance by using Alewife prefetching and Sparc double word loading instructions.

- **Window size adjustment.** Currently we choose our window size as 16, which means we allow at most 16 outstanding messages per connection. Since the shared memory protocol used by MGS rarely sends another request to the same node before the previous one has been replied, and the maximum cluster size in our configuration is only 8, our windows almost never fill up during the application run time. This ensures performance will not suffer due to blocking when windows fill up. In addition, the Alewife network is actually a reliable network, which means messages never need to be retransmitted since they never get lost or corrupted. These effects combined with our efficient software timer design make the overhead introduced by the sliding window protocol trivial compared to UDP and IP overhead.

4.4.2 Statistics collection

To carefully study the performance of our inter-cluster communication system, detailed statistics collection and processing facilities are implemented. Alewife provides a globally available synchronized cycle counter, which greatly simplifies our statistics collection. For each message passing through an INI node, time-stamp for the message is placed in a trace each time the message passes through a major checkpoint in the protocol stack. Traces are kept in physical memory to reduce the invasiveness of monitoring code on overall application behavior. After the application terminates, the trace records are processed, and statistic summarization reports are generated. There are two statistics collection modes in the system. The detailed trace mode trace collects the number of cycles spent in every major module and in the FIFO queue. The non-detailed trace mode, in contrast, only records message type, length, time spent in the INI node, and the instantaneous queue length in the INI node when the message arrives.

Chapter 5

Performance results

This chapter presents the performance results of our scalable parallel inter-cluster communication system. Section 5.1 provides performance measurements on an unloaded INI node. Section 5.3 shows the communication system performance when it is driven by applications. Section 5.4 gives a measurement of the load inbalance of intra-cluster INI nodes for the MGS workload. Section 5.2 presents application performance with different inter-cluster communication system configurations. Finally, in section 5.5, we discuss how the inner-cluster communication system should scale when the cluster size scales.

5.1 Inter-cluster message passing performance on unloaded system

Table 5.1 and 5.2 list the message latency measurement of our inter-cluster communication system. Results are measured in machine cycles on a 20Mhz Alewife machine, and cycle breakdowns for the major processing steps are presented. Results for both short and long messages are reported.

Table 5.1 contains result for the sending path. There are seven columns in the table. The first column contains the length of the measured messages. Refer to Figure 3-1, the next five columns contain the cycle counts that each message spent in the five modules of the protocol stack send path. The last column contains the total number of cycles each

len (bytes)	handler	queue	slid_win	UDP[1]	IP	total
0	357	409	140	380	836	2318
100	391	391	144	375	1000	2557
200	425	460	140	380	1134	2875
1400	900	328	153	380	2702	5865
1600	994	404	153	380	10796	13002
65000	32460	858	323	540	385930	421733

Table 5.1: Time breakdown among sending modules, measured in machine cycles.

out going message spend in the INI node. Our results show that the amount of cycles a message spent in the FIFO queue, the sliding window protocol module, and the UDP send module are pretty stable across messages of different length. The queuing time is very low since there is no contention in the system. The actual UDP overhead does increase with message length since it needs to compute a checksum for the whole message. The results in Table 5.1 counts UDP checksum time in IP overhead, due to the fact that the checksum computation is delayed until in IP send module. Cycles spent in the handler increases linearly with message length. IP send overhead linearly increases along with the message size until 1400 bytes. There is a large leap between the IP overhead of messages of length 1400 and 1600 bytes. This is due to fragmentation. We set the Maximum Transfer Unit (MTU) in our implementation as 1500 bytes. The longest single message deliverable in our system is close to 64k bytes, which is limited by UDP/IP protocol.

Table 5.2 contains receiving path results for the system. Again the time spent in FIFO queue and sliding window does not vary with message length. The overhead in message handler and UDP increases with message length. IP overhead has a large jump between message length of 1400 and 1600 bytes due to IP fragments re-assembly.

5.2 Application Performance

[1]UDP checksum computation is delayed until the message is passed to IP sending module, since it covers a UDP pseudo-header which contains the source and destination IP address of the message.

len (bytes)	handler	queue	IP	UDP	slid_win	total
0	286	340	816	330	145	2513
100	331	340	816	496	145	2794
200	347	339	816	623	145	3005
1400	845	337	816	2199	145	6125
1600	–	–	10475	1893	154	16153
65000	–	–	543513	77666	421	688080

Table 5.2: Time breakdown among receiving modules, measured in machine cycles.

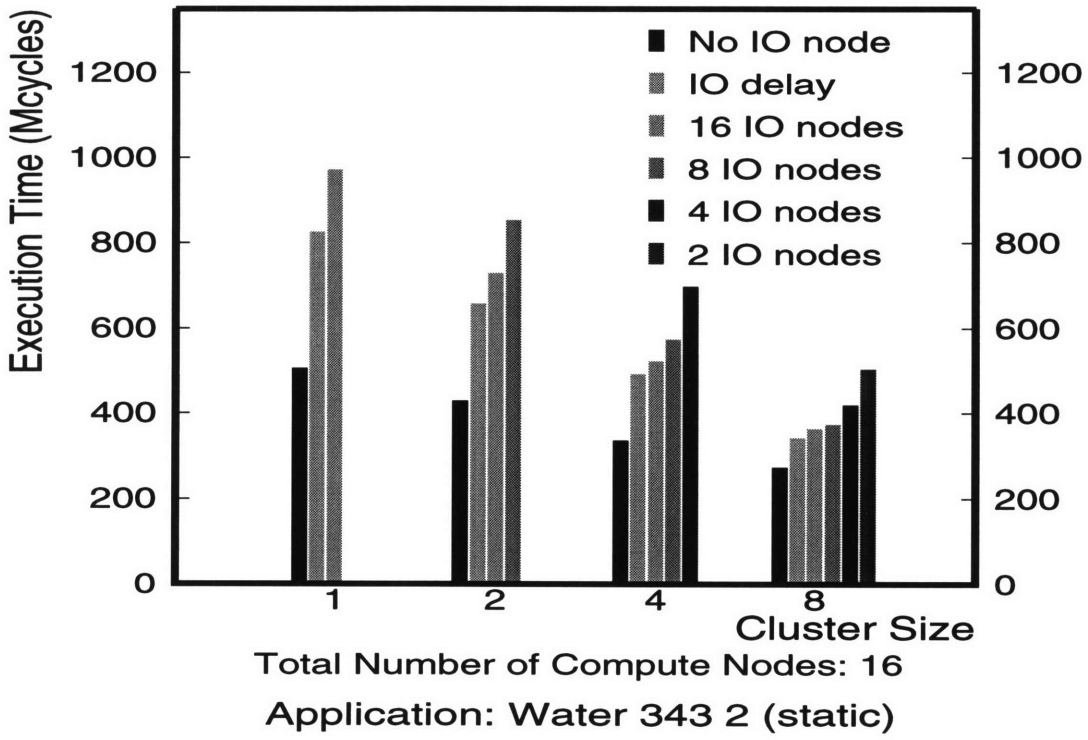

Figure 5-1: Application performance with *static* INI load balance.

Water is a molecular dynamics code taken from the SPLASH [8] benchmark suite. We measured the performance of a version of Water that was ported to the Alewife MGS on our platform. Figure 5-1 shows the application performance on our platform with *static* INI scheduling strategy, results of various inter-cluster communication system configurations are plotted. The problem size of Water measured in this thesis is 343 molecules with 2 iterations.

The results show that our design scales well for all measured configurations. The "No IO node" result in Figure 5-1 represent the run time of the application executing on a version of MGS which directly uses the Alewife network. No inter-cluster communication protocol stack processing is done in this situation, and our inter-cluster communication system is completely bypassed. The "IO delay" results in the figures are the application run time with the inter-cluster messages still directly sent through the Alewife network, but delayed by the latency of a message with the average application message length (5800 cycles), being sent through an unloaded INI node. These results represents the application performance which includes the protocol stack processing overhead, but excludes contention for INI nodes. The results labeled "2 IO nodes" to "16 IO nodes" report application performance with inter-cluster messages using our inter-cluster communication system, with the total number of INI nodes in the machine configured accordingly. From the results we can see that the impact of inter-cluster message contention on the end application performance can be significant, and contention builds up quickly when the number of INI nodes reduces in each cluster. The results also indicate that inter-cluster communication latency can slow Water down by up to a factor of 2.

5.3 Inter-cluster communication performance with shared memory application load

Table 5.3 and 5.4 present INI performance for Water. The application runs on top of MGS. All inter-cluster messages required by MGS memory consistency protocol are sent through our inter-cluster communication system.

Table 5.3 shows results with *static* INI scheduling, while Table 5.4 presents results with

clu size	# of ios per clu	msg len	msg time	queue length	io node utilization
1	1	209	8997	0.537	0.351
2	1	198	10535	0.839	0.464
2	2	198	8026	0.377	0.273
4	1	191	13094	1.310	0.579
4	2	191	8703	0.523	0.348
4	4	190	7198	0.237	0.191
8	1	204	17153	2.157	0.737
8	2	205	10166	0.779	0.449
8	4	205	7675	0.325	0.248
8	8	206	6735	0.151	0.132

Table 5.3: INI node performance running water on MGS, with *static* INI scheduling.

round-robin INI scheduling. Both tables list the average message length, message processing time, which includes cycles spent in both send and receive INI nodes, average queue length in the inter-cluster communication system, and average INI node utilization. With a fixed cluster size, increasing the number of INI nodes assigned to a cluster significantly reduces average message processing time and queue length, which means lower average messaging INI latency and contention. Our prototype scales well all the way to 8 INI nodes per cluster, which is the limit of our machine configuration. The results also show that *round-robin* INI scheduling performs a little better than *static* scheduling.

5.4 Intra-cluster INI node load inbalance measurement

Figure 5-2 shows Water's performance on our platform with *round-robin* INI scheduling. The results are very close to those reported in Figure 5-1. Table 5.5 compares Water run time with *static* and *round-robin* INI scheduling strategies. The results show that the two strategies perform almost equally well, which may be due to the fact that the shared memory application we chose has a vary balanced inter-cluster communication requirement for all compute nodes.

The idea of measuring intra-cluster INI node load inbalance is shown in Figure 5-3.

clu size	# of ios per clu	msg len	msg time	queue length	io node utilization
1	1	209	8775	0.529	0.355
2	1	198	10306	0.833	0.471
2	2	197	7574	0.358	0.275
4	1	191	12492	1.261	0.585
4	2	190	8252	0.504	0.354
4	4	190	6940	0.229	0.191
8	1	204	16793	2.123	0.741
8	2	203	9810	0.771	0.460
8	4	205	7411	0.313	0.247
8	8	206	6579	0.144	0.128

Table 5.4: INI node performance running water on MGS, with *round-robin* INI scheduling.

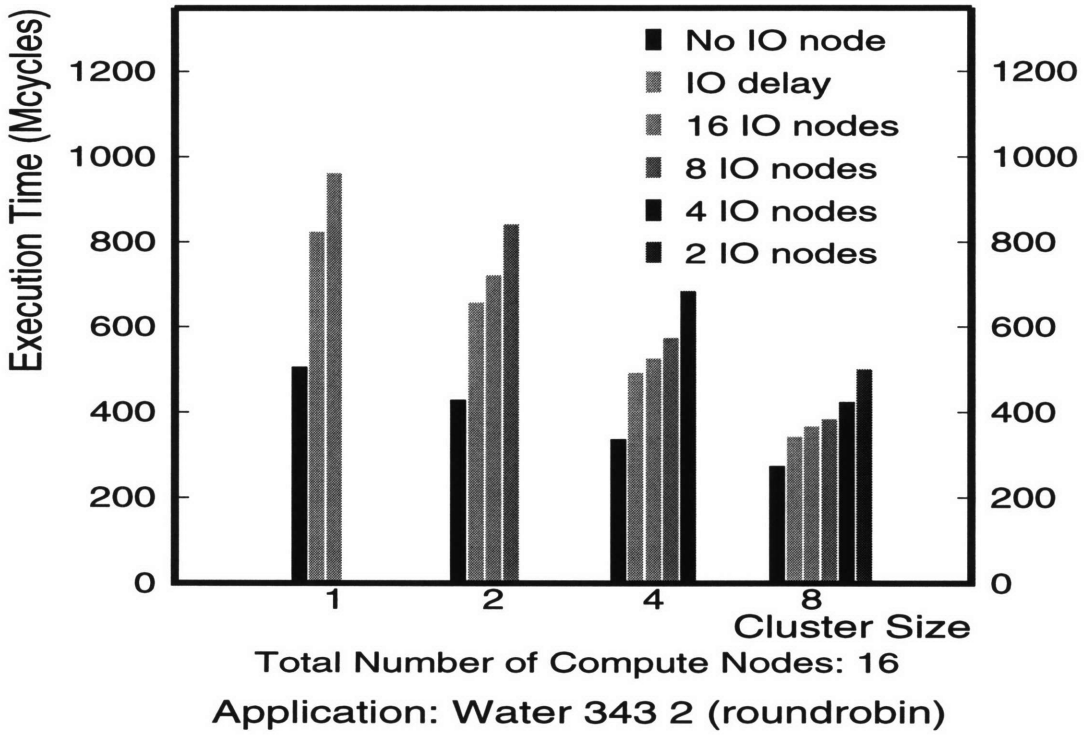

Figure 5-2: Application performance with *round-robin* INI load balance.

clu size	1	2	2	4	4	4	8	8	8	8
clu ios	1	2	1	4	2	1	8	4	2	1
static sched (Meg cycles)	972	729	852	523	574	697	364	374	419	504
round sched (Meg cycles)	962	721	842	526	574	685	367	385	424	500

Table 5.5: Compare application (Water on MGS) performance between *static* INI scheduling and *round-robin* INI scheduling

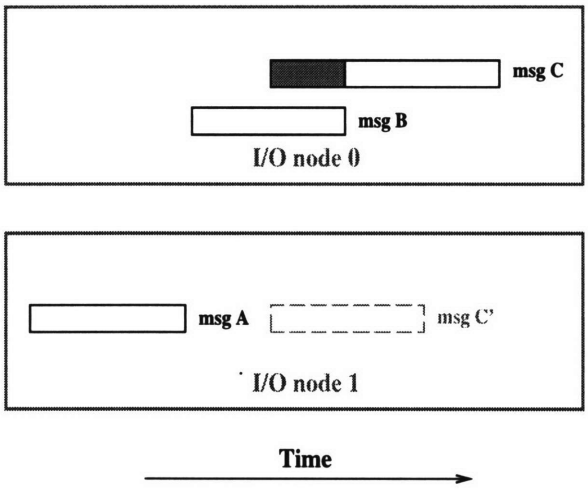

Figure 5-3: Intra-cluster INI node load balance measurement.

Suppose message C arrives at INI node 0 of a cluster while message B is being precessed at the node. C then has to wait a certain amount of time in queue (marked as a gray area in Figure 5-3) before node 0 can start to process it. If by the time C arrives, another INI node of the same cluster is idle, we could potentially save the time C spent in the queue by routing it to the idle node instead. In this sense, we consider C as a misrouted message, and the amount of time it spends in queue (shaded area in the figure) as wasted time.

Table 5.6 and 5.7 report the measurement of INI miss-routing for Water with *static* and *round-robin* INI scheduling. The result is produced by post-processing the inter-cluster message trace at the end of each application run. Two rates are computed. The miss-scheduled time rate is computed by taking the ratio of potentially saved message queuing cycles over the total number of cycles the cluster INI nodes are actively processing inter-cluster messages. The miss-scheduled count rate is the percentage of all inter-cluster messages which can be considered as miss-routed.

clu size	# of ios per clu	mis sched time rate	mis sched count rate
2	2	10.3%	22.9%
4	2	13.0%	26.9%
4	4	6.8%	23.3%
8	2	15.4%	28.4%
8	4	9.7%	29.7%
8	8	2.6%	16.3%

Table 5.6: Intra-cluster INI node load inbalance measurement, with *static* INI scheduling.

clu size	# of ios per clu	mis sched time	mis sched count
2	2	8.6%	18.8%
4	2	11.3%	23.1%
4	4	5.9%	20.9%
8	2	14.1%	25.4%
8	4	9.0%	27.5%
8	8	2.4%	15.0%

Table 5.7: Intra-cluster INI node load inbalance measurement, with *round-robin* INI scheduling.

One thing to note here is that the miss rates reported in Table 5.6 and 5.7 should not be treated in a rigorous way. For example, a 10% miss-scheduled time rate does not mean there exists a way to correctly route messages to achieve this potential. The results are computed from a recorded trace. Any change in any message's latency can potentially change the whole trace. The numbers here should only be viewed as an indication of how well the messages are routed to the cluster INI nodes in a balanced way. Our results show *round-robin* scheduling does better than *static*, but the difference is very limited.

5.5 Scalability of Intra-cluster communication system

[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 ios	1	1.41	2	2.83
msg time	8997	9496	8703	9134
que len	0.537	0.648	0.523	0.591
INI util	0.351	0.385	0.348	0.366
slowdown	1.921	1.868	1.707	1.463

Table 5.8: Scaling inter-cluster communication size with cluster size ^{2}, using *static* INI scheduling.

clu size	1	2	4	8
clu ios	1	1.41	2	2.83
msg time	8775	9174	8252	8816
que len	0.529	0.636	0.504	0.581
INI util	0.355	0.390	0.354	0.372
slowdown	1.900	1.846	1.708	1.490

Table 5.9: Scaling inter-cluster communication size with cluster size ^{2}, using *round-robin* INI scheduling.

An important question we intend to answer in this thesis is when building a balanced clustered multiprocessor, how should an inter-cluster communication system scales 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.

Clustered multiprocessors share information among clusters via message passing through an inter-cluster communication system, which is always very costly compared to accessing information within the same cluster. To achieve good performance, applications should always carefully place data to exploit locality, either manually or automatically using compiler technology. To address locality, we use a simple model. Imagine a clustered multiprocessor built by dividing an MPP with a 2-D mesh network into clusters, and replacing communication between nodes in different clusters using our inter-cluster communication system. If we further assume that applications only require a single node to communicate with its neighboring nodes within a certain distance, then it's clear that the inter-cluster communication load will increase proportionally to the square root of cluster size. This implies that the inter-cluster communication system should scale as the square root of its

cluster size.

Results in Table 5.8 and 5.9 support our argument. When we scale the number of INI nodes per cluster as the square root of cluster size, the message processing time, queue length, and INI node utilization remain relatively the same. The application slowdown improves slowly with the scaling, which may be due to the fact that when cluster size becomes large, the application has smaller inter-cluster communication requirements, since inter-cluster communication latency impacts less on application performance.

Note in the above discussion, we keep total machine size and application problem size fixed while scaling cluster size. The result could be quite different if machine size scales up with cluster size. In this situation, the intuition we discussed above will only be applicable to cases in which inter-cluster communication frequency is dominated by cluster size, not machine size, which can be viewed as a result of communication locality.

Chapter 6

An analytical model

The model presented in this chapter applies Mean Value Analysis (MVA) to an architecture of clustered multiprocessors. It is inspired by the LoPC model [6], which models contention in message passing algorithms on a multiprocessor communicating via Active Messages. LoPC uses the L, o and P parameters, which state for network latency, request and reply processing overhead, and number of processors respectively, to predict the cost of contention in its architecture. Corresponding to the two-level network of our architecture, we denote the network latency and message processing overhead at intra-cluster level using l and o, and at inter-cluster level using L and O. Assuming the average rate of a PE making inter-cluster requests is fixed for a given application and machine configuration. We only keep N, the number of PE nodes per INI node, in our model, and leave out the machine and cluster size.

The reason that we present the model in this thesis is to provide some insight into the amount of contention in inter-cluster communication systems. To keep our model simple, we will only model applications with homogeneous inter-cluster communication load, which means on average, all PEs send about the same amount of inter-cluster messages during the application run time. Other inter-cluster communications patterns can be modeled, with more complicated analysis. We consider the homogeneous communication pattern serves well to our goal.

Parameters	Description
N	Number of PEs per INI node
o	Average message processing overhead at a PE
l	Average wire time (latency) in intra-cluster interconnect
O	Average message processing overhead at an INI
L	Average wire time (latency) in inter-cluster interconnect

Table 6.1: Architectural Parameters of the model.

6.1 Architecture assumption

As shown in Figure 2-1, we assume a clustered multiprocessor has two level of interconnection networks. Message passing in both intra- and inter-cluster networks use an Active Message model. Message processing requirements are precessed in FIFO at both the PEs and the INI nodes, and message handler atomicity is assumed.

Though the operating system may support multiple applications running at the same time, the high cost of buffering messages destined to unscheduled application is typically unacceptable for fine grain applications. In our model, we assume that application threads are always gang scheduled on all processors on the whole machine.

We simplify our analysis by making two assumptions. First we assume both intra- and inter-cluster networks are high throughput networks and contention free. We model contention only for processing resources at the PEs and INIs. Second we assume the message buffers never fill up and message buffering is fast on both PEs and INIs, so messages on networks will not block each other.

6.2 Model parameter description

Table 6.1 describes the architectural parameters used by our model.

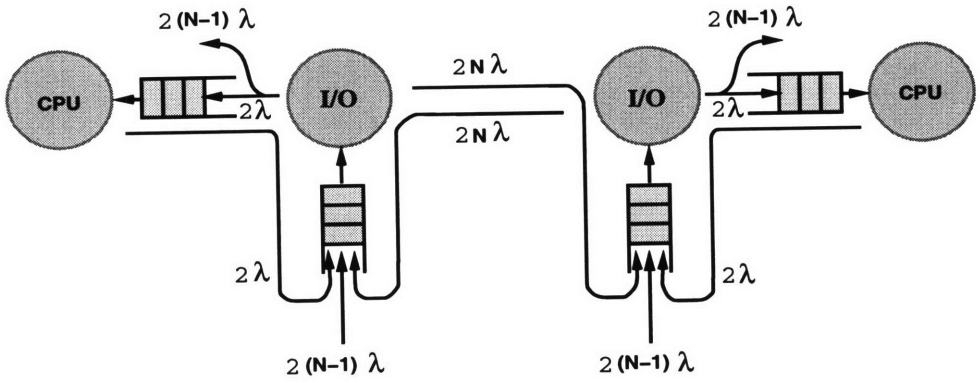

Figure 6-1: Queuing in inter-cluster message passing.

6.3 Model performance for applications with homogeneous inter-cluster communication load

Suppose that a message passing or shared memory application runs on a clustered multiprocessor by running a thread on each PE. Thread T_i runs on PE i. It first spends an average service time of W doing some local work, then sends an inter-cluster request and blocks to wait for the reply. First the request message is sent to one of the cluster's INI nodes, processed and sent to an INI node on its destination cluster. After the inter-cluster communication protocol processing is done at the destination I/O node, the message is routed to its destination PE, say, j. The arriving message will interrupt the thread T_j running on processor j, and invoke its message handler. After an average overhead of o, a reply will be sent from j to i through the inter-cluster communication system. Figure 6-1 and 6-2 show the message path and time-line of a full compute/request cycle respectively. Table 6.2 lists the notations used by the model.

To further simplify our analysis, we assume that the message processing time at all nodes has an exponential distribution, which means we can treat the queuing at all nodes in our system as a Poisson process. Our analysis follows the general techniques of Mean Value Analysis, which is also employed by LoPC.

As shown in Figure 6-2, the average response time is given by:

$$R = R_w + 2l + R_q + 2L + 4R_O + R_y \tag{6.1}$$

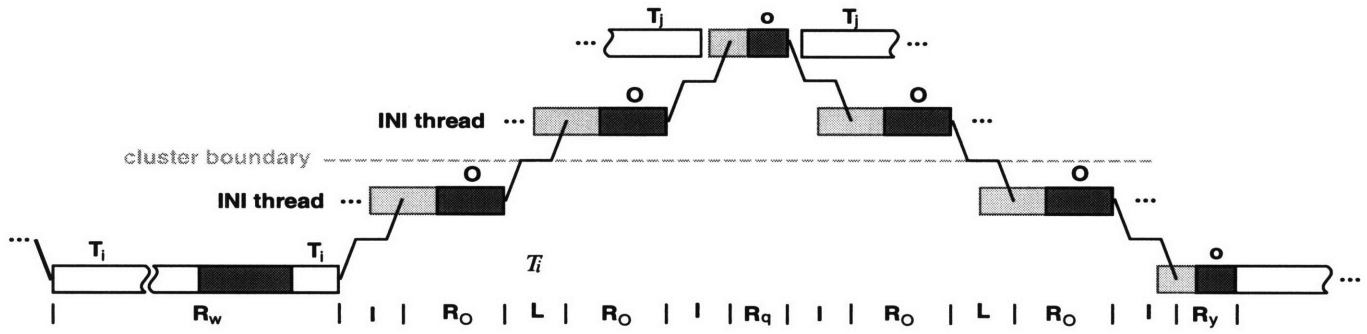

Figure 6-2: Time-line of an inter-cluster compute/request cycle including contention.

Parameters	Description
R	Average round trip time for a compute/request cycle
λ	Average message rate going out from a compute node
W	Average service time between blocking requests
R_w	Average residence time of a computation thread
R_q	Average response time of a request handler
R_y	Average response time of a reply handler
R_O	Average message residence time at an INI
Q_q	Average number of requests queued at a PE
Q_y	Average number of replies queued at a PE
Q_O	Average number of messages queued at an INI
U_q	Average CPU utilization by requests at a PE
U_y	Average CPU utilization by replies at a PE

Table 6.2: Notations used by the model.

From the definition of λ, we have:

$$\lambda = \frac{1}{R} \tag{6.2}$$

By Little's result, we compute the message handler queue length:

$$Q_q = \lambda R_q \tag{6.3}$$

$$Q_y = \lambda R_y \tag{6.4}$$

$$Q_O = 4N\lambda R_O \tag{6.5}$$

At the receiving PE, the message handler can be delayed by handlers of both other requests and replies. The average request response time is given by:

$$R_q = o(1 + Q_q + Q_y) \tag{6.6}$$

Since the requesting thread blocks until a reply is processed, and only one thread is assigned to each PE, reply handler contention needs only to account for contention caused by requests.

$$R_y = o(1 + Q_q) \tag{6.7}$$

The INI response time is given by:

$$R_O = O(1 + Q_O) \tag{6.8}$$

Finally, as in LoPC, we approximate R_w using the BKT approximation:

$$R_w = \frac{W + oQ_q}{1 - U_q} \tag{6.9}$$

Where U_q is the CPU utilization of the request handler. On average, only one request handler needs to run during a compute/request cycle, which means U_q can be expressed as:

$$U_q = \frac{o}{R} \tag{6.10}$$

There are ten unknowns in equations 6.1 through 6.10, they can be found out by solving the ten equations. Contention for processing resources in both INI nodes and PEs are characterized by results of Q_q, Q_y, and Q_O. In our model, application performance is measured by R, the average compute/request cycle time.

6.4 Result discussion

The set of equations in section 6.3 is quite complicated. To find result for various machine and application configurations, automatic tools like Mathematica can be used.

One interesting point is that we can easily find an upper bound of contention in INI nodes by setting W, o, l, and L as 0. This leads to a simple result:

$$Q_{Omax} = N \tag{6.11}$$

This tells us that applications with homogeneous inter-cluster communication pattern, and single block send thread on each PE, the maximum queue length will be no larger than than the number of PEs sharing an INI.

In Figure 6-3 we compare the model results with measurements which are taken from a synthetic application running on Alewife. Only average queue length in INI nodes are plotted. The model result matches the measurements within a few percent. One can notice that in the case when inter-cluster communication overhead dominates application run time, the average queue length in INI nodes increases linearly with N, the number of PEs per INI node.

One more point needs to be mentioned here is that the model does not account contention at request and reply handlers between intra- and inter cluster messages. Again, this is due to our intention to present simple model which only provides us some insight at inter-cluster communication contention.

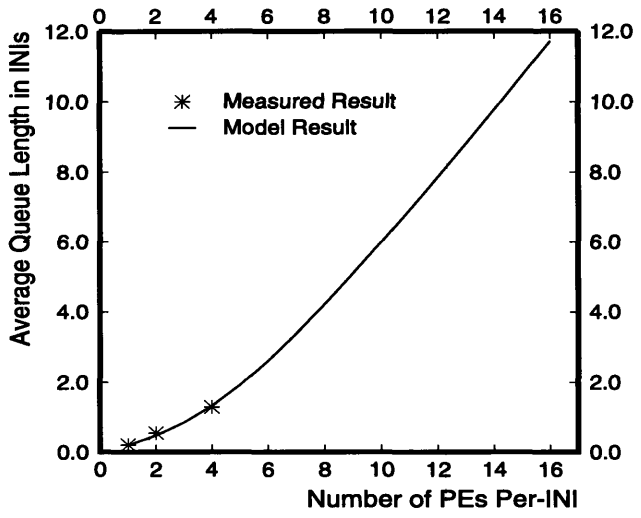

Figure 6-3: Comparison between model and measured queue length at INIs. Measurements are taken from a synthetic application running on Alewife. ($W = 41789$, $o = 742$, $O = 2336$, $l = L = 0$.)

Chapter 7

Conclusion and Future Work

In this thesis we tackle the problem of designing a scalable parallel inter-cluster I/O system for clustered multiprocessors. Our design uses INI nodes as building blocks. By distributing protocol stacks to each INI node, we parallelize the total protocol processing overhead in a cluster. In our design, each INI node has connections to every INI node on all clusters other than the one it belongs to. This allows us the flexibility to be able to route an inter-cluster message through any INI node pair of the message's source and destination cluster, which could potentially improve I/O system performance by applying scheduling algorithms to balance the load on different INI nodes. We also presented an analytical model which models contention in the system.

We integrate our system with MGS, and measure shared memory application performance on the platform. From the measured results and discussions in the previous chapters, we draw the following conclusions:

- To support protocol processing such that good application performance is achieved, the performance of INI nodes should be comparable to PEs.
- Our design is straight forward. By exploiting parallelism in protocol stack processing, it provides reliable connection-oriented inter-cluster communication at high performance. The prototype system scales well in our tests.
- For Water, the shared memory application we tested, intra-cluster INI load balancing problems are insignificant. Complicated INI load balancing algorithms will not likely

increase performance enough to justify the added complexity required to implement the algorithm.

We expect to measure the performance of more applications on our platform in the near future, which will be essential to further evaluate the above conclusions.

Bibliography

[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] 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.

[5] 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.

[6] Matthew I. Frank, Anant Agarwal, and Mary K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. *Submitted to Symposium on Principles and Practices of Parallel Programming*, 1997.

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

[8] 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.

[9] 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.

[10] 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.

6111-43