IMPROVING CLUSTER PERFORMANCE THROUGH THE USE OF PROGRAMMABLE NETWORK INTERFACES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Darius Buntinas, B.S., M.S.

* * * * *

The Ohio State University

2003

Dissertation Committee:

Professor Dhabaleswar K. Panda, Adviser

Professor Ponnuswamy Sadayappan

Professor Mario Lauria

Professor Srinivasan Parthasarathy

Approved by

Adviser
Department of Computer
and Information Science

ABSTRACT

Cluster computing systems are becoming increasingly popular computing environments for day-to-day computational needs because they are cost-effective and affordable. While clusters are considerably less expensive than massively parallel processors (MPPs), MPP communication performance is typically much better than cluster communication performance. Some modern network interface controllers (NICs) have programmable processors which can be used to offload communications processing from the host processor.

Process skew is inherent in cluster communication systems. Some processes may be delayed, relative to other processes, due to various unavoidable causes. Many collective communication operations are implemented in a manner in which all participating processes need to perform the operation in order for the operation to proceed. This means that if one process is delayed, it may cause other processes to be delayed when performing a collective operation.

This dissertation investigates the use of programmable NICs to improve cluster performance. We approach this problem by focusing on improving the performance, scalability, and tolerance to process skew of synchronization operations and collective communication operations through the use of NIC-based operations and NIC-based primitives.

NIC-based operations and primitives improve the performance of cluster systems. Latency is improved in some operations by performing the operation directly at the NIC and avoiding sending messages over the slow I/O bus. Host processor utilization is improved because host processor involvement in the operation is reduced. This also allows computation to be overlapped with the operation. NIC-based operations are also less sensitive to process skew.

To demonstrate the effect of NIC based operations, we have designed and implemented NIC-supported broadcast/multicast, barrier synchronization, reduction and atomic remote memory operations, as well as a application-bypass broadcast. The NIC-supported implementations improved the performance of the operations over the conventional host-based implementations. For instance, our NIC-based barrier operation showed improved latency by a factor of improvement of up to 2.22. The NIC-based reduction operation showed improved host processor utilization by a factor of improvement of up to 2.7. Our NIC-supported application-bypass broadcast showed a factor of improvement of up to 16 in terms of host utilization in the presence of process skew.

VITA

March 22, 1970	Born – Chicago, Illinois, USA
January 1994	B.S. Computer Science, Loyola University Chicago
January 1994	M.S. Computer Science, Loyola University Chicago
January 1995 – June 2003	Graduate Teaching Associate, The Ohio State University
January 1999 – June 2003	Graduate Research Associate, The Ohio State University
June 1999 – September 1999	Graduate Research Assistant, Los Alamos National Laboratory
June 2001 – September 2001	Givens Associate, Argonne National Laboratory

PUBLICATIONS

D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan. "Broadcast/Multicast over Myrinet using NIC-Assisted Multidestination Messages." *Proceedings of Int'l Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC).* January 2000.

D. Buntinas, D. K. Panda and P. Sadayappan. "Fast NIC-Based Barrier over Myrinet/ GM." *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS).* April 2001.

D. Buntinas, D. K. Panda, and P. Sadayappan. "Performance Benefits of NIC-Based Barrier on Myrinet/GM." *Proceedings of International Workshop on Communication Architecture for Clusters (CAC).* April 2001.

D. Buntinas, D. K. Panda, and W. Gropp. "NIC-Based Atomic Remote Memory Operations in Myrinet/GM." *Proceedings of Workshop on Novel Uses of System Area Networks (SAN)*. February 2002.

D. Buntinas and D. K. Panda. "NIC-Based Reduction in Myrinet Clusters: Is It Beneficial?" *Proceedings of Workshop on Novel Uses of System Area Networks (SAN)*. February 2003.

D. Buntinas, A. Saify, D. K. Panda and J. Nieplocha. "Optimizing Synchronization Operations for Remote Memory Communication Systems." *Proceedings of International Workshop on Communication Architecture for Clusters (CAC)*. April 2003.

D. Buntinas, D. K. Panda and Ron Brightwell. "Application-Bypass Broadcast in MPICH over GM." *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID)*. May 2003.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

Computer Architecture	Prof. Dhabaleswar K. Panda
Theory and Algorithms	Prof. Rephael Wenger
Software Methodology	Prof. Neelam Soundarajan

TABLE OF CONTENTS

Page

Abstract ii

Vita iv

List of Tables ix

List of Figures x

Chapters:

1. Introduction 1

    1.1 Software Layers in a Cluster Computing Environment 3
        1.1.1 Communication Operations in Programming Models 4
    1.2 Trends in Network Technologies 5
        1.2.1 Benefits of NIC-Supported Operations 7
        1.2.2 Limitations in NIC Capabilities 9
    1.3 Problem Description 10
    1.4 Our Approach 11
    1.5 Dissertation Overview 14

2. NIC-Assisted Broadcast/Multicast 16

    2.1 NIC-Assisted Multidestination Message Passing 17
    2.2 Broadcast/Multicast with the Multi-send Primitive 20
    2.3 Constructing an Optimal Multicast Tree 22
    2.4 Our Implementation of the Multi-send Primitive 23
    2.5 Experimental Results 25
    2.6 Related Work 28
    2.7 Summary 30

3. NIC-Based Barrier Synchronization 31

3.1 NIC-Based Barrier and Performance Benefits 33
3.1.1 Estimated Performance Improvement 34
3.2 Design Issues 36
3.2.1 Handling Unexpected Barrier Messages 36
3.2.2 Initialization and Cleanup 37
3.2.3 Reliability and In-Order Delivery 38
3.2.4 Multiple Concurrent Barriers 39
3.3 Barrier Algorithm 39
3.3.1 Algorithm Descriptions 39
3.3.2 Algorithm Implementation 40
3.4 Implementation 42
3.4.1 Overview of GM 42
3.4.2 NIC-Based Barrier in GM 44
3.4.3 MPICH Modifications 46
3.5 Performance Evaluation 46
3.5.1 GM-Level Performance 48
3.5.2 MPI-Level Overhead 50
3.5.3 MPI-Level Performance and Scalability 51
3.5.4 Granularity of Computation 51
3.5.5 Varying Arrival Times 58
3.5.6 Synthetic Application Performance 59
3.6 Summary 59

4. NIC-Based Reduction 61

4.1 NIC-Based Reduction 64
4.2 Design and Implementation 65
4.2.1 Unexpected Messages 66
4.2.2 Multiple Instances of the Reduction Operation 66
4.2.3 Generating and Specifying the Tree Structure 67
4.2.4 Performing Floating Point Operations at the NIC 67
4.3 Experimental Results 67
4.3.1 Basic Reduction 68
4.3.2 Larger System Sizes 68
4.3.3 Host CPU Utilization 71
4.3.4 Tolerating Process Skew 73
4.4 Summary 75

5. NIC-Based Atomic Remote Memory Operations 78

5.1 NIC-Based Atomic Remote Memory Operations 79
5.2 Implementation 83
5.2.1 Overview of Myrinet and GM 83
5.2.2 Design Challenges and Our Implementation 85
5.2.3 Serializing Access to Host Memory 87
5.3 Implementing Distributed Locks with Atomic Remote Operations . 89
5.4 Experimental Results 91
5.4.1 Atomic Remote Memory Operations 91
5.4.2 Distributed Locks 93
5.4.3 Host and NIC Processor Utilization 96
5.5 Summary 98

6. NIC-Support for Application-Bypass Broadcast 99

6.1 Application-Bypass 100
6.2 Design and Implementation 102
6.2.1 Design Alternatives 102
6.2.2 Overview of GM and MPICH over GM 103
6.2.3 Our Implementation 105
6.3 Experimental Results 106
6.4 Summary 110

7. Conclusions and Future Research Directions 114

7.1 Summary of Research Contributions 114
7.1.1 NIC-Assisted Broadcast/Multicast 114
7.1.2 NIC-Based Barrier Synchronization 115
7.1.3 NIC-Based Reduction 115
7.1.4 NIC-Based Atomic Remote Memory Operations 116
7.1.5 NIC-Support for Application-Bypass Broadcast 116
7.2 Future Research Directions 117

Bibliography 119

LIST OF TABLES

Table **Page**

5.1 Semantics of atomic memory operations 83

LIST OF FIGURES

Figure		Page
1.1	Block diagram of an MPP.	2
1.2	Block diagram of a cluster of workstations.	2
1.3	Software layers	3
1.4	Communication operations in programming models	4
1.5	Host-based and NIC-supported communication operations	6
1.6	Block diagrams showing host-based and NIC-based broadcast	8
1.7	NIC support for programming models	12
2.1	NIC-supported broadcast for communication subsystems	17
2.2	Multiple host-based point-to-point operations and a NIC-assisted multi-send operation to four destinations.	18
2.3	Timing diagram comparing latencies for sending one packet to four destination using a multi-send operation and host-based point-to-point operations.	19
2.4	NIC-based multicast and a NIC-assisted multicast.	21
2.5	Performance of NIC-assisted multi-send operation versus multiple FM send operations.	25
2.6	Multicast performance for NIC-assisted multicast using an optimal tree *with packet-wise pipelining* (NA-optimal), versus multicast using binomial tree with FM unicast send (FM-binomial).	27

2.7 Multicast performance for NIC-assisted multicast using an optimal tree *without packet-wise pipelining* (NA-optimal), and multicast using binomial tree with FM unicast send (FM-binomial). 28

2.8 Multicast performance for NIC-assisted multicast using an optimal tree *with packet-wise pipelining* (NA-optimal), and multicast using an optimal tree with FM unicast send (FM-optimal). 29

2.9 Multicast performance for NIC-assisted multicast using an optimal tree *without packet-wise pipelining* (NA-optimal), and multicast using an optimal tree with FM unicast send (FM-optimal). 29

3.1 NIC-based barrier for the MPICH middleware 32

3.2 Host-based barrier (left) and NIC-based barrier (right) 33

3.3 Timing diagram comparing latencies for host-based barrier and NIC-based barrier 34

3.4 Block diagram showing the components of GM. 43

3.5 Block diagram of the components of the MCP. 43

3.6 Comparison of NIC-based barrier and host-based barrier for two algorith ms (PE and GB) using the LANai 7.2 and LANai 4.3 NICs . . . 49

3.7 GM barrier latencies and MPI barrier latencies of NIC-based barriers using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs 50

3.8 Performance of NIC-based barrier versus host-based barrier using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs 52

3.9 Performance of NIC-based barrier versus host-based barrier using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs for all number of nodes . . . 53

3.10 Average execution time (compute time and barrier time) per loop for host- and NIC-based barrier on eight nodes using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs 54

3.11 Computation time required to achieve a particular efficiency factor using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs 55

3.12 Total time of computation, varying at each node by 20%, followed by a barrier for NIC-based and host-based barriers over 16 nodes using 33MHz LANai 4.3 NICs. 56

3.13 Difference in execution time between using host- and NIC-based barriers performing computation (± percentage) followed by a barrier (16 nodes; 33MHz LANai 4.3 NICs). 56

3.14 Performance of synthetic benchmarks (total computation time of 360µs and 2,100µs) for host-based and NIC-based barriers using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs 57

4.1 NIC-based reduction for the GM communication subsystem 62

4.2 Block diagrams of Host-based and NIC-based reductions across four nodes. The circles represent the host processor of a node and squares represent the NIC of a node. 63

4.3 Comparison of NIC-based reduction (NB) and host-based reduction (HB) for integer (int) and floating-point (float) operations 69

4.4 Latency of NIC-based reduction (NB) and host-based reduction (HB) for integer (int) and floating-point (float) operations using a 1-degree tree (a chain) of varying depth 70

4.5 Latency of NIC-based reduction (NB) and host-based reduction (HB) for integer (int) and floating-point (float) operations using trees of depth 1 with varying degree 70

4.6 Average time spent by the host processor performing the reduction for host-based (HB) and NIC-based (NB) reductions performing integer (int) and floating-point (float) operations 72

4.7 Average time spent by the host process performing the reduction, with different levels of process skew for host-based (HB) and NIC-based (NB) reductions performing integer (int) and floating-point (float) operations 74

4.8 Average time spent by the host process performing the reduction, with a maximum delay value of 1000µs, for different system sizes for host-based (HB) and NIC-based (NB) reductions performing integer (int) and floating-point (float) operations 76

5.1 NIC-based atomic remote memory operations for the GM communication subsystem 79

5.2 Steps required to perform host-based and NIC-based atomic remote memory operations. 81

5.3 Block diagram showing the components of GM. 84

5.4 Peterson's mutual exclusion algorithm 88

5.5 Example of a distributed lock 91

5.6 Latencies of atomic operations 92

5.7 Locking and unlocking with multiple nodes contending 94

5.8 The number of iterations a process at the home node of a lock can perform in 1,000µs while a process at another node is locking and unlocking the lock at a certain rate 95

5.9 The fraction of time the NIC processor is not idle while a process at another node is locking and unlocking the lock at a certain rate . . . 96

6.1 NIC-supported application-bypass broadcast for MPICH 100

6.2 Broadcast operation over four processes. The large arrows represent timelines for each process. The shaded areas in these timelines represent a call by the application to the broadcast function, and the small arrows represent broadcast messages. 102

6.3 Software and hardware layers for MPICH over GM 103

6.4 Average latency of MPI_Bcast function on 32 nodes. Small messages sizes are 1, 2, 4, and 8 bytes, and large message sizes are 2048, 4096, and 8192 bytes for non-application-bypass MPICH (n) and application-bypass MPICH (ab) 107

6.5 Average latency of signal handler and MPI_Bcast function on 32 nodes. Small messages sizes are 1, 2, 4, and 8 bytes, and large message sizes are 2048, 4096, and 8192 bytes for non-application-bypass MPICH (n) and application-bypass MPICH (ab) 111

6.6 Average latency of signal handler and MPI_Bcast function with a average skew of 333µs for various number of processes. Small messages sizes are 1, 2, 4, and 8 bytes, and large message sizes are 2048, 4096, and 8192 bytes for non-application-bypass MPICH (n) and application-bypass MPICH (ab) 112

CHAPTER 1

INTRODUCTION

Cluster computing systems are becoming increasingly popular computing environments for day-to-day computational needs because they are cost-effective and affordable [55, 6, 58]. Such computing systems consist of high-end commodity workstations connected with a high-performance commodity network. While clusters are considerably less expensive than massively parallel processors (MPPs), the MPP communication performance is typically much better than the cluster communication performance. This is because MPPs are designed for high performance parallel computing, while workstations are designed to be stand-alone machines.

Figure 1.1 shows a block diagram of a typical MPP. The MPP consists of a collection of *nodes* which are connected using a custom high performance interconnect. Each node consists of one or more processing elements (PEs), memory, and a communication assist (CA). The CA provides an interface to the interconnect, and assists in generating outgoing messages and handling incoming messages. This reduces the involvement of the PEs in performing communication operations. Considerable research has been done on efficient communication operations on MPPs [44, 46, 56, 53, 20].

Figure 1.2 shows a block diagram of a cluster of workstations. Such a cluster consists of several high performance workstations connected by a high performance network. A workstation consists of one or more central processing units (CPUs), memory, and various I/O devices including a network interface card (NIC). Notice that workstations do not have a CA, so all communication processing must be performed by the CPU. Also, notice that the NIC is connected to the I/O bus which is slower than the system bus.

These issues lead to poor system performance in clusters of workstations. Communication sent between workstations must pass over the slow I/O bus, resulting in high message latency. Also, the CPU must be involved in performing the communication, which results in poor utilization of the CPU by the application, and makes the overlap of communication and computation difficult.

Furthermore, with the introduction of new high-performance networking technologies available for clusters, such as Quadrics [30], Myrinet [10] and Gigabit Ethernet, these issues become even more important. As network bandwidth and latency

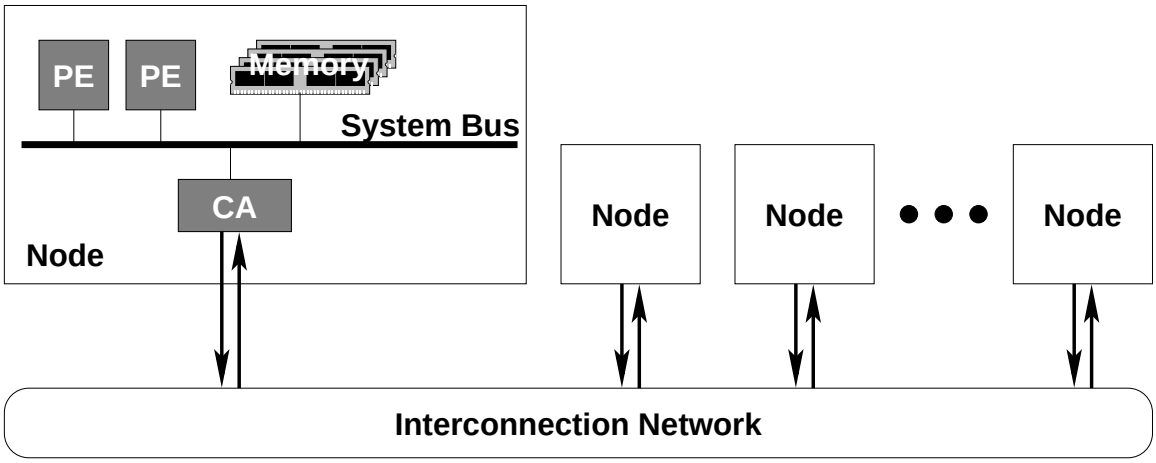


Figure 1.1: Block diagram of an MPP.

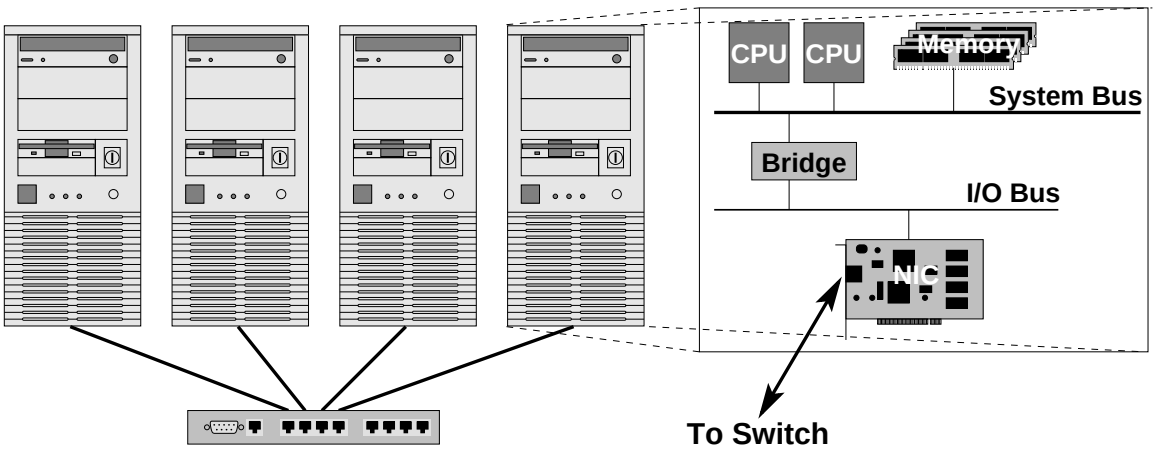


Figure 1.2: Block diagram of a cluster of workstations.

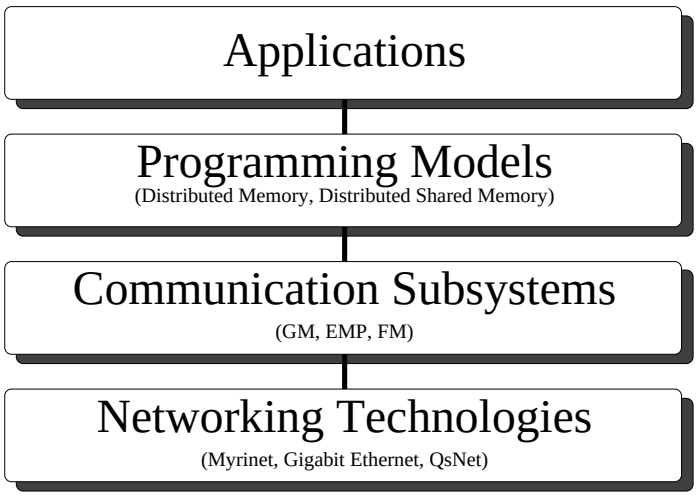


Figure 1.3: Software layers

improve, the issues of the NIC being on the slow I/O bus and the need for CPU involvement in communication become more of a bottleneck for high performance computing.

1.1 Software Layers in a Cluster Computing Environment

Figure 1.3 shows the various software layers in a cluster computing environment. At the bottom layer are the high-performance networking technologies available for clusters. Examples of these are Myrinet, Gigabit Ethernet, and QsNet. Above this layer is the communication subsystems layer. This layer represents the software that provides the communication primitives for a particular networking hardware. Examples of these are GM [34] and FM [41], which are communication subsystem for Myrinet, and EMP [49, 50, 3], which is a communication subsystem for Gigabit Ethernet.

On top of the communication subsystem layer is the programming model layer. A programming model provides a paradigm to the application programmer which is independent of the communication subsystem or hardware that the program is running on. The two major programming models for high-performance parallel computing are Distributed Memory (DM), also called Message Passing, and Distributed Shared Memory (DSM). Typically a middleware library is written to support a particular programming model, such as MPICH [22, 33], which implements the DM programming model, or Global Arrays [37][38, 36], which implements a logical-DSM programming

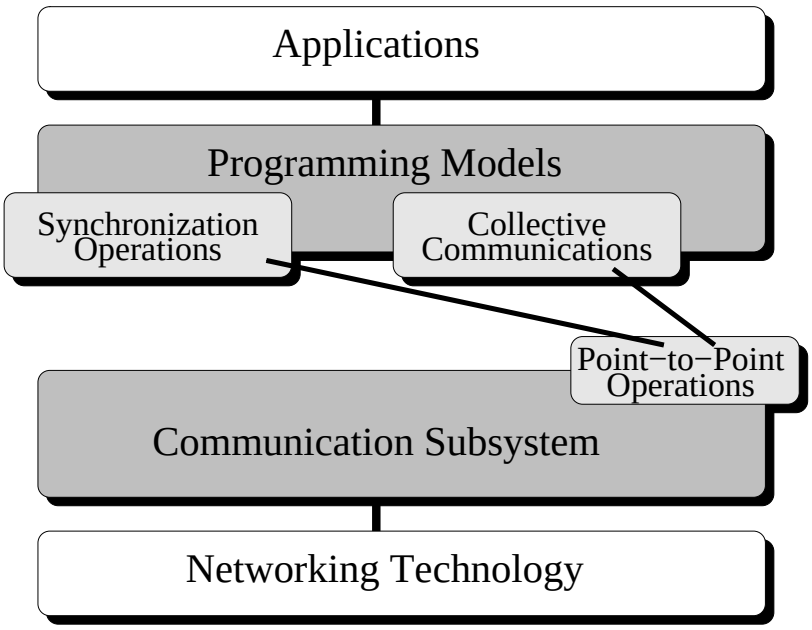


Figure 1.4: Communication operations in programming models

model. At the top layer we find the actual parallel applications which are written using a middleware library.

1.1.1 Communication Operations in Programming Models

Programming models often use synchronization operations and collective communications. These are typically implemented using point-to-point messages. Figure 1.4 shows these communication operations in relation to the software layers and to the point-to-point communication operations, which are implemented at the communication subsystem layer.

ARMCI [38] is a middleware which provides synchronization operations, such as atomic remote memory operations and locking. These operations are traditionally implemented using point-to-point messages and a server thread at each node. When a process performs a synchronization operation it sends a request using a point-to-point message to the server thread at the remote node. The server receives the request performs the operation, and sends a reply back to the intiating process. To avoid wasting computational resources, the server thread would block waiting for an incoming request. The incoming request would trigger an interrupt to wake the server thread.

MPICH is a middleware which uses collective communication operations such as broadcast and reduction. These are also typically implemented using point-to-point messages. In a broadcast operation, messages are broadcast from the root to the other processes over a broadcast tree using point-to-point messages. In order for the broadcast operation to proceed, all of the participating processes need to call the broadcast function. When a process calls the broadcast function, the process waits to receive the message from its parent process, then re-sends the message to its child processes.

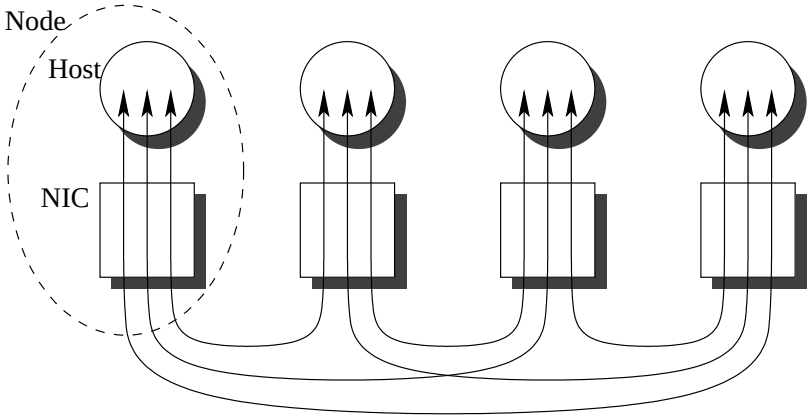
1.2 Trends in Network Technologies

Some modern NICs, such as Myrinet [10], Quadrics [30], and those using the Alteon chip-set (e.g., Netgear 620 [35]) have programmable processors [8, 1]. These programmable NICs can be used to offload communication processing from the host processor. The basic idea of NIC-supported operations is to have the NIC perform, or assist in performing, a communication operation [57, 25, 7, 48, 14, 16, 17, 15, 40, 39, 13]. This raises the question of whether using a programmable NIC in this way can make up for the lack of a CA, and improve cluster performance.

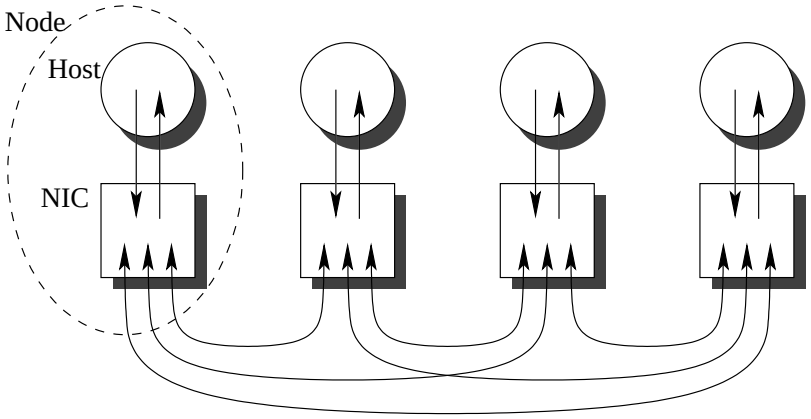
Figure 1.5 illustrates the difference between traditional host-based communication operations and communication operations supported by a programmable NIC. The diagrams show four nodes, consisting of a host and a NIC, performing a generic communication operation where each node sends a message to every other node.

Figure 1.5(a) shows a traditional host-based implementation of this communication operation. In this figure, we see that all communication is initiated by the host, sent through the NIC to the NIC at the receiving node, and finally received at the host of the receiving node. If the operation were to be implemented in a NIC-supported manner using a programmable NIC, the operation may look like that in Figure 1.5(b). Here, notice that the hosts need only initiate the operation, then check for the result later. The operation is performed by the NIC. By offloading the communication processing to the NIC in this way, we reduce the number of times the data is transferred between the host and the NIC, improving the performance of the operation. Also, because the NIC is performing the operation, the host is free to perform other useful computation while the operation is being performed by the NIC, improving the host processor utilization.

In this dissertation, we will refer to a processor at the NIC as a *NIC processor* and a processor at the host as a *host processor*. We also distinguish between two ways that communication operations can be supported by programmable NICs. One way is using a *NIC-based operation* where the operation is completely performed by the NIC. Another way is to use *NIC-based primitives* to assist in performing the operation. In some cases a complete NIC-based implementation would be infeasible, due to the complexity of the operation or because of limited NIC resources. For instance,



(a) Host-based



(b) NIC-supported

Figure 1.5: Host-based and NIC-supported communication operations

a completely NIC-based reduction operation can be implemented for a small number of elements, and for standard arithmetic operations. However, for larger number of elements, or for user-defined arithmetic operations, a completely NIC-based implementation may not be possible. In these cases, a simpler NIC-based primitive can be implemented which could assist the host in performing the operations. The NIC-based primitive would improve the performance of the operation over a completely host-based implementation.

1.2.1 Benefits of NIC-Supported Operations

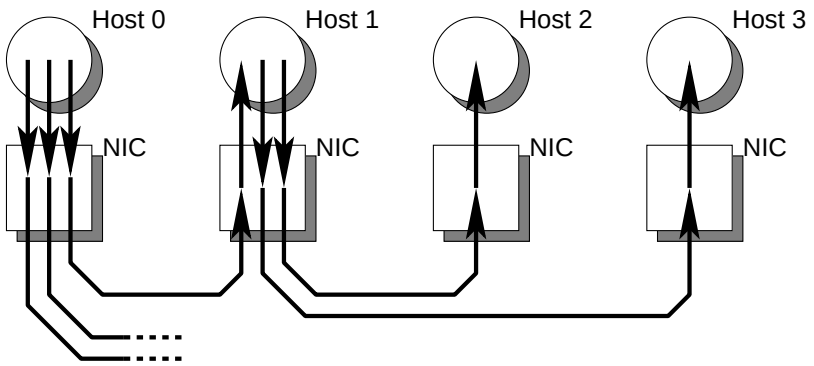
As mentioned, we expect that using NIC-support will improve the latency of the operation, as well as to free-up the host processor for other computation. Let us use a broadcast operation as a specific example to illustrate more benefits of NIC-supported operations.

Figure 1.6 shows block diagrams of a traditional host-based broadcast and a NIC-based broadcast. A broadcast in a point-to-point network is typically performed using a broadcast tree [5]. The root node sends the message to its children which receive the message and then send the message on to their children. Figure 1.6(a) shows four nodes of a traditional host-based broadcast. Node 0 is the root node, and Node 1 is one of its children. The other two children of Node 0 are not shown. Node 1 has two children, Nodes 2 and 3. Nodes 2 and 3 are leaf nodes, and have no children. In this diagram, Host 0 sends the message to be broadcast three times, once to each of its children. The message to Node 1 is received at the NIC, which forwards it to the host. Upon receiving the message the host then sends this same message twice, once to each of its children.

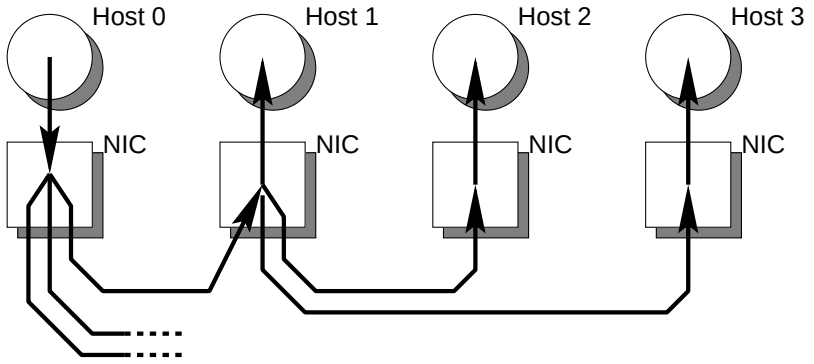
In Figure 1.6(b) we see the NIC-based broadcast. Notice here, that Host 0 sends the message once to the NIC. The NIC, then transmits the message to each child. This avoids having the host initiate multiple messages, and reduces the number of times the data is sent over the PCI bus. At Node 1, we see that when the message is received by the NIC, as well as forwarding the message to the host, the NIC sends the message to Nodes 2 and 3. By having the NIC forward the data to the children, the data does not have to be sent to the host and back down again. This again reduces the traffic on the PCI bus, but also reduces the latency of the operation: as soon as the message is received by the NIC it can be forwarded to the node's children.

This example suggests several benefits of using NIC-supported communication operations over host-based communication operations. We describe these below.

- **Reduced latency of the operation** – This example suggests that a NIC-based broadcast operation would complete faster than the host-based broadcast because the messages at the intermediate nodes need not be passed to the host, only to have the host turn around and send them back through the NIC to its children.



(a) Host-based



(b) NIC-based

Figure 1.6: Block diagrams showing host-based and NIC-based broadcast

- **Reduced host processor involvement in the operation** – We also see from this example that the hosts only to need send one message to the NIC, rather than one per child. Similarly, if the operation were a gather operation, once the process sends its data to the NIC, it can proceed with other useful computation, and need not wait for messages to be received from its child processes or for the operation to complete. This frees up the host processor for other useful computation.

- **Allows for non-blocking or split-phase operations** – Although in this example, the broadcast operation does not lend itself to a non-blocking implementation, other collective operations can benefit from a non-blocking, or split-phase, implementation. For instance, if an all-reduce operation were to be implemented at the NIC, the host process can initiate the operation in one phase, and receive the result in the second phase. In the first phase, the host process provides its data to the NIC and initiates the operation. It can then proceed with other computation that does not depend of the result of the all-reduce operation. When the application needs the result of the operation, it can perform the second phase, and wait for the result from the NIC. This allows the reduction operation to be overlapped with useful compuation at the host.

- **Improved tolerance to process skew** – Another benefit is that even if the host at one of the intermediate nodes is not ready to perform the broadcast, its children can still receive the broadcast message. For instance, in the host-based case, if Host 1 is delayed and did not reach the portion of the program that executes the broadcast, when the broadcast message is received from Node 0, it will not be forwarded to Nodes 2 and 3 until Host 1 executes the broadcast, even if Nodes 2 and 3 are waiting to receive it. In the NIC-based case, because the host is not involved in actually performing the broadcast, even if Host 1 is not ready to receive the message, the NIC will still forward the message to Nodes 2 and 3, allowing them to proceed. This concept is known as *application-bypass* [11]. The concept of application-bypass is to allow communication operations to proceed without the intervention of the application.

1.2.2 Limitations in NIC Capabilities

We have illustrated some of the benefits of moving communication operations from the host to the NIC. However, there are some limitations in implementing operations at the NIC, namely the speed of the NIC processor and the size of the NIC memory. Host processor speed ranges from 700MHz to 2.4GHz, and the host memory capacity ranges from 512MB to 8GB. Whereas the NIC processor speed ranges from 33MHz to 233MHz, and NIC memory capacity ranges from 256KB to 64MB. There is over

a magnitude of difference in processor speed and memory capacity between the NIC and the host.

This difference can affect the performance of the communication operation. Similarly, the limited memory can affect the scalability of the operation, and the number of concurrent instances of the operation that the NIC can support. For this reason, careful design and efficient implementation is critical if the operation is to perform well and scale well.

1.3 Problem Description

The primary objective of this research is to improve cluster performance through the use of programmable NICs. We approach this problem by focusing on improving the performance, scalability, and tolerance to process skew of synchronization operations and collective communication operations through the use of NIC-based operations and NIC-based primitives.

As described in the first section, cluster computing environments typically have poor communication performance when compared to MPP environments due to the lack of a CA and the fact that the NIC is on the slower I/O bus. In order for clusters to take full advantage of modern high performance network technologies, this issue must be addressed. Programmable NICs can potentially be used to compensate for the lack of a communication assist in commodity workstations and improve communication performance. Specifically, programmable NICs can be used to support collective communication operations, such as barrier and reduction, as well as synchronization operations, such as atomic remote memory operations.

Another issue that impacts the performance of collective communication operations in clusters is process skew. During the course of the excution of a parallel program, processes may become skewed in time, i.e., some processes may fall behind other processes. This may be due to various reasons such as imbalanced or asymetric code. Collective communications operations are typically implemented by middleware libraries at the application level. That is, the operation proceeds only once the application at each participating process calls the collective communication function. So, if one process is delayed in calling the function, due to process skew, other processes may be unnecessarily delayed. Dealing with process skew becomes even more important as system sizes increase. This means a scalable collective communication operation must be tolerant to process skew.

Completely NIC-based implementations of collective communication operations are naturally tolerant to process skew because the operation is implemented at the NIC, so the operation can proceed even if the application has not yet called the collective communication function. Similarly NIC-based primitives can be used to improve a collective communication operation's tolerance to process skew by allowing

the operation to bypass the application and proceed, even if the process is delayed and has not yet called the collective communication function.

This dissertation specifically focuses on the following problems:

1. Can programmable NICs be used to support communication operations?
2. Will these communication operations perform well, and scale well?
3. How can NIC-based communication operations be used to support synchronization operations?
4. How can NIC-based primitives be used to assist in performing collective communication operations?
5. Can the use of NIC-based primitives reduce the impact of processor skew on collective communication operations?

We describe our approach to these problems in the next section.

1.4 Our Approach

Figure 1.7 illustrates our approach to to the problems described in the previous section. In the figure, the dotted lines represent the existing method for implementing the communication operations. In this dissertation, we use NIC-based operations to implement synchronization operations and collective communication operations. We also use NIC-based primitives to make collective communication operations which are based on point-to-point operations more tolerant to process skew. These options are represented by the solid lines.

Below we describe in detail our approach to the problems described in the previous section.

Using programmable NICs to support communications operations and improve their performance – We have designed and implemented NIC-supported broadcast/multicast, barrier, reduction, and atomic remote memory operations. We have also designed and implemented a NIC-based primitive which makes a host-based broadcast operation more tolerant to process skew.

Because of the limited resources available at the NIC, careful design and implementation of these operations is crucial in order for the operations to perform well and be scalable. There are many issues that need to be addressed when designing and implementing NIC-based operations and primitives, such as, choosing an appropriate algorithm, guaranteeing the reliable delivery of operation messages, handling multiple concurrent instances of the operation, and dealing with unexpected messages.

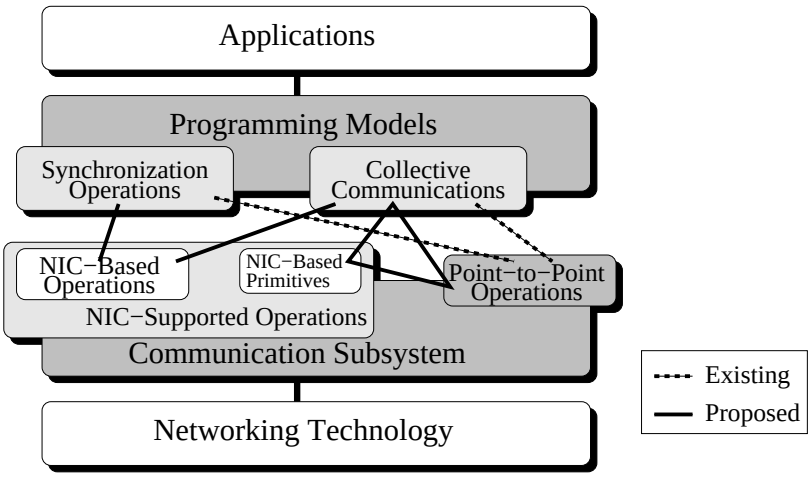


Figure 1.7: NIC support for programming models

Because NIC-based operations and primitives are performed by the NIC, intermediate messages do not have to be passed to the host, reducing the number of times the data passes over the PCI bus. This can improve performance when large amounts of data needs to be transferred, due to the limited bandwidth of the PCI bus compared to the bandwidth of the network. Even when the amount of data is small, the latency of having the message be sent to the host only to have another message be sent back down is eliminated in NIC-based operations, and reduced in NIC-based primitives. This offsets the fact that the operation is being performed by the NIC processor which is slower than the host processor.

Using NIC-based operations to reduce host processor involvement in synchronization operations – In synchronization operations, such as atomic remote memory operations, it is desirable to eliminate the involvement of the remote host process. In a typical host-based implementation, a server thread would be used on each node. This thread would be used to handle asynchronous requests. The thread could be implemented either to poll for incoming requests or to block while waiting for the requests. However, these options either waste processor resources or give poor performance.

We have implemented NIC-based atomic remote memory operations which eliminate the need for the server thread. Because the operation is performed directly by the NIC, the computational thread at the host is not disturbed. This leads to better performance for the operation and better host processor utilization than the host-based implementation.

Because synchronization operations may need to modify data-structures used by the host process, access to these data-structures needs to be serialized to avoid race conditions. For host-based implementations, these issues have long been studied and various efficient solutions are available. A new set of constraints is introduced by moving the operation to the NIC. For instance, while a host process can access NIC memory using loads and stores, the latency of accessing NIC memory over the PCI bus is considerably higher than accessing local memory. Futhermore, the NIC can only access host memory using DMA. We have redesigned the traditional synchronization scheme taking these constraints into account, resulting in an efficient algorithm to serialize access to host data.

Using NIC-based primitives to reduce the effects of process skew – As described previously, during the course of the excution of a parallel program, processes may become skewed. Process skew can affect the performance and scalablity of collective communication operations. Collective communication operations are typically implemented by middleware libraries at the application level. So if one process is delayed in calling the function due to process skew, other processes may be unnecessarily delayed.

For instance, the typical host-based broadcast operation is performed by forwarding messages sent by the root along a broadcast tree. When a non-root process calls the broadcast function, it waits to receive the message from its parent, then sends the message to its children. Notice that if a process is delayed, and does not call the broadcast function, even if the message from its parent is received, the message will not be forwarded on to its children until the process finally calls the broadcast function. So all of the processes which are its decendants in the broadcast tree will have to wait for the process to catch up. Note that process skew also affects the scalability of a collective communication operation. For instance, as the number of processes participating in the broadcast increase, the depth of the broadcast tree increases, so the effect of a delayed process higher up in the tree also increases.

We have implemented an *application-bypass* broadcast operation which uses point-to-point messages along with a NIC-based primitive. In an application-bypass operation, the operation will proceed independently from the application. This can be implemented by using interrupts and a signal handler, or server thread. However generating an interrupt for every message affects performance. In our implementation, the NIC-based primitives is used to selectively interrupt the host only when necessary. This allows the process to poll for incoming messages when it is ready to receive the message, avoiding the cost of the interrupt, but it will interrupt the process when the process is not actively polling for the message, allowing the operation to proceed.

1.5 Dissertation Overview

In this dissertation, we describe how we improve cluster performance through the use of programmable NICs. We present the design, implementation and evaluation of NIC-supported collective communication operations, NIC-supported synchronization opertions, and an application-bypass collective communication operation using a NIC-based primitive. The NIC-supported collective communication operations, broadcast, barrier synchronization and reduction are described in Chapters 2, 3 and 4, respectively. Chapter 5 describes our research on NIC-based atomic remote memory operatons, and Chapter 6 describes our research on application-bypass broadcast using a NIC-based primitive.

Chapter 2 describes our design of a NIC-supported broadcast operation. This operation was added to the Illinois Fast-Messages [42, 29] communication subsystem. We designed the broadcast operation using a NIC-based *multi-send* primitive which requires minimal assistance from the NIC. This primitive improves the performance of sending the same message to multiple destinations, by having the host process send one copy of the message to the NIC and having the NIC send a copy of the message to each destination. To fully utilize the benefits of this primitive, we also propose a method for constructing an optimal multicast tree using the new primitive. When evaluating our NIC-supported broadcast, we find significant improvements in latency over the traditional host-based broadcast operation.

We describe our NIC-based barrier synchronization operation in Chapter 3. This was implemented by modifying the GM communication subsystem. The barrier operation is implemented in a completely NIC-based manner. By reducing the round-trips over the PCI bus, the latency of the operation is reduced significantly. We implemented the operation using two different algorithms, namely, pairwise-exchange and gather-and-broadast, to evaluate the performance benefits. We also modified MPICH-GM, an implementation of MPI over GM, to use the NIC-based operation. We also evaluate our implementations using two different generations of NICs, to see the impact of using faster NIC processors.

Our NIC-based reduction operation is described in Chapter 4. In this operation, the host processes send their data, and topology information to the NICs, and the NICs perform the reduction. The root process then reads the result from the NIC. One major challenge to this design was the lack of floating-point support at the NIC. In order to be able to perform floating-point operations, the operations had to be performed in software. Despite the added overhead of performing floating-point operations in software, the NIC-based operation performs better than the host-based implementation. A major benefit of using NIC-based reduction is the improved tolerance to process skew. We saw a sigificant improvement in host processor utilization in the presence of process skew when using NIC-based reduction versus host-based reduction.

In Chapter 5 we present our NIC-based atomic remote memory operations. By implementing atomic operations at the NIC, the host process is not interrupted to perform the operation. This improves the host processor utilization. Also, because of the efficient implementation, the load on the NIC processor is less with the NIC-based implementation than with the host-based implementation, making the NIC-based operation perform better, and scale better. Because the operation may be used to modify data-structures that are being accessed by the host process, we implemented a mutex operation between the NIC and the host to serialize access to the host memory and avoid race conditions. The mutex had to be designed carefully to reduce polling over the PCI, and to consider the fact that the NIC does not have load/store access to host memory. We implemented distributed locking using the NIC-based atomic remote memory operations, and found a considerable performance improvement over the host-based implementation.

We describe our application-bypass broadcast implementation using NIC-based primitives in Chapter 6. We implemented the broadcast using host-based point-to-point messages, but used a NIC-based primitive to allow the broadcast operation to efficiently proceed even if the process has not called the broadcast function. Our primitive allows interrupts to be generated only for certain messages, and only when the process is not actively polling for the messages. This allows broadcast to be implemented in an application-bypass manner without negatively impacting the performance of the operation with unnecessary interrupts. We see a signifiant improvement in host processor utilization in the presence of process skew when using the application-bypass broadcast over the traditional broadcast. We also see the impact of process skew on the traditional broadcast implementation increase with system size, while the application-bypass broadcast was virtually unaffected. This indicates that application-bypass broadcast would scale much better than the traditional broadcast.

Finally, in Chapter 7 we present our conclusions and describe topics for further research.

CHAPTER 2

NIC-ASSISTED BROADCAST/MULTICAST

Broadcasting and multicasting are common operations in parallel and distributed programs. For example, MPI [33] has a broadcast operation defined as part of the standard. It would be beneficial to be able to reduce the latency of this operation as much as possible. In Section 1.2.2 we described the difference in speeds between the NIC processor and the host processor. This may limit the amount of work that can be done by the NIC without compromising performance. This raises a challenge whether new communication mechanisms can be developed for clusters to support broadcast/multicast with minimal NIC assistance, while delivering good performance.

In this chapter we take on this challenge and introduce a NIC-based multidestination message passing primitive. Such a mechanism has been developed earlier for router-based parallel systems [46, 44, 56] to support efficient collective communication. We describe our design and implementation of a *multi-send* primitive to support efficient broadcast/multicast that requires minimal assistance from the NIC. Our scheme is designed with the idea that as much processing as possible should be done by the host processor. This gives us more flexibility with, for example, creating multicast trees which would be optimal for a particular message size, or choosing a multicast tree dynamically based on requirements of bandwidth versus latency for a particular message. Also, because the proposed scheme does less processing at the NIC, the impact of adding such NIC-assisted multicast operation to a communication subsystem is very small, less than 500ns for non-multi-send packets.

We have implemented the multi-send primitive as a modification to Illinois Fast-Messages (FM) 2.1 [42, 29] running over a Myrinet [10] network. To fully utilize the benefits of this primitive, we propose a method for constructing an optimal multicast tree using the new primitive. We have evaluated this scheme and obtained a factor of improvement of up to 1.85 for multicasting 16K messages with 16 nodes.

Figure 2.1 illustrates our approach to adding NIC-supported broadcast to the FM communication subsystem. The dotted line indicates how the broadcast operation is traditionally implemented using point-to-point messages. The solid line shows our implentation using NIC-based primitives to improve the performance of the operation.

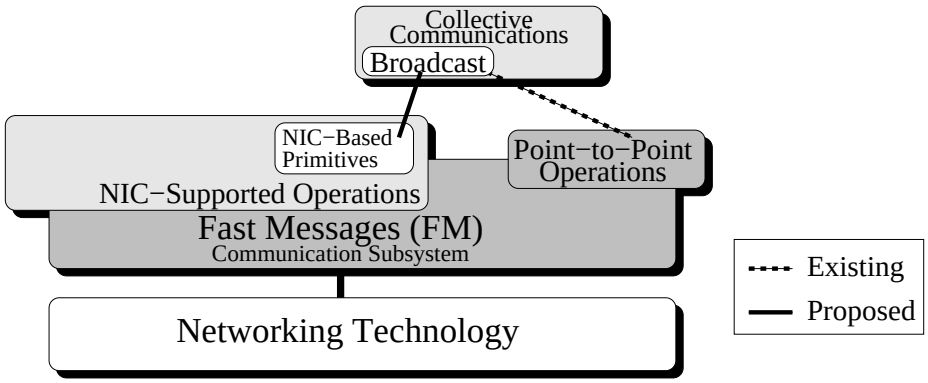


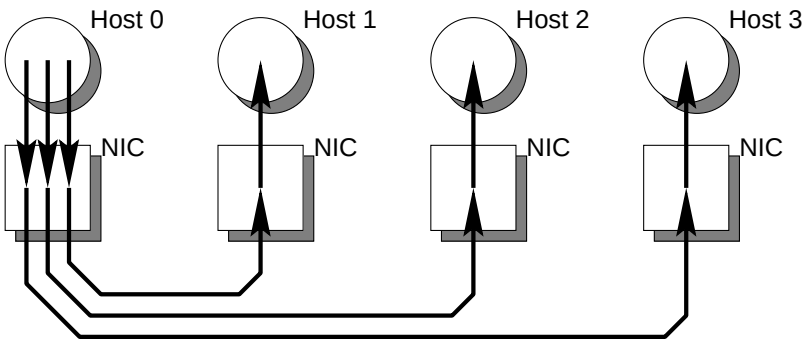
Figure 2.1: NIC-supported broadcast for communication subsystems

The rest of this chapter is organized as follows. Section 2.1 describes the new multi-send primitive, followed by Section 2.2 which describes broadcasting and multicasting using the new primitive. Construction of the optimal multicast tree is described in Section 2.3. The implementation details are discussed in Section 2.4 followed by our experimental results in Section 2.5. Related work is discussed in Section 2.6. We summarize our work in Section 2.7.

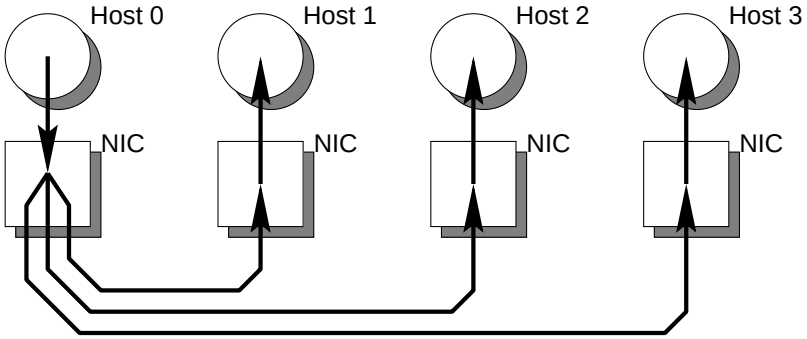
2.1 NIC-Assisted Multidestination Message Passing

The basic idea is to create a *multi-send* primitive in which the host writes the packet data to the NIC only once followed by a list of destinations. The NIC would then transmit copies of the packet to each of those destinations. Figure 2.2 shows two diagrams where Host 0 is sending packets to Hosts 1 through 3. Figure 2.2(a) shows Host 0 making three unicast (point-to-point) sends, each of which is forwarded by the NIC to its destination. Figure 2.2(b) shows the host making a single multi-send operation to the NIC which then forwards a copy of the packet to each of the destinations.

Figure 2.3(a) shows the timing diagram for a multi-send operation sending a packet to four destinations, and the receive time for the last destination. Figure 2.3(b) shows the corresponding timing diagram for host-based point-to-point operations. In the figure the interval marked *Send* corresponds to the time it takes the host to assemble a packet and write it to the send queue on the NIC, and the interval marked $Xmit$ corresponds to the time it takes the NIC to transmit the packet from the send queue to the network. The interval marked *Recv* corresponds to the combined time for the NIC to receive the packet, (including the network latency), for the NIC to forward the packet to the host, and for the host to process the packet. Notice that in both

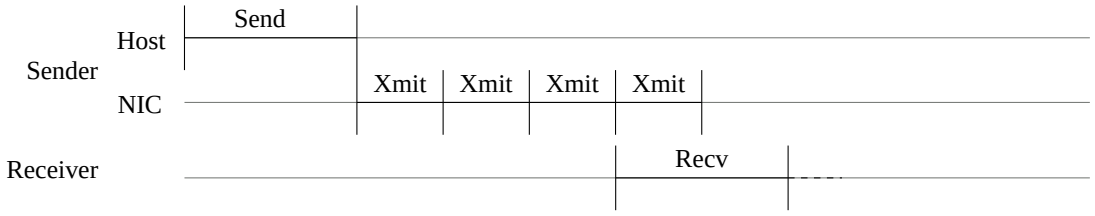


(a) Host-based

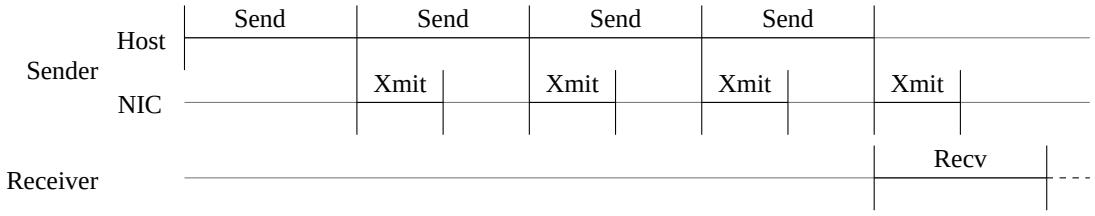


(b) NIC-assisted

Figure 2.2: Multiple host-based point-to-point operations and a NIC-assisted multi-send operation to four destinations.



(a) Multi-send



(b) Host-based point-to-point

Figure 2.3: Timing diagram comparing latencies for sending one packet to four destination using a multi-send operation and host-based point-to-point operations.

diagrams the receive time for a packet at the last receiver is overlapped with the transmission time at the sender for that same packet. In Figure 2.3(b), for the first three packets, the network transmit time of one packet is overlapped with the host send time of the next. As indicated below, timing parameters for FM over Myrinet are such that this will always be the case, regardless of the packet size. Though it is not shown in these figures, packet reception can also be pipelined between the NIC and the host.

Let us compare the latency of sending a packet to k destinations using a multi-send operation with the latency of sending a packet to k destinations using the usual host-based point-to-point operations. The time for the kth destination to receive the packet using a multi-send operation would be $(t_{Send} + (k-1) \times t_{Xmit} + t_{Recv})$, and $(k \times t_{Send} + t_{Recv})$ using host-based point-to-point operations. We have timed the host send, the NIC transmit and the receive operations in FM 2.1 on our cluster consisting of 300MHz Pentium II machines with 33MHz LANai 4.3 Myrinet cards. We measured the send time to be (2.7863μs+0.0301μs$\times m$), the NIC transmit time to be (1.3958μs+0.0075μs$\times m$) and the receive time to be (4.2820μs+0.0230μs$\times m$), where m is the packet size in bytes. Thus for sending a 1,536 byte packet to six destinations, t_{Send} would be 49.0199μs, t_{Xmit} would be 12.9158μs, and t_{Recv} would be

39.61μs. The multi-send operation would take 153.2089μs and the usual host-based point-to-point method would take 333.7294μs. This leads to a factor of improvement of 2.18. We can see that multi-send is a powerful primitive.

The estimates above were made without write-combining support[1]. Without write-combining, we achieve about 31.7 megabytes per second throughput when the host is writing a packet to the NIC. Assuming write-combining is enabled, we could get 90 megabytes per second throughput [1]. Then the send time would be (2.7863μs+0.0106μs$\times m$) which would be 19.0679μs for a 1,536 byte message. The multi-send operation would then take 123.2569μs while the usual host-based point-to-point method would take 154.0174μs. This would lead to a factor of improvement of 1.25. While the improvement is not as great as without write-combining, it is still significant.

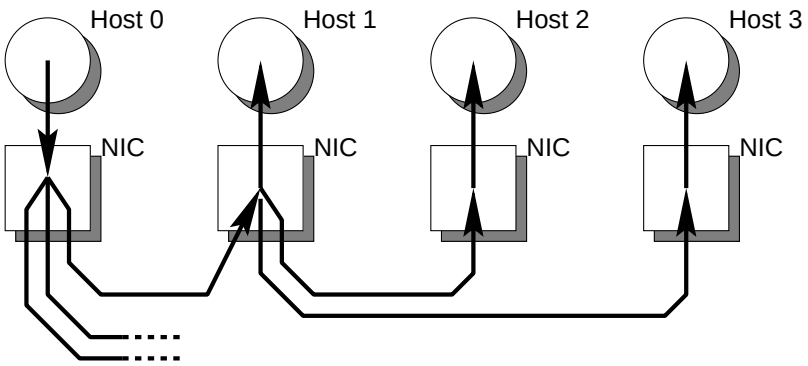
2.2 Broadcast/Multicast with the Multi-send Primitive

While the multi-send primitive is powerful, for covering a large number of destinations we need to perform broadcasts and multicasts hierarchically in order to minimize the overall broadcast/multicast latency. This can be done by having the host at each intermediate node receive the message, then issue another multi-send operation to forward the message to its child nodes. This raises a challenge: how to create an optimal tree? It is also interesting to analyze how this scheme is different from the NIC-based multicast schemes described in [7, 57, 52, 28].

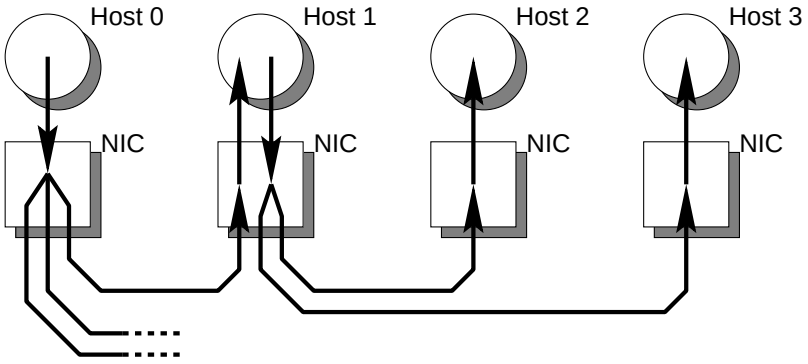
In the NIC-based scheme [7, 57, 52, 28], the incoming multicast packet is transmitted to the child nodes by the NIC after it has been forwarded to the host. Figure 2.4 illustrates the difference in the two methods. This figure shows two diagrams where Node 0 sends a message to multiple destinations, one of which is Node 1. Node 1 then sends to Nodes 2 and 3. Figure 2.4(a) shows a completely NIC-based scheme where the packet coming into Node 1 from Node 0 is sent to Nodes 2 and 3 by the NIC after being forwarded to the host. Figure 2.4(b) shows Node 1 forwarding the incoming packet to the host. The host then issues a multi-send operation to transmit the packet to Nodes 2 and 3.

While it may seem that the completely NIC-based scheme would always be better than the method we are proposing, we believe that that is not the case when the NIC has a very slow processor. The completely NIC-based approach puts more responsibility on the NIC which, as previously mentioned, has a processor 5-15 times slower than the host. We believe that the additional processing power available at the host will allow greater flexibility in, for instance, tree construction. So that depending on the message size and quality of service requirements, a tree optimal in either bandwidth or latency, for that message size can be used for sending the message. This

[1]In our configuration, the write-combining feature was not working in the Myrinet driver supplied with the FM distribution.



(a) NIC-based



(b) NIC-assisted

Figure 2.4: NIC-based multicast and a NIC-assisted multicast.

could be done on a per-message basis. This chapter will examine how to construct a multicast tree that is optimal for latency, and study the performance of such a tree.

2.3 Constructing an Optimal Multicast Tree

In this section, we show how to construct an optimal multicast tree using the proposed new multi-send primitive. The basic idea is to construct a tree such that the maximum number of nodes will be sending at any time. Such a tree would be optimal in terms of latency.

Bar-Noy and Kipnis [4] have shown that in the postal model, a broadcast tree optimal in terms of latency is based on the following recurrence relation. In the postal model, a broadcast to $F_\lambda(t)$ nodes can be completed in t time.

$$F_\lambda(t) = \begin{cases} 1 & \text{if } 0 \leq t < \lambda, \\ F_\lambda(t-1) + F_\lambda(t-\lambda) & \text{if } t \geq \lambda. \end{cases}$$

In this recurrence relation, λ is defined as the ratio of (i) the total amount of time from when the sender of a packet starts sending it to when the receiver receives the complete packet and (ii) the amount of time that the sender spends sending the packet. One unit of time is defined as the time the sender spends sending a packet. It then takes λ time for a recipient to fully receive a packet after the sender starts sending it. It is assumed that as soon as a node receives a packet, it will start sending it to its children nodes.

Intuitively, one can think of $F_\lambda(t)$ as the number of nodes which have received the packet at time t. This is equal to the number of nodes which had already received the packet previously (i.e. $F_\lambda(t-1)$), plus the number of nodes which have just received the packet. The packets which were just fully received at time t must have been sent at λ time before then. Since, at that time there were $F_\lambda(t-\lambda)$ nodes which would have sent these packets, there would be that many new nodes receiving the packet at time t.

In our design, the NIC will be transmitting the packets to different nodes. So we need to apply the postal model from the point of view of the NIC. Specifically, when a packet is sent from a node, say, node 0, to another node, Node 1, which will receive it and send it to a third node, Node 2, then λ is defined as the ratio of i) the time from when the NIC at Node 0 starts transmitting the packet to Node 1 until the NIC at Node 1 is ready to transmit the packet to Node 2 and ii) the time it takes the NIC to finish transmitting the packet. Note that part i) of the ratio is simply the one way latency of a packet (i.e. the time it takes for the NIC to transmit the packet plus the time it takes for the host to receive it plus the time for the host to send the packet to the NIC).

To construct the tree, we used an algorithm similar to the "simple top-down greedy algorithm" in [12]. The tree is stored as a list of destination lists, one per host. The

algorithm uses two queues called the *new* queue and the *old* queue. We start with the root node and enqueue it in the *new* queue with time 0. Then, for each node p in $(p_1, p_2, \ldots p_{n-1})$ we do the following:

- dequeue the node q that has the minimal time t of both of the queues.
- add p to the destination list of q
- enqueue p onto the *new* queue with time $t + \lambda$
- enqueue q onto the *old* queue with time $t + 1$

While there are algorithms which may construct the tree faster [12], this algorithm is simple and general. Because we were constructing the tree off-line in our test program, we were not concerned with the running time of this algorithm. This algorithm does, however, produce a tree that is optimal in the postal model for a given λ [12, 4].

So far we have focused on multicasting a single packet. For multi-packet messages, there are two ways for intermediate nodes to forward the message to it children. One way would be for the node to receive the whole message, then send the message to its children. The other method would be for the node pipeline the message in a packet-wise fashion. In other words, the node would receive the message one packet at a time, and forward the packets to its children as soon as they are received rather than waiting for the whole message to be received.

The decision on whether to pipeline the message or not will mostly depend on whether the communication subsystem supports it. FM, for instance, does not support message pipelining in general (though we were able to do this in a special case). In Section 2.5 we will show performance results with and without pipelining.

2.4 Our Implementation of the Multi-send Primitive

We used Illinois Fast Messages (FM) version 2.1 from the HPVM 1.0 distribution for the base of our experiments. This version was used because it is the latest version of FM available for Linux.

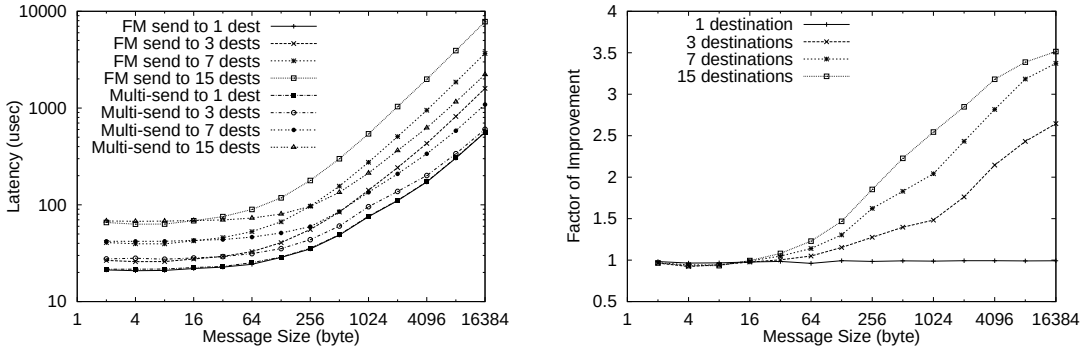
In FM 2.1, a message is associated with a stream. In order to send a message, a stream is created between the node and the destination with the FM API call `FM_begin_message()`. Data can then be sent on the stream using the `FM_send_piece()` API call. When enough data is sent to fill a packet, FM creates a packet and writes it to the NIC which in turn transmits it to the destination node. On the receiving side, FM calls a message handler to optionally re-assemble the message, packet by packet, into the user buffer. A stream is terminated using the `FM_end_message()` API call, at which time any remaining data is assembled into a packet and written to the NIC.

To avoid loosing packets due to buffer overflow at the receiver, FM uses a flow control scheme using credit management. One credit corresponds to one packet buffer at the receiver. On starting FM, each sender starts with a certain amount of credits for each receiver. Before any packet is sent FM checks if the sender has sufficient credits for the receiving node. If there are no available credits at the receiver, then the sender blocks until it receives more credits, otherwise, the sender decrements the number of credits it has for that receiver and sends the packet. Whenever a node receives a packet it increments a counter of the number of credits it needs to return to the sender. These credits are returned to the sender either by piggybacking the credits on a data packet sent to that node, or via an explicit credit packet.

We modified FM by adding a new API call, `FM_begin_message_multi()` which creates a stream between the sender and all destinations specified in the destination list given as an argument. As with a regular FM stream, data is sent using `FM_send_piece()` and the stream is terminated with `FM_end_message()`. The flow control mechanism was also modified. Before a packet is sent credits are checked for every one of the destinations. If there are not enough credits for any destination, the operation blocks until there are. Basically, rather than just verifying that we have credits for a single destination, we check that we have enough credits for each destination. For returning credits normal FM unicast packets have a field for piggybacking them. In our scheme we use piggyback credits for each destination as described below.

When a packet is ready to be sent, the packet is assembled for the first destination and written to the send frame on the NIC. Next a list of information on each additional destination is assembled and written to a new field which is added to the send frame. Each entry in the list holds the logical node number, the physical node number and the returned credits for that destination. These are be used to update the corresponding fields in the packet header so that the packet can be sent to each destination.

Since the host assembles the packet such that it is ready to be sent to the first destination, the NIC can transmit the packet without checking whether it is a multi-destination packet. Only after the NIC has initiated the transmit DMA and is waiting for it to complete, will it check for additional destinations. This way our modifications add no overhead at the NIC for sending standard FM messages. The NIC then updates the logical node number, credit and route fields of the packet for the next destination. This is done as soon as the transmit DMA pointer has passed those fields but without waiting for the entire DMA to finish. This allows the header field updates to be overlapped with the DMA for all but the smallest messages. After updating the fields, the NIC waits for the DMA to finish then the DMA is immediately initiated to transmit the packet to the next destination. This is repeated for each additional destination.



(a) Latency

(b) Factor of improvement

Figure 2.5: Performance of NIC-assisted multi-send operation versus multiple FM send operations.

2.5 Experimental Results

The performance tests were run on a cluster of 16 300MHz Pentium II machines each with 128MB of RAM running RedHat 5.2 Linux with kernel version 2.0.36. The machines are connected by a Myrinet LAN network with LANai 4.3 cards via a 16 port switch.

We tested the performance of the multi-send primitive and compared it with multiple unicast sends. In every iteration of our test routine, the root would send messages to the destinations, and then the last destination would send a zero byte message back to the root after receiving a copy of the message. This was timed for 1,000 iterations, then the average was taken for the result. This was done varying the message size and number of destinations. Figures 2.5(a) and 2.5(b) show the results of this test. Notice that for messages less than 32 bytes, our scheme performs slightly worse than the host-based scheme. This is due to the fact that the NIC transmit time is not smaller than the host send time and due to the overhead of adding the multi-send primitive to FM. However, the NIC-assisted scheme performs clearly better than the host-based scheme for larger messages. It can be observed that the multi-send primitive achieves a factor of improvement of 3.51 for sending a 16K message to 15 destinations.

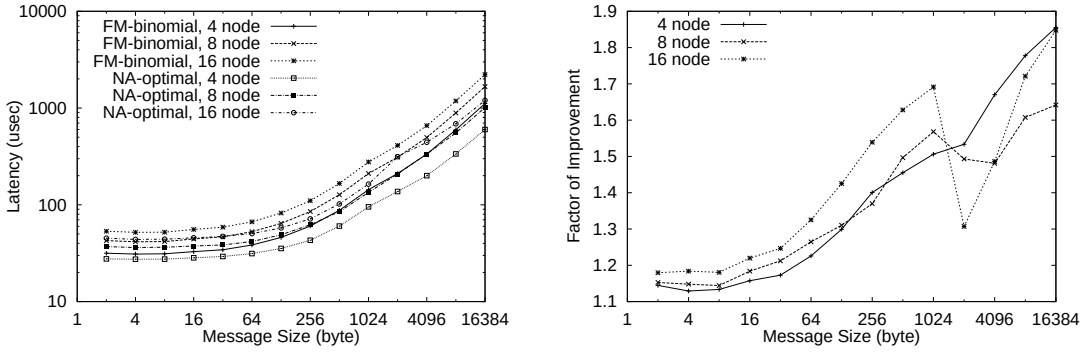
To test the performance of a multicast operation using the multi-send primitive, we ran a test similar to the one described above. One iteration of the test routine would send a message along a multicast tree, then one of the leaf nodes would send a zero byte message back to the root. This was timed for 1,000 iterations then the

average was taken. Because we couldn't be sure which leaf node would receive the message last, we ran the loop several times, each time changing the leaf node which would return the message, then taking the maximum value. In order to cut down on the number of leaf nodes to test we only tested the leaf nodes which were the last children of their parents. This was then varied for message size and number of destinations. When we were using the optimal multicast tree in our tests, we needed to use a value for λ which would produce the best performance. Since the value of λ depends on the message size, in our test program, for each message size we constructed a new tree based on integer λ values from 1 to the number of nodes participating, and took the minimum. So each point in our multicast performance graphs is the minimum over each tree, of the maximum for each responding leaf node in that tree, of the average of 1,000 iterations.

Also, to study the performance impact of pipelining for multi-packet messages, we incorporated modifications to our test program as outlined in Section 2.3. As soon as a packet was received by the intermediate node, it would be sent out, rather than wait for the whole message to arrive. To pipeline a message, the application would have to open a stream to its destination(s), then receive the incoming message one packet at a time until it has received it completely, then close the stream.

This cannot normally be done in FM 2.1 because while a stream is open, FM does not allow the application to make a call to receive parts of a message. This is done to avoid certain deadlock conditions. When a stream is opened with `FM_begin_message()` or `FM_begin_message_multi()`, FM sets a lock so that any calls to receive a message return immediately without receiving. To get around this, we used a version of this call, `FM_unsafe_begin_message_multi()` which does not set a lock. This version is intended to be used only inside a message handler. We used it outside a message handler in our main routine. Since the lock is not set when we open the stream, we are then able to receive the packets of the incoming message and send them out. Though this approach may lead to deadlock in the general case, because our test programs ran one multicast at a time, there were no cyclical dependencies and no deadlock could occur. We used this method, not to show how to pipeline messages in FM, but rather to demonstrate the performance of our scheme when pipelining is possible.

We will next compare NIC-assisted multicasting to host-based multicasting. Because the binomial tree is most often used for host-based multicast operations, as used in MPI, we will use the binomial tree for the host-based multicast, and compare it to the NIC-assisted multicast using an optimal tree as discussed in Section 2.3. Then, because the binomial tree may not be the best tree for host-based multicast, we will use an optimal tree for the host-based multicast and compare that with the NIC-assisted multicast also using an optimal tree. In each case, we will show the impact of message pipelining.



(a) Latency

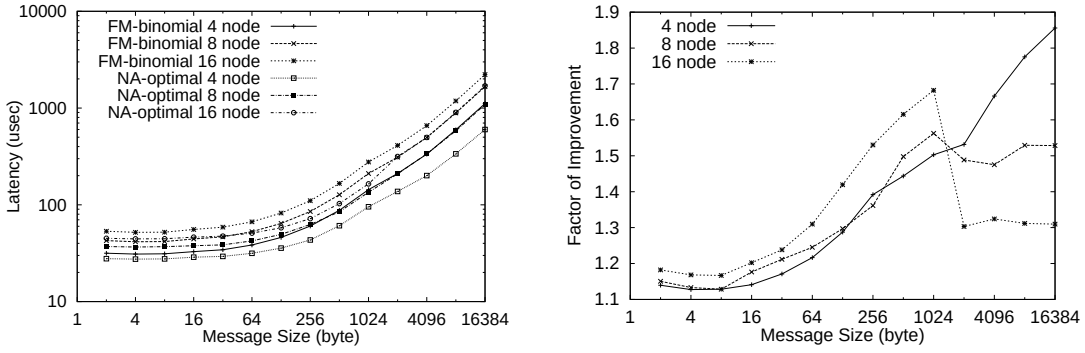
(b) Factor of improvement

Figure 2.6: Multicast performance for NIC-assisted multicast using an optimal tree *with packet-wise pipelining* (NA-optimal), versus multicast using binomial tree with FM unicast send (FM-binomial).

Figure 2.6 compares NIC-assisted multicast using an optimal tree with pipelining against the host-based multicast using a binomial tree. Notice that the NIC-assisted scheme is better for every message size and every number of destinations. The dip in the factor of improvement for 2048 byte messages with 16 nodes is due to packetization effects. The graph also shows a factor of improvement of 1.86 for multicasting a 16K message to four nodes and a factor of improvement of 1.85 for multicasting a 16K message to 16 nodes.

Figure 2.7 shows the same comparison as above but without message pipelining. We can see that the when compared to the pipelined case, performance does not change for one packet messages (FM packets are 1,536 byte) or for four nodes of any size. This is because pipelining does not occur for single packet messages, and for a four node broadcast the optimal tree is flat ($\lambda = 2$), i.e., the root sends the message directly to all the three destinations, so again no pipelining would occur. While the performance improvement is not as great for multi-packet messages with eight and 16 nodes, when compared to the case when messages are pipelined, there is still a 1.53 and 1.30 factor of improvement for 16K messages with eight and 16 nodes, respectively.

To see what impact the shape of the tree had on the performance we are seeing, the optimal tree algorithm was used with the host-based unicast primitive and compared with the same data for the multi-send primitive above. Figures 2.8 and 2.9 show the results of this comparison. In this test, our scheme performs a little worse than the host-based method for messages less than 32 bytes. We believe that this is due to



(a) Latency

(b) Factor of improvement

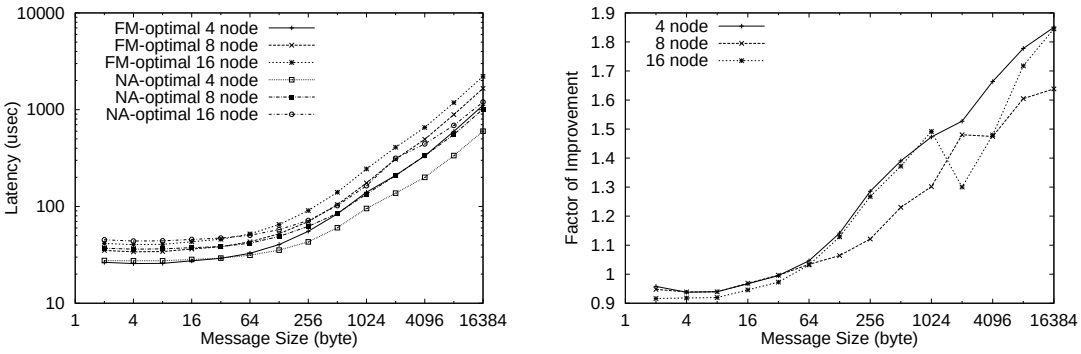
Figure 2.7: Multicast performance for NIC-assisted multicast using an optimal tree *without packet-wise pipelining* (NA-optimal), and multicast using binomial tree with FM unicast send (FM-binomial).

the overhead of the additions to FM similar to that observed for Figure 2.5(a). For multi-packet messages, the optimal tree for the host-based method turned out to be a binomial tree (i.e. $\lambda = 1$), so the performance improvement for those messages is the same as in the previous graphs.

2.6 Related Work

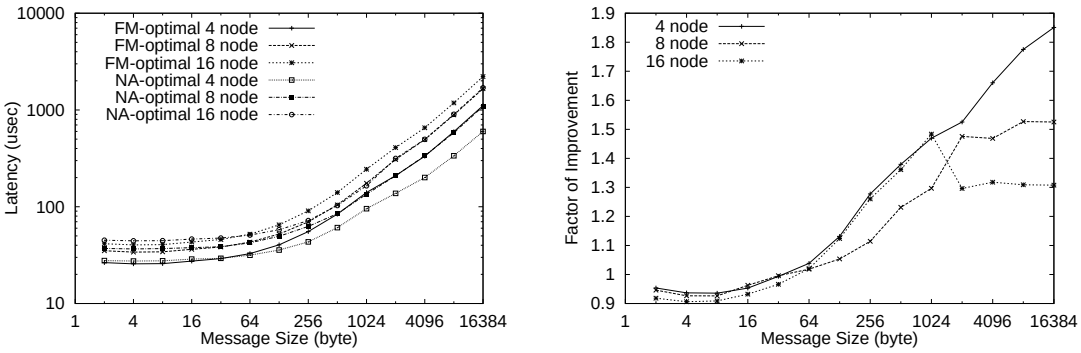
NIC-based multicasting has been previously proposed by Verstoep et.al. [57] and Bhoedjang et.al. [7]. Their schemes perform the entire multicast operation at the NIC, as opposed to our scheme which uses a multi-send operation as a primitive for multicast.

Verstoep, et. al. extend FM 1.1 to include NIC-based multicasting. In their scheme (called FM/MC) [57], the multicast is performed completely at the NIC-level. At intermediate nodes, the packet is forwarded to the host, but then immediately transmitted to the child nodes without involving the host. The host receive queues are divided into multicast queues and unicast queues. In order to prevent buffer overflow, credits are managed separately for each. One multicast credit corresponds to one packet buffer in *each* host in the network, rather than just for one host, as with unicast credits. One NIC on the network is designated as a credit manager which distributes and collects the multicast credits. A NIC must request credits from the manager before multicasting a packet. However, once a multicast has been initiated,



(a) Latency (b) Factor of improvement

Figure 2.8: Multicast performance for NIC-assisted multicast using an optimal tree *with packet-wise pipelining* (NA-optimal), and multicast using an optimal tree with FM unicast send (FM-optimal).



(a) Latency (b) Factor of improvement

Figure 2.9: Multicast performance for NIC-assisted multicast using an optimal tree *without packet-wise pipelining* (NA-optimal), and multicast using an optimal tree with FM unicast send (FM-optimal).

the intermediate nodes do not need to check for credits and can immediately forward the packet to their children when they receive it.

Bhoedjang et. al. propose a new message passing protocol called Link-level Flow Control (LFC) [7]. This system does all flow control at the NIC and also performs the multicast completely at the NIC. In this system data for a packet is copied to a buffer pool on the NIC by the host, then the host writes a descriptor to the send queue. The NIC polls the send queue and transmits the packets when there are sufficient credits. In order to send a multicast packet, the host simply has to add one descriptor for each destination but refer them to the same data. At an intermediate node, the NIC receives the packet then adds descriptors for its children to the send queue.

The main difference between our scheme and the schemes described above, is that a multicast operation in the schemes described above is done completely at the NIC. Also, our scheme keeps the receive queues and credit management unified for both unicast and multicast messages.

2.7 Summary

We have introduced a new NIC-based multi-send primitive, which uses the NIC to transmit multiple copies of a packet to different destinations. We then showed how this primitive can be used in a multicast tree to further improve performance for large numbers of destinations.

The multi-send primitive gave us a 3.51 factor of improvement over conventional host-level iterative sends for 16K messages. We also observed a 1.85 factor of improvement for 8K messages and 16 nodes when using the primitive in a multicast operation using an optimal tree versus using a binomial tree with the traditional host-level unicast sends.

CHAPTER 3

NIC-BASED BARRIER SYNCHRONIZATION

Barrier synchronization is a common operation in parallel and distributed systems. An efficient implementation is important because while processors are waiting on a barrier, generally, no computation can be performed, which impacts parallel speedup. The efficiency of barrier operations also affects the granularity of a parallel computation. If the barrier latency is high, then the granularity must also be high. With a lower latency barrier operation finer-grained computation can be supported. So it is important to minimize the amount of time spent waiting on the barrier.

Earlier generation SMP systems and MPP systems, such as the Cray T3E and CM-5, had special hardware to perform barriers. However clusters built from commodity components lack the communication assists to perform these operations. Dietz [18] had proposed providing hardware barrier over a separate network for clusters of workstations. Such approach requires two networks and may not be cost effective.

Most current clusters use software barriers based on *host-based* point-to-point communication. With *host-based* communication, each message is initiated by the host, passed to the NIC, then to the NIC on the receiving node and finally to the receiving host. The one way latency of such a host-based message may be as high as 30µs. Depending on the algorithm a software barrier would take $\log_2 N$ (e.g., a pairwise-exchange algorithm as used in MPICH [22]) to $2\log_2 N$ (e.g., a gather-and-broadcast algorithm as described in [31]) steps, where N is the number of participating processors. So a barrier across 16 processors would take 120 to 240µs per barrier. This provides high overhead for a barrier and does not lead to scalable or fine grained parallel implementations.

In a barrier operation, often the reception of one message triggers the sending of another message. A NIC-based implementation improves the responsiveness of the barrier by eliminating the round trip over the PCI bus. In this chapter, we investigate the design issues of such an implementation, such as being able to handle multiple concurrent barriers with different processes which use the same NIC, being able to handle multiple consecutive barriers, assuring reliable, in-order delivery of the barrier messages, and initialization of barrier state at the NIC. We also describe the implementation of a NIC-based barrier as an addition to Myricom's GM communication subsystem.

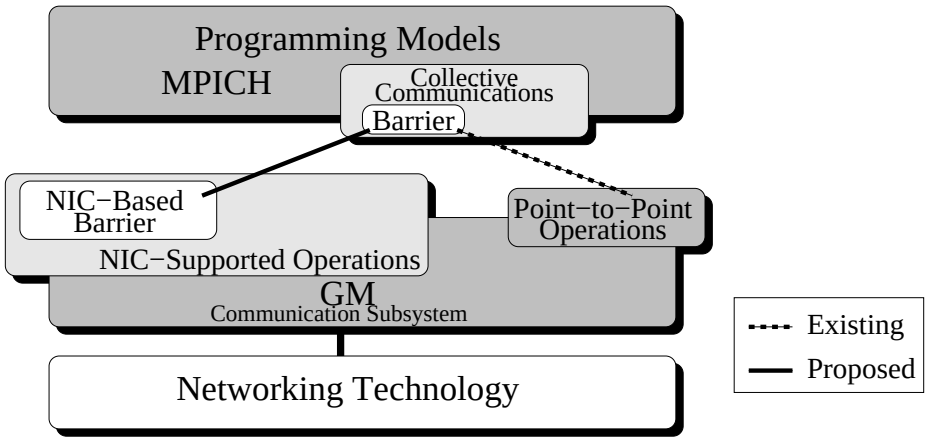


Figure 3.1: NIC-based barrier for the MPICH middleware

Our NIC-based barrier implementation achieved a barrier latency of 102.14µs for 16 processes which is a 1.78 factor of improvement over the host-based barrier for the same algorithm using LANai 4.3 cards. This factor of improvement is expected to increase with the size of the system and with the speed of the NIC processor. Using LANai 7.2 cards, which has a faster processor, we achieved a 1.83 factor of improvement for just eight processes. We expect that the factor of improvement will also increase if an additional programming layer, such as MPI, is added over GM because of the additional overhead the layer adds to each message sent or received. Another feature of our NIC-based barrier implementation is better utilization of the host processor. Because the barrier algorithm is performed at the NIC, the processor is free to perform computation while polling for the barrier to complete. This is known as a *split-phase barrier* [23]. Our NIC-based barrier operation promises scalable fine-grained parallel computation over clusters of workstations.

Figure 3.1 illustrates our approach to adding NIC-based barrier to the GM communication subsystem, and how this new functionality is provided to the MPICH middleware library. The dotted line indicates how the barrier operation is traditionally implemented using point-to-point messages. The solid line shows our implentation using the NIC-based barrier operation.

This chapter is organized as follows. Section 3.1 describes the basic idea of NIC-based barriers. We describe the design issues involved in implementing NIC-based barrier in Section 3.2. The implementation details are discussed in Section 3.4. The details of the barrier algorithms used are described in Section 3.3 followed by an evaluation of our implementation in Section 3.5. We summarize our work in Section 3.6.

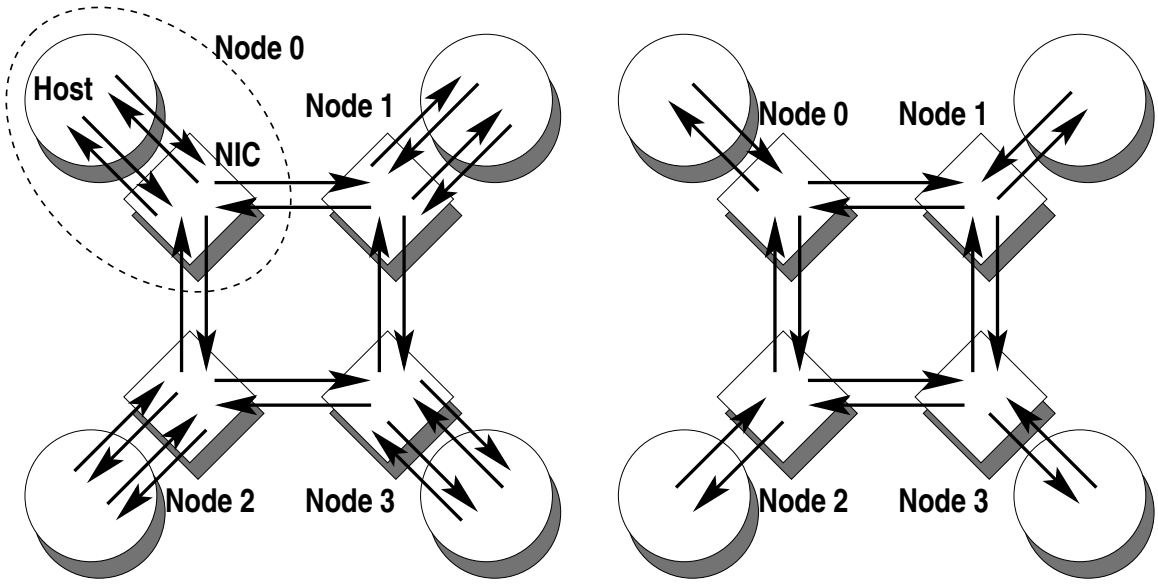
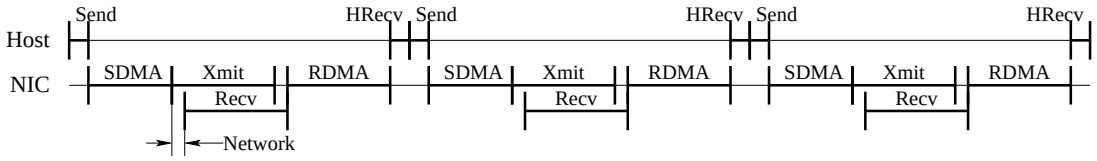


Figure 3.2: Host-based barrier (left) and NIC-based barrier (right)

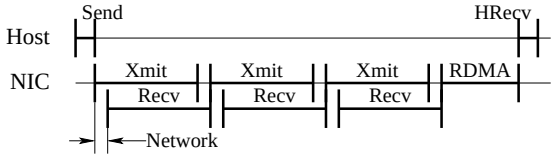
3.1 NIC-Based Barrier and Performance Benefits

The basic idea of the NIC-based barrier is to have the host initiate a barrier operation at the NIC and have the NIC notify the host when it has completed the barrier. Figure 3.2 shows block diagrams comparing host-based barrier to NIC-based barrier. The diagrams show barrier operations, where processes at Nodes 0 and 1 exchange messages at the same time as processes as Nodes 2 and 3 exchange messages, after which the processes at Nodes 0 and 3 exchange messages at the same time as the processes at Nodes 1 and 2 exchange messages. The diagram on the left in Figure 3.2 shows a host-based barrier. In the host-based barrier, in order for a message to be sent the host transfers the message to the NIC which transmits it on the network to the receiving NIC. The receiving NIC receives the message and transfers it to the host. Once the host receives the message, it can initiate sending a message to the next host.

By basing the barrier operation at the NIC, rather than at the host, the intermediate messages need not be transferred between the host and the NIC. The diagram on the right in Figure 3.2 shows a NIC-based barrier. In the NIC-based barrier model, the host sends a message to the NIC to initiate the barrier operation and waits for notification from the NIC that the barrier has completed. The barrier messages are then exchanged between NICs and need not be transferred to the host. As soon as a NIC receives a barrier message, the message to the next process can be sent directly.



(a) Host-based



(b) NIC-based

Figure 3.3: Timing diagram comparing latencies for host-based barrier and NIC-based barrier

3.1.1 Estimated Performance Improvement

In this section we estimate the performance improvement of using a NIC-based barrier over using a host-based barrier. This estimate is based on a pairwise-exchange algorithm similar to the one used in MPICH[22]. To perform a barrier with N processes using this algorithm, each process exchanges messages with $log_2 N$ other processes (This algorithm is described in more detail in Section 3.3.). Figure 3.3 compares the latency of a host-based barrier with a NIC-based barrier for eight processes. In these diagrams it takes three message exchanges per process to complete the barrier. Each timing diagram shows the breakdown of a barrier operation at a single node. For simplicity, we assume that each node has only one process and that all processes start the barrier at the same time, so the timing diagrams for all eight nodes would look the same. We also assume that the NICs have separate receive and transmit channels[2] to the network, so that one message can be received while another is being transmitted. In these diagrams, $Send$ corresponds to the time from when the host initiates the send until the NIC detects it. $SDMA$ is the time it takes for the NIC to transfer the data for the message from the host memory to the NIC transmit buffer. $Xmit$ corresponds to the time for the NIC to transmit the message on the network. We assume that the network is wormhole routed. Thus the time between when the transmit starts at the sender and when the receive starts at the receiver is small.

[2]Current Myrinet NICs support this feature.

This time is represented as $Network$ in the diagrams. $Recv$ represents the time for a message to be received by the NIC. The time to transfer a message from the NIC to the host is represented as $RDMA$. Finally, $HRecv$ corresponds to the time it takes the host to process the message once it has been transferred from the NIC.

Figure 3.3(a) shows the barrier latency for a host-based barrier. After the host transfers the message for the first destination to the NIC, the NIC starts transmitting it. The NIC will start receiving a message after a delay of $Network$ once the message has started being transmitted. Since we assume that the barriers started at the same time on all nodes, the NIC will receive a message sent to it after a delay of $Network$ after it has started transmitting its message. The diagram shows these transmit and receive events occurring concurrently. Once the message has been received, the NIC transfers the message to the host which processes it. The host then initiates sending a message to the second destination and the process is repeated. After the process is repeated again for the third destination, the barrier has completed.

Figure 3.3(b) shows the latency for a NIC-based barrier. Here, the host transfers a message to the NIC to initiate the barrier operation. The NIC starts transmitting the message to the first destination. As before, the NIC starts receiving a message from the corresponding node after a delay of $Network$. After the message has been received by the NIC, the NIC starts transmitting the message for the second destination. Again the message from the corresponding node is received while the second message is being transmitted and, similarly, for the third message. After the third message has been received, the NIC transfers a notification to the host. Once the host processes the notification, the barrier has completed.

From these diagrams, we can see that the latency for an eight node host-based barrier is $3 \times (Send + SDMA + Network + Recv + RDMA + HRecv)$, while the latency for a NIC-based barrier is only $Send + 3 \times (Network + Recv) + RDMA + HRecv$. More generally, for an N process system, the host-based barrier latency would be:

$$T_{Barrier}^{Host} = \log_2 N \times (Send + SDMA + Network + Recv + RDMA + HRecv) \quad (3.1)$$

And for the NIC-based barrier it would be:

$$T_{Barrier}^{NIC} = Send + \log_2 N \times (Network + Recv) + RDMA + HRecv \quad (3.2)$$

The factor of improvement of the NIC-based barrier over the host-based barrier is given by:

$$\begin{aligned} \text{Factor of Improvement} &= \frac{T_{Barrier}^{Host}}{T_{Barrier}^{NIC}} \quad (3.3) \\ &= \frac{\log_2 N \times (Send + SDMA + Network + Recv + RDMA + HRecv)}{Send + \log_2 N \times (Network + Recv) + RDMA + HRecv} \end{aligned}$$

From Equation 3.3 we can predict that as the host send overhead increases, say from the addition of another programming layer such as MPI, the factor of improvement will increase. The factor of improvement will also increase as the number of nodes increases and as the network performance increases.

3.2 Design Issues

There are several major issues in designing a NIC-based barrier operation. One issue is how to handle unexpected barrier messages that are received by a node which hasn't initiated a barrier. Another issue is initializing barrier data structures at the NIC when a process opens an endpoint, and similarly cleaning up data structures after an endpoint is closed. Reliability and in-order delivery of barrier messages must also be addressed. The last issue is to handle multiple concurrent barriers at the same NIC.

In this section we first describe our system model and then identify these design issues and present some solutions. In the next section, we identify the solutions we have implemented.

System model: A system consists of a collection of *nodes*. Each node consists of one or more *programmable NICs* and one or more *host processors*. The nodes are connected, through the NICs, by a *communication network*. *Processes* run on a host processor and can communicate with each using the NICs by using an abstraction called a *communication endpoint*. A process can allocate one or more such endpoints. An endpoint is associated with a particular NIC at the node, so that messages sent or received by the process are handled by that NIC. Messages are sent from one endpoint to another. Similarly, a barrier operation is associated with endpoints. A barrier operation synchronizes the processes which are attached to the specified endpoints.

3.2.1 Handling Unexpected Barrier Messages

If all processes start the barrier operation at the same time, then keeping track of which messages were received would be easy, because the barrier messages would be received in the same order as they are expected. In practice, however, processes may initiate barrier operations in an asynchronous manner. Thus, a NIC may receive barrier messages before the NIC is ready for them and possibly even before the host has initiated the barrier. To make matters worse, there may be multiple consecutive barriers with different subsets of processes, so the NIC may receive barrier messages from future barriers. In the worst case, a process might perform multiple consecutive two-process barriers, one with every other process in the system. Then, if that process is slower than the others and the other processes reach their barriers first, the associated NIC would receive $N - 1$ unexpected barrier messages, where N is

the number of processes in the system. So the NIC must be prepared to receive a barrier message from any process on any node in any order at any time. However, once a process initiates a barrier operation and is waiting for it to complete, it will not initiate another one until that barrier completes. So the NIC can receive *at most* one unexpected message from every other process on every node.

One method of handling unexpected barrier messages is to accept and record the reception of every unexpected barrier message in a *unexpected barrier message record*. For instance, a flag could be allocated for every possible communication endpoint on every possible node. When a barrier message is received, the flag corresponding to the endpoint that sent the message would be set. Then, when the NIC is ready to receive a barrier message from a particular endpoint, the NIC would simply have to check the corresponding flag to see if that message has already been received. The flags are then reset after they are read to allow another unexpected message to be recorded from that same endpoint. Representing the flags as a bit array is the most space efficient representation and also setting, resetting or checking the flags would take constant time. Because GM allows only eight endpoints per NIC, this overhead is only one byte per connection.

3.2.2 Initialization and Cleanup

Another problem is how to initialize the data structures recording these messages, and how to clean them up after a partially completed barrier is aborted. For example, let's say process A on Node 0 initiates a barrier with process B on Node 1, and that process B dies before a barrier message is received. When the NIC at Node 1 receives the message it will record it as an unexpected message, possibly destined to a process that hasn't started yet. Now, say, process A is killed, and two new processes A' and B' are started on Nodes 0 and 1 respectively, and reuse the same endpoints as the previous processes. If process B' initiates a barrier, the NIC will see that it has received a message from Node 0 from the same endpoint that process A' is using and will assume that it has received a barrier message from A' even though that message was sent by A. It is possible now for B' to complete the barrier before A' starts the barrier.

Before we discuss possible solutions, we need to make certain assumptions about the state of the system when a process is started. Processes that will communicate may not all start at the same time. Because of this, it is possible that when a node sends a message to a remote endpoint the remote process to which the message was sent may not have started yet. This may be unavoidable, but is usually benign. In the worst case this message would have to be retransmitted. It is also possible that a different process is still using that endpoint. This has more serious implications. Messages may be sent between the nodes, each one thinking that the message has been sent or received by a different process. To avoid this possibility, it is sufficient

to require that if a process p will communicate with a process q through a remote endpoint e, then when process p is started the endpoint e cannot be owned by a process other than q. Furthermore, no old messages can be in the communication channels between p and q. While this may seem like a rather strict requirement, this usually happens in practice. For instance when a parallel program is started on several machines, if a resource, such as an endpoint, is not available to one process, the whole program is aborted and restarted once the resource is available. A way around this requirement is to include a mechanism to distinguish messages of one parallel program from another.

One naive solution is to simply clear the unexpected barrier message record of all messages destined for a particular endpoint when that endpoint is opened. This may solve the problem mentioned above, but that does not allow barrier messages to be received for a process that hasn't started, or opened an endpoint. This may happen, if, for instance, the first action of a program is to do a barrier in order to make sure all its peers have started.

A better solution is to have the NIC reject any barrier messages for a closed endpoint. Then, the sender of the barrier message will resend the message, but only if the endpoint that initiated the barrier has not closed since the message was sent. Then, with the above requirement about the state of the system when a process starts, we know that once a process opens an endpoint, and starts accepting barrier messages, no old messages will be received. While this method may increase the latency of the barrier, this will only be the case if a participating endpoint has not been opened yet.

Another solution is to record received barrier messages for a closed port, but then reject those messages once the endpoint is opened. Then, the NICs which sent those messages will then resend them, but only if the endpoints which initiated the barriers have not closed since the original message was sent. This has the same drawbacks as the previous solution, except, it would require only one retransmission, rather than an unbounded number. Thus we adopt this approach in our implementation.

3.2.3 Reliability and In-Order Delivery

A lost barrier message could hang processes indefinitely. Therefore it is important to provide a mechanism to deliver barrier messages reliably. Related to this issue is the guarantee of the order in which the messages will be delivered. There are two design options with regard to the delivery order of barrier messages relative to non-barrier messages. If barrier messages are guaranteed to be delivered in-order with regard to non-barrier messages, then messages sent before a barrier is initiated by the sending process will be received before the barrier completes at the receiving process. Similarly messages sent after a barrier completes on the sending process, will not be delivered before the barrier completes on the receiving process. This will not be true

if the relative order of barrier messages and non-barrier messages is not preserved. Instead, the order of messages will be maintained separately among barrier messages and among non-barrier messages.

One method of preserving the relative order between barrier messages and non-barrier messages is to use the same mechanism to provide in-order and reliable delivery for both types of messages. This is the approach we adopt in our implementation.

3.2.4 Multiple Concurrent Barriers

Because barriers using message passing do not depend on holding shared resources, independent barriers on separate nodes can occur concurrently. However, if a NIC can be used by more than one process, then the NIC-based barrier mechanism must be designed to allow multiple processes to initiate barrier operations concurrently. If two processes using the same NIC are participating in the same barrier, it may be possible to provide an optimization, where a barrier message need not actually be sent, but rather just have a flag set to indicate that it has been received. Our initial implementation allows multiple instances of barriers to exist concurrently on the same NIC. We intend to incorporate the above optimization in our final implementation.

3.3 Barrier Algorithm

In this section we describe two algorithms for performing barriers and how we implemented them on the NIC. The first is a gather-and-broadcast algorithm (GB) [31], and the second is a pairwise exchange algorithm (PE) that is used in MPICH [22].

3.3.1 Algorithm Descriptions

The GB algorithm constructs a fixed dimensional tree of the nodes participating in the barrier. The algorithm then proceeds in two phases: gather and broadcast. In the gather phase, each node, except the root, waits to receive a gather message from each child, then sends a gather message to its parent. The root waits for a gather message from all its children, then sends a broadcast message to each of them and exits the barrier. As each other node receives the broadcast message, it sends the broadcast to each child then also exits the barrier. We would expect that the dimension of the tree would impact the performance of the barrier. Thus, depending on the parameters of the communication subsystem and the size of the barrier one could use a different dimension tree to get the best performance.

The PE algorithm runs as follows. Assuming we have a power-of-two number of nodes, we start the algorithm by placing each node into its own group; i.e., if we have N nodes, then there are N groups each with one node. Then the algorithm proceeds recursively by merging groups, two at a time, until there is only one group. The

algorithm is then finished. To merge two groups, each node in one group exchanges messages with exactly one node in the other group. The nodes in those two groups then form one new group. Because the message exchanges can happen concurrently, this can be accomplished in one step. The algorithm runs in a total of $\log_2 N$ steps if N is a power of two.

If N is not a power of two, then we divide the nodes into a set S and a set S' such that $|S|$ is the largest power of two less than N. We then pair each node in S' with a node in S. Each node in S' will send a message to its corresponding node in S, which waits for the message. Next, the nodes in S perform a barrier as described above. Finally, the nodes in S which had received a message from the nodes in S' send a message back to their corresponding node in S'. For non-power-of-two number of nodes the algorithm should run in $\lfloor \log_2 N \rfloor + 2$ steps.

NIC processors are typically much slower than the host processors (e.g., Myrinet NIC processor speeds range from 33MHz to 233MHz while processor speeds for a typical host processor might range from 300MHz to 3GHz). For this reason it may be more efficient to have the host processor perform some parts of the algorithm.

An issue here is the construction of the tree for the GB algorithm. One alternative is to pass the list of participating nodes to the NIC and have the NIC construct the tree. However, the tree construction is a relatively computationally intensive task which can easily be computed at the host. The host at a particular node needs to inform the NIC only of the children and parent of the node, rather than all the nodes in the barrier. This also reduces the amount of data that has to be transferred to the NIC. Similarly, for the PE algorithm, the task of determining the pairings can be done either at the NIC, or at the host. Again, this can be done much quicker at the host and also the whole list of nodes need not be transferred to the NIC.

3.3.2 Algorithm Implementation

Both algorithms were implemented on the NIC. We will first describe the changes to the GM API, then describe the implementation details at the NIC.

We added two new functions to the GM API to support NIC-based barriers: `gm_provide_barrier_buffer()` and `gm_barrier_send_with_callback()`. Before initiating a barrier the host calls `gm_provide_barrier_buffer()` to provide the NIC with a receive token. To perform a barrier, the host must compute the barrier tree (for the GB algorithm), or the list of processes with which to exchange messages with (for the PE algorithm). Then, the process then calls `gm_barrier_send_with_callback()`. For the GB algorithm, the process specifies, in the function call, the parent node id and port id, and the node and port ids of each of the children. For the PE algorithm, the process specifies the list of nodes and port numbers with which to exchange messages. Next, the host polls `gm_receive()` until it receives a `GM_BARRIER_COMPLETED_EVENT`. The reception of this event indicates the completion of the barrier. Because we

separate the barrier initiation from the polling of the barrier completion, a *split-phase barrier* [23] can be performed, where some bounded computation can be done while polling for the barrier completion.

In the GB algorithm, the `gm_barrier_send_with_callback()` function creates a send token with the node list and passes it to the token queue on the NIC. There is a separate packet type for each phase. When the SDMA state machine receives the barrier send token from the process, it first sets the send token pointer in the port structure to this send token, then it checks if it received a barrier *gather* packet from each of its children. If so, it clears the bits for the received packets, and queues the send token for the parent node. If it has not received a gather packet from each child, then it must wait until all have been received.

When a barrier gather packet is received, the packet is recorded, then, if the send token pointer in the port data structure is non-zero, the RDMA state machine checks to see if gather packets have been received from all the children, and, if so, the send token is prepared to send a barrier *gather* packet with the parent's port id and is queued in the send queue for the parent's node id.

When the root node receives gather messages from each child, or when a child receives a barrier *broadcast* packet, the RDMA state machine sends a receive token to the host indicating that the barrier has completed, and sets the send token pointer in the port data structure to zero. Then the send token is prepared to send a barrier *broadcast* packet to the first child, and is enqueued on the connection to the node of the first child. Once the SDMA state machine has prepared the packet to be transmitted, the send token is updated to be sent to the next child, and it is re-queued. This continues until a broadcast packet has been sent to each child. Then the send token is returned to the port.

In the PE algorithm, the `gm_barrier_send_with_callback()` function creates a send token with the node list and passes it to the token queue on the NIC. When the SDMA state machine receives the barrier send token from the host, it sets the *node index* to point to the first node in the node list, sets the destination node id and port id of the send token to this first node and port, then sets the *send token pointer* in the port data structure to point to the send token to indicate that a barrier has been initiated. Then, after the SDMA state machine prepares the packet to be sent, it checks to see if a barrier packet has been received from that same destination. If it has, it 1) clears the bit for that message, 2) increments the index in the send token to point to the next destination, 3) writes the next destination's port number in the send token, 4) removes the send token from the current queue and 5) queues the send token in the queue for the connection for the next destination. If the expected barrier packet was not yet received, then the send token is simply removed from the queue.

When a barrier packet is received, the RDMA state machine checks if the port that the message is addressed to has received a barrier send token from the host by checking if the pointer to the send token is non-zero. If so, and if this is the

expected barrier message, then the send token is updated and enqueued for the next destination. In all other cases, the reception of the message is simply recorded.

Once the packet to the last destination has been sent and the corresponding packet has been received, the barrier is complete. The NIC DMAs a receive token to the host, returns the send token, and sets the send token pointer in the port data structure to zero.

3.4 Implementation

In this section we describe our implementation of a NIC-based barrier as an addition to Myricom's message passing system, GM[34], version 1.2.3. First, we describe GM, then identify our design choices and describe the implementation details of each.

3.4.1 Overview of GM

GM consists of a driver, a library and a Myrinet control program (MCP). The driver loads the MCP on to the NIC when it is loaded. During the execution of a program the driver is used mainly for opening *ports*, pinning and unpinning memory, and to put a process to sleep or to wake a process for blocking functions. A *port* is a data structure through which a process can communicate with the NIC while bypassing the operating system. A port also serves as a communication endpoint. Once a port is opened, the process can communicate with the NIC, bypassing the OS and avoiding system call overhead. In GM version 1.2.3, each NIC can support a maximum of eight ports, some of which are reserved.

At the host level GM is connectionless, but provides reliability by maintaining reliable connections between NICs of different nodes. Flow control is used between the NIC and the host to avoid buffer overflows. To provide this reliability GM uses the concept of tokens. When a process opens a port, it has a certain number of *send tokens* and *receive tokens*. Each send token corresponds to a send event. For sending a message the process fills-in a send token describing the send event and queues it on the send queue. Once the NIC has completed the event, and has freed the resources corresponding to that event, the send token is returned to the process.

In order to receive a message, the process must allocate a buffer into which the message will be received and pass a receive token describing the buffer to the NIC. Once the NIC has DMAed the data into the buffer, the receive token is returned to the process. The process must poll to detect returned receive tokens. Messages may only be sent from and received into buffers which are pinned in memory. Memory is pinned using special functions supplied by GM.

Figure 3.4 is a block diagram of GM where a process has two ports through which send tokens and receive tokens are transferred to and from the MCP without

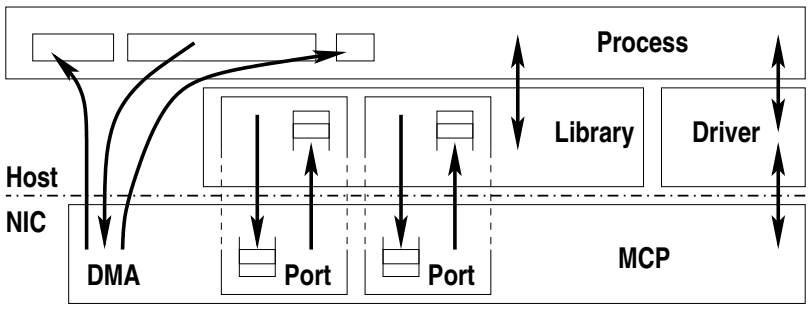


Figure 3.4: Block diagram showing the components of GM.

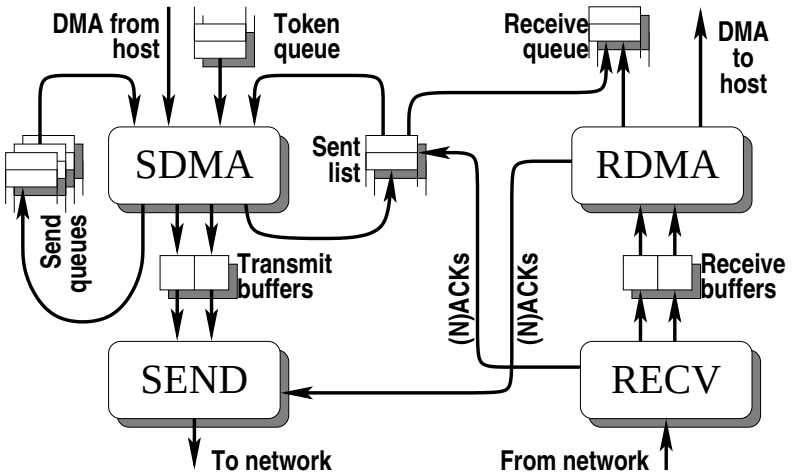


Figure 3.5: Block diagram of the components of the MCP.

going through the driver. The figure also shows DMA operations which transfer data directly to and from the process' memory.

The NIC has a data structure for each local port, which contains the send and receive queues. The NIC also has data structures each corresponding to a connection to one node in the system. The connection structure contains information about the state of the connection and the port from which to send next.

Figure 3.5 shows a block diagram of the MCP. The MCP consists of four state machines called SDMA, SEND, RECV and RDMA. The SDMA state machine polls for new send tokens and queues them on the queue for the appropriate connection. The SDMA state machine is also responsible for initiating a DMA to transfer data from the host memory to the transmit buffers in the NIC and to prepare the packet for

transmission. Once the packet is ready to be transmitted, the send token is moved to the sent list. The SEND state machine is responsible for transmitting packets which were prepared by the SDMA state machine and any acknowledgment packets which may be pending. The RECV state machine receives incoming packets into receive buffers and handles acknowledgment and negative acknowledgment packets. When the RECV state machine receives an acknowledgment it removes the token associated with that send from the sent list and passes it back to the host. The RDMA state machine prepares acknowledgment and negative acknowledgment packets and DMAs the data to the host buffer corresponding to an appropriate receive token. The RDMA state machine also adds receive tokens in the receive queue to notify the process that the receive has completed.

3.4.2 NIC-Based Barrier in GM

We added two new procedures to GM called `gm_provide_barrier_buffer()` and `gm_barrier_with_callback()`. The `gm_provide_barrier_buffer()` procedure transfers a barrier receive token to the NIC. The NIC will return this token to the process when the barrier has completed. This procedure is actually a misnomer because no buffer is needed by the barrier. It was named this because it is the analog of `gm_provide_receive_buffer()`.

The `gm_barrier_with_callback()` procedure fills in a send token describing the nodes and ports with which to exchange messages and queues it on the send queue. The NIC, upon receiving the token, will perform the barrier operation, then return the receive token, previously provided by a `gm_provide_barrier_buffer()` call, to the process indicating that the barrier has completed. Note that the send token need not be returned when the receive token is returned. For instance, in the PE algorithm, if there is a non-power of 2 number of nodes participating in the barrier, then some nodes in the set S, described in Section 3.3, will be sending messages to the nodes in the set S'. In this case, the NIC need not wait for this last message to be sent before returning the receive token to notify the process. The send token will then be returned when this last send is complete.

Multiple Concurrent Barriers

In order for our implementation to support multiple concurrent barriers, we must allow multiple barriers to exist on the same NIC. Since each port may participate in a barrier independently, the NIC must keep the state of each barrier separate. We do this by putting the state information in the *send token* and keeping a pointer in the port data structure to this send token. This way, when a barrier packet is received, the RDMA state machine can access the state of the barrier by simply dereferencing the pointer. The token will store a list of the port ids and node ids with which barrier

messages will be exchanged, as well as an index, *node index*, into this list to indicate which is the next node to receive from or to send to.

Handling Unexpected Barrier Messages

To handle unexpected barrier messages, we used an unexpected barrier message record. Because there is already a data structure per connection, and each connection has at most eight ports, the record was implemented as a bit array for each connection. When an unexpected barrier message is received, the bit corresponding to the source port and connection is set. The NIC can then check for received messages by checking the appropriate bit. After a bit is checked, the bit is cleared.

Reliability and In-Order Delivery, and Initialization and Cleanup

The difficulty in providing reliability is that in GM, when a reliable packet is transmitted, the send token is added to the sent token list. Only once the packet is acknowledged is the send token de-queued and returned to the process. If a packet is negatively acknowledged, all packets sent after that packet must be resent. This is done by pushing the contents of the sent list back on the send queue.

In our current implementation, which uses unreliable barrier packets, once a barrier packet has been transmitted, it is de-queued then re-queued in the queue for the next destination. Now, since our barrier scheme uses only one send token, and the token can potentially be used to send to multiple destinations, if two or more barrier packets need to be retransmitted, the same token would have to be queued twice.

One solution is to have the barrier event use one token for every destination. Then the NIC will queue a send token separately for each packet sent. Another solution is to provide a separate retransmission mechanism just for barrier messages. Under this solution, the barrier messages will be acknowledged separately and will have separate sequence numbers. This will require separate acknowledgment packet types and structures to keep track of sequence numbers, as well as routines to resend the barrier messages. As discussed in Section 3.2.3, barrier messages and non-barrier messages will not necessarily be received in the same order that they were sent.

We have implemented some of the components necessary to provide reliability. We intend to complete the implementation soon. As described in Section 3.2.2, a barrier message retransmission mechanism is necessary for handling barrier messages which are sent to ports which are closed. Since we have not finished implementing this mechanism, when performing our benchmark programs, we must ensure that any barrier that is initiated completes normally (i.e., no participating port is closed during a barrier operation).

3.4.3 MPICH Modifications

MPICH is designed using a layered approach, such that all that needs to be done to port MPICH to use a new communications device, is to write a new *channel interface* for that device. The channel interface defines a set of low level data-transfer primitives which are used by the upper layers. The host-based `MPI_Barrier()` barrier operation is normally implemented at an upper layer using the high level `MPI_Sendrecv()` call. However, if a barrier operation is implemented in a channel interface, then by defining the `MPID_Barrier` and `MPID_FN_Barrier` macros, that operation will be used instead of the upper layer one.

We modified MPICH-GM version 1.2..3 to use our NIC-based barrier operation. As we will describe in Section 3.5, the PE algorithm performs better than the GB algorithm. For this reason we only used the PE algorithm for MPICH-GM. In the GM channel interface, a send is queued at the host until there is a send token available to issue a `gm_send()` call. When the token is returned, the entry for that send is marked as complete. Receive requests are similarly queued, and marked as complete when the receive token is returned. The `MPID_DeviceCheck()` procedure receives messages from the NIC and marks the completed sends as complete. It also keeps track of the token counts and sends pending messages when send tokens are available.

We implemented a low level procedure called `gmpi_barrier()` to perform the NIC-based barrier. The macros described above were defined to refer to this function. This function first determines the list of nodes with which the NIC will exchange messages. This is done using the same basic algorithm that the MPICH host-based barrier algorithm uses, which was described in Section 3.3. Next, the procedure calls `MPID_DeviceCheck()` until all pending sends and receives have completed and until there is at least one send token and at least one receive token available. Now the procedure is ready to perform the barrier. The procedure calls `gm_provide_barrier_buffer()` followed by `gm_barrier_with_callback()` and decrements the send and receive token counts. The procedure now sets a flag `barrier_done` and polls `MPID_DeviceCheck()` until the flag is set. The procedure `MPID_DeviceCheck()` was modified to set this flag when a barrier receive token is received. When the callback function associated with the barrier is called, the send token count is incremented.

3.5 Performance Evaluation

In this section, we evaluate performance benefits of our implementation. The evaluation is done along multiple angles:

GM-level performance – We evaluate the basic performance of the barrier operation at the GM-level. We compare the performance of the two algorithms using LANai 4.3 NICs with 33MHz processors, and LANai 7.2 NICs with faster,

66MHz processors. This will indicate the effects of faster NICs on the performance of the operation.

MPI-level overhead – We have evaluated the amount of overhead that the MPI layer adds to the barrier latency compared to the GM-level barrier. This will indicate whether applications at the MPI level can effectively utilize the performance of the NIC-based barrier.

MPI-level performance and scalability – Scalability is an important factor in collective communications. Modern clusters can have 1,000 or more nodes, so it is important that collective communication operations such as barrier perform well as the system size increases. We evaluated the performance of the NIC-based barrier and compared it to the host-based barrier at the MPI level. We also compared the scalability of the NIC-based barrier to the host-based barrier.

Granularity of computation – The latency of a barrier operation affects the granularity of computation. If the cost of performing a barrier is high, then the amount of computation performed between barriers will have to be large, otherwise the efficiency of the program suffers. So, by reducing the latency of the barrier, the program can be written using finer granularity without loosing efficiency. We evaluated the performance of a computational loop with barrier for varying granularity of computation.

Varying arrival times – In most real applications, nodes participating in a barrier may arrive at the barrier at different times. We evaluated the performance of the NIC-based and host-based barriers while varying the delay between the barriers.

Performance evaluation with synthetic application – Because the barrier latency in many message passing systems has been high, few benchmarks exist which use barrier heavily. For instance, the NAS[2] benchmarks use very few barriers. In order to simulate a higher granularity application, we wrote a synthetic benchmark. Each synthetic application performs several phases of computation each followed by a barrier. The length of the computation varies from one phase to the next. We compared the overall execution times of the applications using NIC-based barrier and host-based barrier.

The performance results were run on a cluster of 16 dual 300MHz Pentium II machines each with 128MB of RAM, running RedHat 6.0 with SMP kernel version 2.2.5. The machines are connected by a Myrinet[10] LAN network using NICs with 33MHz LANai 4.3 processors. These are connected to a 16 port switch. Eight of these machines are also connected by another Myrinet LAN network using NICs with 66MHz LANai 7.2 processors. These are connected to an eight port switch.

We describe the results of our evaluation in the following subsections.

3.5.1 GM-Level Performance

We tested the latency of our NIC-based barrier implementation and compared it to a host-based barrier implementation on GM. We compared the performance for both the GB and PE algorithms. To test the barrier latency, we ran 100,000 barriers consecutively and took the average latency. Tests were performed for 2, 4 and 8 nodes using LANai 4.3 and the LANai 7.2 NICs, and for 16 nodes using LANai 4.3 NICs.

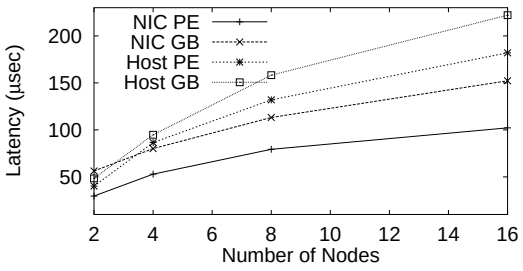
The performance of the GB algorithm on a given system for a given size depends on the dimension of the gather and broadcast tree. In order to find the optimal dimension for the tree, we ran the test for every dimension from 1 to $N-1$, where N is the number of nodes participating in the barrier. The latencies reported in the graphs are the minimum latencies over all dimensions.

Figure 3.6(a) shows the barrier latencies of NIC-based and host-based barriers for each algorithm using the LANai 4.3 cards. Notice that the NIC-based PE barrier performed better than all other barriers, with a 16-node barrier latency of 102.14µs. Also, the NIC-based GB barrier performed better than either host-based barrier except for the two node barrier. The NIC-based GB barrier performed worse for the two node barrier than the host-based GB barrier because of the overhead of processing the barrier algorithm at the NIC. The 16 node barrier latency of the NIC-based GB barrier is 152.27µs. The host-based PE barrier performed better than the host-based GB barrier.

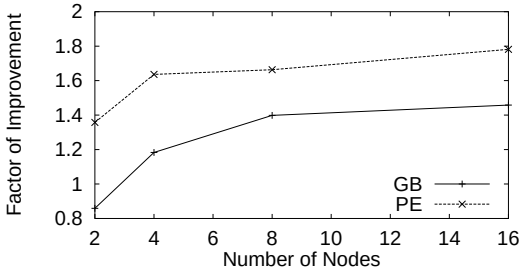
Figure 3.6(b) shows the factor of improvement of the NIC-based barrier over the host-based barrier for both algorithms using the LANai 4.3 cards. For a barrier with 16 nodes, the NIC-based PE barrier gave a 1.78 factor of improvement over the host-based PE barrier, while the NIC-based GB barrier gave a 1.46 factor of improvement over the host-based GB barrier.

One possible reason why the factor of improvement for the GB algorithm is not as large as that for the PE algorithm is that in the broadcast phase of the host-based barrier, the messages sent by the host are pipelined through the NIC, i.e., after the host transfers a send event to the NIC, it is free to transfer the next send event while the NIC is processing the first one. So part of the overall send time of one message is overlapped with that of the subsequent message. There is no such overlapping in the PE algorithm because the host must wait to receive a message before sending the next one.

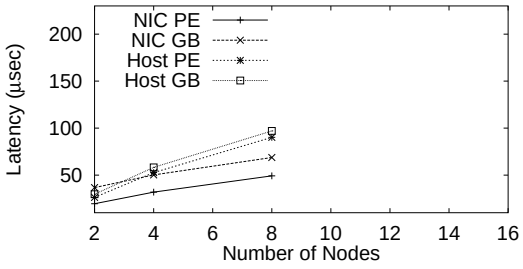
We also ran similar tests using the LANai 7.2 cards. Because we only have eight of these cards, we show the results for up to only eight nodes. Figure 3.6(c) shows the barrier latencies for NIC-based and host-based barriers for each algorithm using these cards. Notice that the faster NIC processor improved the performance of all implementations. With the faster NICs the NIC-based barrier using the PE algorithm performed a barrier in 49.25µs compared to 90.24µs for the host-based PE barrier for eight nodes.



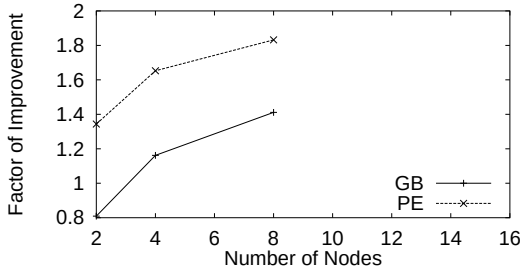
(a) Latency LANai 4.3



(b) Factor of Improvement LANai 4.3



(c) Latency LANai 7.2



(d) Factor of Improvement LANai 7.2

Figure 3.6: Comparison of NIC-based barrier and host-based barrier for two algorith ms (PE and GB) using the LANai 7.2 and LANai 4.3 NICs

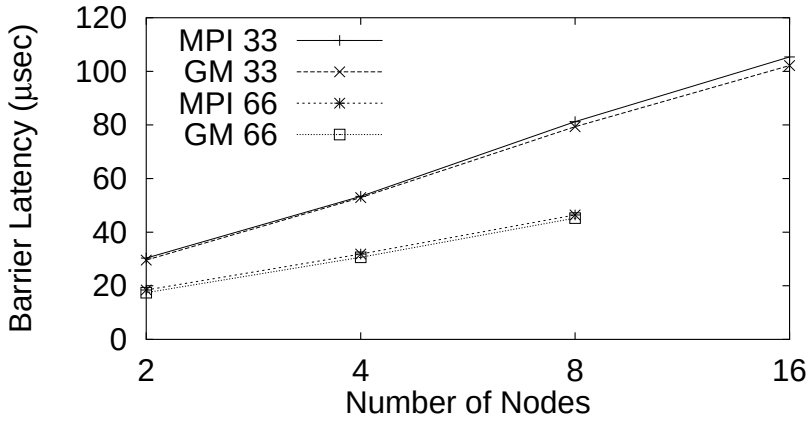


Figure 3.7: GM barrier latencies and MPI barrier latencies of NIC-based barriers using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs

Figure 3.6(d) shows the factor of improvement of the NIC-based barrier over the host-based barrier for both algorithms using the LANai 7.2 cards. This shows a 1.83 factor of improvement of the NIC-based barrier over the host-based barrier using the PE algorithm for eight-nodes. This is a greater factor of improvement than we saw for the LANai 4.3 cards for eight nodes which was 1.66.

3.5.2 MPI-Level Overhead

To evaluate the MPI overhead, we compared the latencies of the barrier operations at the GM level and at the MPI level. To determine the latency of the barriers, we performed 10,000 consecutive barriers and took the average latency of each barrier at each node. Figure 3.7 shows the results of these experiments using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs. When using the 33MHz NICs there was a 3.22µs overhead for the 16 node NIC-based barrier. For the 66MHz NICs, there was only a 1.16µs overhead for the eight node NIC-based barrier. Note that the overhead of MPI operation to initiate a NIC-based barrier does grow slightly with the number of nodes. For the pairwise-exchange algorithm that we are using, it grows at a rate of $\log_2 N$, where N is the number of nodes participating in the barrier. By taking this into account, it can be observed that our MPI-level barrier implementation to use the GM-level NIC-based barrier is extremely efficient.

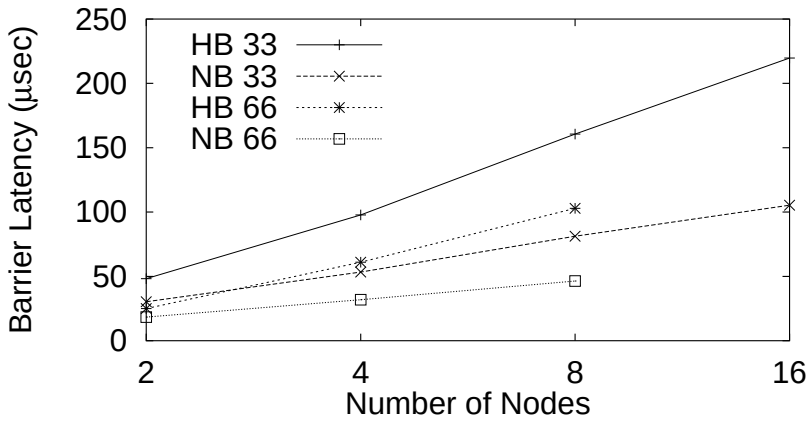
3.5.3 MPI-Level Performance and Scalability

To evaluate the performance of NIC-based barriers at the MPI-level, we performed 10,000 consecutive barriers using the `MPI_Barrier()` function, and took the average latency of each barrier at each node. These experiments were performed for host-based and NIC-based barriers using both LANai 4.3 NICs and LANai 7.2 NICs. Figures 3.8(a) and 3.8(b) show the results of these experiments for power-of-two numbers of nodes. Figure 3.8(a) shows a latency of 105.37µs for the NIC-based barrier (NB) compared to 216.70µs for the host-based (HB) barrier using the 33MHz LANai 4.3 NICs for a 16 node barrier. Similarly, a barrier latency of 46.41µs is observed for the NIC-based barrier using the 66MHz LANai 7.2 NICs for an eight node barrier, compared to 102.86µs for the host-based barrier.

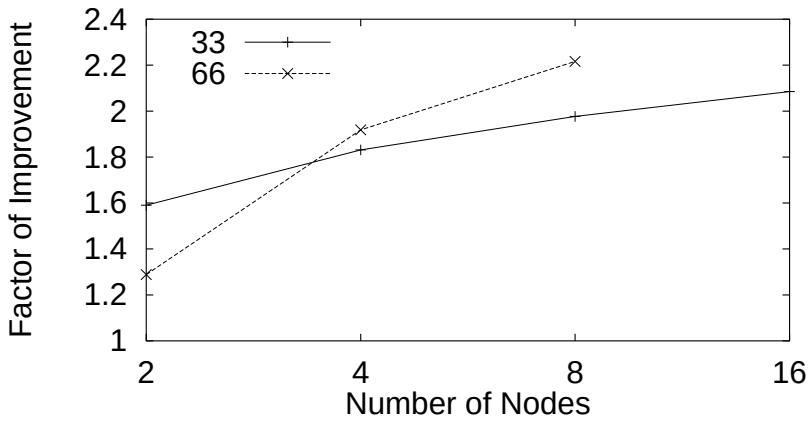
Figure 3.8(b) shows the factors of improvement for the NIC-based barrier over the host-based barrier. Notice that the NIC-based barrier delivered a 2.09 factor of improvement for 16 nodes using the LANai 4.3 NICs and a 2.22 factor of improvement for 8 nodes using the LANai 7.2 NICs. Notice also that for both NICs the factor of improvement increases with system size. This indicates that the NIC-based barrier scales better than the host-based barrier. Figure 3.9(a) shows the barrier latency for all (including non-power of two) nodes and Figure 3.9(b) the factor of improvement of the NIC-based barrier versus the host-based barrier for these nodes. From these graphs we see that even with non-power-of-two numbers of nodes, the NIC-based barrier scales better than the host-based barrier. Notice that, in some cases, the latency of performing a barrier with a non-power-of-two number of nodes is greater than the latency of performing a barrier with a greater power-of-two number of nodes (e.g., 7 nodes v.s. 8 nodes for NIC-based using the LANai 4.3). This is because for a barrier with a non-power-of-two number of nodes, two extra steps must be taken to send and receive from the nodes in set S', as described in Section 3.3.

3.5.4 Granularity of Computation

To examine the effects of NIC-based barrier on granularity of computation, we performed 10,000 loops of computation at the MPI-level followed by an MPI-level barrier. We varied the length of the computation to simulate different levels of granularity. In Figure 3.10, we varied the length of computation from 1.50µs to 129.75µs to examine the effects of NIC-based barrier for very fine levels of granularity. Figure 3.10 shows the average execution time (computation time and barrier time) per loop as the computation time varies. Results are presented for both NIC-based (NB) and host-based (HB) barriers on eight nodes using 33MHz LANai 4.3 (33) and 66MHz LANai 7.2 (66) NICs. Notice that for the host-based barriers, we see a flat spot where the execution time does not increase much for computation time per loop going up to around 17µs for the LANai 4.3 NICs and around 8µs for the LANai 7.2 NICs. This

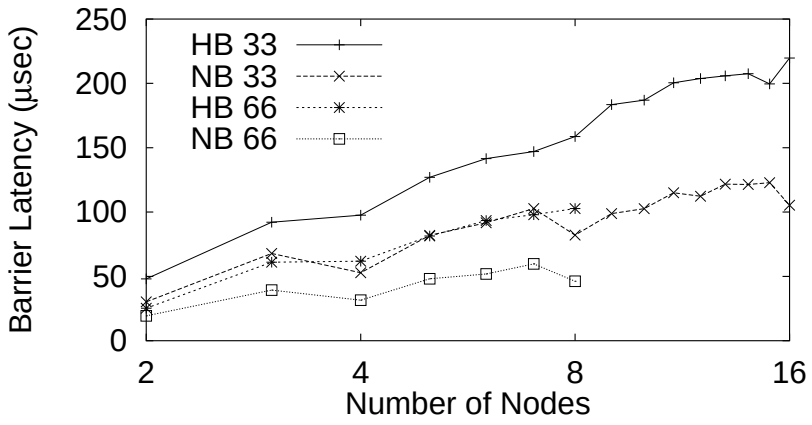


(a) Latency

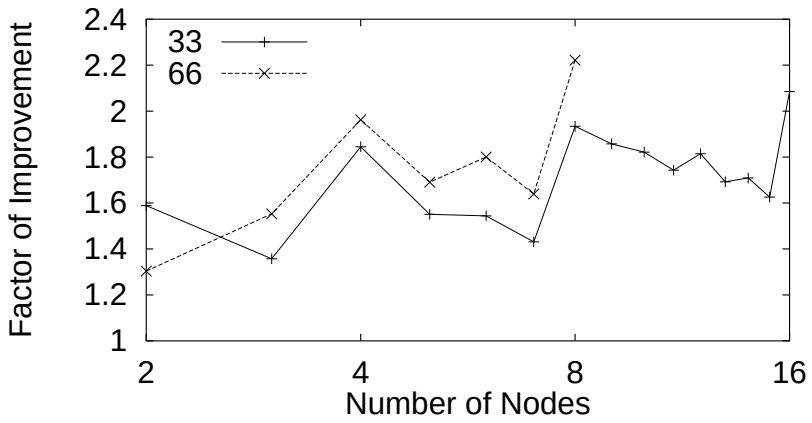


(b) Factor of improvement

Figure 3.8: Performance of NIC-based barrier versus host-based barrier using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs



(a) Latency



(b) Factor of improvement

Figure 3.9: Performance of NIC-based barrier versus host-based barrier using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs for all number of nodes

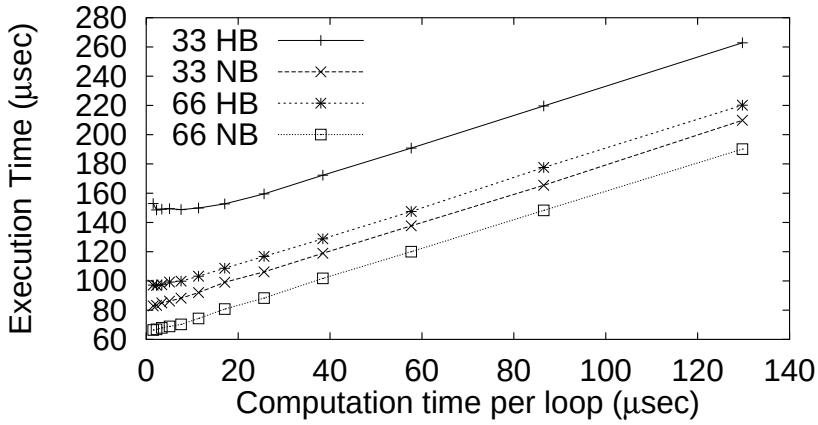
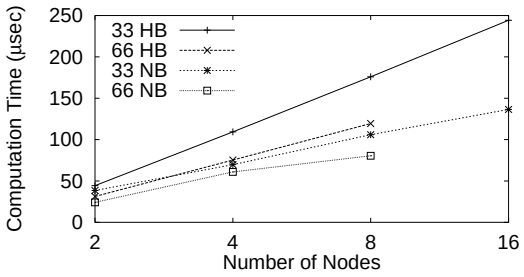


Figure 3.10: Average execution time (compute time and barrier time) per loop for host- and NIC-based barrier on eight nodes using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs

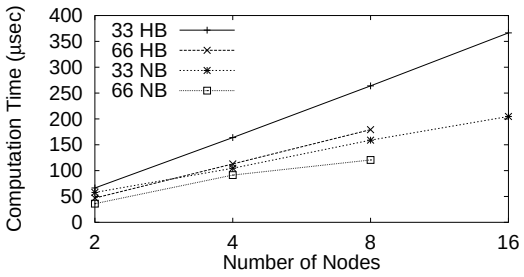
is due to to fact that when the host sends the last message of a barrier and completes the barrier, the NIC may still be transferring the message from the host to the transmit buffer or transmitting the message when the next barrier call is made. The next barrier will send a message which must be delayed until the NIC has finished the previous message. We don't see the same effect with the NIC-based barrier because the NIC does not notify the host that the barrier has completed until just before it starts transmitting the last message. By the time the notification reaches the host and the host initiates the next barrier, the message will have been transmitted.

To compare the granularity of computation possible using NIC-based barriers versus using host-based barriers, we plotted graphs which show the minimum computation time required between barriers for a program to have a certain efficiency factor. We assume that the program performs computation followed by a barrier, and performs no other communication. We define our efficiency factor as the ratio of computation time to the total execution time (i.e., computation time and barrier time). Figures 3.11(a) through 3.11(d) show the computation time required to achieve efficiency factors of 0.25, 0.50, 0.75 and 0.90 for both the LANai 4.3 and the LANai 7.2 NICs.

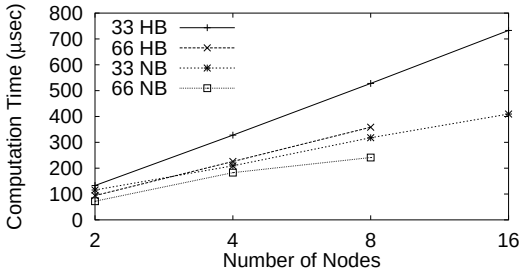
We see from these figures that the minimum computation time for a particular efficiency factor when using NIC-based barriers is less than that when using host-based barriers. For instance, Figure 3.11(b) shows the graph for a 0.50 efficiency factor.



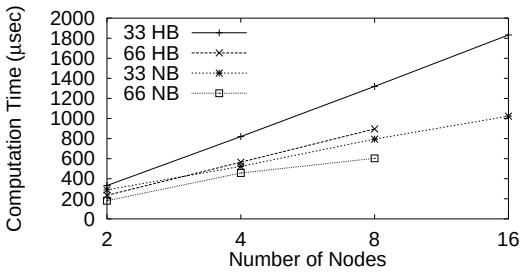
(a) 0.25 efficiency factor



(b) 0.50 efficiency factor



(c) 0.75 efficiency factor



(d) 0.90 efficiency factor

Figure 3.11: Computation time required to achieve a particular efficiency factor using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs

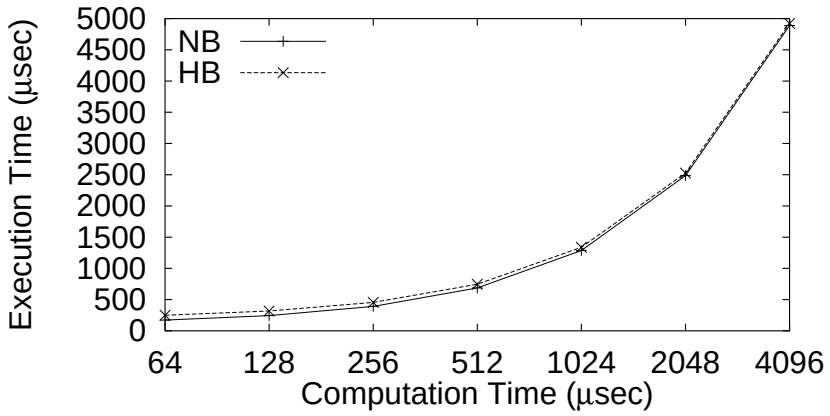


Figure 3.12: Total time of computation, varying at each node by 20%, followed by a barrier for NIC-based and host-based barriers over 16 nodes using 33MHz LANai 4.3 NICs.

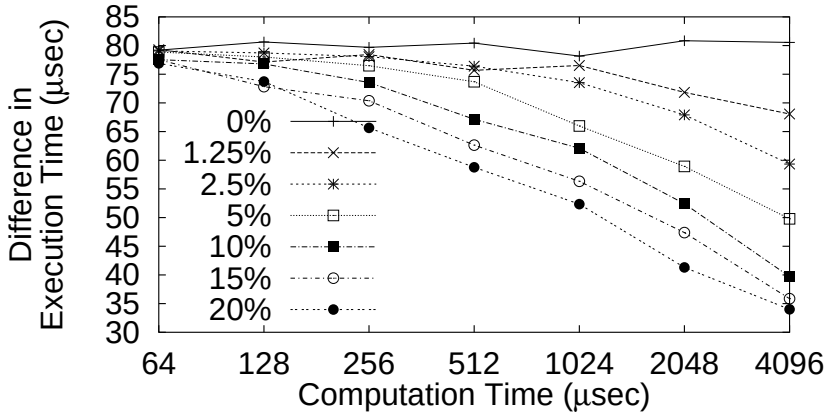
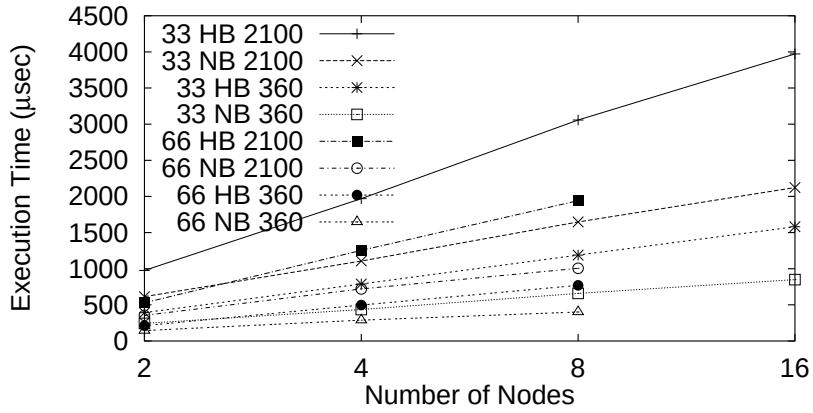
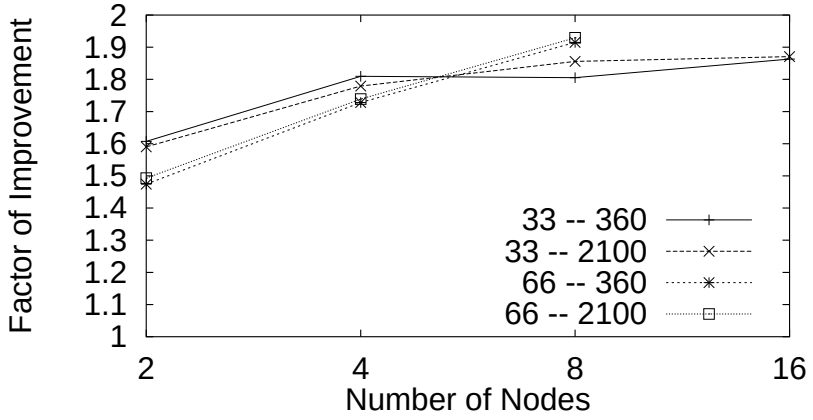


Figure 3.13: Difference in execution time between using host- and NIC-based barriers performing computation (± percentage) followed by a barrier (16 nodes; 33MHz LANai 4.3 NICs).



(a) Execution time



(b) Factor of Improvement

Figure 3.14: Performance of synthetic benchmarks (total computation time of 360µs and 2,100µs) for host-based and NIC-based barriers using 33MHz LANai 4.3 and 66MHz LANai 7.2 NICs

The minimum computation time for 16 nodes is 366.40µs for the host-based barrier and 204.76µs for the NIC-based barrier using the LANai 4.3 NICs. For the LANai 7.2 NICs over eight nodes, the host-based barrier requires 179.18µs of computation between barriers to maintain the efficiency factor, while the NIC-based barrier needs only 120.62µs of computation. For a 0.90 factor of efficiency, Figure 3.11(d) shows that using the LANai 4.3, the host-based barrier requires 1,831.98µs, as compared to 1,023.82µs for the NIC-based barrier to maintain the efficiency factor. Using the LANai 7.2 NIC, the host-based barrier needs 895.91µs of computation time while the NIC-based barrier needs only 603.11µs for 0.90 factor of efficiency. From these results we can see that finer granularity programs can be written using NIC-based barrier without losing efficiency.

3.5.5 Varying Arrival Times

In real applications, the nodes participating in a barrier do not always reach the barrier at the same time. Often some nodes reach the barrier before others. To examine the effects of varying arrival times on barrier performance, we performed 10,000 loops of computation followed by a barrier. The length of computation at each node was varied by a percentage of the mean in both directions from the mean (e.g., 4096µs ±20%). Figure 3.12 shows the execution time of this benchmark for 16 nodes with the computation time varying from 64µs to 4096 µs with a 20% variation in the computation time using LANai 4.3 NICs. Notice that the difference in the execution time of the benchmarks using NIC-based barriers and host-based barriers gets smaller as the computation time gets larger. This is because the total variation in the arrival times gets larger. Figure 3.13 shows the difference between the benchmarks using host-based and NIC-based barriers over 16 nodes using LANai 4.3 NICs. Notice that the difference gets smaller as the computation time and percent variation increases, i.e., as the total variation in arrival time increases. Notice also, that for 0% variation the difference does not decrease. This indicates that the amount of computational delay itself does not affect the difference in execution time, but rather it is the total amount of variation that affects the difference. So, as the variation in arrival times increases, the execution time of the benchmark using NIC-based barriers increases slightly faster than that for the benchmark using host-based barriers. This indicates that the host-based barrier is not as sensitive to a variation in arrival time as the NIC-based barrier. Even though the NIC-based barrier is more sensitive to the variation in arrival time than the host-based barrier, it always performs better than the host-based barrier.

3.5.6 Synthetic Application Performance

In order to evaluate how the barrier performance would affect an application, we ran two synthetic MPI-level applications. The synthetic applications consist of several steps each of which consists of computation followed by a barrier. The mean computation time varies from one step to the next. Within each step, the computation time varies randomly from one node to the next by ±10% from the mean. The execution time of each synthetic application was taken over 10,000 runs. The first application was designed such that it performed eight steps and had computation times of 10, 20, 30, ... 80µs, for the respective steps, for a total of 360µs of computation. The second application had 20 steps and had computation times of 10, 20, 30, ... 200µs, for a total of 2,100µs of computation. Figure 3.14(a) shows these results for host-based (HB) and NIC-based (NB) barriers using 33MHz LANai 4.3 (33) and 66MHz LANai 7.2 (66) NICs. Notice that in all cases the NIC-based barrier performs better than the host-based barrier. Figure 3.14(b) shows the factor of improvement for the applications using NIC-based barriers versus the applications using host-based barriers. Using the LANai 4.3 NICs, we see a 1.86 factor of improvement in the 360µs application and a 1.87 factor of improvement in the 2,100µs application, for 16 nodes. Using the LANai 7.2 NICs, which have faster processors, we see a 1.92 factor of improvement for the 360µs application and a 1.93 factor of improvement for the 2,100µs application over only eight nodes. These results indicate that a NIC-level barrier implementation on large clusters using NICs with faster processors can deliver very good performance benefits at the application level.

3.6 Summary

In this chapter, we presented our design, implementation and evaluation of NIC-based barrier. We implemented the NIC-based barrier in GM and showed that it performed better than the host-based barrier. We then modified MPICH-GM version 1.2..3, to use the NIC-based barrier. This was done in an efficient manner which added only 3.22µs overhead to the GM implementation of the NIC-based barrier over 16 nodes using the 33MHz LANai 4.3 NICs. When comparing the performance of the barriers at the MPI level, we found a 2.09 factor of improvement for 16 nodes using the LANai 4.3 NICs and a 2.22 factor of improvement for 8 nodes using the 66 MHz LANai 7.2 NICs. Furthermore, the factor of improvement increased with the number of participating nodes. This indicates that the NIC-based barrier scales better than the host-based barrier.

We also evaluated the impact of the NIC-based barrier on the granularity of computation. We found that for a program to have a 0.90 factor of efficiency using the LANai 4.3 NICs, at least 1831.98µs of computation must be performed per barrier if the host-based barriers are used, but only 1023.82µs if a NIC-based barrier is used. This value is 44% lower than for the host-based barrier. So, using the NIC-based

barrier allows for finer grained programs without lowering the efficiency. We noticed that the NIC-based barrier is more sensitive to variation in arrival times than the host-based barrier. However, the NIC-based barrier always performed better than the host-based barrier. To evaluate the impact of using the NIC-based barrier on applications, we used synthetic applications. We found up to a 1.93 factor of improvement when using NIC-based barriers versus using a host-based barrier. This indicates that using the NIC-based barrier in applications which perform many barrier calls will deliver significant performance benefits.

CHAPTER 4

NIC-BASED REDUCTION

Reduction-to-one and reduction-to-all operations are common operations in parallel and distributed systems. These collective operations can involve many processes. It is therefore important to make these operations fast and efficient. Research has previously been done to make these operations efficient in MPP environments by taking advantage of the particular characteristics of the underlying architecture [21, 45]. However, clusters lack the communication assists which could be used to implement these operations efficiently. We have shown how programmable NICs can be used in cluster environments to support reduction operations. A NIC-based reduction implementation would not only provide an efficient reduction operation, but would also provide significant potential for overlap of computation with communication. In this chapter we explore the benefits of NIC-based support for reduction operations for integer and floating-point operations. It is worthwhile to note that a large fraction of reduction operations are performed using small data sizes of just a few elements [27]. This means that a specialized reduction operation which can efficiently perform the operation on a small number of elements would be useful.

Using NIC-based reduction can significantly reduce host CPU utilization. This is because the NIC processor is performing the operation rather than the host processor. Another benefit of NIC-based reduction is that a parallel program using these operations is less sensitive to process skew. Processes of a parallel program can become unsynchronized, or skewed, during the course of running the application. This can happen as a result of unbalanced or asymmetric code, through random, unpredictable causes such as a process being context switched, or because the processes may not have been started at the same time. Such skew can have a significant impact on the performance of a parallel program when host-based collective communications operations, such as reduction are used. In a reduction operation, at a particular node, data from certain nodes must be received before the arithmetic operation can be performed and the result can be forwarded to other nodes. If this reduction operation is implemented at the application-level then upon calling the reduction function, the process waits to receive all of the messages, performs the operation, and sends the result on. This means that if a process is delayed and hasn't performed the reduction

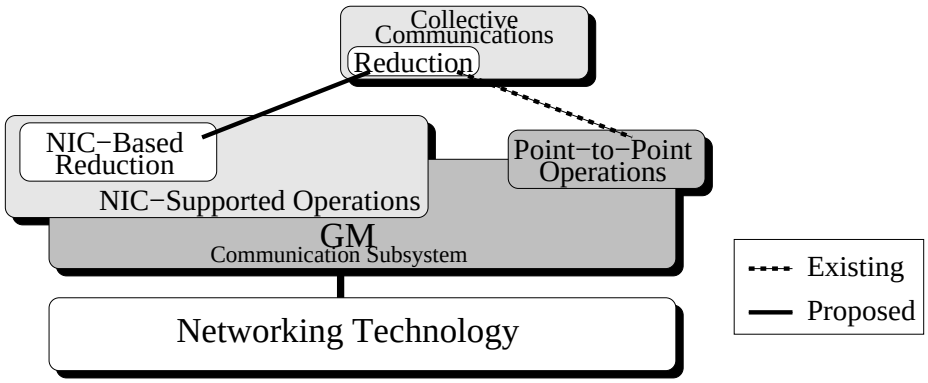


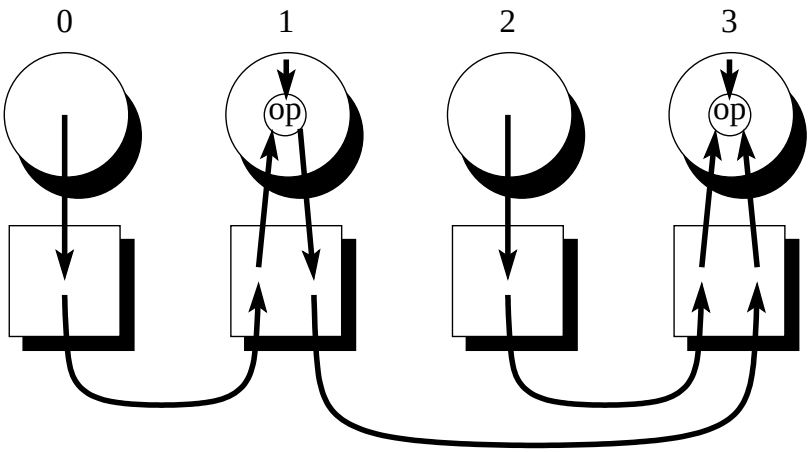
Figure 4.1: NIC-based reduction for the GM communication subsystem

operation, then another process may also be delayed waiting for that message, and so cannot continue with useful computation. However, because the NIC-based reduction operation is performed by the NIC, and not the host, once the process passes its data to the NIC, it can continue with its computation, and will not be stalled due to a delayed process.

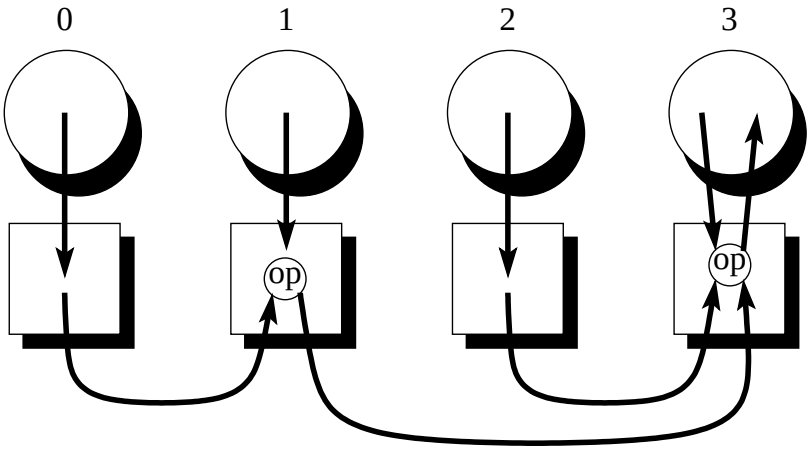
We have implemented a NIC-based reduction-to-one operation to perform integer and floating-point operations on single 64 bit elements. In this chapter we describe the design and implementation of this operation as well as the evaluation of the implementation. Our implementation achieves a 1.19 factor of improvement over the traditional host-based implementation when performing integer operations on a 16 node system. We show that NIC-based reduction would be even more beneficial for larger system sizes. Our evaluation also shows that using NIC-based reduction can reduce host CPU utilization with a factor of improvement of 2.7, and can reduce the effects of process skew with a factor of improvement of up to 4.5.

Figure 4.1 illustrates our approach to adding NIC-based reduction to the GM communication subsystem. The dotted line indicates how the redution operation is traditionally implemented using point-to-point messages. The solid line shows our implentation using the NIC-based reduction operation.

The rest of this chapter is organized as follows. In the next section, we describe the general concept of a NIC-based reduction operation. In Section 4.2 we describe our design and implementation, followed by the evaluation of our implementation in Section 4.3. Finally we summarize our work in Section 4.4.



(a) Host-based



(b) NIC-based

Figure 4.2: Block diagrams of Host-based and NIC-based reductions across four nodes. The circles represent the host processor of a node and squares represent the NIC of a node.

4.1 NIC-Based Reduction

Before we describe the general concept of NIC-based reduction, we will briefly describe traditional host-based reduction. In traditional host-based reduction in a message-passing system, messages are passed between processes running at the host, and the arithmetic operations are performed by the host processors. When the operation is complete, the result of the operation will be located at one of the processes. Processes participating in the reduction operation are organized in a logical tree. Each process receives reduction messages from its children, which contain a partial result from the subtree of that child. Next, each process performs the arithmetic operation on its data and the partial results received from its children. The process then sends this result to its parent. Figure 4.2(a) shows a block diagram of a host-based reduction operation across four nodes. Node 0 sends its data to Node 1. When Node 1 receives this message it performs the arithmetic operation on the data from Node 0 and its data. Node 1 then sends this result to Node 3. Node 3 receives data from Node 2 and Node 1, and performs the arithmetic operation on its own data and the data sent by Nodes 1 and 2.

In a NIC-based reduction, each process sends its data to the NIC. The NIC will then wait for the messages from its children, perform the arithmetic operation, and either send a message to its parent, or if this node is the root of the tree and has no parent, it will forward the result to the host. Figure 4.2(b) shows a block diagram of a NIC-based reduction operation across four nodes. Here we see each process sending its data to the NIC. The NICs at Nodes 0 and 2 immediately forward their data to their parents, since they are leaf nodes. The NIC at Node 1 receives the message from Node 0 and performs the arithmetic operation on this data and the data sent from the host. It then sends this result to the NIC at Node 3. The NIC at Node 3 receives the messages from the NICs at Nodes 1 and 2, performs the arithmetic operation on this data and on the data sent from the host, then forwards this result to the host.

Notice that in the host-based reduction, messages received at intermediate nodes, such as Node 2, are received by the NIC, forwarded to the host, which performs the arithmetic operation and sends another message that is sent down to the NIC to be transmitted. In the NIC-based case, because the arithmetic operation is performed at the NIC, such messages do not have to be passed between the NIC and the host. Only the initial data needs to be passed from the host to the NIC. This can improve the performance of the reduction operation.

Another potential benefit of NIC-based reduction is reduced host involvement in the operation. For non-root nodes, once the host has sent its data to the NIC, it no longer has to be involved in the reduction operation. In the host-based case, the host process must either wait to receive the messages from its children, or it must be interrupted when the messages arrive so that it can perform the arithmetic operation

on them. Since interrupts are time consuming operations, using these can lead to poor reduction latency, and is not commonly used for reduction operations.

However, the alternative of waiting for the messages has its own drawbacks. If processes are skewed, meaning that some processes are performing the reduction operation while others are lagging behind and have not yet started the operation, then intermediate processes may be waiting for other processes in their subtree to catch up. This can lead to poor overall application performance. By using NIC-based reduction the host needs only to supply the data to the NIC. It can then proceed on with other useful computation. This allows greater overlap of computation and communication operations. Furthermore, because the host is not involved in actually performing the operation, NIC-based reduction is a non-blocking operation. The root process need not wait idle for the result after it sends its data to the NIC. It can proceed with other computation and only get the result from the NIC when it needs it. This can further reduce host involvement.

4.2 Design and Implementation

We implemented our NIC-based reduction operation as a modification to the GM message passing system [34] which uses the Myrinet network [10], a popular system area network for clusters. Before we describe the design and implementation of our NIC-based reduction, we will give some background on GM and Myrinet.

Myrinet is a high-performance full-duplex 2Gbps network which uses NICs with programmable processors. GM is a user-level message passing system which uses the programmable NICs for much of the protocol processing. GM consists of three components: a kernel module, a user-level library, and a control program which runs on the NIC processor. When a user application wishes to send a message it calls the appropriate function from the library. This function constructs a *send descriptor* which describes what data is to be sent and to which process to sent it to. This descriptor is then written to the NIC using PIO. The NIC detects that a new descriptor has been written and processes it, DMAing the data from the host buffers and transmitting the message. In order to receive a message, the process must provide memory buffers in host memory into which the NIC will DMA the message data. This is done by sending the NIC a *receive descriptor* which describes such a buffer. When the NIC receives a message it DMAs the data into one of the buffers, then DMAs a notification to the host process that a message has been received. The host process can either poll for these notifications, or can block while waiting. In the latter case the NIC will signal an interrupt after it DMAs the notification.

We implemented a NIC-based reduction operation by modifying GM version 1.6.3. Our implementation can perform binary *AND* and *OR* operations, as well as integer and floating point *SUM*, *MIN* and *MAX* operations on a single 64 bit data element. The host process passes a descriptor to the NIC describing the reduction operation.

As the NIC receives reduction messages from the network, it performs the arithmetic operation on the data and stores the result. Once messages from all of the children have been received and processed, if the process initiating the reduction operation is the root, the NIC DMAs a notification to the host, including the result, indicating that the reduction has completed, otherwise, the NIC transmits the result to the parent NIC.

There are several design issues in this implementation, namely, dealing with unexpected messages, dealing with multiple instances of the reduction operation, generating and specifying the tree structure, and performing floating point operations at the NIC. In the rest of this section we will discuss these issues.

4.2.1 Unexpected Messages

Because processes are not always synchronized, it is possible that some processes may execute the reduction operation before others. This means that a NIC may receive reduction messages from other NICs before the host process has initiated the reduction operation and send its data. Since the host has not informed the NIC which processes to expect data from, and what arithmetic operation to perform, the NIC cannot process the messages. Such a message can be handled in one of two ways. One option is to reject the message and request that the sender retransmit it later. Another option is for the NIC to store the data until the host has initiated the corresponding reduction operation. The first option can lead to high latency because the messages need to be retransmitted after a delay. While the second option gives better performance, it requires NIC memory to be allocated for storing this data. Since NIC memory is limited, this may limit the number of messages that can be stored.

We used a hybrid approach where we provided a limited number of buffers to store unexpected data, and reject messages once these are full. When the NIC receives a descriptor from the host for a reduction operation, it checks the list of unexpected messages. If it finds any unexpected messages that match, it performs the operation on that data, and frees that unexpected message buffer.

4.2.2 Multiple Instances of the Reduction Operation

When a non-root process initiates a reduction operation, after it sends the data to the NIC, it can proceed with its computation. This means that a process can initiate a second reduction operation before the NIC has completed the first. The NIC needs to be able to process multiple instances of the operations in the correct order. We did this by keeping a queue of instances of reduction operations for each host process. When a reduction message is received from the network for a particular process, the NIC searches the list of instances for that process looking for a matching instance. If a matching instance is found, the arithmetic operation is performed for

that instance, otherwise the message is considered an unexpected message and is handled as described above.

4.2.3 Generating and Specifying the Tree Structure

The tree structure can be generated by either the NIC or the host process. However, because NIC processors are typically much slower than host processors, it would be more efficient to have the host construct the tree and pass a list of children and the parent to the NIC. We used this option. The send descriptor was only 64 bytes so we are limited as to the number of children that can be specified. Four bytes are needed for each child: two bytes are needed to specify a GM node, one byte is needed to specify the GM *port*, and one byte is used in the algorithm to indicate whether a reduction message has been received from this process. Since we also include the eight-byte data in the descriptor, and 12 more bytes are used in the descriptor for other fields, there is only room to specify nine children, and one parent.

The shape of the reduction tree is also an important design issue. The latency of the operation increases with each level of the tree, so a very deep tree may not be desirable. On the other hand, a very shallow tree increases network contention as many child nodes transmit their data to one parent node. The exact shape of the tree depends on the performance characteristics of the reduction operation. We have not fully investigated the optimal tree shape for NIC-based reduction. For our evaluation we used a binomial tree because this is the most common tree used for reduction operations, e.g., MPICH [22] uses a binomial tree.

4.2.4 Performing Floating Point Operations at the NIC

The Myrinet NIC processors do not have floating point units. So in order to be able to perform floating point operations, we had to use floating point operations implemented in software. We used the SoftFloat [24] library for these operations. SoftFloat is a free software implementation of the IEC/IEEE Standard for Binary Floating-point Arithmetic, and supports all functions dictated by the standard for 32, 64, and 128 bit floating point formats. We used only the 64 bit format in our implementation.

4.3 Experimental Results

In this section, we evaluate our implementation on a cluster of 16 quad-SMP 700MHz Pentium-III nodes with 66MHz/64bit PCI. The nodes are connected to a Myrinet2000 network. The NICs are PCI64B cards with 2MB of memory and 133MHz LANai 9.1 processors. These are connected to 16 ports of a 32 port switch. We compare our NIC-based reduction implementation, which is based GM version 1.6.3,

to a host-based reduction implementation using the same version of GM. We evaluate the basic reduction operation, the host CPU utilization of the reduction operation, and its tolerance of process skew.

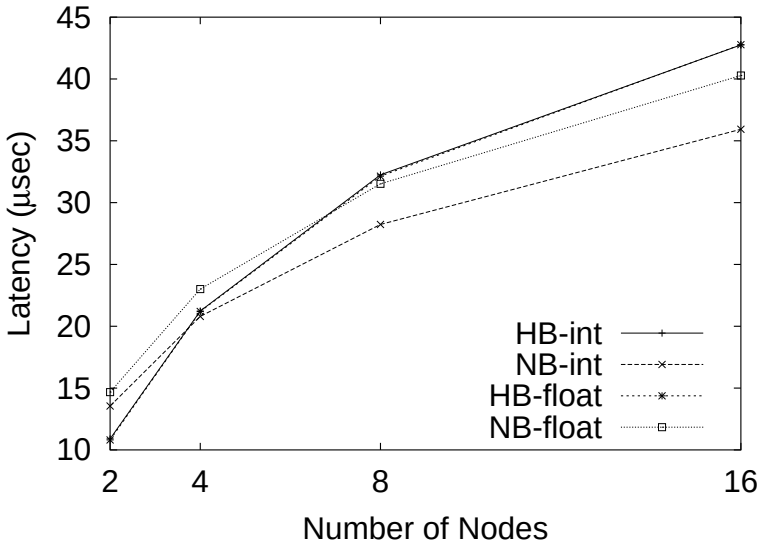
4.3.1 Basic Reduction

To evaluate the performance of our reduction implementation, we compare the time from when the leaf node furthest away from the root initiates the operation until the root node receives the result. We performed the test in the following manner. All of the nodes perform the reduction operation. As soon as the root node completes the operation and receives the result, it sends a message to the last leaf node of the tree. Once this node receives the message it takes the time between when it initiated the reduction operation and when it received the message, then subtracts off the one way latency time. We take the average time over 10,000 iterations.

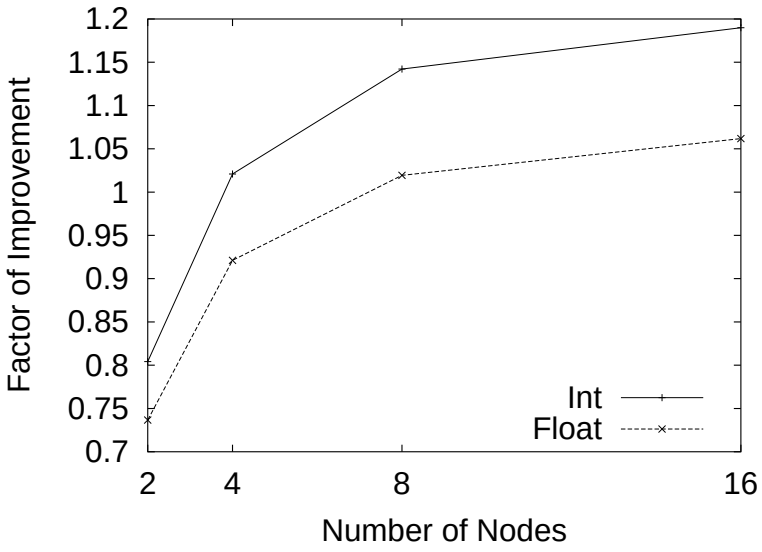
We performed the evaluation for 2, 4, 8 and 16 nodes using integer operations and floating-point operations. Figure 4.3 shows the results of this evaluation. The figures show the results for the integer and floating point SUM operation. Notice also that the results for the host based floating point and integer operations were very similar, so the two lines on the graph are on top of one another. The graphs show that for integer operations, NIC-based reduction performs better than host-based for all but the two node case. We see that the floating-point operations add some overhead, but that NIC-based reduction is still better than the host based reduction for all but the two and four node cases. We see up to a 1.19 factor of improvement for the integer operation, and up to a 1.06 factor of improvement for floating point operations.

4.3.2 Larger System Sizes

The factor of improvement for NIC-based reduction increases with the number of nodes. This indicates that for larger system sizes, the NIC-based reduction operation may be even more beneficial. In order to investigate how the relative performance of NIC-based reduction would change with an increase in system sizes we compared the performance of the operations using a 1-degree tree, in other words a chain, and varied the depth of the tree. Figure 4.4 show the results of this comparison. Notice again in this graph that the lines for host-based floating-point and integer operations overlap. This graphs shows us that as the depth of the tree increases the latency of the host-based operation increases faster than the NIC-based operation. In fact the time for host-based integer reduction increases at a rate of 3.70µs per level of depth faster than that for NIC-based integer reduction. Similarly, the latency of host-based floating-point reduction increases at a rate of 2.64µs per level of depth faster than that of NIC-based floating-point reduction. We see that for a tree of depth 1 host-based reductions perform better than NIC-based reductions. Similarly, host-based floating-point reduction performs better than NIC-based floating-point



(a) Latency



(b) Factor of Improvement

Figure 4.3: Comparison of NIC-based reduction (NB) and host-based reduction (HB) for integer (int) and floating-point (float) operations

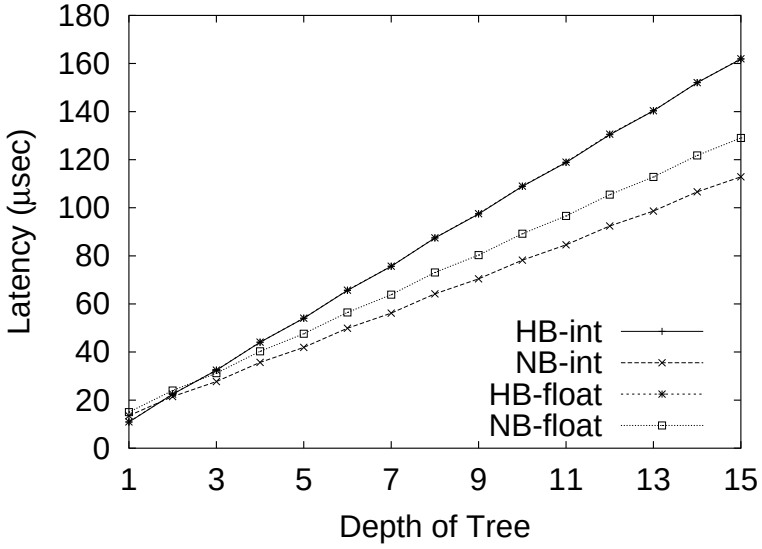


Figure 4.4: Latency of NIC-based reduction (NB) and host-based reduction (HB) for integer (int) and floating-point (float) operations using a 1-degree tree (a chain) of varying depth

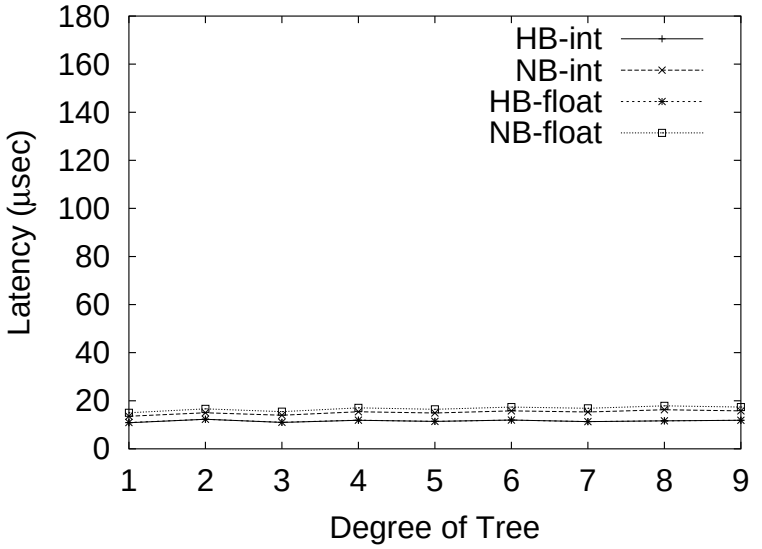


Figure 4.5: Latency of NIC-based reduction (NB) and host-based reduction (HB) for integer (int) and floating-point (float) operations using trees of depth 1 with varying degree

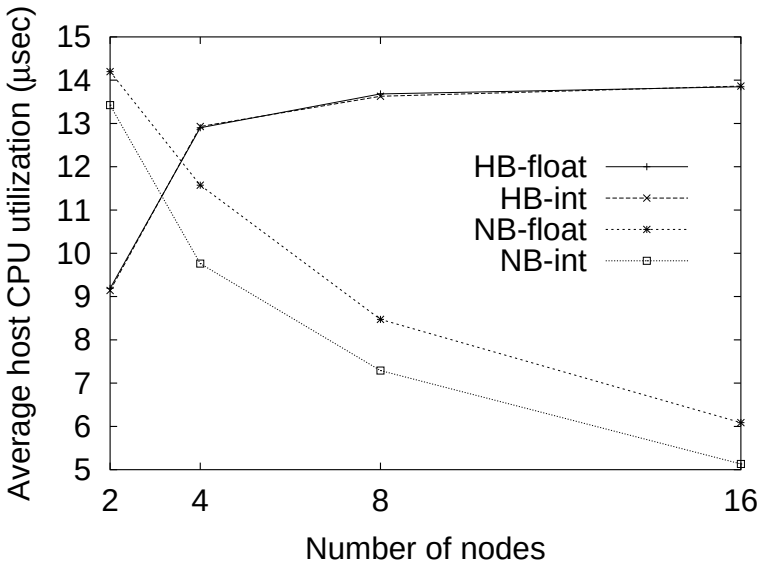
reduction for the tree of depth 2. We believe that this is because of the overhead of the more complicated operation at the slower NIC processor. As the depth increases, the number of times messages have to be sent between the NIC and the host at intermediate nodes increases in host-based reduction. Since NIC-based reduction avoids this overhead, it performs better for deeper trees.

As system sizes increase, and trees get larger, the maximum degree of the tree also increases. To study the effect of increasing the degree of a tree, we compared the latency of NIC-based and host-based reduction operations for trees of depth 1 with varying degree. Figure 4.5 shows the results of this test. Again the host-based floating-point and integer lines overlap. We see here that host-based reductions perform better than NIC-based for any of the trees. However host-based integer reduction performs only about 2.19µs better, and host-based floating-point reduction performs only 3.63µs better. Furthermore, while there is a slight increase in overhead for NIC-based reductions as the degree of the tree increases, it is quite small, 0.22µs per degree for the integer reduction, and 0.24µs per degree for the floating-point reduction. For trees such as binomial trees the depth of the tree increases at the same rate as the depth of the tree. This indicates that the NIC-based reduction will continue to perform better than the host-based reduction for large system sizes.

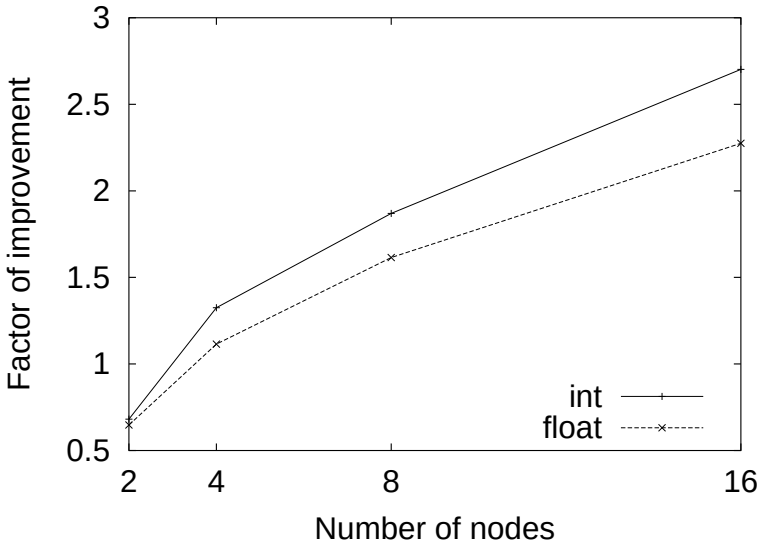
4.3.3 Host CPU Utilization

One of the major benefits of NIC-based reduction is that it reduces the load on the host processor. Once the host process sends its data to the NIC, it is free to perform useful computation. To compare the host CPU utilization for NIC-based and host-based reduction, we timed how long the host process spends performing the reduction at each node. Our test consists of each process performing a barrier synchronization followed by a reduction operation. Figure 4.6 shows the average of 10,000 iterations of this test.

In Figure 4.6(a) we see the average host CPU utilization for NIC-based and host-based reduction for various numbers of nodes. Notice that for all but two node reductions, NIC-based reductions use the host CPU less than host-based reductions. Also note that the average CPU utilization for host-based reduction increases as the number of nodes increases. This is because, in the reduction tree, as the number of nodes increases, there are more interior nodes, which are waiting for reduction messages from their children. In NIC-based reduction, we see that the average host CPU utilization actually decreases as the system size increases. This is because for non-root nodes, the host process simply has to construct the send descriptor and send it to the NIC. So regardless of how many nodes are performing the reduction, the host CPU utilization at non-root nodes is only a few hundred nanoseconds. In this test, the root node waits for the result of the reduction after it sends its data to the NIC, so the CPU utilization for the root node will increase with the latency to



(a) Average host CPU utilization



(b) Factor of Improvement

Figure 4.6: Average time spent by the host processor performing the reduction for host-based (HB) and NIC-based (NB) reductions performing integer (int) and floating-point (float) operations

perform the reduction, as it does for host-based reduction. However, as the number of nodes increases, the CPU utilization at the root increases slower than the number of nodes performing the reduction, so the *average* host CPU utilization decreases with the number of nodes.

Note, that NIC-based reduction allows for a *non-blocking* implementation, where the root process does not wait for the result from the NIC after it sends its data. Instead, the process can go on to perform other useful computation, that does not depend on the result, and it would only read the result from the NIC once it needs that data. This would allow a further reduction in CPU utilization.

Figure 4.6(b) shows the factor of improvement in CPU utilization for NIC-based reduction over host-based reduction. We see a factor of improvement of 2.7 for integer operations and 2.3 for floating point operations. Notice that the factor of improvement increases as the system size increases.

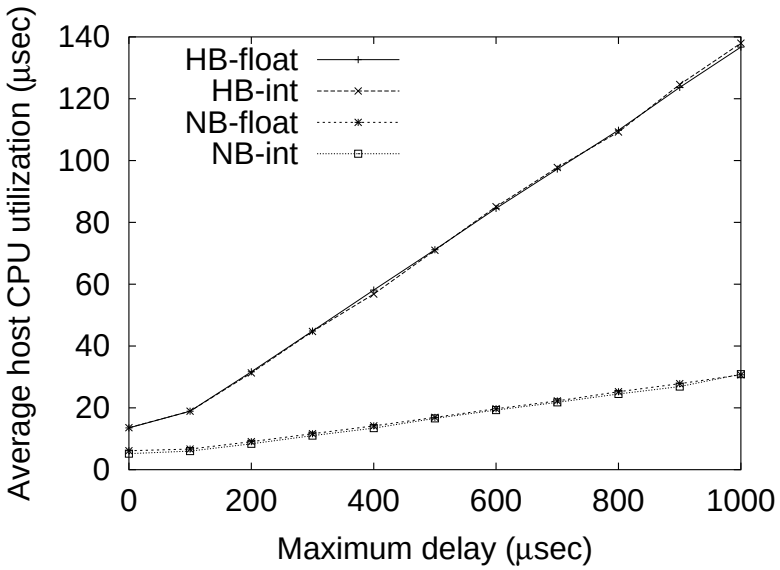
4.3.4 Tolerating Process Skew

Another major benefit of NIC-based reduction over host-based reduction is its tolerance to process skew. In order to see what effect process skew has on host CPU utilization, we timed how long each host process spends performing the reduction operation, while varying the skew between processes.

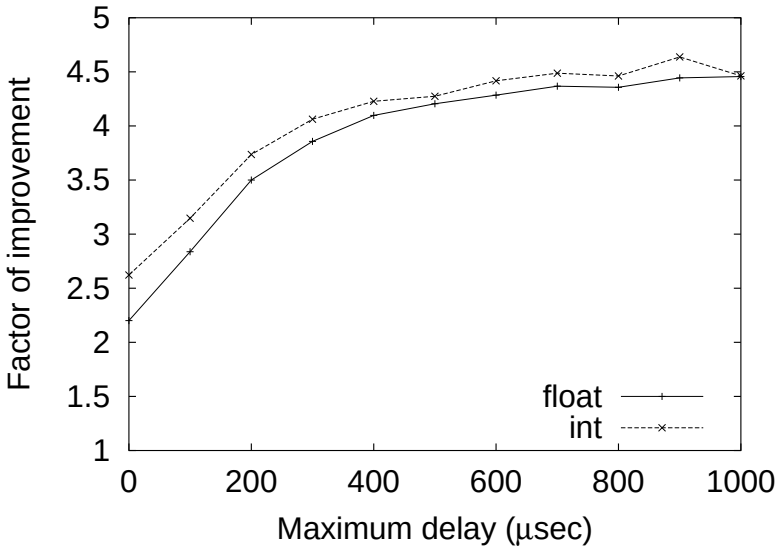
In this test, the processes perform a barrier synchronization, followed by a delay, the length of which is chosen at random between 0 and a maximum delay value. The processes then perform a reduction operation. This is repeated 10,000 times. By varying the maximum delay value, the level of skew between processes varies. As the maximum delay value increases, the skew between processes increases.

The results of this test are shown in Figure 4.7. Figure 4.7(a) shows the average time spent by all of the host processes of 16 nodes performing the reduction, as the maximum delay value is varied. Recall, that in a tree-based reduction, a node must receive all messages from its children, as well as provide its own data, before it can pass the result on to its parent. When a child node or, more generally, any descendant node is delayed, the reduction operation at that node cannot proceed. We would expect, then, that as process skew increases, the number of descendants of a process which are delayed increases, as well as the amount by which they are delayed. For host-based reduction, where the host processes must wait for messages from child processes, process skew results in processes that are stalled and cannot perform useful computation.

We see, in Figure 4.7(a), that for host-based reduction, as the maximum delay increases, the CPU utilization increases dramatically. However, for NIC-based reductions, only the root node is delayed by skewed processes. Because the reduction operation is performed at the NIC, non-root processes simply have to pass their



(a) Average host CPU utilization



(b) Factor of Improvement

Figure 4.7: Average time spent by the host process performing the reduction, with different levels of process skew for host-based (HB) and NIC-based (NB) reductions performing integer (int) and floating-point (float) operations

data to the NIC. This makes non-root processes unaffected by process skew. In Figure 4.7(a), we see process skew has relatively little effect on the average time host processes spend on reduction.

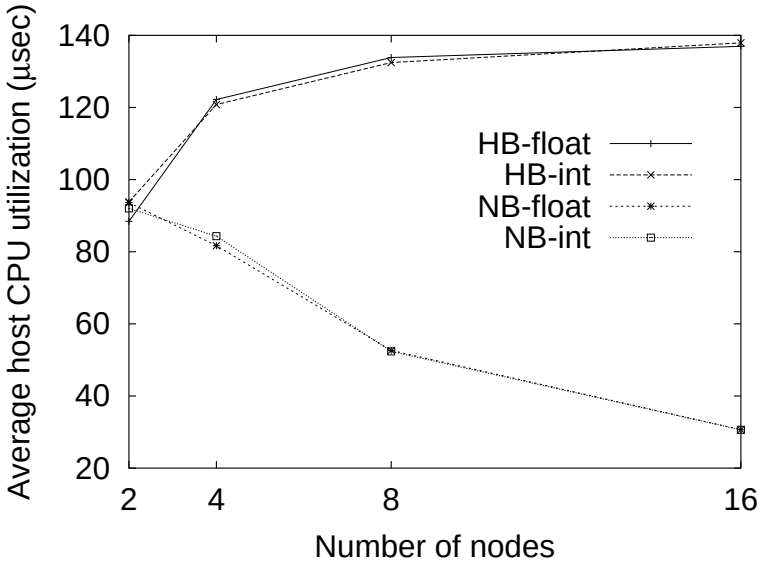
Figure 4.7(b) shows the factor of improvement in CPU utilization for NIC-based reduction over host-based reduction. We see up to a 4.5 factor of improvement for both integer operations and floating point operations. Furthermore we see significant improvements even when process skew is small. For instance we see a 3.7 factor of improvement for integer operations even when the maximum delay value is only 200µs.

We also evaluated the effect of system size on host CPU utilization by fixing the skew and varying the number of nodes performing the reduction. In Figure 4.8(a) we see that as the number of nodes increases, the average host CPU utilization for host-based reduction increases. However, for NIC-based reductions, we see that the average host CPU utilization decreases. This is because in the NIC-based case, only the root process is affected by process skew. As the number of processes increases the time spent by the root process contributes a smaller fraction to the average. For the host-based case, each process can be affected by skew, and as the number of processes increases, there are more processes which can be delayed, relative to their parents and ancestors, so the average host CPU utilization increases as the system size increases. Figure 4.8(b) shows the factor of improvement for NIC-based reduction over host-based reduction. We see up to a 4.5 factor of improvement for both integer operations and floating point operations.

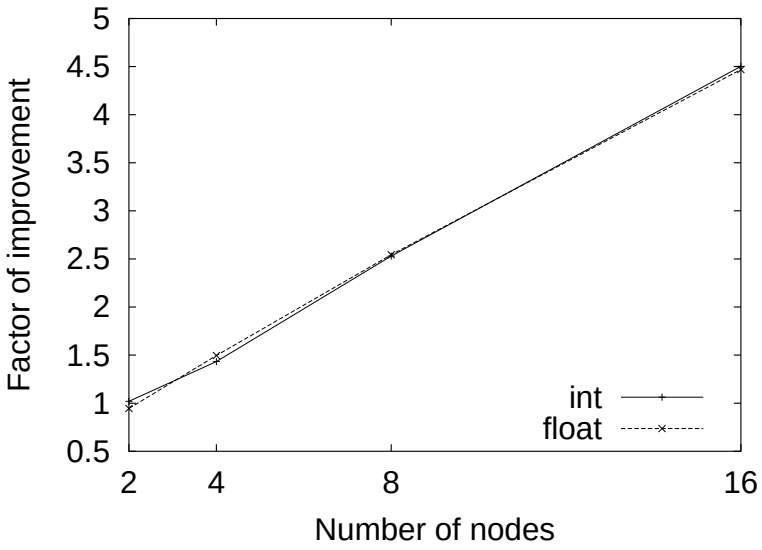
These results indicate that NIC-based reduction operations are much more tolerant to process skew than host-based reduction. Furthermore we see that process skew has a greater impact on host-based reduction as system size increases. This means that using NIC-based reduction can significantly increase the scalability of a system in the presence of process skew.

4.4 Summary

We have presented our implementation of a NIC-based reduction operation, and evaluated it. We found up to a 1.19 factor of improvement for integer reduction and 1.06 factor of improvement for floating-point reduction. We also give evidence that NIC-based reduction will perform better than host-based reduction in larger systems. Though this improvement is not very large, the fact that the operation does not involve the host allows useful computation at the host to be overlapped with the reduction operation at the NIC. In fact, we have shown a 2.7 factor of improvement in CPU utilization when using NIC-based reduction for integer operations and a 2.3 factor of improvement when using floating point operations. We further note that NIC-based reduction can be used in a non-blocking fashion which would further improve the CPU utilization.



(a) Average host time



(b) Factor of Improvement

Figure 4.8: Average time spent by the host process performing the reduction, with a maximum delay value of 1000µs, for different system sizes for host-based (HB) and NIC-based (NB) reductions performing integer (int) and floating-point (float) operations

We have also shown that NIC-based reduction is much more tolerant to process skew than the host-based implementation. In the presence of process skew NIC-based reduction gives a 4.5 factor of improvement in CPU utilization over host-based reduction. We also noticed that when the system size increases, the effect of the skew impacts host-based reduction much more than the NIC-based reduction. This indicates that NIC-based reduction would greatly improve the scalability of certain applications.

CHAPTER 5

NIC-BASED ATOMIC REMOTE MEMORY OPERATIONS

Efficient implementations of synchronization operations such as locks and semaphores is important in parallel and distributed systems. These operations can be efficiently implemented using hardware atomic Read-Modify-Write (RMW) memory operations, such as test&set, compare&swap, etc., on shared memory machines [19]. As clusters are becoming more cost effective and popular, other methods for implementing locks are necessary, since such atomic RMW operations have not been available which operate across nodes of a cluster. Synchronization operations for clusters are typically implemented with *lock manager* process running on one or more nodes which performs the operation. Such a process serves only to handle the synchronization operations and does not directly contribute to the computation. In fact, because it uses computational resources at the node it is running on, it negatively impacts the computation because it reduces useful processor utilization at that node.

By using remote atomic memory operations which are supported by the communication layer, such as those described in the InfiniBand Architecture (IBA) standard [26], locks can be performed without the intervention of the remote host. This means that lock manager processes are not needed, leading to improved processor utilization. Furthermore, because context switches at the host processor are not needed to handle the lock requests, locks implemented using communication layer remote atomic memory operations can lead to better lock performance.

In this chapter we describe our implementation and evaluation of NIC-based remote atomic memory operations. We implemented these operations by modifying the GM message passing system [34] which uses programmable Myrinet [10] network cards. We found up to a 1.25 factor of improvement for performing a remote atomic operation using our NIC-based approach over using the best host-based implementation. When we implemented a distributed lock algorithm using the remote atomic operations our NIC-based implementation gave up to a 2.6 factor of improvement over the host-based implementation. Furthermore, we found that locks implemented with host-based atomic operations had a significant impact on host processor and NIC

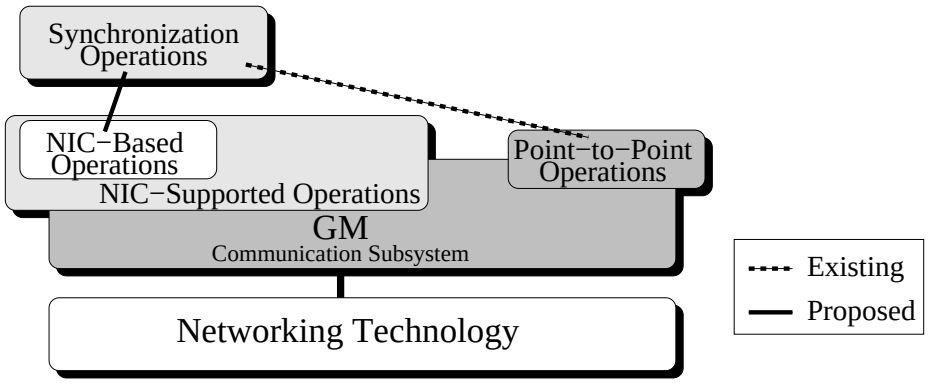


Figure 5.1: NIC-based atomic remote memory operations for the GM communication subsystem

processor utilization, while locks implemented with NIC-based atomic operations had little to no impact.

Figure 5.1 illustrates our approach to adding NIC-based atomic remote memory operations to the GM communication subsystem. The dotted line indicates how the atomic operations are traditionally implemented using point-to-point messages. The solid line shows our implentation using the NIC-based atomic remote memory operations.

The rest of the chapter is organized as follows. In Section 5.1 we describe the basic concept of NIC-based atomic remote memory operations. Section 5.2 describes the implementation of these operations, and Section 5.3 describes how to implement distributed locks using the operations. In Section 5.4 we present our experimental results, and summarize our work in Section 5.5.

5.1 NIC-Based Atomic Remote Memory Operations

The basic idea of the NIC-based remote atomic operations is to have the NIC perform the operation directly rather than dedicate a separate thread at the host to perform the operation. The application initiating the remote atomic operation would send a message to the NIC at the remote node indicating which operation to perform along with the operands. The remote NIC, upon receiving the message, would perform the operation atomically on the memory at the host. The atomicity of an operation is guaranteed, with respect to other NIC-based atomic operations, by ensuring that the NIC does not perform any other operations on that memory region until that operation has completed. In order to guarantee atomicity of the NIC-based operation with respect to the host process, either all local operations need

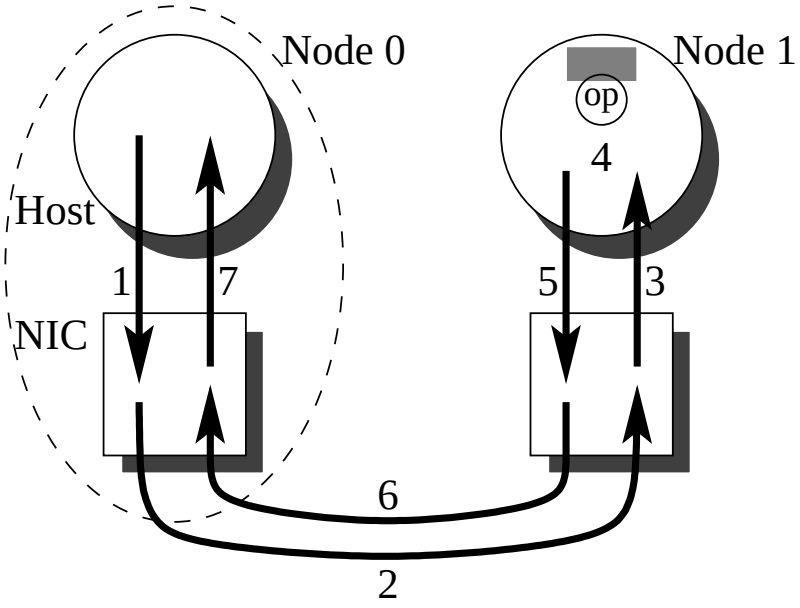
to be performed by the NIC, or a mutex can be used to serialize access to the host memory. Figure 5.2(a) shows an example of an atomic operation being performed by a thread running on the host processor (host-based atomic operation) and Figure 5.2(b) shows an example of an atomic operation being performed by the NIC (NIC-based atomic operation).

In order to perform an atomic operation on a remote memory region without using NIC-based atomic operations, the remote node would need to have a thread which receives the requests, performs the operations and returns the result. This is shown in Figure 5.2(a). Here, an application at node 0 sends a request for an atomic operation to a thread at node 1 which performs the operation. This is performed in seven steps:

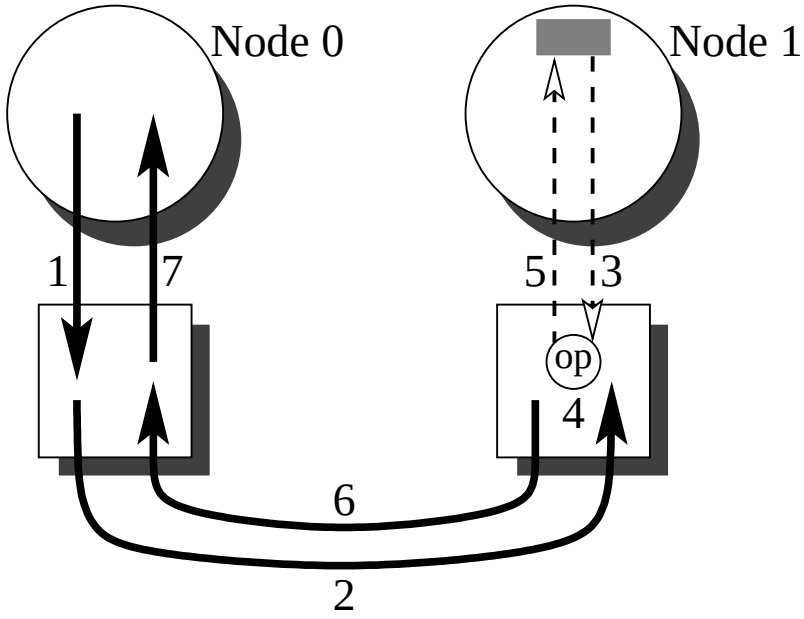
1. A message is generated by the application at the host of Node 0 and is sent to the NIC.
2. The NIC then transmits it to the NIC at Node 1.
3. The NIC at Node 1 receives the message and forwards it to the thread which is handling the atomic operations.
4. Upon receiving the message, the thread at Node 1 performs the operation specified in the message on the host's memory.
5. This thread then sends a reply message to the NIC.
6. The NIC at Node 1 then transmits it back to node 0.
7. This reply is received by the NIC at Node 0 and is forwarded to the application.

Using NIC-based atomic operations, no thread is needed at the remote host to handle the atomic operations. Instead, the operations are performed directly by the NIC. Figure 5.2(b) shows a NIC-based atomic operation on remote memory. This operations is performed as follows:

1. An application at Host 0 sends a special atomic operation message to the NIC.
2. This NIC transmits it to the NIC at Node 1.
3. Upon receiving the message, the NIC copies the value stored at the memory location specified in the message using DMA.
4. The NIC then performs the operation using this value.
5. If necessary, the NIC copies the new value back to the memory location, again using DMA.



(a) Host-based



(b) NIC-based

Figure 5.2: Steps required to perform host-based and NIC-based atomic remote memory operations.

6. The NIC transmits the result back to the NIC at Node 0. Note that this step can be performed concurrently with the previous step.

7. The NIC at Node 0, upon receiving the result message, forwards it to the application.

The main advantage of using the NIC-based approach is that the operation can be performed without the intervention of a host thread. With the host-based approach, the atomic operation requests need to be handled at the host. This can be done by having the main application periodically poll for these messages, however, this can lead to poor response time for the operation if the main application polls infrequently. Another option for the host-based approach is to have a separate *server thread* to handle these requests. When using a server thread, unless a separate processor at the host can be dedicated to this thread, the thread should block while waiting for the requests. The main thread is then interrupted when an incoming request is received so that the server thread can process the request. When the server thread and main thread share a CPU, blocking the server thread while it is idle will lead to better utilization of the CPU by the main thread. However, when there are many such requests, the repeated interrupts can lead to poor performance of the main application. By using the NIC-based approach atomic operations can be performed without interfering with any processes at the host. The application process can be running on the host CPU, while the NIC is performing the atomic operations directly on host memory.

Just about any atomic operation can be implemented using this scheme. The only constraint is the processing power of the NIC processor. Typically, the NIC processor is much slower than the host processor. For instance, the LANai processors on the Myrinet NICs range from 33MHz to 233MHz, while host processors may range from 300MHz to 2GHz. Furthermore, NIC processors may not have floating point units, so any floating point operation would have to be simulated using integer operations. For this reason it would probably not be beneficial to perform complex operations. Another constraint is the NIC processor's access to the host memory. Most NICs do not support PIO access to the host memory from the NIC. Rather any transfers of data from host memory initiated by the NIC must be done using DMA. While DMA performs well for transferring large data, there is an overhead to setting up the DMA which makes it less efficient for performing small data transfers. So the number of data transfers between the NIC and host memory should be limited.

We implemented the following three atomic primitives: fetch&add, fetch&write and compare&swap. The fetch&add and fetch&write operations take four parameters: the target node id, the target port id, the remote virtual memory address, and a 32-bit data word. The compare&swap operation takes one additional 32-bit parameter which is used for the compare part of the operation. Table 5.1 describes the semantics of the operations. For each operation, the table shows the value of the memory location

Operation	Memory Contents		Return Value
	Before Op	After Op	
fetch&add($data$)	X	$X + data$	X
fetch&write($data$)	X	$data$	X
compare&swap($data$, $compare$)	X	$\begin{cases} data & \text{if } compare = X \\ X & \text{otherwise} \end{cases}$	X

Table 5.1: Semantics of atomic memory operations

before the operation and after, as well as the value returned to the caller. The *data* value in the table represents the data that is to be written to the memory location and the *compare* value in the table represents what the value stored in the target memory location is compared to.

5.2 Implementation

In this section we describe our implementation of the NIC-based remote atomic operations as a modification of Myricom's message passing system GM[34] version 1.5. We will first give a brief overview of Myrinet[10] and GM, then describe our implementation.

5.2.1 Overview of Myrinet and GM

Myrinet is a low latency, high bandwidth, wormhole routed network. The links are full-duplex and have either 1.28+1.28 gigabits per second or 2+2 gigabits per second link rate. The newer NICs and switches provide the 2+2 Gbps link rate.

The Myrinet NIC consists of a programmable *LANai* processor, memory, one DMA engine for transferring data between the NIC memory and the host memory, one DMA engine for transmitting data from the NIC memory and the network and another DMA engine for receiving data from the network to the NIC memory. Depending on the revision of the card, the LANai processor runs at either 33, 66, 133, or 200MHz, and has between 1 and 4 MB of SRAM. The programmable processor runs a control program which allows host processes to directly interact with the NIC bypassing the operating system (OS-bypass) for low latency communication.

GM is a user-level message passing system that uses the Myrinet network. GM consists of a kernel module, a library and a Myrinet control program (MCP). The driver loads the MCP on to the NIC when it is loaded. During the execution of a program, the driver is used mainly for opening *ports*, pinning and unpinning memory, and to put a process to sleep for blocking functions. A *port* is a data structure through

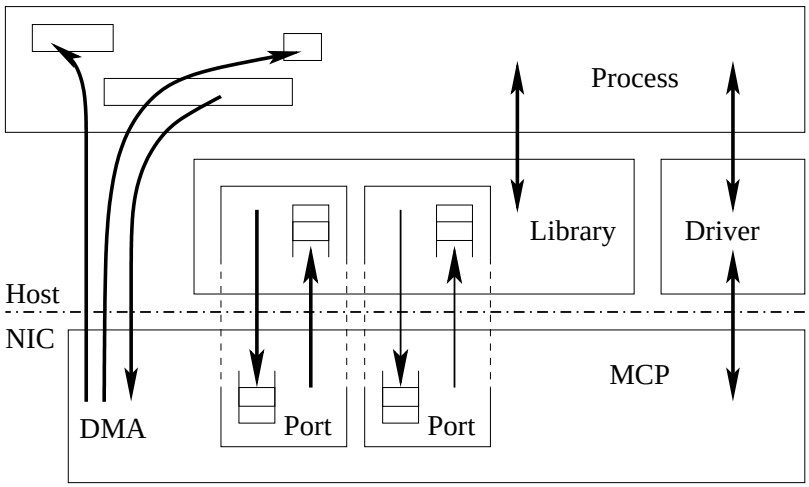


Figure 5.3: Block diagram showing the components of GM.

which a process can communicate with the NIC. Once a port is opened, the process can communicate with the NIC, bypassing the operating system and avoiding system call overhead.

Figure 5.3 is a block diagram of GM where a process has two ports through which send tokens and receive tokens are transferred to and from the MCP without going through the kernel. The figure also shows DMA operations which transfer data directly to and from memory regions of the process.

At the host level GM is connectionless, but it provides reliability by maintaining reliable connections between NICs of different nodes. When a packet is sent by the NIC, the NIC keeps a *send record* with enough information to reconstruct the packet. The send record also has a timestamp of when the packet was sent. Until the packet is acknowledged, the NIC checks the timestamp of each send record to determine if a packet needs to be resent. If a packet times-out, using the information in the send record, the NIC DMAs the data for the packet again from host memory, reconstructs the packet and transmits it. Upon receiving an acknowledgment for a packet, the corresponding send record is deleted. Packets also have sequence numbers which are used to ensure correct packet ordering. If a packet is received out of order, or if a duplicate packet is received, it is dropped and an acknowledgment packet is sent re-acknowledging the last packet correctly received.

Flow control is used between the NIC and the host to avoid buffer overflows. To provide this flow control GM uses the concept of tokens. When a process opens a port, it has a certain number of *send tokens* and *receive tokens*. Each send token corresponds to a send event. For sending a message the process fills-in a send token describing the send event and passes it to the NIC. The message may consist of

several packets. The NIC takes care of packetizing the data. Once the NIC has finished sending the message and all of the packets have been acknowledged, the NIC returns the send token to the process in a callback function.

Data can only be sent from or received into pinned memory regions. This is necessary so that the pages that contain the data are not paged out by the operating system while the data is being transmitted by the NIC. GM provides special functions which pin memory and inform the NIC of the physical address and virtual address of the pages to be used for address translation when DMAing the data. This is known as *registering* memory.

In order to send or receive a message, the process must pass a receive token describing the buffer to the NIC. Once the NIC has DMAed the data into the buffer, the receive token is returned to the process. The process can either poll to detect returned receive tokens, or block and wait for the receive tokens.

5.2.2 Design Challenges and Our Implementation

We implemented two ways for the return value of the operation to be provided to the host process. In one method a receive token is created by the NIC which contains the return value. In this method, the process would call `gm_receive()`, or one of its variants, to get the return value. In the other method, the NIC would DMA the return value directly to a memory location specified by the process when the operation is initiated. This method saves the time taken by the NIC to create a receive token, and is slightly faster.

We added three functions to the GM API, and modified the MCP. The `gm_provide_atomic_buffer()` function passes a receive token to the NIC. This function is used if the process is to receive the return value with a receive token. The process must ensure that the NIC has sufficient receive tokens to receive the return values for the atomic remote memory operations. The `gm_atomic_send_with_callback()` function is used to initiate an atomic remote memory operation in which the return value is provided to the process with a receive token. The function builds a send token and passes it to the NIC initiating the atomic remote memory operation. The send token describes the atomic operation and includes all of the necessary parameters plus a *tag* value which the application will use to match the atomic operation request with the return value. The process then checks for the completion of the operation using the `gm_receive()` function or one of its variants. When the operation is completed, the `gm_receive()` function returns a receive token with the return value of the operation along with the tag value which was provided in the corresponding call to `gm_atomic_send_with_callback()`.

The `gm_atomic_send_with_callback_direct()` function is used to initiate an atomic remote memory operation where the return value is DMAed directly to the

process' memory. In this function, the tag value is not needed. Instead, the process supplies the memory address of the location where the return value is to be DMAed.

When the NIC receives the send event, it transmits an *atomic operation packet* to the destination node with all relevant parameters. Upon receiving the packet, the NIC at the destination node checks for packet corruption and correct packet sequence, and when the DMA engine to the host is free, performs the operation.

The operation is performed in the following manner. The NIC DMAs the data from the target host memory location to a temporary location. The NIC then calculates the new value of the target memory location (as described in Table 5.1), and DMAs the new value to the host memory. Because the NIC performs this operation without interruption, the atomicity of the operation is guaranteed with regard to other atomic requests. We will describe guaranteeing atomicity with regard to the host process in the next section. The return value is stored in a table and a *reply packet* is sent back to the initiating node. The reply packet also serves as an acknowledgment for the atomic operation packet. Upon receiving the reply packet, the NIC checks for packet corruption, processes the acknowledgment and sends a *receive token* to the application. The NIC performs each of the operations using one or two DMAs.

In our implementation, we had to address the following challenges: how to inform the NIC that a particular memory region should be used for atomic operations, how to notify the calling process of the return value, and how to provide reliability.

We addressed the first challenge of how to inform the NIC that a particular memory region should be used for atomic operations by using the same method that GM uses to provide *directed sends*. In GM, directed sends are messages where the data is written directly to the receiver's memory without the receiver calling `gm_receive()`. The sender of the message provides the address of the buffer at the receiver. This type of communication is sometimes called RMA (for remote memory access) or RDMA (for remote direct memory access). With directed sends, the sender can specify any registered memory region at the receiver as the destination address. We used the same idea. Atomic operations can be performed on any 32-bit word in any registered memory region at the target node.

The number of memory locations that can be used by the atomic remote memory operations is limited only by the amount of memory that a process can register in GM, which, for GM version 1.5 on Linux, is 7/8 of the physical memory of the host. We can specify a remote memory location by a triple: a node id, to identify a particular machine on the network; a port id, to identify a particular process on that machine; and the virtual address of the memory location in the address space of that process. The method of distributing this triple to other nodes in the system is left up to the programmer (e.g., by simply sending the triple in a message).

The second challenge was how to notify the calling process of the return value. Atomic remote memory operations produce a return value which the calling process must receive. We needed a mechanism by which the calling process can receive the

value. We used the two methods described above: using receive tokens, and DMAing the value directly to host memory. The receive token had space for small message data to be sent to the host, this allowed for the result to be included in the token. When the NIC receives the return value from the NIC at the remote node, it passes the process a receive token containing the return value. In order for the calling process to match the call to the atomic operation with the return value, the process specifies a *tag* value when the operation is initiated which is then included in the receive token along with the return value.

In the second method, upon received a reply packet, the NIC will DMA the return value followed by a flag value to the address specified in `gm_atomic_send_with_callback_direct()` when the operation was initiated. The flag value is used to indicate to the process that the return value has been DMAed, so before calling `gm_atomic_send_with_callback_direct()`, the process must reset the flag. By using a different address for each outstanding atomic operation, a tag value is not needed to match return values to a particular instance of an atomic operation.

The third implementation challenge was to provide reliability for the atomic operations messages. To do this, we used mechanisms similar to the ones used by GM with two differences. First the reply packet doubles as an acknowledgment packet to the atomic operation packet, so a separate packet is not needed. Second, we handle duplicate atomic operation packets differently. A duplicate atomic operation packet cannot be dropped because the initiating node needs the return value of the atomic operation. Furthermore, because the operations are not idempotent we cannot repeat the operation to get the return value. Instead, the NIC keeps a table of return values and sequence numbers. Upon receiving a duplicate atomic operation packet, the NIC looks up the return value and re-sends a reply packet. Because there are a limited number of entries in the table of return values, each NIC must limit the number of unacknowledged atomic operations it sends, otherwise it is possible that some of the return values for outstanding packets will not be stored. This method of using a table to record return values is similar to the one described in the InfiniBand Architecture standard[26].

5.2.3 Serializing Access to Host Memory

Allowing NIC-based atomic operations to modify data which the host process may be accessing, may lead to race conditions. One way to avoid race conditions is to only allow the host process to access a variable when it is known that the NIC will not perform an atomic operation on it. This may be possible in some cases, but not in general. Another option is to force the host process to use the NIC to access these variables. While this will increase the latency of accessing those local variables, this option may be acceptable in cases where the host process rarely accesses those variables.

```
1    char turn;
2    char interested[2];
3
4    enter_region (int process) {
5      interested[process] = TRUE;
6      turn = process;
7      while (interested[1-process] == TRUE && turn == process) skip;
8    }
9
10   leave_region (int process) {
11     interested[process] = FALSE;
12   }
```

Figure 5.4: Peterson's mutual exclusion algorithm

A third option is to use a mutex to serialize access between the NIC and the host. There are many mutex algorithms available [47, 51]. Since we only needed to provide mutual exclusion between two parties, the NIC and one host process, we used Peterson's algorithm because it is simple and efficient in this context. Figure 5.4 shows Peterson's algorithm. When a party, either the NIC or the host process, wants to enter the critical region and modify a variable, it sets its `interested` variable to `TRUE` to indicate that it is attempting to enter the critical section (line 5). It then sets the `turn` variable to its identifier, in line 6. Next, in line 7, it waits until the other party is not interested, or the `turn` variable is no longer set to its identifier. The `turn` variable is used to break ties. If both parties are attempting to enter the region at exactly the same time, one of the parties will set the `turn` variable before the other. Then when they enter the while loop on line 7, eventhough both parties have their `interested` variables set, the `turn` variable will be set to one of the parties' identifiers, allowing the other party to exit the while loop, and enter the critical section. Once the party has entered the critical section, it performs the atomic operation, then exits the critical section by setting its `interested` variable to `FALSE`, in line 11.

There are several constraints in applying this algorithm to NIC-based atomic operations. First the NIC does not have load/store access to the host memory. Instead, it must use DMA to modify or read host memory. DMA transfers are optimized for large data sizes, and moving single variables using DMA is much higher than accessing the data using loads and stores. Another constraint is the time it takes for the host process to access NIC memory over the PCI bus. Although the host can access NIC memory using loads and stores, the time to access the memory over the PCI

bus, which is uncached, is considerably longer than the time to access local memory, especially if it is cached. These constraints mean that it is impractical to have the NIC poll on a variable stored at the host. Similarly, it is very costly to have the host poll on a variable stored at the NIC.

We modified the algorithm to account for these constraints. First we located the NIC's `interested` variable in host memory, and the host's `interested` variable in NIC memory. We then located the `turn` variable at the NIC. So when the NIC requests the critical section it performs one DMA, to set its `interested` variable at the host, then sets the `turn` variable using a local store. Next, it polls on the host's `interested` variable and the `turn` variable which are both local. When the host requests the critical region, it sets its `interested` variable and the `turn` variable over the PCI bus in NIC memory. Notice that the NIC's `interested` variable is stored in host memory, so polling on this variable is efficient. However the host must still check the `turn` variable which is located in NIC memory. In order to limit the effect of polling on the `turn` variable, the host polls on the turn variable using an exponential backoff scheme: The host checks `turn` the first iteration of the while loop, then skips the next iteration, checks it on the following one, then skips two iterations, polls the following one then skips four iterations, and so on. We note that the `turn` variable is used only to break ties, and will only change in the time between lines 5 and 7. This means that the longer the host is polling on the mutex, the less likely it is that `turn` will change. We achieved very good performance using this scheme.

5.3 Implementing Distributed Locks with Atomic Remote Operations

One use of remote atomic memory operations is in distributed locks. We implemented a software queuing lock using atomic remote memory operations similar to the MCS[32] lock. The MCS lock is intended for shared memory machines, but we extended the idea for distributed memory machines using atomic remote memory operations. The algorithm creates a distributed linked list of processes waiting for the lock. The process at the head of the queue holds the lock. In this algorithm, each process has a `next` variable which points to the process which has requested the lock immediately after this process, and a boolean `locked` variable which indicates whether the node is waiting for the lock. These two variables should be stored so that atomic remote memory operations can be performed on them. The `lock` itself is a variable which points to the last node to request the lock. The `lock` variable is stored at the *home node* of the lock.

When a process, p, requests a lock, it first sets its `next` variable to NIL. Next, it performs a fetch&write(p) operation on the `lock` variable to determine which process is currently last on the queue (i.e., its *predecessor*. If the queue is empty (i.e., the predecessor is NIL), then this node has acquired the lock. Otherwise, it sets its own

`locked` variable to true, then performs a remote write to write p to its predecessor's `next` variable, thereby inserting itself in the queue. It then polls on its own `locked` variable until it becomes false.

To release a lock, a process, p, first checks if its own `next` variable is NIL. If it is not, then it performs a remote write operation on its *successor's* `locked` variable setting it to false thereby successfully releasing the lock. Otherwise, it performs a compare&swap(NIL, p) operation on the `lock` variable. If this succeeds, (i.e., operation wrote a NIL to the `locked` variable) then process p was the last node on the queue, and has successfully released the lock. If the operation failed, this indicates that another process has begun requesting the lock, and has updated the `lock` variable, but has not yet updated process p's `next` variable. Process p should then poll on its `next` variable until that process updates it, at which point process p should perform a remote write setting the `locked` variable at that process to false.

Figure 5.5 gives an example of how the lock algorithm works. In the figure, the circle with the `L` in it represents the `lock` variable stored at the home node. The boxes with the numbers in them represent the processes requesting the lock. The arrows coming out of the boxes represent the `next` variable, and the squares in the boxes represent the boolean `locked` variable. A filled in square indicates that `locked` is set to true and that the process is waiting on the lock. Step (a) shows the initial state where there are no processes requesting the lock. Step (b) shows the state after process 1 acquires the lock. In (c) we see the state after processes 2 and 3 have requested the lock, but before process 1 has released the lock. Notice that the `locked` variables for processes 2 and 3 are shown as true. When process 1 releases the lock, it will notice that its `next` variable points to process 2. It will then change process 2's `locked` variable to false, so that process 2 can acquire the lock, as shown in step (d). Step (e) shows the state where only process 3 is left in the queue and has acquired the lock. If process 3 releases the lock before another process requests it, the `lock` variable will be set to NIL, and the state will be the same as in step (a).

Because we implemented the lock algorithm for distributed memory, we cannot use simple memory pointers for the `next` and `lock` variables as used in the original MCS algorithm. As we described in Section 5.2.2, a remote memory location is specified as a triple of the node id, port id, and virtual memory address. Instead of using memory pointers, our lock implementation uses process ranks. Each process then has an array where the ith element stores the remote memory triple describing the location of the ith process' `next` and `locked` variables. For example, when a process releases a lock, and reads a value of 4 in its `next` variable, it gets the remote address triple for its successors `locked` variable by looking up the 4th entry in the array.

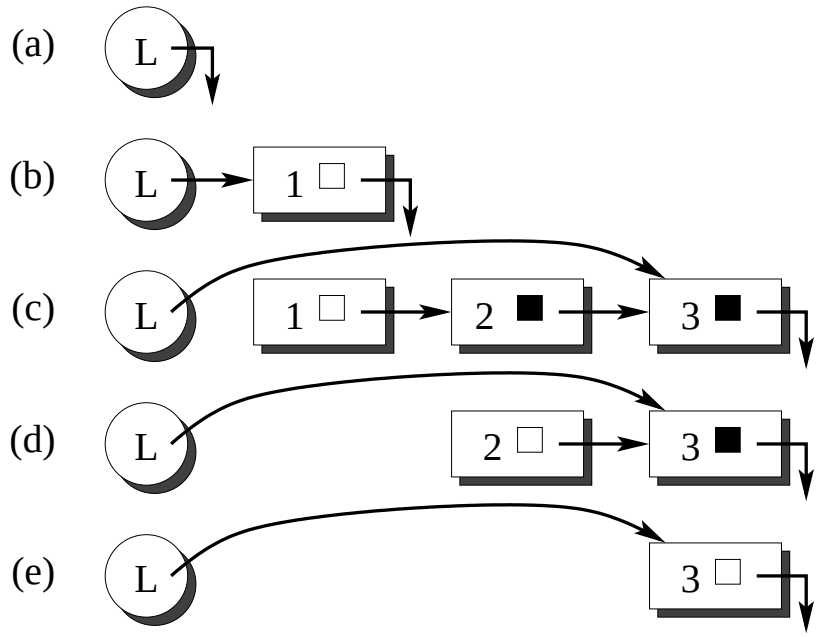


Figure 5.5: Example of a distributed lock

5.4 Experimental Results

In this section, we evaluate our implementation of NIC-based atomic remote memory operations. We first evaluate the individual operations and compare them to host-based implementations. Next we implement a distributed lock using atomic remote memory operations and evaluate the performance of the locks using our NIC-based implementation and the host-based implementations. Finally, we evaluate impact of using NIC-based versus host-based atomic remote memory operation on host processor and NIC processor utilization.

The performance results were run on a cluster of 16 quad 700MHz Pentium III machines. The machines are connected by a Myrinet[10] LAN network using NICs with 133MHz LANai 9.1 processors. These are connected to 16 ports of a 32 port switch.

5.4.1 Atomic Remote Memory Operations

We tested the performance of the NIC-based atomic operations and compared it to host-based implementations. Three different versions of the NIC-based operations were tested. In the first version, *mutex*, a receive token was used to provide the process with the return value, and access to the host memory was serialized using

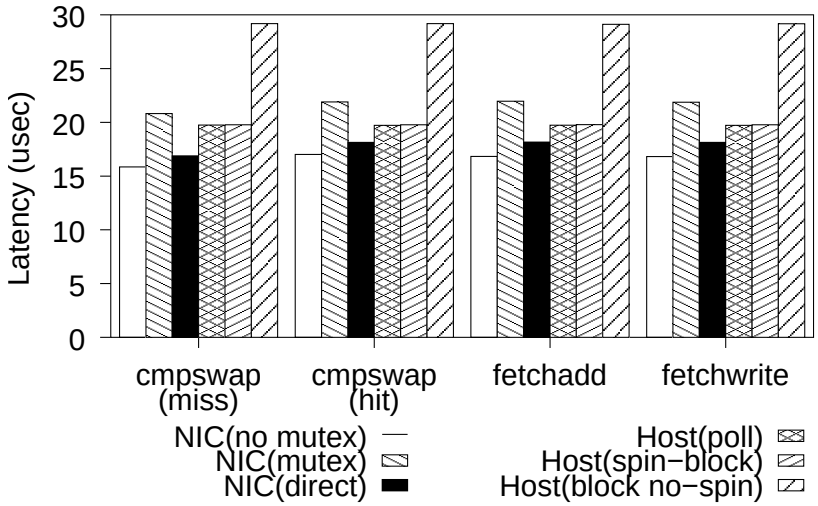


Figure 5.6: Latencies of atomic operations

a mutex. In the second version, *no-mutex*, a receive token was again used, but no mutex was used. In the third version, *direct*, the return value was DMAed to the process, and a mutex was used by the remote NIC to serialize access to host memory.

A host-based test consists of a process which sends atomic operation request messages to a *server* process on another node. The server process receives the request messages, performs the operations and returns reply messages. Because a host-based implementation would most likely to be on a separate thread which would be interrupt driven, we tested three different methods that the server could use for checking for incoming messages. In the first case, *poll*, the server process is polling for the reception of new request messages. In the second case, *spin-block*, the server process polls for a short while, then blocks waiting for the message. In the last case, *block no-spin*, the server process blocks immediately waiting for a new message without polling. These three cases correspond to the GM functions `gm_receive()`, `gm_blocking_receive()` and `gm_blocking_receive_no_spin()`, respectively.

The tests consist of taking the average time of 10,000 iterations of each atomic operation. The compare&swap operation was evaluated in two ways, once where the compare failed (*miss*) so that the swap was not performed, and once there the compare succeeded (*hit*). Figure 5.6 shows the results of these tests. Notice that the NIC-based no-mutex and direct perform better than any of the host-based operations, and that the NIC-based mutex version performs better than the host-based blocking-no-spin. The NIC-based no-mutex compare&swap (hit) operation took an average of 15.9µs as compared to the best host-based implementation, polling, which took

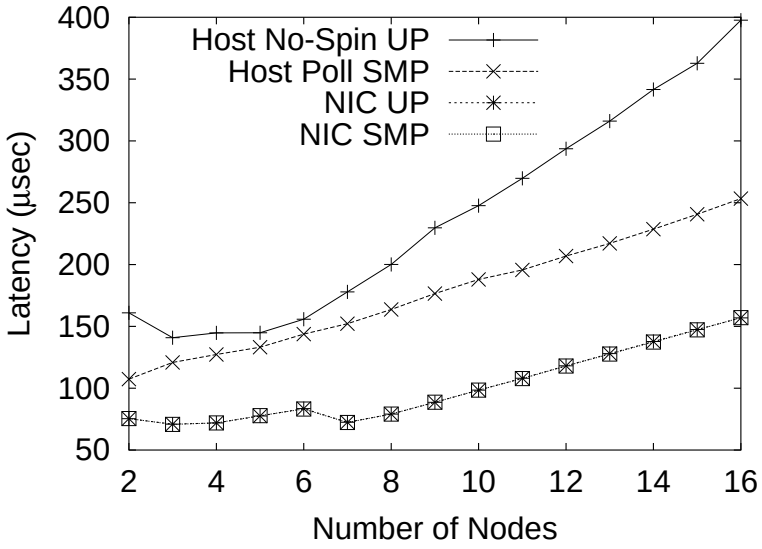
an average of 19.8µs. This gives a 1.25 factor of improvement. The spin-block host-based implementation took an average of 19.8µs, and the blocking-no-spin host-based implementation took an average of 29.2µs. The NIC-based implementation showed a 1.84 factor of improvement over this host-based blocking-no-spin implementation.

One should note, that these tests represent the best-case configurations. At each node there is only one process running. In this situation, for the host-based implementation, the polling and spin-block versions of the server thread perform much better than the block-no-spin version. However, these versions would typically not be used in a situation where the server thread was sharing a single processor with another thread. In such a situation, if a server thread uses polling receives, the CPU will be under-utilized whenever the server thread is scheduled and no messages are coming in. Using blocking-no-spin receives releases the CPU when no messages are waiting to be received, and will not schedule the process until a message comes in. This leads to better performance of the main thread because it get scheduled more often. Using spin-block receives works in a similar manner as using blocking-no-spin receives, except that when there are no messages to receive, the operation polls for a while before blocking. This works well when incoming messages are bursty, so that many messages can be handled with one interrupt. Otherwise, if no new message is received, the time that the server thread spends polling is wasted. This again would lead to poor main thread performance.

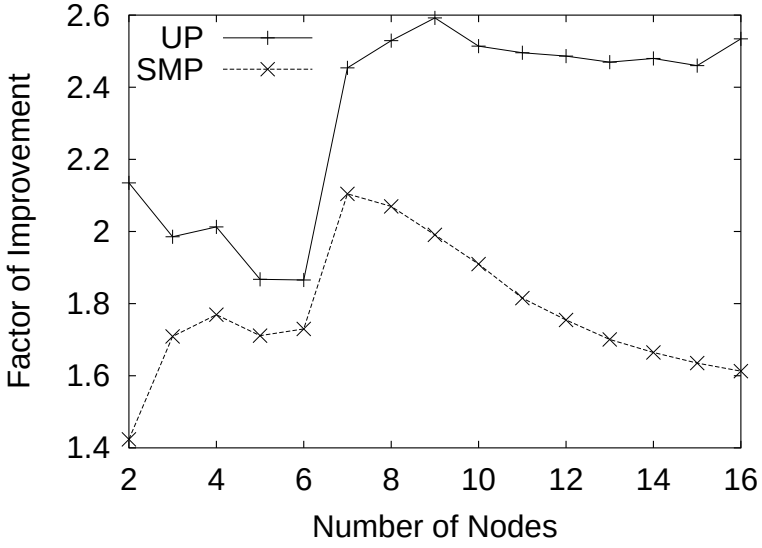
5.4.2 Distributed Locks

The lock test for our host-based implementation consists of the *main thread*, a *server thread* and a shared memory region which both threads can read from and write to. The *main thread* requests and releases locks by sending remote atomic operation requests to the *server threads* at the target node. The server thread performs the operations on the target memory location in the shared memory region. The NIC-based implementation consists of just a single thread which requests and releases locks using NIC-based remote atomic memory operations.

In this test we took the average time it takes for one process at one particular node to repeatedly acquire and release a remote lock. To vary the load, we added more nodes also repeatedly locking and unlocking the same lock. The tests were run using both an SMP enabled kernel, so that one CPU was available for each thread, and using a uniprocessor kernel (UP), in which case both threads shared the same processor. For the tests run on the SMP kernel, the server thread for the host-based implementation used polling receives, since this performed better than the other receive methods when there was no other thread contending for the CPU. While for the tests run on the uniprocessor kernel, the server thread used polling-no-spin receives, because this performed better than the other receive methods when the



(a) Latency



(b) Factor of Improvement

Figure 5.7: Locking and unlocking with multiple nodes contending

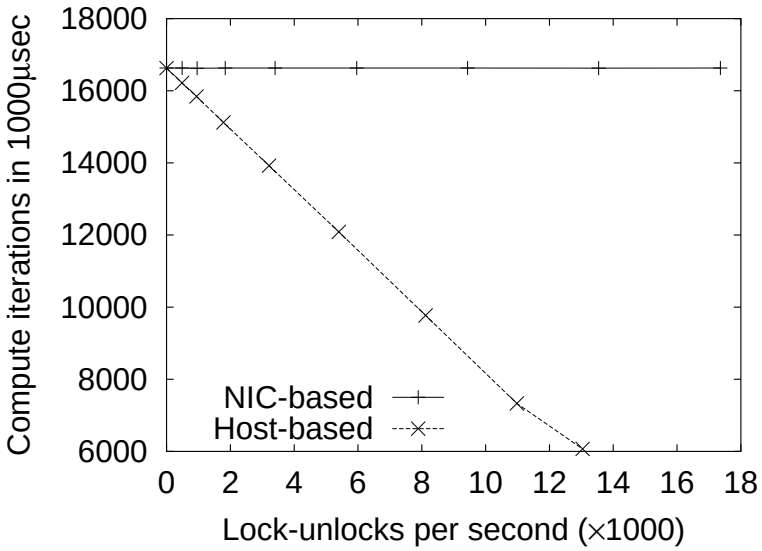


Figure 5.8: The number of iterations a process at the home node of a lock can perform in 1,000µs while a process at another node is locking and unlocking the lock at a certain rate

server thread was sharing the processor with the main thread. For the NIC-based case, we used the implementation which used receive tokens, but no mutex.

Figure 5.7 shows the results of this test. We show the results for the host-based blocking-no-spin using the uniprocessor (UP) kernel, and the host-based polling using the SMP kernel and compare them to the NIC-based implementations on each kernel. Notice that in both graphs the NIC-based implementation outperforms the host-based implementations. Notice also that because there is only one process necessary for the NIC-based implementation, the NIC-based implementation gives the same performance using either kernel. As expected the host-based polling implementation on the SMP kernel outperforms the host-based blocking implementation using the uniprocessor kernel because of the lack of context switching overhead in the SMP case.

Figure 5.7(b) shows the factor of improvement of the NIC-based implementation over the host-based implementation for the SMP and uniprocessor (UP) kernels. We see up to a 2.6 factor of improvement for the UP case and up to a 2.1 factor of improvement for the SMP case.

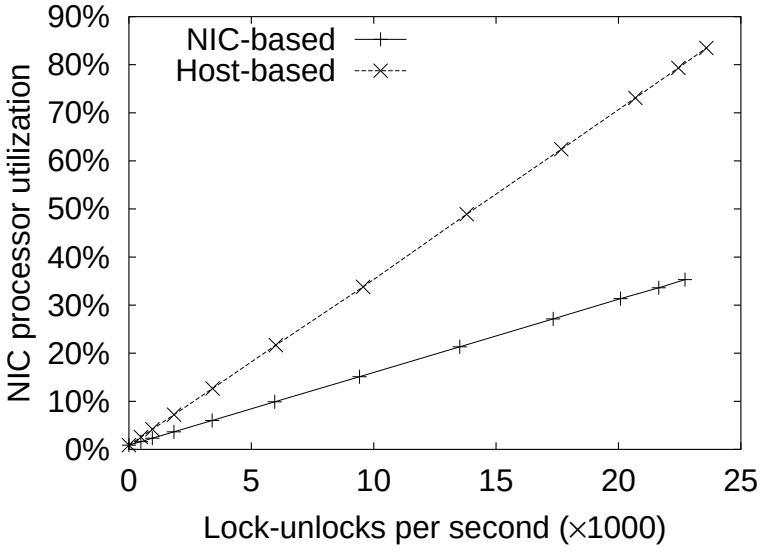


Figure 5.9: The fraction of time the NIC processor is not idle while a process at another node is locking and unlocking the lock at a certain rate

5.4.3 Host and NIC Processor Utilization

To test the impact of using the atomic operations on host processor utilization we performed a test where a process (the *counter process*) at the home node of a lock performs a loop for 1,000µs and counts how many iterations of the loop it was able to perform. While the counter process is doing that, a process at another node repeatedly locks and unlocks the lock. We inserted a delay just after the lock operation and another just after the unlock operation. By varying these delays, we can alter the rate at which the lock-unlocks are performed. We also performed the test where no lock or unlock operations were performed to serve as a baseline (*idle case*). When running a uniprocessor kernel, the number of iterations that the counter process is able to perform in the alloted time gives an indication of the amount of host processor time that is used in handling incoming atomic operation requests. The more time that the processor is spending handling incoming requests, the fewer iterations the counter process is able to complete. In this test we used the uniprocessor (UP) kernel and the blocking-no-spin version of the server thread for the host-based implementation. This way, when atomic operation requests are received, the counter process will be interrupted by the server process, and the impact can be measured.

Figure 5.8 shows the results of this test. The figure shows the number of iterations that the counter process was able to complete while lock-unlock operations were happening at a certain rate. As expected, the NIC-based implementation is unaffected

by the number of atomic operations being performed, because the operations are performed completely by the NIC and require no intervention of the host processor. However, for the host-based implementation we see that the processor utilization by the counter thread at the home node decreases as the number of atomic operations the server node is processing increases. In the host-based implementation, the main thread must be interrupted so that the server thread can process the incoming request leading to decreased processor utilization by the counter process.

We also evaluated the impact of handling atomic remote memory operations on NIC processor utilization. To perform this test we instrument the firmware to count the amount of time that the NIC spends in the idle loop waiting for an event. We perform this test similarly to the host processor utilization test. A node varies the rate at which it repeatedly locks and unlocks a lock located at another node. We then take the percentage of time that the NIC at the home node of the lock is not in the idle loop. This indicates what the load is on the NIC processor.

Figure 5.9 shows the results of this test. We see that as the rate of locking and unlocking increases, the NIC processor utilization increases for both the NIC-based and the host-based implementations. However, the host-based implementation increases much faster than the NIC-based. In fact, the host-based implementation increases about twice as fast as the NIC-based implementation. This indicates that the NIC-based implementation has less impact on the NIC processor than the host-based.

The reason for this is that in the host-based implementation, a process at the requesting node sends a message to the server process at the target node. The NIC at the target node must receive the message, process it, send it to the host process, then send an acknowledgment to the NIC at the requesting node. After the server thread processes the request, it send a reply message back to the requesting process. The NIC has to process the send token, transmit a message, then process the acknowledgment from the NIC at the requesting node. In the NIC-based case, the target NIC only has to handle receiving one message from the requesting node. Once it handles this message it sends the reply back, which also acts as an acknowledgment for the request message. The amount of time to actually perform the operation is small compared to the overhead of processing messages and acknowledgments. The host-based implementation has to handle two messages, whereas the NIC-based implementation has to handle only one message. This seems to fit with the results shown in Figure 5.9, where the utilization for the host-based implementation grows twice as fast as the utilization for the NIC-based implementation.

Notice in Figure 5.7(a) that for the NIC-based case, the latency of locking and unlocking the lock remains relatively constant between 2 to 7 nodes for the NIC-based cases, while in the host-based SMP case it starts increasing immediately. The reason for this is due to the load on the NIC processor. In the host-based case, with two processes performing locks and unlocks, the NIC processor is saturated, so even if the

host processor can handle more requests, the NIC is the bottleneck, and incoming requests are delayed. In the NIC-based case, the NIC processor is not saturated until around 7 processes are contending for the lock, so requests can be processed as soon as they arrive.

5.5 Summary

In this chapter, we presented an implementation of NIC-based atomic remote memory operations. We added support for atomic remote memory operations to GM version 1.5. We then evaluated the performance of the NIC-based atomic operations and compared them with atomic operations implemented at the host level. We found up to a 1.25 factor of improvement for the compare&swap operation when comparing the best NIC-based implementation to the best host-based implementation. Using these atomic operations to implement a distributed lock, we saw up to a 2.6 factor of improvement when using NIC-based atomic operations. Because the NIC-based atomic operations do not use the host processor they gave us better host processor utilization and NIC processor utilization than when using the host-based implementations.

By using the NIC-based remote atomic memory operations along with remote memory access methods provided by some communication layers such as Quadrics and GM, applications can reduce the number of messages that need to be handled by the application. This means that for applications which currently use server threads, the number of interrupts can be reduced, or that the server thread can be eliminated altogether. This would lead to better host processor utilization and performance of the main thread. Such an approach demonstrates potential for designing high performance system area networks for next generation clusters and servers.

CHAPTER 6

NIC-SUPPORT FOR APPLICATION-BYPASS BROADCAST

Process skew is an important aspect in parallel and distributed systems which has not received much attention. Many collective communication benchmarks [43, 54] perform the collective communication with all processes starting the operation at the same time. While this would be ideal when running a parallel application, it is not realistic. Processes can become skewed as a result of unbalanced code, where one process has more computation to perform than others, of asymmetric code, where different processes perform different computation or communication operations, of using heterogeneous systems, where nodes in the system have different performance characteristics, as well as of random, unpredictable effects such as the processes not being started at exactly the same time, or processors receiving interrupts during computation. Process skew may be more severe in georgaphically distributed computing systems, where communication time between remote nodes may be variable. The effects of process skew become more severe as the size of the system grows.

Collective communication operations can impose implicit synchronization. When processes are skewed, such synchronization will cause certain processes to wait idle for other processes to catch-up. With certain collective communication operations this synchronization is unavoidable unless a split-phase approach is used, such as a *reduce-to-all* operation where all processes must provide input and start the operation before any can finish, and with others, such as *barrier*, synchronization is the desired effect. However for other collective operations such as *broadcast* and *reduce-to-one* it is desirable to reduce the amount of implicit synchronization in order to reduce the effects of skew and improve overall system performance.

For instance, the broadcast operation in MPICH [22] is implemented such that a process will not forward the broadcast message until that process has made a call to `MPI_Bcast()` and received the message. If a process is slow to call `MPI_Bcast()`, other processes may be delayed as a result. A more desirable implementation would be to allow the broadcast operation to *bypass the application*. The concept of application-bypass operations was discussed in [11]. We will describe application-bypass in detail in the next section.

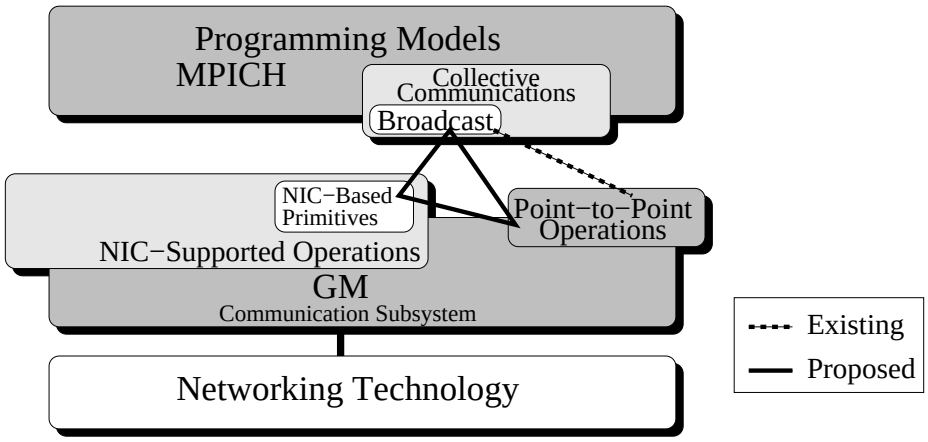


Figure 6.1: NIC-supported application-bypass broadcast for MPICH

We have designed and implemented an application-bypass broadcast operation by modifying GM [34] and MPICH over GM. We then evaluated our implementation and found that using application-bypass broadcast we see a factor of improvement of up to 16 when the processes are skewed. We also noticed that the effects of process skew become more severe as system size increases, and that application-bypass broadcast is considerably less sensitive to process skew than non-application-bypass broadcast. This indicates that that an application-bypass approach is critical for dealing with process skew allowing collective communications operations to be scalable.

Figure 6.1 illustrates our approach to application-bypass broadcast to MPICH using NIC-based primitives. The dotted line indicates how the broadcast operation is traditionally implemented using point-to-point messages. The solid line shows our implentation using NIC-based primitives and point-to-point messages to make the broadcast operation bypass the application.

The outline of the rest of this chapter is as follows. In the next section we describe the basic idea of application-bypass. The design and implementation of our application-bypass broadcast operation are described in Section 6.2. We evaluate our implementation in Section 6.3, the summarize our work in Section 6.4.

6.1 Application-Bypass

The basic idea of an application-bypass operation is that the application need not be involved in order for the operation to proceed. Broadcast is an operation which can be implemented in an application-bypass manner. The broadcast operation in many message passing systems is performed by creating a logical binomial tree over the processes participating in the broadcast. The root process sends a copy of the

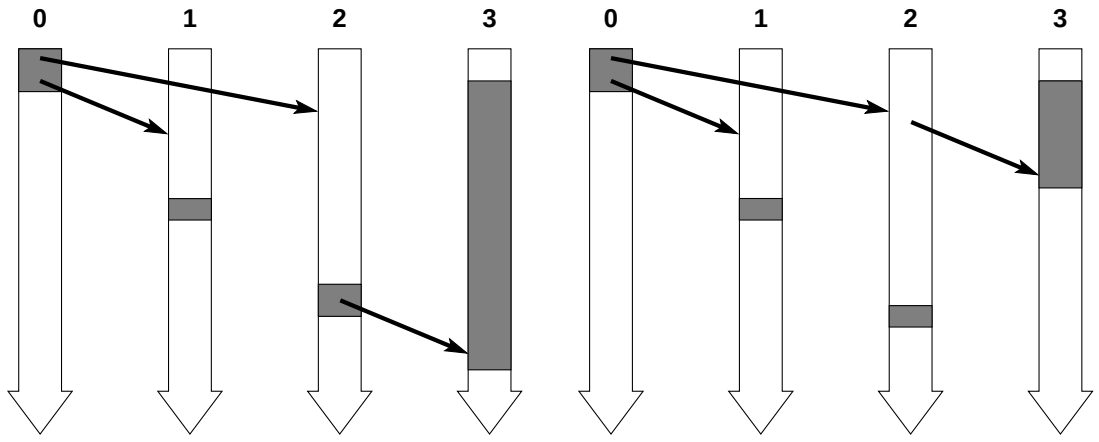
data to each of its children. Each non-root process waits to receive the data, then sends copies of the data to each of its children, if any. At the application level, each process pariticipating in the broadcast will call the broadcast function.

In MPICH, the `MPI_Bcast()` function performs the broadcast by first waiting for the message to be received from the parent process, if this process is not the root, then sending copies of the message to each child process. The broadcast operation in MPICH is not implemented in an application-bypass manner. If a message is received by a process, the message will not be forwarded to its children until the process calls `MPI_Bcast()`. This means that if a non-root, non-leaf node is delayed, the descendants of that process will also be delayed, even if they have called `MPI_Bcast()`, and are waiting for the message. If the broadcast operation were implemented in an application-bypass manner, as soon as a process receives a broadcast message, it would forward the message to its children, regardless of whether the process has called `MPI_Bcast()`.

Figure 6.2 illustrates the concept of an application-bypass broadcast. Figure 6.2(a) shows a non-application-bypass broadcast, while Figure 6.2(b) shows an application-bypass broadcast. These diagrams show four processes. The root process, Process 0, sends messages to its children, Processes 2 and 1. Process 2 receives the message and sends it to Process 3. Notice that in this example, the processes do not call broadcast at the same time, in particular, Process 2 calls the broadcast call much later than Processes 0 and 3. Because of this, in the non-application-bypass case shown in Figure 6.2(a) we see that even though Process 3 had called the broadcast function and was waiting for the message, it did not receive the message until after Process 2 finished its computation and called the broadcast function.

In the application-bypass broadcast operation shown in Figure 6.2(b), as soon as the broadcast message arrives at Process 2, it receives the data into a temporary buffer, and sends a copy to Process 3. Process 3 can receive the message much sooner because it doesn't have to wait for Process 2 to call the broadcast function. Once Process 2 calls the broadcast function, it will copy the data for the broadcast message from the temporary buffer to its final location.

Application-bypass operations can be even more important in large scale or heterogeneous systems. In such systems it is more likely for processes to be skewed, and so collective communication operations may not be called at the same time by all of the processes. Non-application-bypass operations can impose implicit synchronization among the processes, which means some faster processes will sit idle waiting for slower processes to catch up. Application-bypass operations can reduce the amount of synchronization that such operations cause. This can reduce the amount of time processes spend waiting for each other and can improve overall application performance.



(a) Non-application-bypass (b) Application-bypass

Figure 6.2: Broadcast operation over four processes. The large arrows represent timelines for each process. The shaded areas in these timelines represent a call by the application to the broadcast function, and the small arrows represent broadcast messages.

6.2 Design and Implementation

In this section we will describe the design and implementation of application-bypass broadcast. We start by identifying some design alternatives which we considered, next we give an overview of MPICH over GM, then describe our implementation in detail.

6.2.1 Design Alternatives

We identified several options for implementing application-bypass broadcast. One design option is to use a *broadcast thread* to perform the broadcast operation. To perform a broadcast, the main thread would send a message to its broadcast thread. The broadcast thread would be polling for incoming messages and would broadcast the message among the broadcast threads associated with the other processes. After broadcasting the message, the broadcast threads would send the message to their main thread. Because the broadcast thread is constantly polling for incoming messages it consumes processor resources which could be better used by its main thread on a uniprocessor system, or by additional computation threads on an SMP system. For this reason this option may not be practical in a real system.

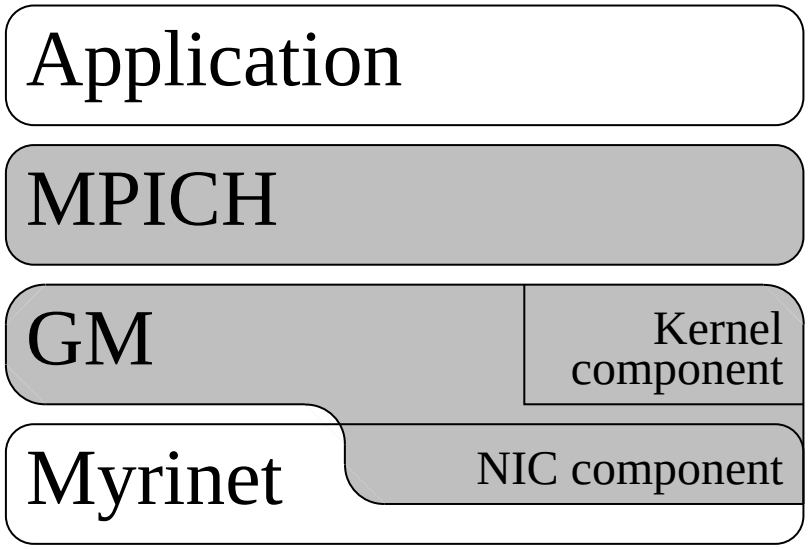


Figure 6.3: Software and hardware layers for MPICH over GM

Another alternative would be to have the broadcast thread block while waiting for an incoming message. This option would not waste processor resources, but would increase the latency of performing a broadcast because of the interrupt overhead. The cost of performing interrupts for every broadcast may make this option impractical.

We chose to implement a third option which uses a single thread and a signal handler. This option does not waste processor resources because the signal hander is only called when a broadcast message needs to be processed. Furthermore, because there is only one thread, when the thread is polling for a message, there is no need for a signal to be generated to process an incoming message. Our implementation allows the thread to disable interrupts at the NIC when polling for a message. This gives us the best of both previous design alternatives: low latency broadcast and low processor usage overhead.

6.2.2 Overview of GM and MPICH over GM

Before we describe our implementation we will briefly describe some internal details of GM and MPICH over GM. Figure 6.3 shows the software and hardware layers associated with MPICH over GM. GM is a user-level communication subsystem over the Myrinet [9] network. The Myrinet network is a 2Gbps full duplex network with network interface cards (NICs) that have programmable processors. GM consists of a kernel component, a user-level library and a NIC component. The kernel component is used for things like setting up new communication endpoints, and registering

memory, and is not used in the critical path. The NIC component is code which is executed on the NIC processor. Almost all of the protocol processing is performed at the NIC. The user-level library is basically used as an interface between the host process and the NIC code.

MPICH is an implementation of the MPI [33] message passing interface standard. MPICH has been ported to many different platforms and networks including GM.

The broadcast operation in MPICH is performed, as described earlier, by propagating messages over a broadcast tree. Each process participating in the broadcast makes a call to `MPI_Bcast()`. The root node and source or destination buffer are specified as parameters to this function. A *communicator* is also given as a parameter which specifies the group of processes which will participate in the broadcast. In the call to `MPI_Bcast()`, each process determines its parent process, if any, then waits to receive the message from this process. Once the message is received, it determines which processes are its children and sends the message to them.

When a process makes a call to receive a message, MPICH checks to see if the message has already been received. It searches a queue called the *unexpected message queue* for messages which match certain criteria specified in the receive call, such as sender id, datatype, and tag. If the message is not found, a *descriptor* is posted describing the anticipated message, as well as the memory location where the data should be received into. MPICH will then optionally poll for incoming messages until the message has been received.

When a message arrives at a process, MPICH first checks the list of posted receives, to see if this message is expected. If it is expected, it copies the data to the location specified in the descriptor, then marks the descriptor that the receive has completed. If MPICH finds no posted descriptor matching the incoming message, the message is copied into the unexpected queue.

MPICH over GM uses two modes in sending messages: *eager* and *rendez-vous*. The eager mode is used to send small messages. In this mode the data for the message is copied into a send buffer and is transmitted from the buffer. The copy to a send buffer is necessary because GM can only send data which is located in a pinned memory region. Pinning a memory region requires a system call and so is faster to copy the data to a pre-pinned buffer and send it from there than to perform the system call to pin the data in its original location. When the message is received, GM places the message in a pre-pinned buffer at the receiver. The data for the message must then be copied out of this buffer to its final location.

For large messages because the cost of copying the data becomes quite large, it is faster to pin the memory of the original source of the data at the sender and the final destination at the receiver, then send the data directly from the original location to the final destination eliminating any copies of the data. The rendez-vous protocol is used to perform this. The sender sends a request-to-send message to the receiver, pins the memory for the source of the data and waits for a response from the receiver.

Upon receiving the request-to-send message, the receiver pins the memory for the final destination of the data and sends the address of this to the sender in a OK-to-send message. When the sender receives this message, it sends the data directly from the source location to the remote destination.

6.2.3 Our Implementation

We modified MPICH over GM version 1.2.4..8a to provide application-bypass broadcast functionality. We also modified GM version 1.5.2.1 to allow signals to be generated when messages are received.

In MPICH, the broadcast operation is performed by a process when the application calls `MPI_Bcast()`. In order for the broadcast operation to bypass the application, the broadcast operation would have to be performed as soon as the broadcast message is received by the MPICH library. We did this by defining a new message type. When such a message type is received by the MPICH *progress engine*, copies of the message are sent to each of the children. Once the copies of the message are sent, the progress engine handles the message the same way as any other received message.

The list of children of a process is calculated by knowing the processes that are participating in the broadcast and which process is the root of the broadcast. Normally these parameters are supplied by the application to the `MPI_Bcast()` call. However, in our implementation, for non-root nodes, the broadcast operation is not performed in the `MPI_Bcast()` function. Instead, we added a field to the header of broadcast messages to identify the root. Also, MPICH includes a *context id* field in each message which can be used to uniquely identify an MPI *communicator*. A communicator specifies which processes are participating in a collective communication. When a communicator is created, we computed the list of children for that communicator, for each possible root. We store this array of lists of children in a hash table hashed on the communicator's context id. Then when an incoming broadcast message is received, we can get the list of children by getting the array from the hash table using the context id of the message, and indexing on the root, which is also given in the message.

In our implementation of application-bypass broadcast, we only considered messages sent in the eager mode. For MPICH over GM these are messages which are less than 16KB. With rendez-vous messages, the final destination must be known in order for the message to be sent. However, the final destination of the broadcast message is not known until the application calls `MPI_Bcast()`. Using a temporary buffer to store a broadcast rendez-vous message would require memory copies which would defeat the purpose of using the rendez-vous mode for large messages. We intend to study this issue in the future.

We added a signal handler which calls the progress engine to process any new messages. In order to avoid race conditions we added a mutex variable which is set when

the process calls the progress engine. When the signal handler is called, it checks the mutex variable and exits if it is set. The mutex variable is reset when the process exits the progress engine. However, this could lead to a case where we could lose a signal for a new broadcast message. For example, a new broadcast message could be received just before the process left the progress engine. The signal handler would be called, but it would exit immediately, because the mutex variable indicates that the process is executing the progress engine. When the process continues executing, it would leave the progress engine without handling the newly received broadcast message. To deal with this situation, we added a loop around the progress engine which keeps calling the progress engine while there are messages waiting to be received. This way, any broadcast message received after the progress engine processed the last message, and before it resets the mutex will be handled, because the loop condition will find that there is a pending receive. Any broadcast message received after the mutex variable is reset will be handled by the signal handler.

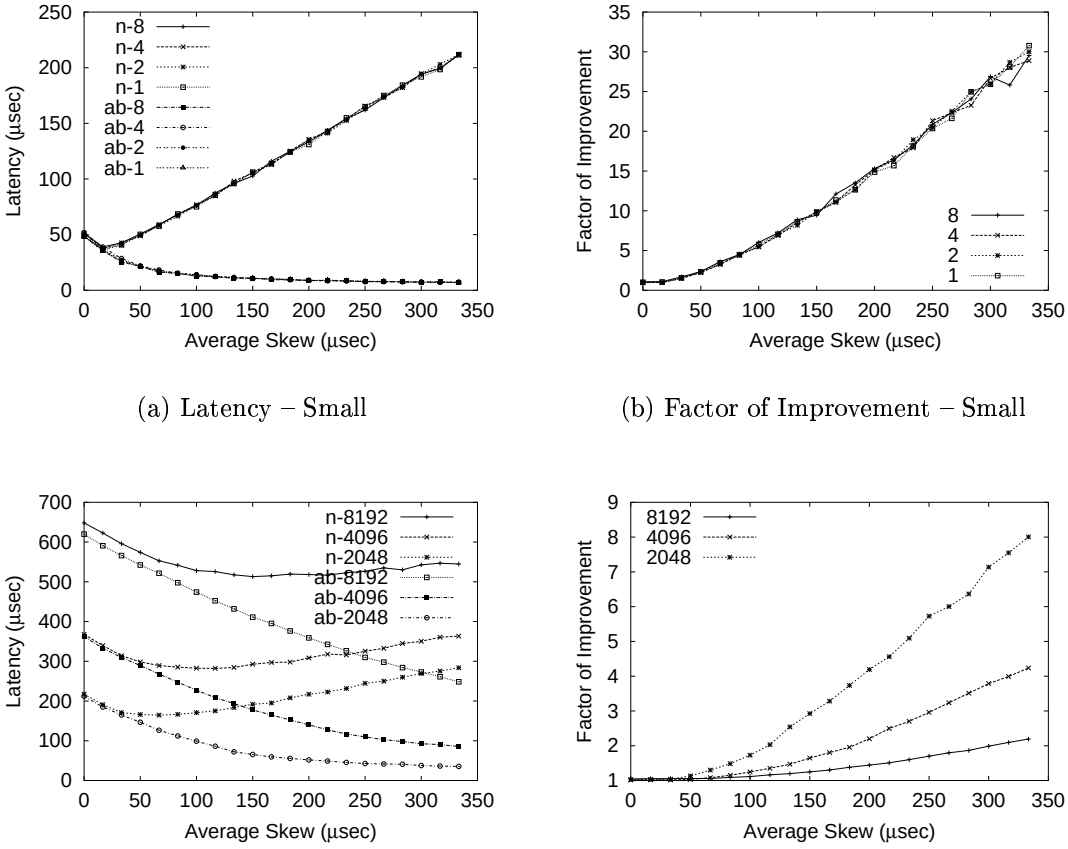
GM does not have the capability to generate signals when a message is received. We modified GM to add this capability. Since performing an interrupt for every incoming message would have a severe impact on performance, we wanted to perform interrupts only when necessary. We did this by defining a new packet type in GM. Only the reception of these packets generates a signal. This way the sender of a message can specify that a signal will be generated for the receiving process when the message is received.

We also allow a process to disable the signal generation at the user level. We added a flag to the data structure at the NIC which is mapped into the process' address space. The process simply has to write to the flag to enable or disable signal generation. Since this flag is located in NIC memory any accesses by the host will go over the PCI bus. This is considerably slower than accessing local memory, and may interfere with other PCI traffic. For this reason care must be taken when accessing this flag not to adversely impact system performance.

We used the signaling capability that we added to GM to generate signals for sending broadcast messages to non-leaf nodes. This limits the number of interrupts to only those cases where it could benefit. Furthermore, we disabled signaling at the NIC whenever the process calls `MPI_Bcast()` or whenever the process is waiting for a receive. This way, if the process is already polling for a message, there is no need to generate a signal to have the broadcast message processed. By eliminating as many interrupts as possible, we reduce the impact of using a signal handler while giving us the advantages of application-bypass broadcast.

6.3 Experimental Results

We performed our evaluation on a 32 node cluster consisting of 16 700MHz quad-SMP Pentium III nodes with 66MHz/64 bit PCI slots, and 16 1GHz dual-SMP



(a) Latency – Small

(b) Factor of Improvement – Small

(c) Latency – Large

(d) Factor of Improvement – Large

Figure 6.4: Average latency of MPI_Bcast function on 32 nodes. Small messages sizes are 1, 2, 4, and 8 bytes, and large message sizes are 2048, 4096, and 8192 bytes for non-application-bypass MPICH (n) and application-bypass MPICH (ab)

Pentium III nodes with 33MHz/32 bit PCI slots. The cluster was connected using a Myrinet-2000 network. The network consists of 28 PCI64B cards with 133MHz LANai 9.1 processors and 4 PCI64C cards with 200MHz LANai 9.2 processors and are connected using fiber cables to a 32 port switch. Each of the nodes ran the 2.4.18 Linux kernel. Our tests were performed using GM version 1.2.5.1 and MPICH version 1.2.4..8a which are the same versions as our modified GM and MPICH.

We evaluated our implementation using micro-benchmarks. The first micro-benchmark compares the average time to perform `MPI_Bcast()`. In this micro-benchmark, the root process calls `MPI_Bcast()` while the other processes perform a delay loop, followed by a call to `MPI_Bcast()`. This test is performed 1,000 times with an `MPI_Barrier()` being called before each test. The number of iterations a process performs in the delay loop is chosen randomly, between 0 and a maximum delay value, by each process each time the test is performed. Notice that increasing the maximum delay value, increases the skew between the processes.

Since we used a heterogeneous system, we wanted to normalize the delay loops. We did this by having each process count how many iterations it can compute in 50µs. The maximum delay value was then incremented by that many iterations. For the 700 MHz machines, this was about 17,500 iterations, while on the 1GHz machines this was about 25,000. The graphs show the *average* skew time in microseconds for convenience.

Figure 6.4 shows the average time each process spent in the `MPI_Bcast()` call over 32 processes. Figure 6.4(a) shows these results for 1, 2, 4 and 8 byte messages. The graph does not show much differentiation between the message sizes, however a large difference is seen between the application-bypass MPICH and non-application-bypass MPICH. Notice that as the skew increases, the average time spent in `MPI_Bcast()` by the non-application-bypass MPICH increases when the average skew is larger then 17µs. This is as we expected because as the skew between processes increases more processes are being delayed longer waiting for one of their ancestors to call `MPI_Bcast()`. However, for application-bypass MPICH, we see that the average time spent in `MPI_Bcast()` actually decreases as the skew increases. We don't see an increase as the skew increases as we did with the non-application-bypass MPICH because even if a non-leaf process is performing the delay loop when a broadcast message is received, the process will be interrupted and the broadcast operation will be allowed to proceed. The reason why the average time spent in `MPI_Bcast()` actually decreases is because as the time each process spends in the delay loop increases, the probability that the broadcast message has arrived and that the broadcast operation has completed before the process calls `MPI_Bcast()` increases. If the broadcast message has arrived before the process calls `MPI_Bcast()`, then all that needs to be performed in the `MPI_Bcast()` function is to copy the received data to the final memory location. Notice that when the average skew is 17µs, the time the non-application-bypass MPICH spends in `MPI_Bcast()` decreases compared to when there is no skew. This is

because some of the time that processes lower down in the broadcast tree would spend waiting for the broadcast message to propagate down the tree is overlapped with the delay loop. Although the broadcast message may be delayed because a process higher up in the tree is delayed, this delay is smaller than the time which is overlapped.

Figure 6.4(b) shows the factor of improvement of performing broadcasts using application-bypass MPICH over non-application-bypass MPICH for small messages. The graph shows a factor of improvement of up to 30 for the time spent in `MPI_Bcast()` when the average skew is 333µs and when broadcasting a 1 byte message. The improvement for 2, 4 and 8 byte messages is similar.

We performed the same evaluations using large messages. Figure 6.4(c) shows the results for 2K, 4K and 8K messages. Again we see that the time spent in `MPI_Bcast()` for application-bypass MPICH processes decreases as the skew increases, while for non-application-bypass MPICH processes the time increases once the skew is large enough that the effect of the overlap of the broadcast and delay loop is no longer seen. Notice that even when the skew is small or even zero, the application-bypass MPICH processes spend less time in `MPI_Bcast()` than the non-application-bypass MPICH processes. We believe that this is because even when the skew is zero, and the processes spend no time in the delay loop, the processes are still skewed slightly due to the nature of a distributed system. Even though a `MPI_Barrier()` is called before each test, not all processes will leave the barrier at exactly the same time. It is possible that some processes may receive the broadcast message while still performing the barrier. In application-bypass MPICH, when the message is to be forwarded to the child processes, the data will be copied from the receive buffer into the send buffers and the messages are sent to the children. In non-application-bypass MPICH, the message is copied into the unexpected queue. When `MPI_Bcast()` is called, the data is copied out of the unexpected queue and into the final memory location, then the data is copied from this memory location and into the send buffers to be sent to the child processes. Notice that the non-application-bypass MPICH has an extra memory copy in the critical path, which explains why we see a larger difference for larger message sizes when the skew is small.

Figure 6.4(d) shows a factor of improvement in the time spent in the `MPI_Bcast()` function of up to 8 for 2K messages, up to 4.2 for 4K messages, and 2.2 for 8K messages when the average skew is 333µs.

Just considering the time a process spends in `MPI_Bcast()` does not consider the time the application-bypass MPICH spends performing the operation when a broadcast message is received before the call to `MPI_Bcast()` is made. In order to evaluate the impact of performing the broadcast operation asynchronously and of the associated interrupts on the computation, we first timed the delay loop when there are no incoming broadcast messages. Then we timed the total time the process spends in the delay loop and the `MPI_Bcast()` call, and subtracted off the time it would have spent in the delay loop had there been no incoming broadcast messages.

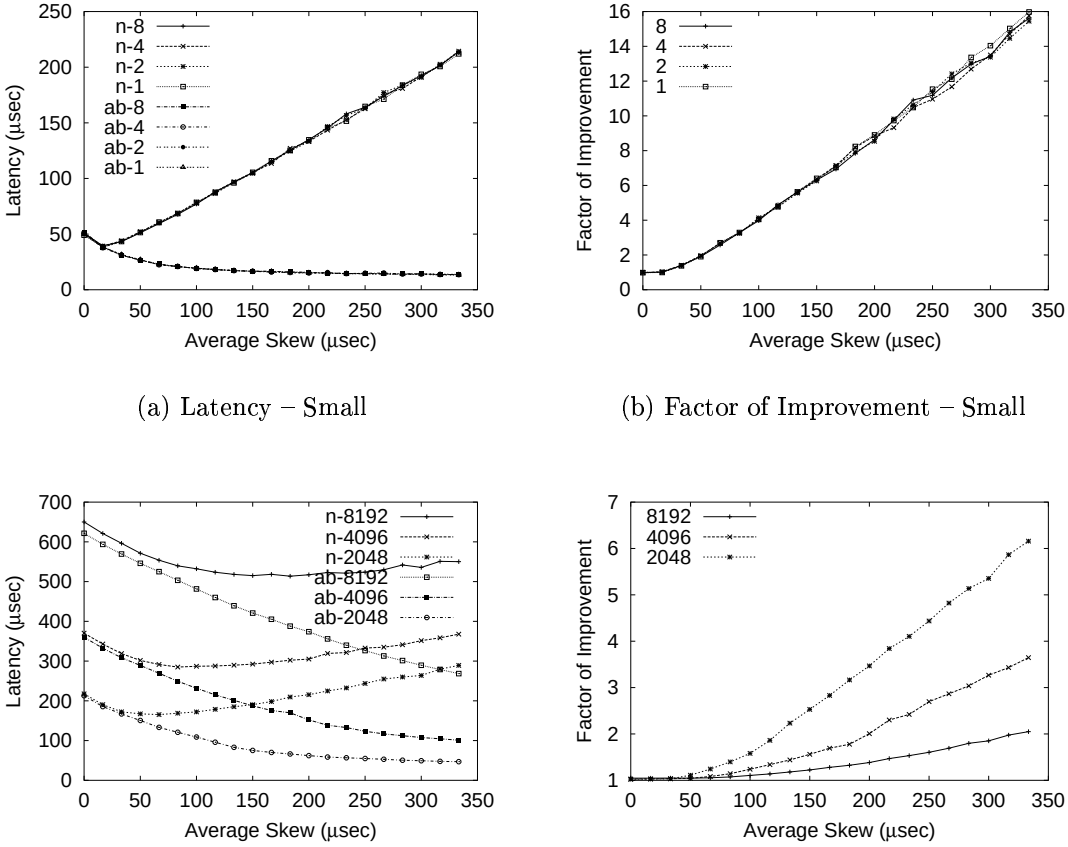
What is left is the time the process spends broadcasting. Figure 6.5 shows the results of these tests for 32 processes. Again the graph for small messages, Figure 6.5(a), does not show much difference between the different message sizes, but a significant difference is seen between the application-bypass MPICH and non-application-bypass MPICH. Notice that these results are very similar to those for just the time spent in `MPI_Bcast()`. There is about a 6µs overhead in the application-bypass case for processing broadcast messages by the signal handler for small messages. Figure 6.5(b) shows a factor of improvement of up to 16 for application-bypass MPICH processes over non-application-bypass MPICH processes. This is a significant improvement over non-application-bypass MPICH.

Figure 6.5(c) shows the results of the same test for large messages. As with the small messages, the results are very similar to the results for just the `MPI_Bcast()` function. For large messages there is about a 11µs, 15µs, and 20µs overhead for processing 2K, 4K, and 8K broadcast messages in the signal handler, respectively. Figure 6.5(d) shows factors of improvement of up to 2 for 8K messages, 3.6 for 4K messages, and 6.2 for 2K messages. Again, these are significant improvements.

In order to see how application-bypass broadcast can benefit systems of different sizes, we performed a test similar to the previous, except we used an average skew of 333µs and varied the number of processes. As before, for 32 processes, we used both the 700MHz machines and 1GHz machines, but for 16 and fewer processes, only the 700MHz machines were used. Figure 6.6 shows these results. Notice that for small messages, in Figure 6.6(a), as the system size increases the time taken by the application-bypass MPICH processes remains almost constant. This is because for all but the two process case, all of the delay of propagating the message down the broadcast tree is overlapped by the delay loop. The increase in time seen in the non-application-bypass MPICH results as the number of nodes increases is due primarily to process skew. As the number of processes participating in the broadcast increases, the number of processes waiting in `MPI_Bcast()` for an ancestor to finish the delay loop and perform the broadcast also increases. We see a similar effect in Figure 6.6(c) for large messages. The time increases much slower as the system size increases for application-bypass MPICH versus non-application-bypass MPICH. These results indicate that the effects of process skew become more severe as system size increases. Furthermore, they indicate that an application-bypass approach is critical for dealing with process skew allowing collective communications operations to be scalable.

6.4 Summary

We have described our design implementation of application-bypass broadcast using NIC-based primitives and evaluated our implementation. Our evaluation shows



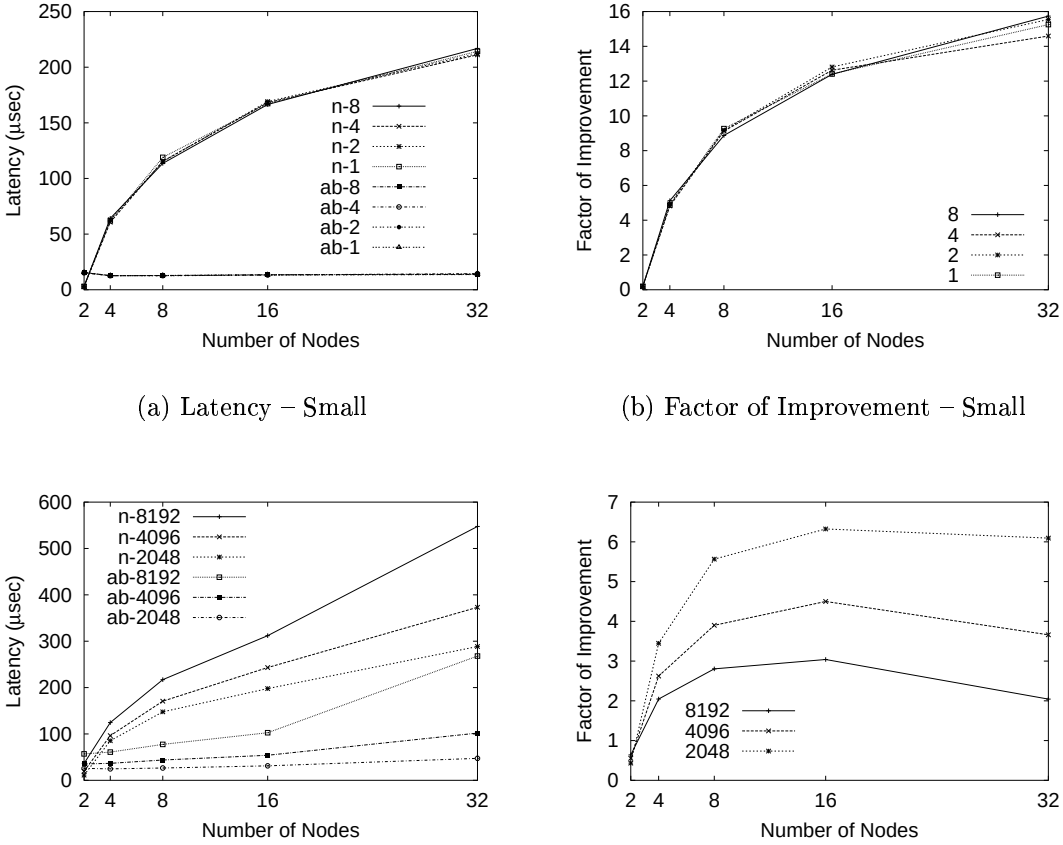
(a) Latency – Small

(b) Factor of Improvement – Small

(c) Latency – Large

(d) Factor of Improvement – Large

Figure 6.5: Average latency of signal handler and MPI_Bcast function on 32 nodes. Small messages sizes are 1, 2, 4, and 8 bytes, and large message sizes are 2048, 4096, and 8192 bytes for non-application-bypass MPICH (n) and application-bypass MPICH (ab)



(a) Latency – Small

(b) Factor of Improvement – Small

(c) Latency – Large

(d) Factor of Improvement – Large

Figure 6.6: Average latency of signal handler and MPI_Bcast function with a average skew of 333µs for various number of processes. Small messages sizes are 1, 2, 4, and 8 bytes, and large message sizes are 2048, 4096, and 8192 bytes for non-application-bypass MPICH (n) and application-bypass MPICH (ab)

that an application-bypass broadcast is not as sensitive to process skew as non-application-bypass broadcast. In fact, using the application-bypass broadcast, we have seen a factor of improvement of up to 16 when processes are skewed. Furthermore we see that in a non-application-bypass broadcast the effects of process skew increase as the system size increases. We note that while process skew can be reduced by careful design of parallel programs and close control of the computing environment, we believe that process skew cannot be eliminated altogether. For this reason, we believe that the use of NIC-based primitives to support application-bypass collective communication operations is critical to improving the scalability of those operations, and of the system in general.

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In this dissertation we investigated the use of programmable NICs to improve cluster performance. We approached this problem by focusing on improving the performance, scalability, and tolerance to process skew of synchronization operations and collective communication operations through the use of NIC-based operations and NIC-based primitives. Our research has shown that NIC-supported operations offer improved performance over the conventional host-based implementation. Specifically, the NIC-supported operations offered improved latency, scalability, host processor utilization, and better tolerance to process skew.

7.1 Summary of Research Contributions

We presented our design, implementation and evaluation of NIC-supported broadcast/multicast, barrier synchronization, reduction, atomic remote memory operations, and application-bypass broadcast. Below, we summarize the contributions and results of this dissertation.

7.1.1 NIC-Assisted Broadcast/Multicast

In Chapter 2 we presented a new NIC-based multi-send primitive which we used to improve the performance of broadcast/multicast operations. The primitive uses the NIC to transmit multiple copies of a packet to different destinations. We also presented a method for computing a multicast tree which would be optimal in terms of latency.

When comparing the multi-send to conventional host-based iterative sends, we saw up to a 3.51 factor of improvement for 16K messages. When the primitive was used to support multicast using a multicast tree, we saw up to a 1.85 factor of improvement for 8K messages and 16 nodes.

7.1.2 NIC-Based Barrier Synchronization

We presented our NIC-based barrier operation in Chapter 3. In this chapter, we described design challenges, and the implementation of the operation by modifying the GM communication subsystem. We also presented how the MPICH middleware library was modified to use the more efficient NIC-based barrier over the conventional host-based barrier which was implemented using point-to-point messages.

An extensive evaluation was performed comparing the performance of the NIC-based barrier to the host-based barrier, at both the communication subsystem layer, as well as the programming models layer, using two generations of NICs, the 33MHz LANai 4.3 NICs and 66MHz LANai 7.2 NICs, which have different performance characteristics.

When comparing the performance of the barriers at the programming models level, using MPICH, we found a 2.09 factor of improvement for 16 nodes using the LANai 4.3 NICs and a 2.22 factor of improvement for 8 nodes using the 66 MHz LANai 7.2 NICs. Furthermore, the factor of improvement increased with the number of participating nodes, indicating that the NIC-based barrier scales better than the host-based barrier.

We also evaluated the impact of the NIC-based barrier on the granularity of computation. We found that for a program to have a 0.90 factor of efficiency using the LANai 4.3 NICs, at least 1831.98µs of computation must be performed per barrier if the host-based barriers are used, but only 1023.82µs if a NIC-based barrier is used. This value is 44% lower than for the host-based barrier. So, using the NIC-based barrier allows for finer grained programs without lowering the efficiency. We noticed that the NIC-based barrier is more sensitive to variation in arrival times than the host-based barrier. However, the NIC-based barrier always performed better than the host-based barrier. To evaluate the impact of using the NIC-based barrier on applications, we used synthetic applications. We found up to a 1.93 factor of improvement when using NIC-based barriers versus using a host-based barrier. This indicates that using the NIC-based barrier in applications which perform many barrier calls will deliver significant performance benefits.

7.1.3 NIC-Based Reduction

Chapter 4 presents our design, implementation and evaluation of a NIC-based reduction operation. We designed our NIC-based reduction operation to perform integer and floating point operations, and implemented it by modifying the GM communication subsystem. The basic operation showed a factor of improvement of up to a 1.19 for integer reduction and 1.06 for floating-point reduction. We also give evidence that NIC-based reduction will perform better than host-based reduction in larger systems.

The real benefit of this operation is the fact that it allows better overlap of computation at the host with the reduction operation at the NIC. We saw up to a 2.7 factor of improvement in processor utilization when using NIC-based reduction for integer operations and a 2.3 factor of improvement when using floating point operations. The NIC-based implementation also allows the operation to bypass the application making it more tolerant to process skew than the host-based implementation. In the presence of process skew NIC-based reduction gives a 4.5 factor of improvement in processor utilization over host-based reduction. We also noticed that when the system size increases, the effect of the skew impacts host-based reduction much more than the NIC-based reduction. This indicates that NIC-based reduction would greatly improve the scalability of certain applications.

7.1.4 NIC-Based Atomic Remote Memory Operations

We designed and implemented NIC-based atomic remote memory operations by modifying the GM communication subsystem. This work was presented in Chapter 5. The NIC-based implementation allows atomic operations to be performed at a remote node without intervention of the remote node's host processor.

We evaluated our implementation, comparing it with a conventional host-based implementation. Our design included a method for serializing access to host memory to ensure the atomicit of the operation being performed at the NIC. We found up to a 1.25 factor of improvement for the compare&swap operation when comparing the best NIC-based implementation to the best host-based implementation. We used atomic remote memory operations to implement distributed locking. The locking algorithm implemented with the NIC-based operations gave up to a a 2.6 factor of improvement over the implementation with host-based operations. The NIC-based operations gave better host processor utilization, and NIC processor utilization. This means that the NIC-based implementation could handle more atomic requests than the host-based implementation before the NIC processor was saturated. This work demonstrates the potential for designing NIC-based operations which would allow asynchronous operations to be performed with minimal impact on host processor utilization.

7.1.5 NIC-Support for Application-Bypass Broadcast

Some degree of process skew is inevitable in cluster environments. We have seen that the effect of process skew on collective communication operations which do not bypass the application increases with system size. With the increase in size of modern cluster systems to thousands of nodes, the effect of process skew must be addressed. In Chapter 6, we describe how we used NIC-based primitives to implement an application-bypass broadcast operation in the MPICH middleware library.

We evaluated our implementation and saw a factor of improvement of up to 16 when processes are skewed. We noticed that as system size increased, the effect

of process skew increased for the conventional broadcast, but the application-bypass broadcast operation showed little to no effect. This research indicates that implementing collective communication operations in an application-bypass manner is critical for scalability in large systems.

7.2 Future Research Directions

As technology improves, NIC processors are becoming more powerful. This gives the opportunity for improved performance for NIC-based operations, as well as the ability to implement more complex operations. This leaves many interesting research directions to pursue. Below we describe some of these areas of future research.

Non-blocking or split-phase collective communication operations – In many middleware libraries collective communication operations are blocking operations. In a blocking operation, when the application calls the function initiating the operation, the function does not return until the operation has completed. This makes overlapping collective communication operations with computation difficult, expecially for operations such as all-reduce and all-gather, where each process supplies data to the operation and and receives data from the operation. If the operation is implemented as a NIC-based operation, after a process initiates the operation, it must wait idle until the operation completes, and it receives the result of the operation. In a non-blocking or split-phase implementation, the process can initiate the operation, then proceed with other computation which does not depend on the result. When the process finally needs the result, it can then wait for the operation to complete. Depending on how much computation can be performed that doesn't need the result of the operation, some or all of the operation can be overlapped with useful computation.

Research needs to be done to determine how best to incorporate non-blocking collective operations into existing middleware libraries, and how to modify existing applications to make best use of these operations.

Using fast-trap interrupts for NIC-based reduction operations – NIC processors typically lack support for floating-point operations. For this reason, floating-point operations must be emulated in software at the NIC. This severely impacts performance of reduction operations. One solution to this would be to have the host processor perform the floating-point operations, while the NIC performs the rest of the reduction operation. The NICs would exchange messages, and collect the data on which the arithmetic operation will be performed. The NIC will then send an interrupt to the host, and the host would perform the operation. To do this efficiently, a fast-trap interrupt should be used. A fast-trap interrupt is a light-weight interrupt, which doesn't do a full context switch, such as flushing page tables, performing bottom-half handlers, etc. Because a

full context switch is not performed, the fast-trap handler would not be able to modify kernel data structures, or access process address spaces, however, it would be able to read data from the NIC using a hardware address supplied by the NIC, perform the floating-point operations and write the result back down the the NIC. Research should be done to investigate the potential advantages of using such an approach, especially for reduction operations data with several elements.

Other NIC-based collective operations – We have implemented NIC-spported broadcast, barrier and reduction. Further work needs to be done so provide support for other collective operations, such as all-reduce, all-to-all, all-gather, etc.

NIC support for software distributed shared memory systems – As we demonstrated with NIC-based atomic remote memory operations, NIC-based operations can be used to eliminate or reduce the need for a server process. Such an approach can also be used for supporting operations in software distributed shared memory (SDSM) systems. Page invalidation, optimistic page fetching, and distributed locking are examples of operations which could benefit from NIC support. By implementing such opertations at the NIC, the host processor need not be interrupted, giving better processor utilization by the computational thread.

NIC support for high-performance distributed filesystems – As with SDSM systems, high-performance distributed filesystems can also benefit from NIC support. Again, eliminating the interaction with the server thread improves performance and processor utilization. Research needs to be done to investigate how best to use NIC support for locking of file regions, and optimistic access to cached file data.

BIBLIOGRAPHY

[1] S. Araki, A. Bilas, C. Dubnicki, J. Elder, K. Konishi, and J. Philbin. User-space communication: A quantitative study. In *Proceedings of the 1998 SC'98 Conference*, November 1998.

[2] D. H. Bailey, E. Barszcz, L. Dagum, and H.D. Simon. NAS Parallel Benchmark Results. Technical Report 94-006, RNR, 1994.

[3] P. Balaji, P. Shivam, P Wycoff, and D. K. Panda. High performance user-level sockets over gigabit ethernet. In *Cluster 2002*, September 2002.

[4] A. Bar-Noy and S. Kipnis. Designing broadcast algorithms in the postal model for message-passing systems. *Mathematical Systems Theory*, pages 381–390, September 1994.

[5] Michael Barnett, David G. Payne, Robert A. van de Geijn, and Jerrell Watts. Broadcasting on meshes with worm-hole routing. *Journal of Parallel and Distributed Computing*, 35(2):111–122, 1996.

[6] Donald J. Becker, T. Sterling, D. Savarese, E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A Parallel Workstation for Scientific Computation. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pages 11–14, 1995.

[7] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Proceedings of the 27th International Conference on Parallel Processing (ICPP '98)*, pages 381–390, August 1998.

[8] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11), November 1998.

[9] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[10] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. Seizovic, and W. Su. Myrinet - a gigabit per second local area network. In *IEEE Micro*, February 1995.

[11] Ron Brightwell, Rolf Riesen, Bill Lawry, and A. B. Maccabe. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC)*, April 2002.

[12] J. Bruck, L. De Coster, N. Dewulf, C. Ho, and R. Lauwereins. On the design and implementation of broadcast and global combine operations. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):256–265, March 1996.

[13] D. Buntinas and D. K. Panda. NIC-based reduction in Myrinet clusters: Is it beneficial? In *Proceedings of the Workshop on Novel Uses of System Area Networks (SAN)*, February 2003.

[14] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan. Broadcast/Multicast over Myrinet using NIC-Assisted Multidestination Messages. In *Proceedings of Int'l Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC)*, pages 115–129, 2000.

[15] D. Buntinas, D. K. Panda, and W. Gropp. NIC-based atomic remote memory operations in Myrinet/GM. In *Workshop on Novel Uses of System Area Networks (SAN-1)*, February 2002.

[16] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001, (IPDPS)*, April 2001.

[17] D. Buntinas, D. K. Panda, and P. Sadayappan. Performance benefits of NIC-based barrier on Myrinet/GM. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS '01*, April 2001.

[18] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle. Dynamic Barrier Architecture for Multi-mode Fine-grain Parallelism using Conventional Processors. In *International Conference on Parallel Processing*, Aug 1994. to appear.

[19] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware-Software Approach*. Morgan Kaufmann, March 1998.

[20] D. Dai and D. K. Panda. Building Efficient Limited Directory-Based DSMs: A Multidestination Message Passing Based Approach. Technical Report OSU-CISRC-4/96-TR21, Dept. of Computer and Information Science, The Ohio State University, Apr 1996. Symposium on Parallel and Distributed Processing (SPDP), under review.

[21] R. A. Van de Geijn. On Global Combine Operations. *Journal of Parallel and Distributed Computing*, 22:324–328, 1994.

[22] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[23] R. Gupta. The Fuzzy Barrier: A Mechanism for the High Speed Synchronization of Processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, 1989.

[24] John Hauser. SoftFloat. http://www.cs.berkeley.edu/~jhauser/arithmetic/SoftFloat.html.

[25] Y. Huang and P. K. McKinley. Efficient Collective Operations with ATM Network Interface Support. In *Proceedings of the International Conference on Parallel Processing*, pages I:34–43, Chicago, IL, Aug 1996.

[26] InfiniBand trade association, InfiniBand architecture specification, volume 1, release 1.0. http://www.infinibandta.com.

[27] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of Supercomputing (SC2001)*, November 2001. Available from http://www.sc2001.org/papers/pap.pap255.pdf.

[28] R. Kesavan and D. K. Panda. Optimal Multicast with Packetization and Network Interface Support. In *Proceedings of International Conference on Parallel Processing*, pages 370–377, Aug 1997.

[29] M. Lauria, S. Pakin, and A. Chien. Efficient layering for high speed communication: Fast Messages 2.x. In *Proceedings of the 7th High Performance Distributed Computing Conference (HPDC7)*, July 1998.

[30] Quadrics Supercomputers World Ltd. QsNet high performance interconnect. http://www.quadrics.com/website/pdf/qsnet.pdf.

[31] P. K. McKinley, Y.-J. Tsai, and D. F. Robinson. A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers. Technical Report MSU-CPS-94-35, Dept. of Computer Science, Michigan State University, 1994.

[32] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[33] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[34] Myricom GM Myrinet software and documentation. http://www.myri.com/scs/GM/doc/gm_toc.html, 2000.

[35] Netgear. http://www.netgear.com.

[36] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A non-uniform-memory-access programming model for high performance compuers. *The Journal of Supercomputing*, 10:197–220, 1996.

[37] J. Nieplocha, R. J. Harrison, and R. L. Littlefield. Global arrays: A portable "shared memory" programming model for distributed memory computers. In *Supercomputing 94*, 1994.

[38] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99*, April 1999.

[39] Ranjit Noronha and Nael Abu-Ghazaleh. Early cancellation: An active NIC optimization for time-warp. In *16th Workshop on Parallel and Distributed Simulation*, May 2002.

[40] Ranjit Noronha and Nael Abu-Ghazaleh. Using programmable NICs for time-warp optimization. In *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.

[41] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively Parallel Processors. *IEEE Concurrency*, pages 60–73, April-June 1997.

[42] S. Pakin, M. Lauria, and A. A. Chien. High performance messaging on workstations: Illinois fast messages on myrinet. In *Proceedings of Supercomputing 95*, November 1995.

[43] Pallas MPI benchmarks - PMB, part MPI-1. ftp://ftp.pallas.com/pub/PALLAS/PMB/PMB-MPI1.pdf.

[44] D. K. Panda. Fast Barrier Synchronization in Wormhole k-ary n-cube Networks with Multidestination Worms. In *International Symposium on High Performance Computer Architecture*, pages 200–209, 1995.

[45] D. K. Panda. Global Reduction in Wormhole k-ary n-cube Networks with Multidestination Exchange Worms. In *International Parallel Processing Symposium*, pages 652–659, Apr 1995.

[46] D. K. Panda, S. Singal, and R. Kesavan. Multidestination Message Passing in Wormhole k-ary n-cube Networks with Base Routing Conformed Paths. *IEEE Transactions on Parallel and Distributed Systems*, 10(1):76–96, Jan 1999.

[47] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12:115–116, June 1981.

[48] Ian Philp and Yin-Ling Liong. The scheduled transfer protocol (ST). In *Workshop on Communication, Architecture and Applications for Network-Based Parallel Computing (CANPC-99)*, January 1999.

[49] P. Shivam, P. Wyckoff, and D. K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Int'l Conference on Supercomputing (SC '01)*, November 2001.

[50] P. Shivam, P. Wycoff, and D. K. Panda. Can user level protocols take advantage of multi-CPU NICs? In *International Parallel and Distributed Processing Symposium (IPDPS'02)*, April 2002.

[51] Abraham Silberschatz and Peter Galvin. *Operating System Concepts*. Addison Wesley, 4th edition, 1994.

[52] R. Sivaram, R. Kesavan, D. K. Panda, and C. B. Stunkel. Where to Provide Support for Efficient Multicasting in Irregular Networks: Network Interface or Switch? In *Proceedings of the 27th International Conference on Parallel Processing (ICPP '98)*, pages 452–459, August 1998.

[53] R. Sivaram, D. K. Panda, and C. B. Stunkel. Efficient Broadcast and Multicast on Multistage Interconnection Networks using Multiport Encoding. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1004–1028, October 1998.

[54] Sphinx parallel microbenchmark suite. http://www.llnl.gov/CASC/sphinx/sphinx.html.

[55] T. Sterling, J. Salmon, D. Becker, and D. F. Savarese. *How to Build a Beowulf*. MIT Press, 1999.

[56] C. B. Stunkel, R. Sivaram, and D. K. Panda. Implementing Multidestination Worms in Switch-Based Parallel Systems: Architectural Alternatives and their Impact. In *Proceedings of the 24th IEEE/ACM Annual International Symposium on Computer Architecture (ISCA-24)*, pages 50–61, June 1997.

[57] K. Verstoep, K. Langendoen, and H. Bal. Efficient Reliable Multicast on Myrinet. In *Proceedings of the International Conference on Parallel Processing*, pages III:156–165, Aug 1996.

[58] M. S. Warren, D. J. Becker, M. P. Goda, J. K. Salmon, and T. Sterling. Parallel supercomputing with commodity components. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 1372–1381, 1997.