DESIGNING HIGH PERFORMANCE AND SCALABLE MPI OVER INFINIBAND

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Jiuxing Liu, B.S., M.S.

* * * * *

The Ohio State University

2004

Dissertation Committee:

Professor Dhabaleswar K. Panda, Adviser

Professor Ponnuswamy Sadayappan

Professor Srinivasan Parthasarathy

Approved by

Adviser

Graduate Program in Computer and Information Science

ABSTRACT

Rapid technological advances in recent years have made powerful yet inexpensive commodity PCs a reality. New interconnecting technologies that deliver very low latency and very high bandwidth are also becoming available. These developments lead to the trend of *cluster computing*, which combines the computational power of commodity PCs and the communication performance of high speed interconnects to provide cost-effective solutions for computational intensive applications, especially for those grand challenge applications such as weather forecasting, air flow analysis, protein searching, and ocean simulation.

InfiniBand was proposed recently as the next generation interconnect for I/O and inter-process communication. Due to its open standard and high performance, InfiniBand is becoming increasingly popular as an interconnect for building clusters. However, since it is not designed specifically for high performance computing, there exists a semantic gap between its functionalities and those required by high performance computing software such as Message Passing Interface (MPI). In this dissertation, we take on this challenge and address research issues in designing efficient and scalable communication subsystems to bridge this gap. We focus on how to take advantage of the novel features offered by InfiniBand to design different components in the communication subsystems such as protocol design, flow control, buffer management, communication progress, connection management, collective communication, and multirail network support.

Our research has already made notable contributions in the areas of cluster computing and InfiniBand. A large part of our research has been integrated into our MVAPICH software, which is a high performance and scalable MPI implementation over InfiniBand. Our software is currently used by more than 120 organizations worldwide to build InfiniBand clusters, including both research testbeds and production systems. Some of the fastest supercomputers in the world, including the 3rd ranked Virginia Tech Apple G5 cluster, are currently powered by MVAPICH. Research in this dissertation will also have impact on designing communication subsystems for systems other than high performance computing and for other high speed interconnects.

I dedicate this dissertation to my wife, my parents and my sister.

ACKNOWLEDGMENTS

I would like to thank my adviser Prof. D. K. Panda for his guidance during my PhD study. I am grateful for his tremendous effort and time that he has dedicated to my dissertation as well as his patience and understanding.

I would also like to thank the other members of my dissertation committee, Prof. P. Sadayappan and Prof. S. Parthasarathy, for their valuable comments and suggestions.

I gratefully acknowledge the financial support provided by The Ohio State University, National Science Foundation (NSF), and Department of Energy (DOE).

I am grateful to Jiesheng Wu for his help and friendship. I have learned a lot from him on both technical and non-technical matters. It has always been a good experience working with him.

I am grateful to Dr. Mohammad Banikazemi, Dr. Bulent Abali, and Dr. Craig Stunkel of IBM T. J. Watson Research Center for their guidance and support during my summer intern and afterwards. I am especially grateful to Mohammad, who was also my senior student when he was at Ohio State. I would not have gone this far without his help.

I would also like to thank Dr. Pete Wyckoff of the Ohio Supercomputer Center and Prof. Jose Duato of the Universidad Politecnica de Valencia for their help and support during the course of my PhD study.

I am very fortunate to have worked with many excellent current and former members of the NOWLAB: Dr. Darius Buntinas, Dr. Hyun-Wook Jin, Sushmitha Kini, Balasubraman Chandrasekaran, Weikuan Yu, Weihang Jiang, Amith Mamidala, and Abhinav Vishnu. I am grateful for their valuable suggestions, friendship, and support.

I am also grateful to many other members of the NOWLAB, especially, Pavan Balaji, Lei Chai, Wei Huang, Sundeep Narravula, Ranjit Noronha, Gopalakrishn Santhanaraman, Shuang Liang, Sayantan Sur, and Karthikeyan Vaidyanathan, for their friendship and many helpful discussions on various technical and non-technical topics.

Finally, I would like to thank my family: my wife, my parents, and my sister. It is their love, understanding, and support that make my life worthwhile.

VITA

1975	Born – Chongqing, China
June 1997	B.S. Computer Science, Shanghai Jiao Tong University, China
June 1999	M.S. Computer Science, Shanghai Jiao Tong University, China
September 1999 – August 2000	University Fellow, The Ohio State University
September 2000 – May 2001	Graduate Research Associate, The Ohio State University
June 2001 – August 2001	Summer Intern, IBM T. J. Watson Research Center
September 2001 – May 2002	Graduate Research Associate, The Ohio State University
June 2002 – August 2002	Summer Intern, IBM T. J. Watson Research Center
September 2002 – June 2003	Graduate Research Associate, The Ohio State University
July 2003 – September 2003	Summer Intern, IBM T. J. Watson Research Center
October 2003 – June 2004	Graduate Research Associate, The Ohio State University
July 2004 – September 2004	Presidential Fellow, The Ohio State University

PUBLICATIONS

Research Publications

J. Liu, J. Wu, D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand", *International Journal of Parallel Programming,* June, 2004.

J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, P. Wyckoff, D. K. Panda, "Micro-Benchmark Performance Comparison of High-Speed Cluster Interconnects", *IEEE Micro,* January/February, 2004.

J. Liu, A. Vishnu, D. K. Panda, "Building Multirail InfiniBand Clusters: MPI-Level Designs and Performance Evaluation", *SuperComputing 2004 Conference (SC 04),* November, 2004, to be presented.

A. Mamidala, J. Liu, D. K. Panda, "Efficient Barrier and Allreduce over InfiniBand using InfiniBand Multicast and Adaptive Algorithm", *2004 IEEE International Conference on Cluster Computing (Cluster 04),* September, 2004, to be presented.

W. Jiang, J. Liu, H. Jin, D. K. Panda, D. Buntinas, R. Thakur, W. Gropp, " Efficient Implementation of MPI-2 Passive One-Sided Communication over InfiniBand Clusters", *Euro PVM/MPI 2004 Conference,* September, 2004, to be presented.

J. Liu, A. Mamidala, A. Vishnu, D. K. Panda, "Performance Evaluation of InfiniBand with PCI Express", *Hot Interconnect 12 (HOTI 04),* August, 2004.

J. Liu, A. Mamidala, D. K. Panda, " Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support", *Int'l Parallel and Distributed Processing Symposium (IPDPS 04),* April, 2004.

J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, B. Toonen, "Design and Implementation of MPICH2 over InfiniBand with RDMA Support", *Int'l Parallel and Distributed Processing Symposium (IPDPS 04),* April, 2004.

J. Liu, D. K. Panda, "Implementing Efficient and Scalable Flow Control Schemes in MPI over InfiniBand", *Workshop on Communication Architecture for Clusters (CAC 04),* Held in Conjunction with *Int'l Parallel and Distributed Processing Symposium (IPDPS 04),* April, 2004.

W. Jiang, J. Liu, H. Jin, D. K. Panda, W. Gropp, R. Thakur, "High Performance MPI-2 One-Sided Communication over InfiniBand", *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 04),* April, 2004.

J. Liu, D. K. Panda, M. Banikazemi, "Evaluating the Impact of RDMA on Storage I/O over InfiniBand", *3rd Annual Workshop on Novel Uses of System Area Networks (SAN-3),* Held in Conjunction with *The 10th International Symposium on High Performance Computer Architecture (HPCA-10),* February, 2004.

J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, D. K. Panda, "Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics", *SuperComputing 2003 Conference (SC 03),* November, 2003.

S. Kini, J. Liu, J. Wu, P. Wyckoff, D. K. Panda, "Fast and Scalable Barrier using RDMA and Multicast Mechanisms for InfiniBand-Based Clusters", *Euro PVM/MPI 2003 Conference,* September, 2003.

J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, P. Wyckoff, D. K. Panda, "Micro-Benchmark Level Performance Comparison of High-Speed Cluster Interconnects", *Hot Interconnect 11 (HOTI 03),* August, 2003.

J. Liu, J. Wu, S. P. Kinis, P. Wyckoff, D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand", *17th Annual ACM International Conference on Supercomputing (ICS 03),* June, 2003.

J. Liu, M. Banikazemi, B. Abali, D. K. Panda, "Design and Performance Evaluation of A Portable Client/Server Communication Middleware over System Area Networks", *2nd Annual Workshop on Novel Uses of System Area Networks (SAN-2),* Held in Conjunction with *The 9th International Symposium on High Performance Computer Architecture (HPCA-9),* February, 2003.

J. Wu, J. Liu, P. Wyckoff, D. K. Panda, "Impact of On-Demand Connection Management in MPI over VIA", *2002 IEEE International Conference on Cluster Computing (Cluster 02),* September, 2002.

M. Banikazemi, J. Liu, D. K. Panda, and P. Sadayappan, "Implementing TreadMarks over VIA on Myrinet and Gigabit Ethernet: Challenges, Design Experience, and Performance Evaluation", *2001 International Conference on Parallel Processing (ICPP 01),* September 2001.

M. Banikazemi, J. Liu, S. Kutlug, A. Ramakrishna, P. Sadayappan, H. Shah, and D. K. Panda, “VIBe: A Micro-benchmark Suite for Evaluating Virtual Interface Architecture (VIA) Implementations”, *Int'l Parallel and Distributed Processing Symposium (IPDPS 01),* April 2001.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

Computer Architecture	Prof. Dhabaleswar K. Panda
Software Systems	Prof. Mario Lauria
Computer Networking	Prof. Dong Xuan

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Today's distributed and high performance applications require high computational power as well as high communication performance. In the past few years, the computational power of commodity PCs has been doubling about every eighteen months. At the same time, network interconnects that provide very low latency and very high bandwidth are also emerging [71, 77]. This trend makes it very promising to build high performance computing environments by *clustering*, which combines the computational power of commodity PCs and the communication performance of high speed network interconnects. However, performance of the underlying hardware is not delivered to applications in traditional communication protocols such as TCP/IP due to their high protocol overhead, heavy kernel involvement and extra data copies in the communication critical path [15, 64].

Recently, the research and industry communities have been proposing and implementing many communication systems such AM [98], U-Net [97], VMMC [8] and FM [74] to address some of the problems associated with the traditional networking protocols. In these systems, the involvement of operating system kernel is minimized and the number of data copies is reduced. As a result, they provide much higher communication performance to the application layer. In the recent past, the Virtual Interface Architecture (VIA) [20, 17, 4] was proposed to standardize these efforts.

More recently, InfiniBand Architecture [37] has been proposed as the next generation interconnect for I/O and inter-process communication. In InfiniBand Architecture, computing nodes and I/O nodes are connected to the switched fabric through Channel Adapters (CAs). InfiniBand provides a Verbs interface which is similar to VIA. This interface is used by hosts to access communication functions provided by Host Channel Adapters (HCAs).

In the area of high performance computing, Message Passing Interface (MPI) [90, 29] has been the *de facto* standard for writing parallel applications. To achieve optimal performance in a cluster, it is very important to implement MPI efficiently on top of the cluster interconnect. High speed interconnects such as Myrinet [71] and Quadrics [81] were designed for high performance computing environments. As a result, their hardware and software were specially optimized to achieve better MPI performance [44, 66, 77]. Unlike Myrinet and Quadrics, InfiniBand was initially proposed as a generic interconnect for inter-process communication and I/O. Since it was not designed specifically for high performance computing, there exists a semantic gap between the communication interface of InfiniBand and that of MPI. However, besides its high performance, InfiniBand provides many novel features that can potentially benefit MPI design. These features include different communication semantics, multiple transport services, hardware multicast, communication management infrastructure, flexible completion and event handling mechanisms, Quality-of-Service, etc.

In this dissertation, we focus on how to design a communication subsystem that can bridge the gap between InfiniBand and high performance computing middleware layers such as MPI. In designing our communication subsystem, we aim to achieve the following goals:

1. *High performance.* Despite the semantics gap, our communication subsystem needs to preserve the performance of the underlying InfiniBand layer.

2. *High scalability.* Our design needs to scale with system size, despite that certain underlying components of InfiniBand may not be scalable.

Specifically, we investigate how to exploit different features in InfiniBand to address design issues in MPI. These include designing MPI communication protocols, flow control, collective communication, multirail network support, performance evaluation framework, as well as other issues such as buffer management, connection management and communication progress,

The remaining part of this dissertation is organized as follows: In Chapter 2, we provide background information of our research by introducing InfiniBand and MPI. In Chapter 3, we present the problem statement and methodology of our research. A basic MPI design over InfiniBand is described in Chapter 4. In Chapter 5, we describe how to implement efficient and scalable MPI flow control mechanisms over InfiniBand. In Chapter 6, we present a novel MPI design which is based on InfiniBand Remote Direct Memory Access (RDMA) operations. In Chapter 7, we discuss how to achieve fast and scalable MPI level broadcast over InfiniBand. We describe research issues in supporting multirail InfiniBand networks at the MPI level in Chapter 8. In Chapter 9, we present a comprehensive framework for evaluating the performance of different MPI implementations. Our MPI software package derived from our research and its impact are described in Chapter 10. In Chapter 11, we conclude and discuss some of the future research directions.

CHAPTER 2

BACKGROUND

Before we get to the details of the design of our communication subsystem, we provide some background information for both MPI and InfiniBand. In the following sections, we will first introduce MPI and its design issues. Then we provide an overview of the InfiniBand Architecture.

2.1 MPI Overview

Message Passing Interface (MPI) [90] was proposed as a standard communication interface for parallel applications. It specifies an Application Programming Interface (API) and its mapping to different programming languages such as Fortran, C and C++. Since its introduction, MPI has been implemented in many different systems and has become the *de facto* standard for writing parallel applications. The main communication paradigm defined in MPI is *message passing*. However, MPI has also been implemented in systems that supports shared memory [26, 36]. Therefore, parallel applications written with MPI are highly portable. They can be used in different systems as long as there are MPI implementations available.

2.1.1 MPI Point-to-Point Communication

In an MPI program, two processes can communicate using MPI point-to-point communication functions. One process initiates the communication by using MPI_Send function. The other process receives this message by issuing MPI_Recv function. Source or destination process needs to be specified in the functions. In addition, both sides specify a *tag*. A send function and a receive function match only if they have compatible tags.

MPI_Send and MPI_Recv are the most frequently used MPI point-to-point functions. However, they have many variations. MPI point-to-point communication supports different *modes* for send and receive. The mode used in MPI_Send and MPI_Recv is called *standard* mode. There are other MPI functions that support other modes such as *synchronous, buffered* and *ready* modes. Communication buffers specified in MPI_Send and MPI_Recv must be contiguous. However, there are also variations of MPI_Send and MPI_Recv functions that support non-contiguous buffers. Finally, any send or receive functions in MPI can be divided into two parts: one to initiate the operation and the other one to finish the operation. These functions are called *non-blocking* MPI functions. For example, MPI_Send function can be replaced with two functions: MPI_Isend and MPI_Wait. By using MPI non-blocking functions, MPI programmers can potentially overlap communication with computation, and therefore increase performance of MPI applications.

2.1.2 MPI Collective Communication

In addition to point-to-point communication functions, MPI offers collective communication functions that allow a group of processes to perform communication in a coordinated manner. Examples of MPI collective functions include MPI_Barrier, MPI_Bcast, MPI_Reduce, MPI_Alltoall, MPI_Allreduce, etc. The groups of processes participating in collective communication functions are specified by *communicators* in MPI. Collective communication functions not only provide simple and intuitive interfaces to programmers for performing these operations, but also give MPI designers and implementers more opportunity to optimize them.

2.1.3 MPI One-Sided Communication

Since its introduction, MPI has undergone many changes. The new MPI-2 standard [63, 28], which was proposed in 1996, brought many enhancements to the original MPI standard. One of these enhancements is *one-sided communication.*

In MPI One-Sided communication, data transfer operations are carried out only at one side, which specifies both data source and destination. Therefore, it looks more like remote memory operations. To guarantee mutual exclusion and completion of one-sided data transfer, MPI-2 also introduced synchronization operations. The original Send/Receive based communication in MPI is also called *two-sided communication.*

There are two kinds of MPI one-sided communication: *active target communication* and *passive target communication.* In both cases, only one side is involved in data transfer operations. However, in active target communication, both sides participate in synchronization operations. In passive target communication, even synchronization operations involve only one side.

We have carried out research on how to support MPI one-sided communication efficiently over InfiniBand in [48, 40, 39]. However, in this dissertation, we focus on two-sided communication in MPI.

2.2 MPI Design Issues

To support an efficient and scalable MPI implementation, the communication subsystem must address many design issues related to point-to-point communication and collective communication. In this section, we will provide general discussion of these design issues. In later parts of this dissertation, we will present more details in the context of InfiniBand Architecture.

2.2.1 Point-to-Point Communication

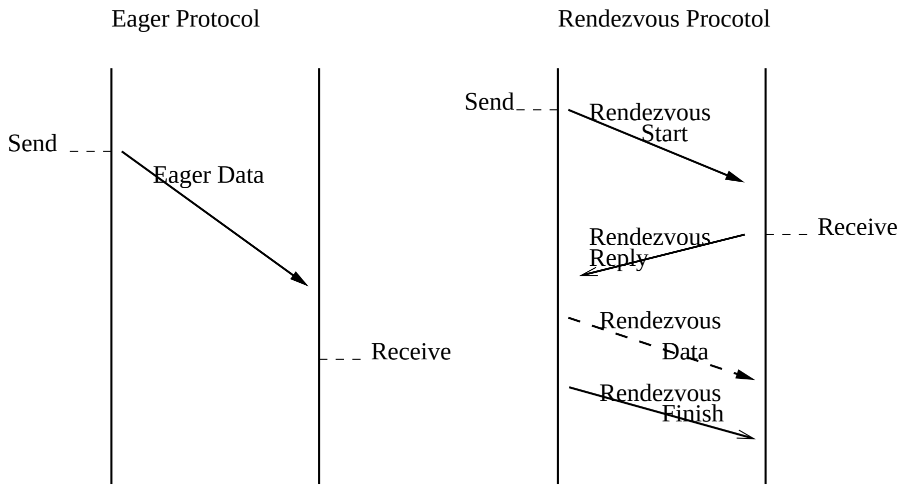

Figure 2.1: Eager and Rendezvous Protocols

Design issues in point-to-point communication include the following:

- Mapping MPI protocols to low level communication operations. MPI defines four different communication modes: *Standard*, *Synchronous*, *Buffered*, and *Ready* modes. Two internal protocols, *Eager* and *Rendezvous*, are usually used to implement these four communication modes. In Eager protocol, the message is pushed to the receiver side regardless of its state. In Rendezvous protocol, a handshake happens between the sender and the receiver via control messages before the data is sent to the receiver side. Eager and Rendezvous protocols are shown in Figure 2.1. Usually, Eager protocol is used for small messages and Rendezvous protocol is used for large messages. In these protocols, there are both data transfer operations and control messages that must be implemented by using low level communication operations. To obtain an efficient MPI implementation, two requirements must be met: First, the low level communication operations must provide good communication performance. Second, the mapping must have very low overhead.

- Communication Buffer Management. The implementation of MPI makes use of many internal communication buffers. These buffers serve multiple purposes. Some buffers are used to send control messages. Other buffers are used to send data messages. (In the latter case, the messages might need to be copied from user buffer.) To achieve optimal communication performance, it is necessary to manage these buffers efficiently. Specifically, the allocation and deallocation of buffers must be very fast. Another important issue is that the total amount of buffer space used. Ideally, the buffer space used should be small and increase gracefully with the size of the applications.

- Flow control. In order to receive a message, a certain amount of resource needs to be consumed at the receiver side. To prevent a fast sender from overwhelming a receiver, a flow control mechanism is needed. This mechanism should be able to slow down the sender when the receiver cannot keep up. However, in normal cases, it should impose very little overhead.

- Connection management. MPI communication assumes a fully connected topology. A process can send or receive messages from any other processes after initialization. However, at the low level communication layer, an explicit connection setup phase may need to be carried out before data transfer can happen. How to handle connection setup in an efficient and scalable manner is very important for an MPI implementation.

- Communication progress. In typical parallel applications, processes need to perform computation tasks in addition to communicating with other processes. Ideally, after being initiated, a communication operation should make progress independently without host intervention. In this way, the host can be freed from communication and computation and communication can overlap, resulting in better overall application performance. However, independent communication progress is not easy to achieve, especially for collective operations.

- Taking advantage of multirail networks. In multirail networks, processing nodes are connected with multiple separate networks. In large scale systems, network congestion may create a bottleneck for parallel applications. One way to address this issue is to use multiple communication connections which follow different paths between processes and schedule communication traffic across these connections. This can be regarded as a special type of multirail networks. By using multiple connections, we can also achieve higher bandwidth by striping large messages. In InfiniBand, multiple connections can be set up between a single pair of ports. However, we can also set up them on different ports or even different HCAs to avoid bottlenecks such as link bandwidth or I/O bus bandwidth.

2.2.2 Collective Communication

Collective communication algorithms have been studied extensively in the literature [94]. To implement collective communication operations efficiently, we must carefully design the communication protocols that map the algorithms to low level communication operations. If possible, we should take advantage of collective communication support in the interconnects to achieve both high performance and scalability.

When implementing collective communication operations, their interaction with point-to-point operations must also be considered. First, they should be implemented carefully so that semantics of either collective communication or point-to-point communication is not violated. Second, their implementation should not have an adverse impact on the performance of point-to-point communication.

2.3 InfiniBand Overview

The InfiniBand Architecture (IBA) [37] defines a high speed network for interconnecting processing nodes and I/O nodes. It provides the communication and management infrastructure for inter-processor communication and I/O. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Channel

Adapters (CA). Channel Adapters usually have programmable DMA engines with protection features. They generate and consume IBA packets. There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs sit on processing nodes. Their semantic interface to consumers is specified in the form of InfiniBand Verbs. The architecture of InfiniBand is shown in Figure 2.2.

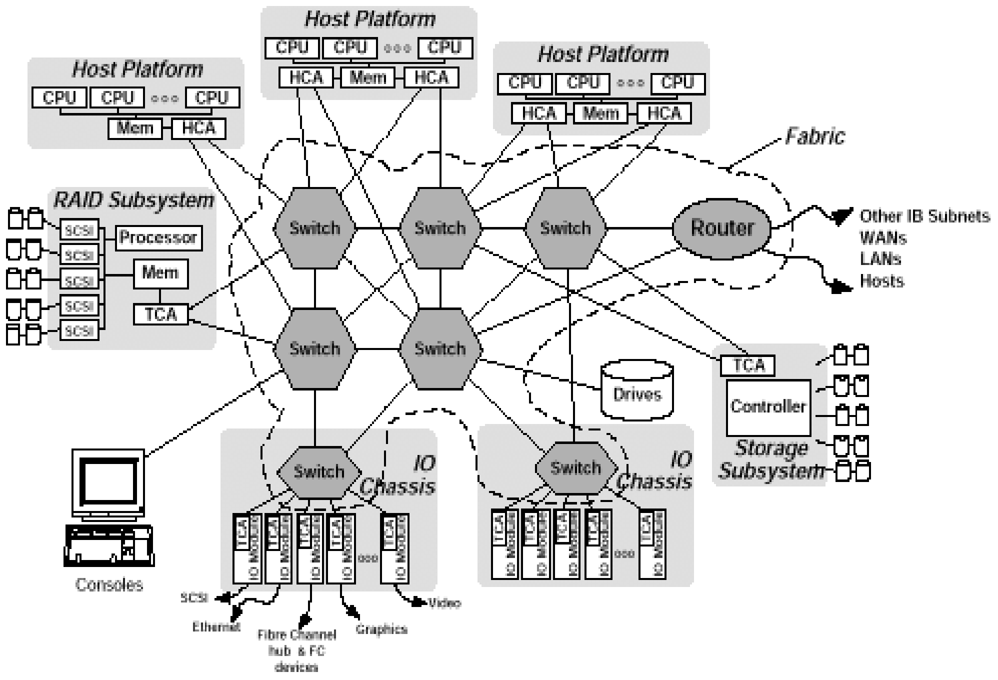

Figure 2.2: InfiniBand Architecture (Courtesy InfiniBand Trade Association)

2.3.1 Queue Pair Based Communication Model

The InfiniBand communication stack consists of different layers. The interface presented by channel adapters to consumers belongs to the transport layer. A queue pair based model is used in this interface. A *queue pair* in InfiniBand Architecture consists of two queues: a *send queue* and a *receive queue*. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication operations are described using Work Queue Requests (WQR), or *descriptors*, and submitted to the queue pairs. Once submitted, a Work Queue Request becomes a Work Queue Element (WQE). WQEs are executed by Channel Adapters. The completion of work queue elements is reported through Completion Queues (CQs). Once a work queue element is finished, a completion queue entry can be placed in the associated completion queue. Applications can check the completion queue to see if any work queue request has been finished. This process is illustrated in Figure 2.3.

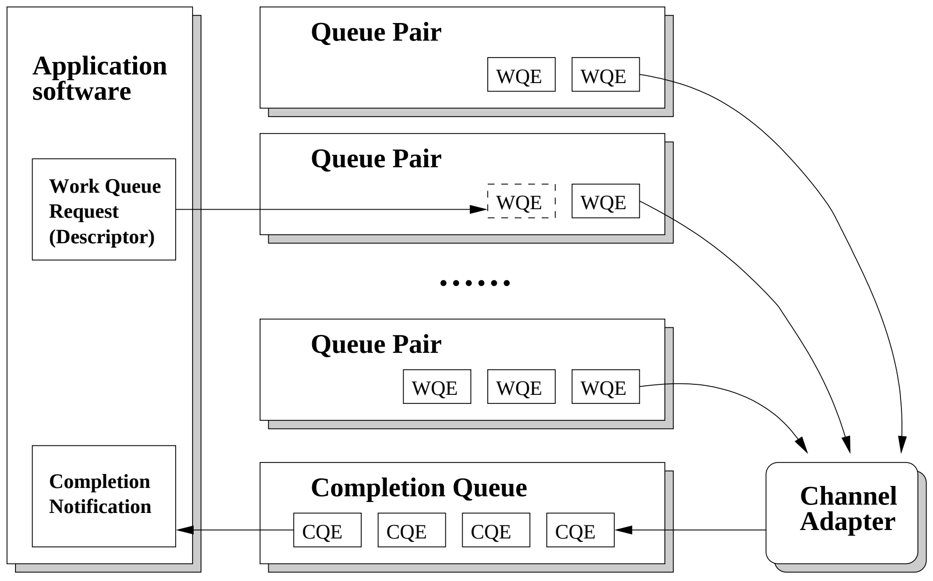

Figure 2.3: InfiniBand Queue Pairs and Completion Queue

In InfiniBand, before a buffer can be used for communication, it must be registered. After communication, the buffer can be de-registered. Buffer registration and de-registration are usually expensive operations.

2.3.2 Channel and Memory Semantics

InfiniBand Architecture supports both channel and memory semantics. In channel semantics, send/receive operations are used for communication. To receive a message, the programmer posts a receive descriptor which describes where the message should be put at the receiver side. At the sender side, the programmer initiates the send operation by posting a send descriptor. The send descriptor describes where the source data is but does not specify the destination address at the receiver side. When the message arrives at the receiver side, the hardware uses the information in the receive descriptor to put data in the destination buffer. Multiple send and receive descriptors can be posted and they are consumed in FIFO order. The completion of descriptors are reported through CQs.

In memory semantics, Remote Direct Memory Access (RDMA) operations are used instead of send and receive operations. These operations are one-sided and do not incur software overhead at the other side. The sender initiates RDMA operations by posting RDMA descriptors. A descriptor contains both the local data source addresses (multiple data segments can be specified at the source) and the remote data destination address. At the sender side, the completion of an RDMA operation can be reported through CQs. The operation is transparent to the software layer at the receiver side.

2.3.3 Transport Services

InfiniBand Architecture supports multiple classes of communication services at the transport layer. A queue pair can be configured for the following classes of services:

- Reliable Connection (RC)
- Reliable Datagram (RD)
- Unreliable Connection (UC)
- Unreliable Datagram (UD)
- Raw Datagram

In the Reliable Connection service, each queue pair can only communicate with one queue pair at each remote node. Before communication, a connection must be established between the local queue pair and the remote queue pair. By using mechanisms such as acknowledgment and retransmission, InfiniBand ensures that the connection is reliable. Unreliable Connection is similar to the Reliable Connection service. The difference is that reliability is not guaranteed.

Reliable Datagram and Unreliable Datagram both provide connectionless transport services. Therefore, a single queue-pair can communicate with multiple queue-pairs on remote nodes. In Reliable Datagram, reliability is provided by using *End-to-End Context*. UD service does not guarantee any reliability. Another restriction of UD is that the length of message size cannot exceed the Maximum Transfer Unit (MTU) of the InfiniBand network.

The purpose of Raw Datagram service is to provide interoperability of InfiniBand Architecture and other networks. Its usage is out of the scope of this dissertation.

2.3.4 Management Infrastructure

Unlike many other interconnects, InfiniBand Architecture has a comprehensive management infrastructure. InfiniBand networks usually consist of smaller networks called *subnets*. In each subnet, InfiniBand defines the methods for a subnet manager to discover and configure the nodes and manage the fabric. InfiniBand also provides *General Management Services*, which handle tasks such as connection setup, performance monitoring, configuration management, etc.

2.3.5 Hardware Multicast

InfiniBand provides hardware support for multicast. In some cases, this mechanism can greatly reduce communication traffic as well as latency and host overhead. InfiniBand also provides flexible mechanisms to manage multicast groups. However, multicast is only available for the Unreliable Datagram service. Therefore, tasks such as fragmentation, acknowledgment and retransmission may be needed on top of UD to make multicast work reliably for large messages.

2.3.6 Atomic Operations

As we have mentioned, InfiniBand provides the ability to access a remote node's memory with RDMA operations. However, accesses to the same memory location may lead to unpredicted results if they are not coordinated. Atomic operations provide stronger consistency guarantee. Therefore they can be used in these cases. InfiniBand supports two kinds of atomic operations: Fetch-and-Add and Compare-and-Swap. These operations are initiated in a way similar to RDMA operations.

2.3.7 Completion and Event Handling Mechanisms

In InfiniBand Architecture, the Completion Queue (CQ) serves as an efficient and scalable mechanism to report the completion of communication operations. However, there is also overhead with the handling of CQs and generation of CQ entries. At the sender side, InfiniBand provides a mechanism called *Unsignalled Completion* that reduces the overhead of completion by forcing certain operations not to generate completion entries.

In addition to generating CQ entries, InfiniBand can also invoke an event handler upon the completion of a communication operation. Event handler invocation usually involves interrupts and context switches. Therefore it is a relatively expensive operation. To give upper layer software more control over the invocation of event handler at the remote side, InfiniBand provides a mechanism called *Solicited Event.* In this mechanism, a send operation leads to the invocation of event handler at the remote side for the matching receive only when a "solicited event" bit is set in the request. By using this mechanism, the sender side can suppress many unnecessary event invocation and improve the performance.

2.3.8 End-to-End Flow Control

In InfiniBand, send operations match with receive operations posted at the receiver side in FIFO order. In cases where the sender is too fast, it may happen that there are not enough receive operations posted to match all the send operations. In this case, InfiniBand provides an End-to-End flow control mechanism which effectively slows down the sender transmission rate to let the receiver catch up. In addition, the sender can retry the send operations until corresponding receive operations are posted. The upper layer software can control this behavior by specifying parameters such as timeout value and retry delay for queue-pairs.

2.3.9 Quality-of-Service Support

InfiniBand has built-in Quality-of-Service (QoS) support that consists of three components: Virtual Lanes (VLs), Service Levels (SLs), and Service Level to Virtual Lane mapping. In an InfiniBand network, each physical link is divided into up to 16 Virtual Lanes. Each Virtual Lane has different QoS characteristics. At end-points, Service Levels are assigned to communication and packets are marked with their service levels. As packets travel through the network, they are assigned to different Virtual Lanes on each link according to SL to VL mapping. Researchers have shown that this mechanism can accommodate different Quality-of-Service Schemes [3].

CHAPTER 3

PROBLEM STATEMENT AND METHODOLOGY

3.1 Problem Statement

In this dissertation we focus on efficient support of two-sided communication in MPI. InfiniBand Architecture can provide high performance and scalability. However, due to the semantic gap between MPI and InfiniBand, this performance and scalability are not seen by MPI applications unless the communication subsystem that supports MPI is designed in an efficient and scalable manner. For example, InfiniBand supports both send/receive operations and RDMA operations. However, RDMA operations are more efficient than send/receive operations. Thus, to achieve optimal performance, it is desirable to take advantage of RDMA operations in InfiniBand. InfiniBand also provides a rich set of other features. A carefully designed communication subsystem should take advantage of these features to increase its performance and scalability. Therefore, the question we address in this thesis is:

How can we design an efficient and scalable communication subsystem to support MPI by taking advantage of the novel features of InfiniBand?

Given the MPI design components discussed in the previous chapter and the features of InfiniBand, there are many ways to design each component by using different features, as shown in Figure 3.1. However, there are also trade-offs in using many of the features. For example, hardware supported multicast can potentially reduce traffic volume and improve latency for MPI collective communication operations. However, since multicast is only supported in the Unreliable Datagram service, extra effort is needed to ensure reliability and in-order delivery. Thus, we need to understand these features and selectively use them.

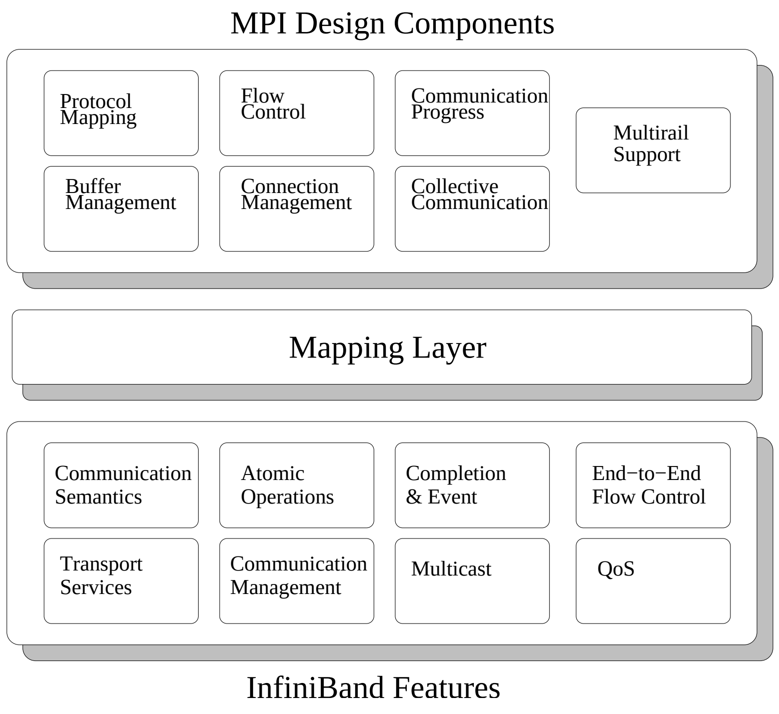

Figure 3.1: Design MPI Components Using InfiniBand Features

In this dissertation, We will investigate the benefits of InfiniBand features in each MPI design component as outlined below:

- **Implementing MPI communication protocols efficiently** — InfiniBand offers both send/receive and RDMA operations. For transferring large data messages in Rendezvous protocol, RDMA can be used because the target buffer addresses can be obtained through the handshake process. However, control messages and eager data messages seem to match better with send/receive operations because the message needs to be pushed to the other side even though the destination address is not known yet. Another advantage of send/receive operations is that the CQ mechanism provides an efficient and scalable mechanism for detecting incoming messages at the receiver side. In Chapter 4, we present an MPI design which follows this idea. This design is also documented in [56, 70]. However, as we will discuss, send/receive operations do not perform as well as RDMA operations. Furthermore, send/receive operations incur overhead of managing receive descriptors at the receiver side. To achieve better performance, we also explore using RDMA operations for eager data messages and control messages. There are several challenges in this approach. First, we need to know the target addresses of RDMA operations beforehand. Second, since RDMA operations are transparent to the remote side, we need to provide a scalable mechanism to detect the arrival of incoming messages. Some related issues have been discussed in the literature for interconnects that support remote memory access [87, 5, 36, 11]. However, by combining both send/receive and RDMA operations, we have a larger design space and have more opportunities to address the performance and scalability issues.

The two techniques we propose to address the above issues are *persistent buffer association* and *RDMA polling set*. In persistent buffer association, we have a buffer pool at the sender side and the receiver side respectively for each connection. Each buffer at the sender can only be RDMA written to its corresponding buffer at the receiver side. We keep using the buffer in a fixed, circular order so that the receiver always knows where to expect the next message. To detect incoming messages, the receiver needs to poll on all connections. This leads to scalability problems in applications with a large number of connections. To address this problem, we introduce the concept of RDMA polling set. Each receiver maintains such a set which contains a subset of all incoming connections. A sender uses RDMA operations only if the corresponding connection is in the receiver's RDMA polling set. Otherwise it falls back on send/receive operations. Therefore, the receiver only needs to poll the connections in the RDMA polling set for all incoming RDMA messages. For all other connections, it can poll the CQ for incoming messages. Based on the research results in [95], the polling set can be kept small to accommodate all connections for many MPI applications. Therefore, RDMA polling set combined with CQ can greatly reduce the polling time and increase scalability for large MPI applications. In Chapter 6, we present details of this RDMA based design and evaluate its performance. The results are also presented in [55, 52]

- **Implementing effective flow control mechanisms with minimum overhead** — Flow control is an important issue in communication subsystems. In InfiniBand Architecture, each send operation needs a receive to be posted at the remote side. If there is no receive posted, InfiniBand provides an end-to-end flow control mechanism that can reduce the rate of the sender and retry the transmission.

  Our MPI design can take advantage of this mechanism without implementing our own flow control. For each connection, we pre-post a number of receives.

When a message comes, we process the message and re-post the receive as soon as possible. In the case that the receiver cannot keep up with the rate of the sender, the end-to-end flow control mechanism automatically comes into play and makes sure all messages will eventually be reliably delivered. The above scheme eliminates the overhead of flow control at the MPI level. However, since we have very little control over the flow control behavior at the InfiniBand level, the scheme may result in unnecessary communication traffic or long communication delay in some cases. An alternative is to develop our own credit-based flow control mechanism. In this way, we have more flexibility. However, it may incur more run-time overhead than the previous scheme.

Another issue we need to study is how to decide the number of receive buffers pre-posted for each connections. In a static scheme, the number is the same for all connections and never changes during run-time. If this number is too small for a certain application, the flow control will be triggered frequently and reduce the communication performance. If the number is too large, buffer space may be wasted most of the time. Excessive buffer usage also significantly reduces the scalability of MPI applications since the number of buffers increases with the number of connections.

We will investigate the impact of different flow control schemes on MPI applications in Chapter 5. We show that dynamically deciding the number of buffer pre-posted will improve both performance and efficiency of buffer usage. The results of our research is also presented in [50].

- **Designing high performance and scalable collective communication protocols** — By default, MPI collective communication is implemented using MPI point-to-point communication. To improve the performance and scalability of collective communication operations, we can implement them directly by taking advantage of InfiniBand features. In particular, we study how to take advantage of hardware multicast to design high performance and scalable collective communication operations. We also study the use of RDMA operations in designing collective communication. With RDMA operations, the receiver overhead can be greatly reduced. In many cases, it can also reduce overhead by avoiding message header and receiver side message matching in point-to-point messages. Hardware multicast can greatly reduce the communication traffic and latency for those operations in which identical messages are sent to multiple processes. Our approach is to combine both of them to design high performance and scalable collective communication. For example, in barrier operations, we can use a two-phase approach as shown in Figure 3.2. In the first phase, RDMA operations are used to wait for all processes (gather). In the second phase, a multicast operation is used to inform all processes that the barrier operation is finished. We should note that to increase scalability, the first step can be done in a hierarchical manner if the number of processes is large.

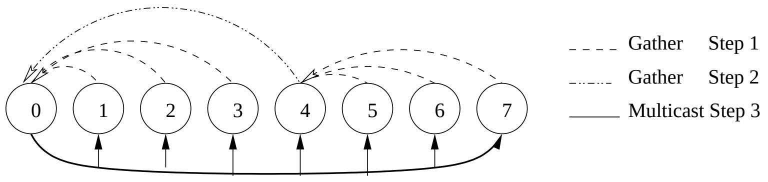

Figure 3.2: RDMA and Multicast Barrier Protocol

As a part of our previous work, we have already studied the use of RDMA operations and hardware multicast in designing efficient and scalable MPI_Barrier [43, 42]. In Chapter 7, we will extend this idea further and study how to support another important MPI collective operations MPI_Bcast. We investigate different multicast and RDMA based collective communication protocols and their effectiveness in improving both performance and scalability. The results are also presented in [49].

- **Taking advantage of multirail InfiniBand networks** — The basic idea of multirail networks is to have multiple independent networks (rails) to connect nodes in a cluster. Using multirail networks helps us in several ways. First, by carefully choosing the path or switching from one path to another, we can possibly avoid congested area in the network and improve communication performance. Second, by striping large messages across multiple connections, we can dramatically improve bandwidth and avoid bottlenecks in the network, at the link or at the host I/O bus. InfiniBand allows multiple connections which follow different paths to be set up between processes on different nodes. These connections can be on a single port, different ports on a single HCA, or different ports on different HCAs. In Chapter 8, we present an MPI communication framework that can support multirail InfiniBand networks. It supports multiple paths for a single port, multiple ports and multiple HCAs. Our framework is also policy-driven. Messages can be scheduled to be sent through multiple connections based on the policy. We not only address design issues in the framework, but also challenges in providing good policies based on information from different components in the system. Our design is also presented in [54].

There are several other important issues we need to consider besides the above designs. Although we do not have chapters dedicated to these issues, they are very important for the performance and scalability of our MPI design. We will discuss some of these issues in the context of different designs in Chapters 4, 5, 6, 7, and 8.

- **Providing efficient buffer management** — MPI internal communication buffers are used for eager data messages and control messages. These buffers are pre-registered to avoid the overhead of registration in the communication critical path. To improve communication performance, it is desirable that these buffers are aligned. However, aligned buffers tend to have worse cache performance since the beginning blocks of the buffers need to compete for a less number of cache blocks. Another issue we need to consider is the size of these buffers. If it is too small, an eager message may need to be divided into multiple buffers and sent out using multiple send/receive or RDMA operations, which increases the communication overhead. However, if it is too large, more buffer space will be wasted due to internal fragmentation. These trade-offs need to be considered carefully in the context of real applications.

  For transferring large messages in the Rendezvous protocol, extra copies can be avoided by directly sending and receiving from user buffers. However, the user buffers needs to be registered before they can be used for communication. To reduce this overhead, the pin-down cache technique [32] can be used. We investigate how to efficiently implement this technique in the context of InfiniBand Architecture and study its effectiveness in different applications.

- **Designing scalable connection management schemes** — InfiniBand Architecture requires that a connection be setup between a sender and a receiver before communication in its connection based transport services. Since MPI applications assume a fully-connected topology, connection setup must be handled in the MPI implementation. A simple scheme is to setup all connections at the MPI initialization time. However, this leads to several problems for applications using a large number of processes. First, the connection setup may become very time-consuming. Second, this may reduce the scalability of the application because the resources allocated increase with the number of connections. To improve the scalability, our previous study [100] introduces the concept of *on-demand connection*. The work was done for VIA and we have shown that on-demand connection increases the scalability of MPI applications. It can also be used in a similar way in InfiniBand.

- **Ensuring communication progress** — To achieve better computation and communication overlap, an MPI implementation needs to make communication progress independent of the host processor. This issue is especially important for collective communication. This is because slowing down one process may lead to performance degradation of the entire operations since the processes are usually dependent on each other to make progress. In general, the decoupling of communication progress from the computation of application processes is called *application bypass* [10]. Work in our group has shown that it is an effective approach to ensure communication progress and tolerate process skew [12] in collective communication.

Another important problem we have addressed is how to evaluate the performance of an MPI implementation in a meaningful manner. There are many different aspects of an MPI implementation. To provide a comprehensive performance evaluation of MPI implementations, we have proposed a framework which consists a set of micro-benchmarks and a set of application characteristics. These micro-benchmarks include traditional measurements such as latency, bandwidth and host overhead. In addition to those, we have also included the following micro-benchmarks: communication/computation overlap, buffer reuse, memory usage, intra-node communication and collective communication. The objective behind this extended micro-benchmark suite is to characterize different aspects of the MPI implementations and get more insights into their communication behavior. We also use in-depth profiling to obtain different communication characteristics for applications. By combining results from micro-benchmarks and application profiling, we expect to achieve a much better understanding of the impact of MPI implementations on application performance. We will discuss details of this performance evaluation framework in Chapter 9. It is also presented in [53].

We use Figure 3.3 to summarize our research work to design efficient and scalable MPI components by taking advantage of various InfiniBand feature. Please note that certain InfiniBand features, such as Atomic Operations and QoS, are not used in our current design. We will cover them when discussing future research topic in Chapter 11.

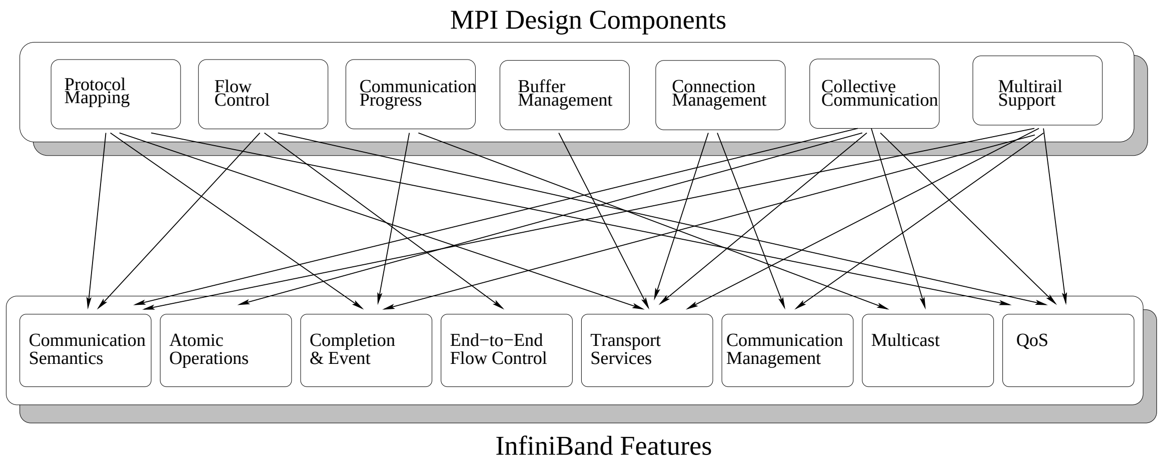

Figure 3.3: Research in Designing MPI Components over InfiniBand

3.2 Methodology

Currently one of the most popular MPI implementations is MPICH [27] from Argonne National Laboratory. MPICH uses a layered approach in its design. The platform dependent part of MPICH is encapsulated in an interface called Abstract Device Interface (ADI), which describes the communication functions used by the MPI implementation. To port MPICH to a new communication architecture, only the ADI functions need to be implemented. More sophisticated ADI functions, such as collective communication calls, are usually implemented by using point-to-point functions. However, the implementation architecture of ADI is very flexible. To achieve optimal performance, collective functions can be implemented directly over the messaging layer provided by the interconnect.

We have based our implementation on MPICH. Our MPI implements a new ADI layer which uses InfiniBand as the underlying communication interconnect. However, we have implemented collective communication operations directly on top of InfiniBand instead of using point-to-point operations to achieve optimal performance. Our implementation is also derived from MVICH [45], which is an ADI2 implementation for VIA.

Currently, MPICH does not support features in the MPI-2 standard. Therefore, there is no mechanism to support functionalities such as one-sided communication in MPICH. MPICH2 [59] is the next generation MPI implementation being developed at Argonne National Laboratory. MPICH2 will support full MPI and MPI-2 standards. Similar to MPICH, MPICH2 uses a layered approach in its design. It also has an ADI layer that makes it easy to be ported to other interconnects.

Our InfiniBand platform consists of InfiniHost HCAs and an InfiniScale switch from Mellanox[60]. InfiniHost provides a programming interface called VAPI [61]. InfiniScale is a full wire-speed switch with eight 10 Gbps ports. The InfiniHost MT23108 HCA connects to the host through PCI-X bus. It allows for a bandwidth of up to 10 Gbps over its ports. For most of our experiments, we use a cluster system consisting of 8 SuperMicro SUPER P4DL6 nodes. Each node has dual Intel Xeon 2.40 GHz processors with a 512K L2 cache at a 400 MHz front side bus. The machines were connected by Mellanox InfiniHost MT23108 DualPort 4X HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The HCA adapters work under the PCI-X 64-bit 133MHz interfaces. We have two slightly different kinds of InfiniHost HCAs called A0 and A1, respectively. A1 cards are optimized and give slightly better performance. The machines were also connected by Myrinet network using NICs with 200MHz LANai 9 processors through an 8-port Myrinet-2000 switch. Myrinet adapters use 64-bit 66MHz PCI bus for all experiments. The Quadrics Elan3

QM-400 cards were attached to these fours nodes. They were connected with each other through an Elite16 switch. The QM-400 card also uses a 64-bit 66MHz PCI slot. We used the Linux Red Hat 7.2 operating system.

CHAPTER 4

BASIC IMPLEMENTATION

First, we describe the design of a basic MPI implementation over InfiniBand. This implementation is based on the MPI implementation over Virtual Interface Architecture (VIA) [17] from the Lawrence Berkeley National Laboratory [45]. Although this implementation is complete, many of its components are not necessarily optimized. We will discuss the design choices we have made and some of the implementation techniques. Then we evaluate its performance by using micro-benchmarks.

4.1 MPI Design Issues

As we have discussed, there are many design issues involved in putting an MPI layer on top of InfiniBand. InfiniBand provides two kinds of communication semantics: send/receive and RDMA. Although it is possible to implement MPI using only one of them, it is better to combine them so that we can take advantage of both semantics. MPI assumes that the underlying communication layer provides such functionality as reliability, buffer management, and flow control. MPI applications should not be involved in these issues. Since the Verbs Interface requires that communication buffers should be registered and upper layer software should take care of flow control, there is a functional mismatch between the MPI and the Verbs layer.

Thus, the most important task for implementing MPI on top of the Verbs layer is to bridge this gap. InfiniBand provides different classes of services including reliable connection (RC) and reliable datagram (RD). If these two services are used for MPI, reliability can be guaranteed. However, buffer registration and flow control issues still need to be handled explicitly by the MPI implementation.

In addition to basic communication operations, InfiniBand also offers advanced features such as end-to-end flow control, atomic operations and QoS support. It is possible to improve MPI performance by taking advantage of these advanced features. However, for the basic implementation we do not focus on these features.

Next, we briefly discuss several components in this implementation.

4.1.1 Send/Receive vs RDMA

MPI implementations usually use two internal protocols to handle communication: Eager and Rendezvous. In Eager protocol, a message is sent to the receiver even though the corresponding receive has not been issued. In this case, the message is put into an unexpected queue and later copied to the receive buffer. In Rendezvous protocol, the actual data transfer takes place only after both send and receive have been issued. Eager protocol matches well with send/receive operations in InfiniBand. For Rendezvous protocol, RDMA can also be used. This is because during Rendezvous protocol, the sender and the receiver exchange information through control messages before the actual data transfer, and the destination address needed by RDMA operation can be put into these control messages.

When to switch from one protocol to another depends on the messages size. Usually, small messages uses Eager protocol and large messages uses Rendezvous protocol.

The switch point is important and it must be carefully chosen to match the performance characteristics of the underlying platform.

4.1.2 Handling Buffer Registration

Buffer registration and de-registration in InfiniBand Architecture are expensive operations, because they not only involve the operating system kernel, but also need some interaction between the NIC and the host. Therefore we would like to avoid these operations on the communication critical path if possible.

One way to address this problem is to maintain a pre-registered buffer pool. When the message is being sent, it is first copied to a buffer in the pool. Similarly, on the receiver side, messages are first received into buffers in the pre-registered pool, and then copied to the destination buffers. This method avoids the buffer registration overhead completely at the cost of two extra copies for very messages.

Another way is to use a technique called Pin-down Cache which is first proposed in [32]. The idea is to maintain a cache of registered buffers. When a buffer is first registered, it is put into the cache. When the buffer is unregistered, the actual unregister operation is not carried out and the buffer stays in the cache. Thus the next time when the buffer needs to be registered, we need not to do anything because it is already in the cache. A buffer is unregistered only when it is evicted from the cache. The effectiveness of Pin-down Cache depends on how often the application reuses its buffers. If the reuse rate is high, most of the buffer registration and de-registration operations can be avoided.

4.1.3 Flow Control

For send/receive operations in InfiniBand, when a message is sent out and there is no corresponding receive posted on the receiver side, a retry mechanism is triggered and the performance may drop significantly. In order to avoid buffer overrun on the receiver side, a flow control scheme is needed for the MPI implementation. Even when we use RDMA operations, flow control is still needed because the control messages may still use send/receive operations.

To deal with this problem, a credit-based flow control mechanism is used. When a message is sent out, the sender's credit is decremented. When the receiver reposts receive requests, it can inform the sender that new credits are available. If the number of credits is low, the sender will switch from Eager protocol to Rendezvous. If there is no credit available, send operations will be blocked until enough credits arrive from the receiver. In this scheme, the number of credits may be important as it may affect the communication performance.

4.2 Performance

In this section we present performance evaluation for our basic MPI implementation using micro-benchmarks. We also compare our results with those from Myrinet/GM and Quadrics.

Figures 4.1 and 4.2 show the latency for different MPI implementations. The latency tests were carried out in a ping-pong fashion. In the bandwidth test, the sender keeps sending 1000 messages to the receiver and then waits for a reply. Then the sender calculates the bandwidth based on the elapsed time and number of bytes it has sent. In the latency tests, blocking version of MPI functions (MPI_Send and

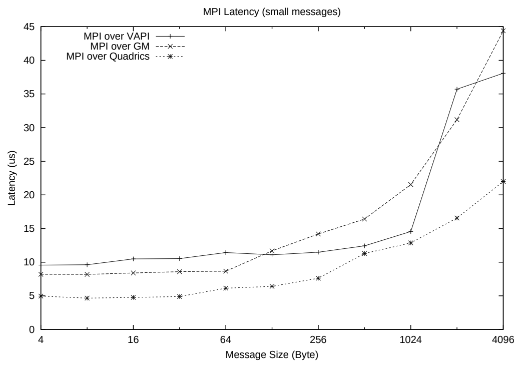

Figure 4.1: Small Message Latency for MPI

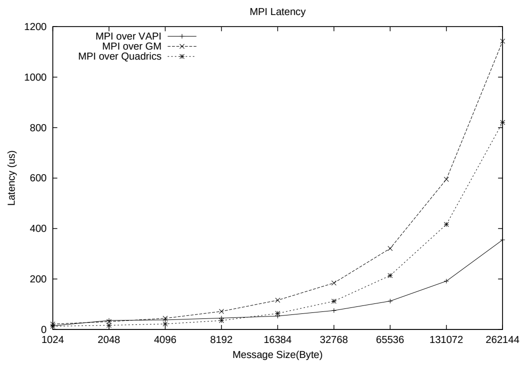

Figure 4.2: Large Message Latency for MPI

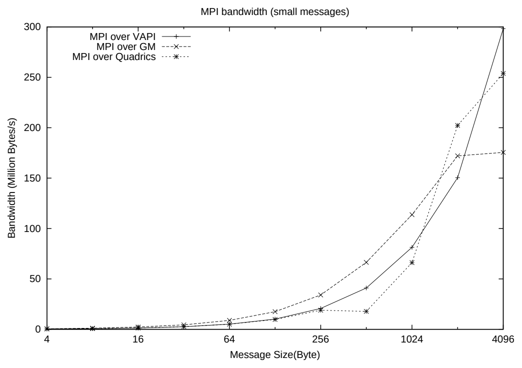

Figure 4.3: Small Message Bandwidth for MPI

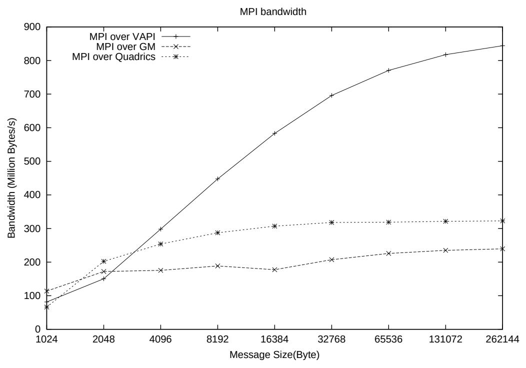

Figure 4.4: Large Message Bandwidth for MPI

MPI_Recv) were used. In the bandwidth tests, unblocking version of MPI functions (MPI_Isend and MPI_Irecv) were used. The smallest latency we have achieved is around 9.5 microseconds for MPI over VAPI. Comparing our implementation with MPI over GM, we find that the two perform comparably for messages up to 4 KBytes. (GM performs better for messages smaller than 128 bytes.) However, for large messages MPI over VAPI performs much better. MPI over Quadrics performs best for message range 0 to 16 KBytes and its smallest latency is around 4.3 microseconds.

The bandwidth graphs in Figure 4.3 and 4.4 show that our MPI implementation is able to achieve over 844 Million Bytes/second (844 MB/s) peak bandwidth using RDMA.

MPI over GM performs slightly better than MPI over VAPI in the message range up to 1 KBytes. For all other message sizes, MPI over VAPI gives better performance.

4.3 Summary

In this chapter, we presented a basic implementation to support MPI on top of InfiniBand's verbs layer. Our implementation uses InfiniBand Send/Receive operations to transfer small messages. For large messages, we have implemented an RDMA write based scheme that can achieve zero-copy. Our performance evaluation shows that this design achieves very good performance. We also carried out performance comparison with contemporary cluster interconnects such as Myrinet and Quadrics. Our results show that current InfiniBand hardware is capable of delivering significant performance benefits and in some cases even better than Myrinet and Quadrics. This basic implementation also serves as the basis for our later research on optimizing MPI performance over InfiniBand.

CHAPTER 5

DESIGNING EFFICIENT AND SCALABLE FLOW CONTROL

One of the key issues in designing MPI over InfiniBand is flow control. Since current MPI implementations are based on InfiniBand Reliable Connection (RC) service, multiple receive buffers have to be posted for each connection in order for the sender to have multiple outstanding messages. However, if the sender sends too fast, these buffers can be exhausted. The flow control mechanism is to prevent a fast sender from overwhelming a slow receiver and exhausting its resources such as buffer space in this case. Flow control is an important issue in MPI design because it affects both the performance and the scalability of an MPI implementation.

5.1 Flow Control in MPI over InfiniBand

We have shown different types of messages in MPI Eager and Rendezvous protocols in Figure 2.1. Among these messages, *Eager Data* and *Rendezvous Start* messages are sent to the receiver regardless of its current state. Therefore, these messages are *unexpected* with respect to the receiver. Since the sender can potentially initiate a large number of send operations in MPI, it is possible that the receiver can be overwhelmed by too many unexpected messages because each of these messages consumes resources such as buffer space at the receiver side. This issue is even more important for InfiniBand because communication buffers must be pinned down and they cannot be swapped out during communication.

To accommodate unexpected messages, every process can pre-post a number of receiver buffers for each connection. Every message will be received into one of these buffers. After the receiver finishes processing a buffer, it immediately re-posts the buffer. As long as the number of outstanding unexpected messages for a connection is less than the number of pre-posted buffers, these messages can always be received safely. However, if the number of outstanding unexpected messages is too large, a flow control mechanism needs to be implemented to stall or slow down the sender so that the receiver can keep up.

Thus, there are two important problems in flow control. First, we need to study the impact of the number of pre-posted receiver buffers. Using too many buffers will adversely affect application performance and limit the scalability of the MPI implementation. However, if the number is too small, senders may have to stall or slow down frequently and wait for the remote process to re-post the buffers. As a result, the sender and the receiver become tightly coupled in communication, leading to degraded performance. Ideally, the number should be determined by application communication pattern to achieve both performance and scalability. The second issue is what mechanism we use to stall or slow down the sender when the receiver cannot keep up. This mechanism should be effective and have negligible run-time overhead during normal communication.

Based on the basic implementation, we propose several designs to address the flow control issue. Flow control can be handled in the MPI implementation. In this case, we call it a *user-level scheme.* However, since InfiniBand itself provides end-to-end flow control, an alternative is to let InfiniBand hardware handle this task. We call this a *hardware-based scheme.* Flow control schemes can also be classified by the way they choose the number of pre-posted buffers. In a *static scheme*, this number is determined at compilation or initialization time and remains unchanged during execution of the application. On the other hand, in a *dynamic scheme*, this number can be changed during program execution.

Flow control is an important issue in cluster computing and has been studied in the literature. Similar to our schemes, Flow control in FM [74] is also credit-based. In [7], a reliable multicast scheme is implemented by exploiting link-level flow control in Myrinet. GM [65] is a messaging layer over Myrinet developed by Myricom. In GM, a sender can only send a message when it owns a *send token.* This is essentially a credit-based flow control scheme. Work in [22] proposed an automatic tuning mechanism for TCP flow control. Unlike the above, our work concentrates on flow control schemes in MPI over InfiniBand.

MVICH [45] is an MPI implementation over VIA [17]. It uses a user-level static flow control scheme. Our original MPI implementation over InfiniBand [56] was based on it and used a similar flow control scheme. In this work, we carry out a detailed study and comparison of different flow control schemes.

In order to improve the scalability of MPI implementations, an on-demand connection set-up scheme was proposed in [100]. In this scheme, connections are set up between two processes when they communicate with each other for the first time. If there is no communication between them, no connection will be set up and therefore no buffer space will be used. Our proposed dynamic flow control scheme can be combined with on-demand connection setup to further improve the scalability of MPI implementations.

5.2 Hardware-Based Flow Control

In hardware-based schemes, there is no flow control at the MPI level. All outgoing messages are submitted immediately to the send queue. If too many send operations are posted, the HCA at the sender side will reduce its sending rate because of the InfiniBand end-to-end flow control. At the receiver side, when there is no posted receive for an incoming message, this message will be dropped and RNR Nak will be issued. The sender will then wait for a time-out and re-transmit the message. To ensure reliability at the MPI level, the retry count can be set to infinite. Thus eventually the message will be delivered to the receiver side when the receive buffer is posted.

One of the advantages in using hardware-based flow control is that it incurs almost no run-time overhead in normal communication when there are enough pre-posted receiver buffers. This is because there is no need to keep track of flow control information in the MPI implementation. This also means that flow control processing can be done regardless of the communication progress of applications. Therefore, hardware-based schemes also achieve better "application bypass" [10, 12]. However, InfiniBand provides very little flexibility to adjust the behavior of hardware based flow control. Since the flow control algorithm used by InfiniBand may not be optimal for every MPI application, this lack of flexibility may result in performance degradation for some applications. Further, the end-to-end flow control and transmission retries are largely transparent to the software layer and there is no information feedback for the

MPI implementation to adjust its behavior. The lack of information feedback makes it very hard to implement dynamic flow control schemes in which the MPI implementation can adjust the number of pre-posted buffers for each connection based on application communication pattern.

5.3 User-Level Static Flow Control

In this subsection we discuss how to implement user-level flow control at the MPI level. First we will describe static schemes currently used in [45] and [56].

The basic idea of user-level flow control is to use a credit-based scheme. During MPI initialization, a fixed number of receive buffers are pre-posted for each connection. Initially, the number of credits for each sender is equal to the number of pre-posted buffers at the corresponding receiver. Whenever a sender sends out a message that will consume a receiver buffer, its credit count will be decremented. When the credit count reaches zero, the sender can no longer post send operations that consume credits. Instead, these operations will be stored in a *backlog queue.* The operations in the backlog queue will be processed later in FIFO order when credits are available.

At the receiver side, the receiver will re-post a receive buffer after it has finished processing it. The credit count for the corresponding sender will then be incremented. However, since this information about new credits is only available at the receiver side, we must have some kind of mechanism to transfer it to the sender side. Two methods can be used for this purpose: *piggybacking* and *explicit credit messages.* To use piggybacking, we add a credit information field to each message. An MPI process can use this field to notify the other side about credit availability. If the communication pattern is symmetric, each sender will get credit information updates frequently and be able to make communication progress. Explicit credit messages can be used when the communication pattern is asymmetric. When a process has accumulated a certain number of credits and there is still no message sent by the MPI layer to the other side, a special credit message can be sent to transfer the credit information. In the MPI implementation, small messages are usually transferred using Eager protocol. However, when there are no credits, only Rendezvous protocol is used. Because of the handshaking process in Rendezvous protocol, credit information can be exchanged through piggybacking, which can speed up the processing of the send operations in the backlog queue.

Because of possible deadlock situation, explicit credit messages must be used carefully. In [45] and [56], these messages themselves will consume credits because they are implemented using send operations. To prevent deadlock, we proposed an *optimistic scheme* for credit messages. Basically, we do not impose flow control for explicit credit messages. Thus, explicit credit messages are not subject to user-level flow control. These messages are always posted directly without going through the backlog queue even though no credit is available. In this case, the hardware-level flow control mechanism will ensure that the message will be delivered. Since credit messages can always be sent, deadlock will not happen.

Using user-level flow control requires the MPI implementation to manage credit information and take appropriate actions. Therefore, it may have larger run-time overhead than hardware-based schemes. The use of explicit credit messages may increase network traffic in some cases. (We should note that hardware-based schemes may also increase network traffic because of NAK and re-transmission.) However, these overheads can be reduced by an optimized implementation. Another disadvantage of user-level schemes is that flow control processing relies on communication progress. Therefore, it achieves less “application bypass” compared with hardware-based schemes. The major advantage of user-level flow control is that various information regarding flow control is available to the MPI layer. Based on this information, an MPI implementation can adjust its behavior to achieve better performance and scalability. Next, we will discuss a dynamic user-level flow control scheme that takes advantage of this information.

5.4 User-Level Dynamic Flow Control

To achieve both performance and scalability, we propose a dynamic user-level flow control scheme. This scheme uses credit-based flow control at the MPI level, which is similar to the static scheme. The difference is that each connection starts with a small number of pre-posted buffers. During program execution, the number of pre-posted buffers can be gradually increased based on the communication pattern using a feedback-based control mechanism. In this scheme, two important issues must be addressed:

- How to provide feedback information?
- What to do when feedback information is received?

The feedback mechanism should notify the receiver when more pre-posted buffers are needed. We notice that if there are not enough credits, a message will enter the backlog queue and be processed later. Therefore, this information can be used to provide feedback. We add a field to each message indicating whether it has gone through the backlog. When a process receives a message that has gone through the backlog queue, it takes action to increase the number of pre-posted buffers for the corresponding sender. The increase can be linear or exponential depending on the application. If communication pattern changes are relatively slow compared with the time to increase the number of pre-posted buffers, this mechanism can achieve both good performance and buffer efficiency.

In addition to increasing the number of buffers, a dynamic scheme can also decrease the number of buffers when the application no longer needs so many buffers. This may be beneficial to long-running, multi-phase MPI applications whose communication pattern changes in different phases. Currently we only allow increasing the number of buffers. We plan to investigate more along this direction in the future.

5.5 Performance Evaluation

In this section, we present performance evaluation of different flow control schemes using both micro-benchmarks and applications. The micro-benchmarks are latency and bandwidth tests. The applications we use are the NAS Parallel Benchmarks [68]. In the performance evaluation, we concentrate on two aspects of different flow control schemes: normal condition (with plenty of pre-posted buffers or credits) and flow control condition (when the number of outstanding messages exceeds the number of pre-posted buffers or there are not enough credits). Another issue we are interested in is how many pre-posted buffers are generally needed by applications in order to achieve best performance.

5.5.1 Latency

In the latency test, the communication pattern is very symmetric. Since the sender and the receiver send back a message only after processing the previous one, there are always enough receive buffers posted at both sides. For user-level schemes, the credit information is always transferred in time through piggybacking. Therefore, this test shows how different schemes perform under normal conditions.

In the latency test, the hardware-level scheme has the least overhead because there is no need to keep track of information related to flow control. However, from Figure 5.1 we can see that this bookkeeping overhead is negligible and all three schemes perform comparably.

5.5.2 Bandwidth

The bandwidth tests are carried out by having the sender send out a number of back-to-back messages to the receiver and then waiting for a reply from the receiver. The number of back-to-back messages is referred to as *window size*. The receiver sends the reply only after it has received all messages. The above procedure is repeated multiple times and the bandwidth is calculated based on the elapsed time and the number of bytes sent by the sender. We use two different versions of bandwidth tests. In the blocking version, MPI_Send and MPI_Recv functions are used. MPI_Isend and MPI_Irecv are used in the non-blocking version.

In the first group of our tests, we have chosen a fixed message size (4 bytes). The tests are conducted for both blocking version and non-blocking version. The numbers of pre-posted buffers we have chosen for the tests are 10 and 100. Different results are obtained by varying the window size of the bandwidth tests.

When there are 100 pre-posted buffers, the window size does not exceed the number of pre-posted buffers. Thus with enough buffers or credits, all three schemes perform comparably for both blocking and non-blocking version.

Figures 5.2 and 5.3 show the results with only 10 pre-posted buffers. We can observe that when the window size exceeds the number of pre-posted buffers, the user-level dynamic scheme achieves the best performance because it is able to adapt to the communication pattern and increase the number of buffers. On the other hand, user-level static scheme performs the worst because the communication is stalled when there are not enough credits. We also notice that for user-level schemes, blocking version of bandwidth tests achieve better performance. This is because in user-level schemes, when there is no credit available, Rendezvous protocol will be used even for small messages. In the blocking tests, the sender waits for the send operation to finish and therefore is able to get more credits through the handshaking procedure of Rendezvous protocol.

Figures 5.4 and 5.5 show the results with 10 pre-posted buffers for large messages (32K bytes). Since large messages always go through Rendezvous protocol, the communication pattern in these tests becomes more symmetric because of the handshaking procedure. As a result, all three schemes are able to perform well even with less number of buffers. The non-blocking version performs much better than the blocking version because it achieves better communication overlap.

5.5.3 NAS Parallel Benchmarks

To better understand the impact of different flow control schemes on application performance, we have conducted experiments using the NAS Parallel Benchmarks

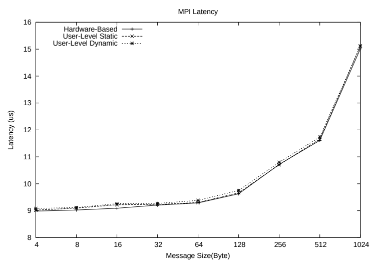

Figure 5.1: MPI Latency

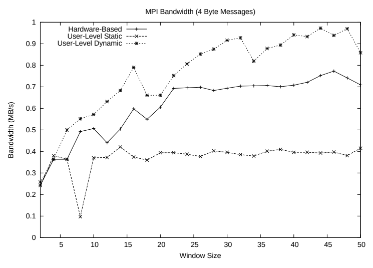

Figure 5.2: MPI Bandwidth (Pre-Post = 10, Blocking)

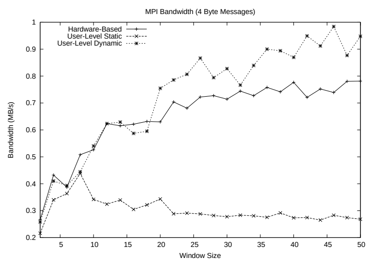

Figure 5.3: MPI Bandwidth (Pre-Post = 10, Non-Blocking)

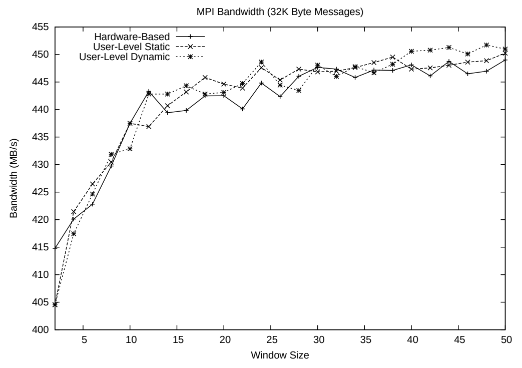

Figure 5.4: MPI Bandwidth (Pre-Post = 10, Blocking)

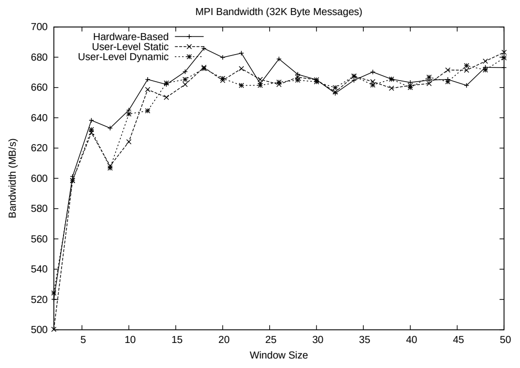

Figure 5.5: MPI Bandwidth (Pre-Post = 10, Non-Blocking)

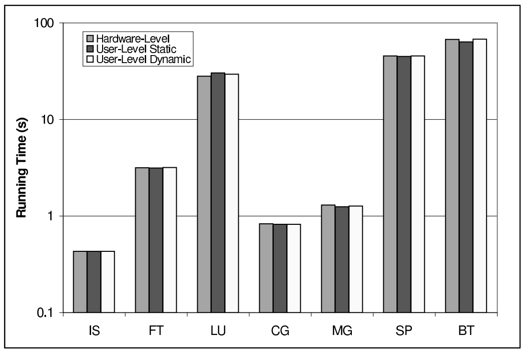

Figure 5.6: NAS Benchmarks (Pre-Post = 100)

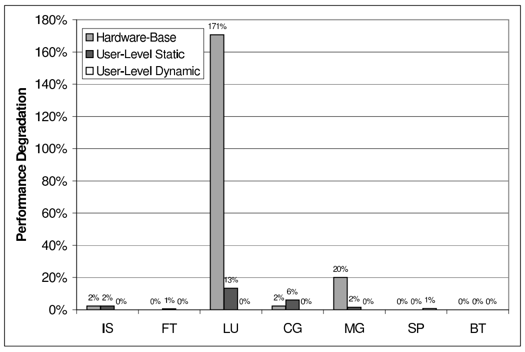

Figure 5.7: NAS Benchmarks (Performance Degradation from Pre-Post=100 to Pre-Post=1)

Table 5.1: Explicit Credit Messages for User-Level Static Scheme

App	# ECM Msg	# Total Msg
IS	0	383
FT	0	193
LU	9002	48805
CG	0	4202
MG	1	1595
BT	0	28913
SP	0	14531

Table 5.2: Maximum Number of Posted Buffers for User-Level Dynamic Scheme

App	# Buffer
IS	4
FT	4
LU	63
CG	3
MG	6
BT	7
SP	7

(Class A). IS, FT, LU, CG and MG tests were carried out using 8 processes on 8 nodes. Since SP and BT tests require the number of processes to be a square number, they were conducted using 16 processes on 8 nodes.

First, we study the impact of different flow control schemes on normal communication where there are always enough pre-posted buffers or credits. We carried out the experiments with 100 pre-posted buffers, which are more than any of the application will need. The results are shown in Figure 5.6. We notice that the three flow control schemes perform comparably for almost all the applications, with at most 2%–3% difference due to random fluctuation. One exception is the LU application. (There are also some discrepancies in the running time of BT. We are currently investigating this issue.) For LU, the hardware-based scheme is the best, which outperforms both user-level schemes by around 5%–6%. The reason why user-level schemes performs worse is that they use explicit credit messages. If the application communication pattern is very asymmetric, these messages have to be generated frequently in order to transfer credit information and the performance will be degraded. Table 5.1 shows the average number of explicit credit messages (ECM) for each connection at each process and the total number of messages (including data and control messages). We can see that for LU, explicit credit messages make up for a significant percentage of the total number of messages (18%). However, there are almost no explicit flow control messages for other applications. We should also note that the number of explicit credit messages depends on a threshold credit value, which suppresses any explicit credit messages if the number of credit to be transferred is below the threshold. Currently we use a relatively small threshold value of 5. Performance can be improved by increasing this value for LU.

5.5.4 Impact of Number of Pre-Posted Buffers

As we have discussed, the number of pre-posted buffers has significant impact on the scalability of applications. In this subsection, we consider an extreme case where there is only one pre-posted buffer for every connection at each process. Figure 5.7 shows the percentage of performance drop when we change the pre-post value from 100 to 1. This case can serve as an "upper bound" of the impact of changing the number of pre-posted buffers.

One surprising findings from Figure 5.7 is that most applications perform quite well even in this extreme condition. For IS, FT, SP and BT, the maximum performance degradation for all three schemes is only 2%. For the hardware-based scheme, performance drops significantly for LU and MG. This drop is due to the large number of time-out and re-transmission happening at the hardware level. For the user-level static scheme, the largest performance drops are for LU (13%) and CG (6%). Since the user-level dynamic scheme is able to adapt its behavior according to the application communication pattern, there is almost no performance degradation. In Table 5.2, we show the maximum number of posted buffers for every connection at every process in the user-level dynamic scheme after program execution. We can see that for all applications except LU, only a very small number of buffers (no more than 7) are needed for each connection. Therefore, the user-level dynamic flow control scheme can potentially achieve both performance and buffer efficiency. If this communication pattern remains unchanged for large number of processes, buffer space will not be the limitation of scalability. We plan to investigate this direction further in the future on large-scale clusters.

5.6 Summary

Flow control is an important issue in cluster computing and has been studied in the literature. Similar to our schemes, Flow control in FM [74] is also credit-based. In [7], a reliable multicast scheme is implemented by exploiting link-level flow control in Myrinet. GM [65] is a messaging layer over Myrinet developed by Myricom. In GM, a sender can only send a message when it owns a *send token*. This is essentially a credit-based flow control scheme. Work in [22] proposed an automatic tuning mechanism for TCP flow control. Unlike the above, our work concentrates on flow control schemes in MPI over InfiniBand.

MVICH [45] is an MPI implementation over VIA [17]. It uses a user-level static flow control scheme. Our original MPI implementation over InfiniBand [56] was based on it and used a similar flow control scheme. In this work, we carry out a detailed study and comparison of different flow control schemes.

In this work, we present a detailed study of the flow control issues in implementing MPI over the InfiniBand Architecture with Reliable Connection service. We categorize flow control schemes into three classes: hardware-based, user-level static and user-level dynamic. These schemes differ in their run-time overhead and how they decide the number of pre-posted buffers for each connection. The hardware-based scheme has the least overhead under normal conditions. However, in user-level schemes, MPI implementation can have more control over the system communication behavior when the receiver is overloaded. In particular, the user-level dynamic scheme is able to adjust the number of pre-posted buffers according to the application communication pattern. Therefore, it can potentially achieve both good performance and high scalability in terms of buffer usage.

We have implemented all three schemes in our MPI implementation over InfiniBand and conducted performance evaluation on our 8-node InfiniBand cluster. We use both micro-benchmarks and the NAS Parallel Benchmarks for the evaluation. We have shown that the overheads of user-level schemes are very small. Our results have also shown that the user-level dynamic scheme can achieve both performance and buffer efficiency by adapting to the communication pattern. Another finding in our performance evaluation is that for most NAS applications, only a small number of pre-posted buffers are required to achieve good performance.

CHAPTER 6

RDMA-BASED DESIGN

We now describe an enhancement of our previous MPI implementation. In this implementation, we propose a method which brings the benefits of RDMA operations to not only large messages, but also for small and control messages. By introducing techniques such as *persistent buffer association* and *RDMA polling set*, we address several challenging issues in the RDMA-based MPI design. Instead of using only RDMA operations for communication, our design combines both send/receive and RDMA. By taking advantage of send/receive operations and the Completion Queue (CQ) mechanism offered by InfiniBand, we are able to simplify our design and achieve both high performance and scalability.

Being the *de facto* standard of writing parallel applications, MPI has been implemented for numerous interconnects, including those with remote memory access abilities [87, 5, 36, 11]. [5] relies on the active message interface offered by LAPI. [36] uses PIO for small messages. Work done in [11] implemented MPI for Cray T3D based on the SHMEM interface. [87] describes an MPI implementation over Sun Fire Link Interconnect, which is based on PIO. MPI over Sun Fire Link uses a sender-managed buffer scheme for transferring messages. In this approach, the sender can choose any buffer at the receiver side for doing remote write. To let the receiver know where the data has been written, another PIO is used to write the buffer address to a pre-specified location. This extra PIO has very little overhead. However, the RDMA operations in InfiniBand Architecture have larger overhead. Therefore, one of our objectives is to use as few RDMA operations as possible. Another difference is that InfiniBand offers both channel and memory semantics and we have shown that it is possible to combine them to achieve scalability. However, none of the existing implementations have information regarding the use of RDMA operations for small data messages and control messages, nor are the scalability issues in RDMA discussed in these references.

RDMA operations have been used to implement MPI collective operations. Work in [82] focuses on how to construct efficient algorithms to implement collective operations by using RDMA operations. Our work can be used in conjunction with their work to efficiently transfer short data messages and control messages.

RDMA operations have also been used to design communication subsystems for databases and file systems [104, 57]. These studies do not address the issue of using RDMA for control messages. [13] evaluated different communication schemes for implementing a web server on a cluster connected by VIA. Some of their schemes use RDMA write for transferring flow control messages and file data. However, their schemes differ from ours in that they have used a sender-managed scheme which is similar to [87].

6.1 Mapping MPI protocols

MPI defines four different communication modes: *Standard*, *Synchronous*, *Buffered*, and *Ready* modes. Two internal protocols, *Eager* and *Rendezvous*, are usually used to implement these four communication modes.

When we are transferring large data buffers, it is beneficial to avoid extra data copies. A zero-copy Rendezvous protocol implementation can be achieved by using RDMA write. In this implementation, the buffers are pinned down in memory and the buffer addresses are exchanged via the control messages. After that, the data can be written directly from the source buffer to the destination buffer by doing RDMA write. Similar approaches have been widely used for implementing MPI over different interconnects [45, 19, 5].

For small data transfer in Eager protocol and control messages, the overhead of data copies is small. Therefore, we need to push messages eagerly toward the other side to achieve better latency. This requirement matches well with the properties of send/receive operations. However, as we have discussed, send/receive operations also have their disadvantages such as lower performance and higher overhead. Next, we discuss different approaches of handling small data transfer and control messages.

6.1.1 Send/Receive Based Approach

In this approach, Eager protocol messages and control messages are transfered using send/receive operations. To achieve zero-copy, data transfer in Rendezvous protocol uses RDMA write operation.

In the MPI initialization phase, a reliable connection is set up between every two processes. For a single process, the send and receive queues of all connections are associated with a single CQ. Through this CQ, the completion of all send and RDMA operations can be detected at the sender side. The completion of receive operations (or arrival of incoming messages) can also be detected through the CQ.

InfiniBand Architecture requires that the buffers be pinned during communication. For eager protocol, the buffer pinning and unpinning overhead is avoided by using a pool of pre-pinned, fixed size buffers for communication. In Rendezvous protocol, data buffers are pinned on-the-fly. However, the buffer pinning and unpinning overhead can be reduced by using the pin-down cache technique [32].

In send/receive based approach, we only need to check the CQ for incoming messages. The CQ polling time usually does not increase with the number of connections. Therefore, it provides us an efficient and scalable mechanism for detecting incoming messages. Figure 6.1 shows the CQ polling time with respect to different number of connections in our InfiniBand testbed.

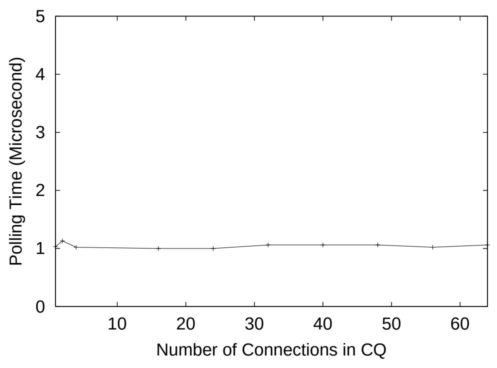

Figure 6.1: CQ Polling time

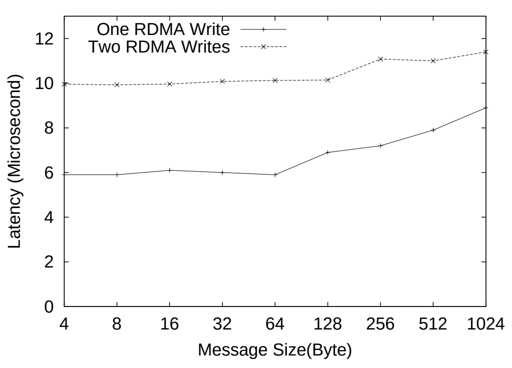

Figure 6.2: Latency of One RDMA Write versus Two RDMA Writes

However, there are also disadvantages for using the send/receive based approach. First, since the performance of send/receive is not as good as RDMA write, we cannot achieve the best latency for small data transfer and control messages. Second, we have to handle tasks such as allocating and de-allocating buffers from the pre-pinned buffer pool and re-posting receive descriptors. These tasks increase the overhead and communication latency.

6.1.2 RDMA-Based Approach

To overcome the drawbacks of the send/receive based approach, we have designed an RDMA write based approach for Eager protocol and control messages. In this approach, the communication of Eager protocol and control messages also goes through RDMA write operations. Therefore, we can achieve lower latency and less overhead. However, two difficulties must be addressed before we can use RDMA for data transfer:

- The RDMA destination address must be known before the communication.
- The receiver side must detect the arrival of incoming messages.

In current generation InfiniBand hardware, RDMA operations have high starting overhead. From Figure 6.2 we can see that the latency increases significantly if we use two RDMA write operations instead of one. Thus, it is desirable that we use as few RDMA operations as possible to transfer a message. Ideally, only one RDMA operation should be used.

To address the first problem, we have introduced a technique called *persistent buffer association*. Basically, for each direction of a connection we use two buffer pools: one at the sender and one at the receiver. Unlike other approaches in which the sender may use many buffers for an RDMA write operation, we have persistent correspondence between each buffer at the sender side and each buffer at the receiver side. In other words, at any time each buffer at the sender side can only be RDMA written to its corresponding buffer at the receiver side. These associations are established during the initialization phase and last for the entire execution of the program. Thus, the destination address is always known for each RDMA operation.

The second problem can be broken into two parts: First, we need to efficiently detect incoming messages for a single connection. Second, we need to detect incoming messages for all connections in a process.

For RDMA write operations, the CQ mechanism cannot be used to report the completion of communication at the receiver side. The basic idea of detecting message arrival is to poll on the content of the destination buffer. We organize the buffers as a ring. The sender uses buffers in a fixed, circular order so that the receiver always knows exactly where to expect the next message. The details of our design will be presented in the next section.

Once we know how to detect incoming messages for a single connection, multiple connections can be checked by just polling them one by one. However, the polling time increases with the number of connections. Therefore this approach may not scale to large systems with hundreds or thousands of processes.

6.1.3 Hybrid Approach

As we can see, RDMA and send/receive operations both have their advantages and disadvantages. To address the scalability problem in our previous design, we have enhanced our previous design by combining both RDMA write and send/receive

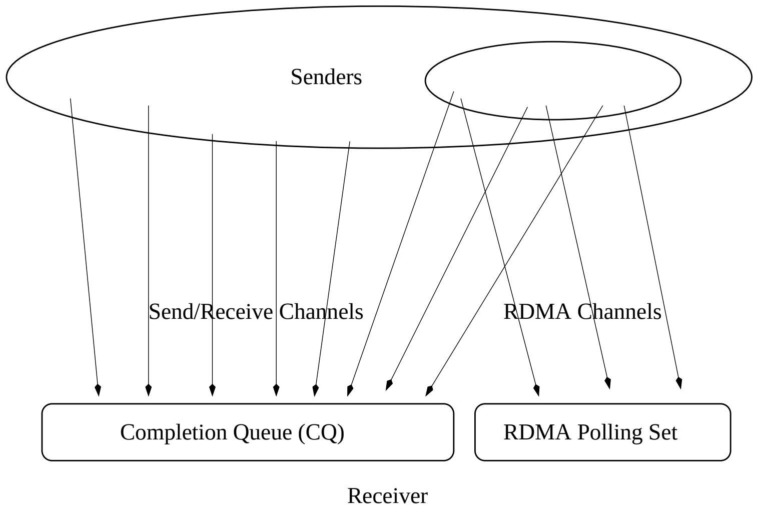

Figure 6.3: RDMA Polling Set

operations. It is based on the observation that in many MPI applications, a process only communicates with a subset of all other processes. Even for this subset, not all connections are used equally frequently. Table 6.1 lists the average number of communication sources per process for several large-scale scientific MPI applications [95, 100] (values for 1024 processors are estimated from application algorithm discussions in the literature). Therefore, we introduce the concept of *RDMA polling set* at the receiver side. In our scheme, each sender has two communication channels to every other process: a send/receive channel and an RDMA channel. A sender will only use the RDMA channel for small messages if the corresponding connection is in the RDMA polling set at the receiver side. If a connection is not in the polling set, the sender will fall back on send/receive operations and the message can be detected through the CQ at the receiver. The receiver side is responsible for managing the RDMA polling set. The concept of RDMA polling set is illustrated in Figure 6.3. Ideally, the receiver should put the most frequently used connections into the RDMA polling set. On the other hand, the polling set should not be so large that the polling time is hurting performance.

Table 6.1: Number of distinct sources per process

Application	# of processes	Average # of sources
sPPM	64	5.5
	1024	6
Sphot	64	0.98
	1024	1
Sweep3D	64	3.5
	1024	4
Samrai4	64	4.94
	1024	10
CG	64	6.36
	1024	11

By restricting the size of the RDMA polling set, the receiver can efficiently poll all connections which use RDMA write for Eager protocol and control messages. Messages from all other connections can be detected by polling the CQ. In this way, we not only achieve scalability for polling but also get the performance benefit of RDMA.

Having two communication channels also helps us to simplify our protocol design. Instead of trying to handle everything through the RDMA channel, we can fall back on the send/receive channel in some infrequent cases.

6.2 Detailed Design Issues

In this section, we discuss detailed issues involved in our design. First, we present the basic data structure for an RDMA channel. After that we discuss the communication issues for a single RDMA channel, including polling algorithm, flow control, reducing sender overhead and ensuring message order. Then we discuss how a receiver manages the RDMA polling set.

6.2.1 Basic Structure of an RDMA Channel

Unlike send/receive channels, RDMA channels are uni-directional. One direction of the connection can use an RDMA channel while the other direction cannot. Figure 6.4 shows the basic structure of an RDMA channel. For each RDMA channel, there are a set of fixed size, pre-registered buffers at both the sender side and the receiver side. Each buffer at the sender side is persistently associated with one buffer at the receive side and its content can only be RDMA written to that buffer.

On both sides, buffers are organized as rings with the respective head pointers and tail pointers. The buffers run out for a sender when the head pointer meets the tail pointer. At the sender side, the head pointer is where the next outgoing message should be copied and RDMA written to the remote side. After the message is written, the head pointer is incremented. Later the receive side detects the incoming message and processes it. Only after this processing, this buffer can be used again for another RDMA write. The tail pointer at the sender side is to record those buffers that are already processed at the receiver side. The sender side alone cannot decide when to advance the tail pointer. This is done by a flow control mechanism discussed later.

At the receiver side, the head pointer is where the next incoming message should go. In order to check incoming messages, it suffices to just examine the buffer pointed by the head pointer. The head pointer is incremented after an incoming message is detected. When we have got an incoming message, the processing begins. After the processing finishes, the buffer is freed and it can be reused by the sender. However, the order in which the buffers are freed may not be the same as the order in which the messages arrive. Therefore we introduce the tail pointer and some control fields at the receiver to keep track of these buffers. The tail pointer advances if and only if the current buffer is ready for reuse.

One concern for the RDMA-based design is memory usage. For each connection, we need to use two pools of pre-pinned buffers. Actually, the same problem exists for the send/receive based scheme because the receiver has to pre-post a number of buffers for each connection. However, we have found that memory consumption is not large for the RDMA-based design (around 100 KBytes for each connection per node). By limiting the number of connections in the RDMA polling set, we can effectively reduce the memory consumption.

6.2.2 Polling for a Single Connection

In order to detect the arrival of incoming messages for a single RDMA channel, we need to check the content of the buffer at the receiver side. In InfiniBand Architecture, the destination buffers of RDMA operations must be contiguous. Therefore a simple solution is to use two RDMA writes. The first one transfers the data and the second one sets a flag. Please note that by using two RDMA writes, we can be sure that when the flag is changed, the data must have been in the buffer because InfiniBand ensures ordering for RDMA writes.

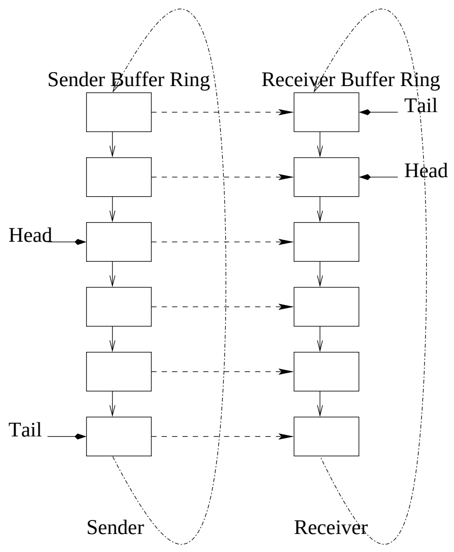

Figure 6.4: Basic Structure of an RDMA Channel

The above scheme uses two RDMA writes, which increase the overhead as we have seen in Figure 6.2. There are two ways to improve this scheme. First, we can use the gather ability offered by InfiniBand to combine the two RDMA writes into one. The second way is to put the flag and the data buffer together so that they can be sent out by a single RDMA write. However, in both cases, we need to make sure that the flag cannot be set before the data is delivered. And to do this we need to use some knowledge about the implementation of hardware. In our current platform, gather lists are processed one by one. And for each buffer, data is delivered in order (the last byte is written last). Thus, we need to put the flag after the data in the gather list, or to put the flag at the end of the data. Since using gather list complicates the

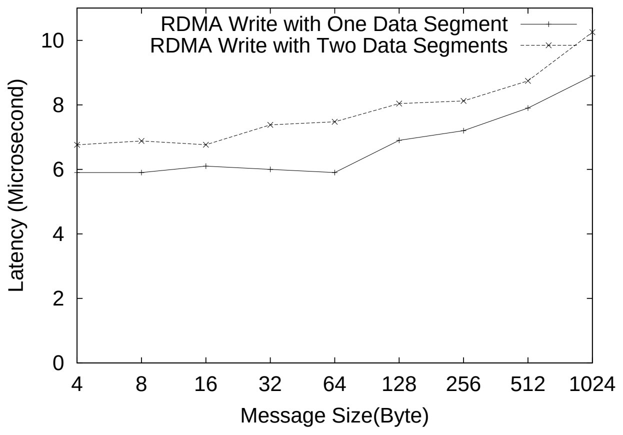

Figure 6.5: Latency of RDMA Write Gather

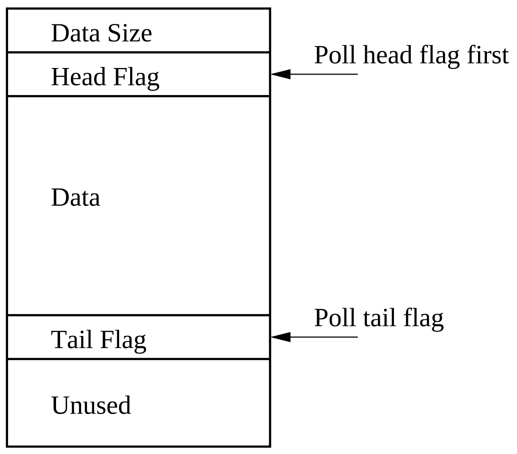

Figure 6.6: RDMA Buffer Structure for Polling

implementation and also degrades performance (more DMA operations needed) as can been in Figure 6.5, we use the second approach: putting the flag at the end of the data buffer. Although the approach uses the in-order implementation of hardware for RDMA write which is not specified in the InfiniBand standard, this feature is very likely to be kept by different hardware designers.

Putting the flag at the end of the data is slightly more complicated than it looks because the data size is variable. The receiver side thus has to know where the end of the message is and where the flag is. To do this, we organize the buffer as in Figure 6.6. The sender side sets three fields: head flag, data size and tail flag before doing the RDMA write. The receiver first polls on the head flag. Once it notices that the head flag has been set, it reads the data size. Based on the data size, it calculates the position of the tail flag and polls on it.

The above scheme has one problem. Since the receive buffer can be reused for multiple RDMA writes, it may happen that the value at the tail flag position is the same as the flag value. In this case, the send side should use two flag values and switch to another value. But how does the sender side know the value of the buffer at the receiver side? We notice that because of the persistent association between buffers, the buffer on the sender side should have the same content as the receiver side. Thus what we need to do at the sender side is the following[1]:

1. Set data size.

2. Check the position of the tail flag. If the value is the same as the primary flag value, use the secondary value. Otherwise, use the primary value.

[1]Another approach called "bottom-fill" was used in [82].

3. Set the head and tail flags.

4. Use RDMA write operation to transfer the buffer.

Initially, the head flag at the receiver side is cleared. The receiver polls by performing the following:

1. Check to see if the head flag is set. Return if not.

2. Read the size and calculate the position of the tail flag.

3. Poll on the tail flag until it is equal to the head flag.

After processing, the receive side clears the head flag.

6.2.3 Reducing Sender Side Overhead

Using RDMA write for small and control message can reduce the overhead at the receiver side because the receiver no longer needs to manage and post receive descriptors. In this section we describe how the overhead at the sender side can also be reduced by using our scheme.

At the sender side, there are two kinds of overheads related to the management of descriptors and buffers. First, before buffers can be sent out, descriptors must be allocated and all the fields must be filled. Second, after the operations are done, completion entries are generated for them in the CQ and the sender side must process them and take proper actions such as free the descriptor and the buffer.

To reduce the overheads of allocating and freeing descriptors, we store them together with the buffer. Since we have persistent association between source and destination buffers, all fields in the descriptors can be filled only once and reused except for the data size field. To deal with the overhead of completion entries in the CQ, we can use the unsignalled operations in the InfiniBand Architecture. These operations will not generate CQ entries.

6.2.4 Flow Control for RDMA Channels

As we have mentioned in the previous subsection, before the sender can reuse an RDMA buffer for another operation, it must make sure that the receiver has already finished processing this buffer. To achieve this, a flow control mechanism is implemented for the RDMA channel:

- At the sender side, the head pointer is incremented after each send in the RDMA channel.
- At the receiver side, the head pointer is incremented after an incoming message is received.
- An RDMA channel cannot be used if its head pointer is equal to its tail pointer at the sender side. In this case, we fall back and use the send/receive channel.
- The receiver maintains a credit count for the RDMA channel. Each time a receiver RDMA buffer is freed, the credit count is increased if the buffer is pointed by the tail pointer. Then the receiver goes on and checks if the following buffers were already freed. If they were, the credit count and the tail pointer are incremented for each buffer. The checking is necessary because although the arrival of messages is in order, the buffers can be freed out of order.
- For each message sent out (either RDMA channel or send/receive channel), the receiver will piggyback the credit count.

- After the sender receives a message with a positive credit count, it increases its tail pointer.

One thing we should note is that when we run out of RDMA buffers, we fall back on the send/receive channel for communication because we have separate flow control for the send/receive channel. However, these cases do not happen often and RDMA channels are used most of the time.

6.2.5 Ensuring Message Order

Since we use Reliable Connection (RC) service provided by InfiniBand for our design, messages will not be lost and they are delivered in order. However, in our RDMA-based approach, there are two potential sources of incoming messages at the receiver side for each sender: the RDMA channel and the send/receive channel. The receiver has to poll on both channels to receive messages. Therefore it might receive messages out of order. This is not desirable because in MPI it is necessary to ensure the order of message delivery. To address this problem, we introduce a *Packet Sequence Number* (PSN) field in every message. Each receiver also maintains an *Expected Sequence Number* (ESN) for every connection. When an out-of-order message arrives, the receiver just switches to the other channel and delays processing of the current packet. It stays on the other channel until the PSN is equal to the current ESN.

6.2.6 Polling Set Management

In this subsection we describe our mechanism to manage polling sets for RDMA channels. Initially, all connections use send/receive channels. Each receiver is responsible for adding or deleting connections to the RDMA polling set. When a receiver decides to add or remove a connection, it tells the sender by piggybacking or explicitly sending a control packet. The sender side takes corresponding actions after receiving this information.

There are different policies that the receiver can use to manage the RDMA polling set (add or remove a connection). For an application with only a small number of processes, all connections can be put into the RDMA polling set because the polling time is small. For large applications, we need to limit the size of RDMA polling sets in order to reduce the polling time. A simple method is to put first N (N is the size of the RDMA polling set) channels with incoming messages into the RDMA polling set. As we can see from Table 6.1, this method works for many large scientific applications. For those applications which have a large number of communication destinations for each process, we can dynamically manage the RDMA polling sets by monitoring the communication pattern. Another method is to take some hints from the applications regarding the communication frequency of each connection. We plan to investigate along some of these directions.

The order of polling in the RDMA polling set can be very flexible. Different algorithms such as sequential, circular and prioritized polling can be used. Polling order can have some impact on communication performance when the size of the polling set is relatively large. We also plan to investigate along some of these directions.

6.3 Performance Evaluation

In this section we present performance evaluation for our RDMA-based MPI design. Unlike the basic implementation, A1 InfiniHost cards and Intel compilers are used for the tests. Base MPI performance results are first given. Then we evaluate the impact of using RDMA based design by comparing it with send/receive based design. We use micro-benchmarks as well as applications (NAS Parallel Benchmarks [68]) to carry out the comparison. Finally, we use simulation to study the impact of number of RDMA channels on RDMA polling performance.

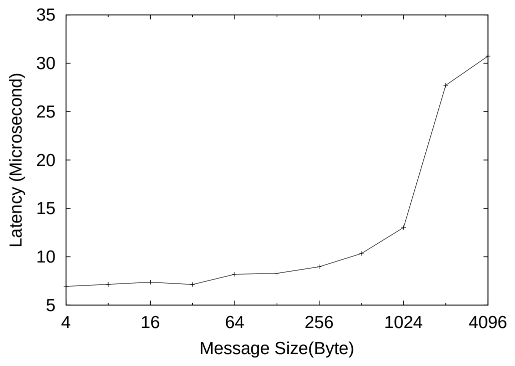

Figure 6.7: MPI Latency

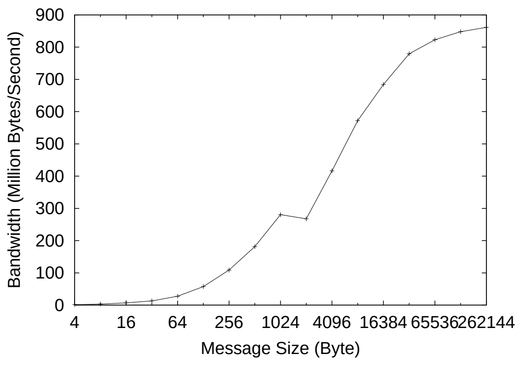

Figure 6.8: MPI Bandwidth

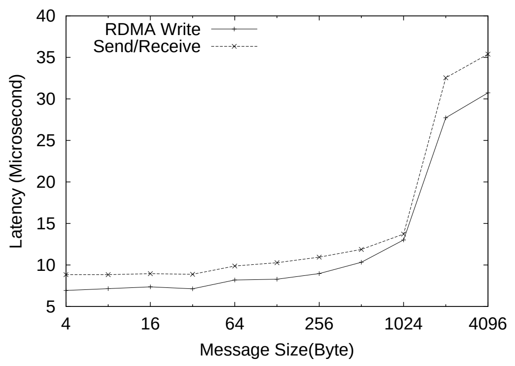

Figure 6.9: MPI Latency Comparison

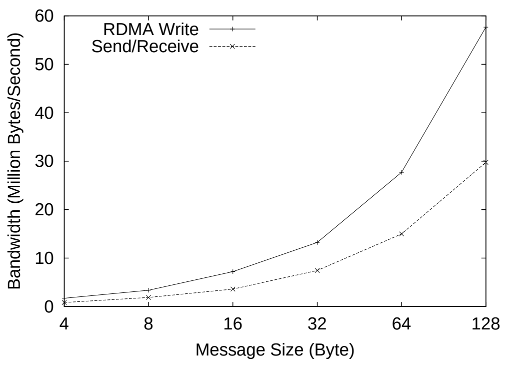

Figure 6.10: MPI Bandwidth Comparison (Small Messages)

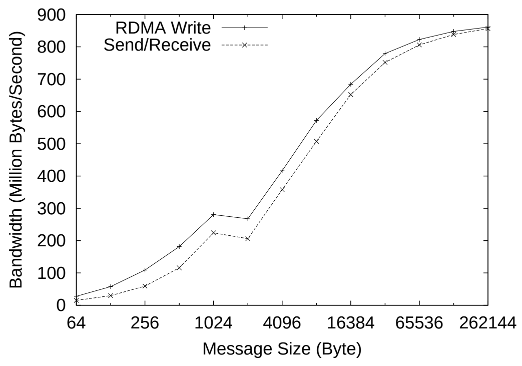

Figure 6.11: MPI Bandwidth Comparison

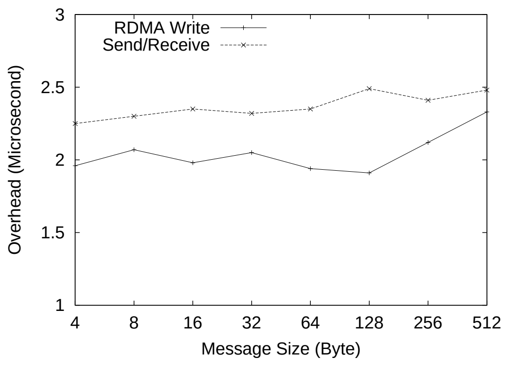

Figure 6.12: Host Overhead in Latency Test

Table 6.2: MPI Performance (Smallest Latency and Peak Bandwidth)

	Latency (us)	Bandwidth (MB/s)
This implementation	6.8	871
Quadrics	4.7	305
Myrinet/GM	7.3	242

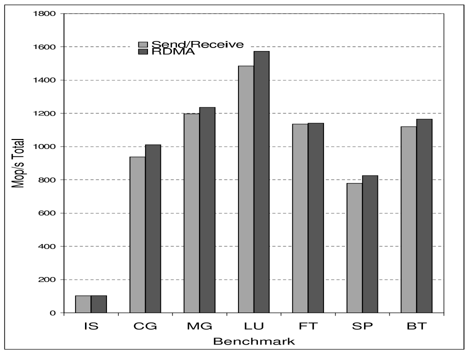

Figure 6.13: NAS Results on 4 Nodes (Class A)

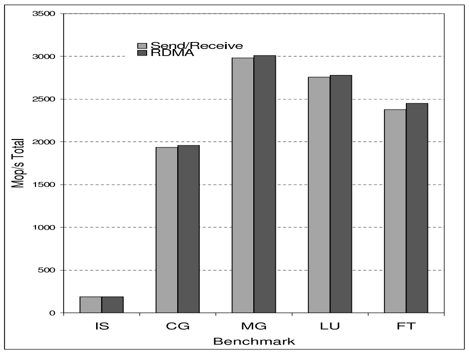

Figure 6.14: NAS Results on 8 Nodes (Class B)

Figures 6.7 and 6.8 show the latency and bandwidth of our RDMA-based MPI implementation. We have achieved a 6.8 microseconds latency for small messages. The peak bandwidth is around 871 Million Bytes (831 Mega Bytes)/second. We have chosen 2K as the threshold for switching from Eager protocol to Rendezvous protocol. Table 6.2 compares these numbers with the results we got from Quadrics Elan3 cards and Myrinet Lanai 2000 cards in the same cluster. (Please note that Myrinet and Quadrics cards use PCI-II 64x66 MHz interface while the InfiniHost HCAs use PCI-X 133 MHz interface.) From the table we see that our implementation performs quite well compared with Quadrics and Myrinet.

The latency test was carried out in a ping-pong fashion and repeated for 1000 times. In Figure 6.9, we can see that RDMA-based design improves MPI latency. For small messages the improvement is more than 2 microseconds, or 24% of the latency time. For large messages which go through the Rendezvous protocol, we can still reduce the latency by saving time for control messages. The improvement for large messages is more than 6 microseconds.

The bandwidth test was conducted by letting the sender push 100 consecutive messages to the receiver and wait for a reply. Figures 6.10 and 6.11 present the bandwidth comparison. It can be seen that RDMA-based design improves bandwidth for all message ranges. The impact is quite significant for small messages, whose performance improves by more than 104%.

LogP model for parallel computing was introduced in [18], which uses four parameters *delay, overhead, gap* and *processors* to describe a parallel machine. The overhead in communication can have significant impact on application performance, as shown by previous studies [58]. In Figure 6.12 we present the host overhead for the latency tests in Figure 6.9. We can see that the RDMA-based design can also reduce the host overhead for communication. For small messages, the RDMA-based approach can reduce the host overhead by up to 22%.

In Figures 6.13 and 6.14 we show the results for IS, MG, LU, CG, FT, SP and BT programs from the NAS Parallel Benchmark Suite on 4 and 8 nodes. (Class A results are shown for 4 nodes and class B results are shown for 8 nodes.) SP and BT require the number of processes to be a square number. Therefore, their results are not shown for 8 nodes. Program IS uses mostly large messages and the improvement of the RDMA-based design is very small. For all other programs, the RDMA-based design brings improvements as high as 7% in overall application performance.

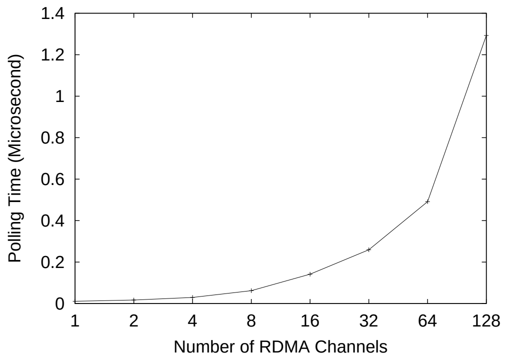

Figure 6.15: Polling Time of RDMA Channels

In order to study the behavior of the RDMA-based design for large systems, we have simulated the polling time with respect to different numbers of connections in the RDMA polling set. Figure 6.15 shows the results. We can see that even though the polling time increases with the number of connections, the time to poll a connection is very small. Even with 128 connections, the polling time is only about 1.3

microseconds. This small polling time means that the size of an RDMA polling can be relatively large without degrading performance. For applications shown in Table 6.1, a polling set with 16 connections is enough even for 1024 processes. The polling time for 16 connections is only 0.14 microseconds. Thus, our proposed design demonstrates potential for being applied to large systems without performance degradation.

6.4 Summary

Being the *de facto* standard of writing parallel applications, MPI has been implemented for numerous interconnects, including those with remote memory access abilities [87, 5, 36, 11]. [5] relies on the active message interface offered by LAPI. [36] uses PIO for small messages. Work done in [11] implemented MPI for Cray T3D based on the SHMEM interface. [87] describes an MPI implementation over Sun Fire Link Interconnect, which is also based on PIO.

RDMA operations have been used to implement MPI collective operations. Work in [82] focuses on how to construct efficient algorithms to implement collective operations by using RDMA operations. Our work can be used in conjunction with their work to efficiently transfer short data messages and control messages.

RDMA operations have also been used to design communication subsystems for databases and file systems [104, 57]. These studies do not address the issue of using RDMA for control messages. [13] evaluated different communication schemes for implementing a web server on a cluster connected by VIA. Some of their schemes use RDMA write for transferring flow control messages and file data. However, their schemes differ from ours in that they have used a sender-managed scheme which is similar to [87].

In this work, we have proposed a new design of MPI over InfiniBand which brings the benefit of RDMA to not only large messages, but also to small and control messages. We have proposed designs to achieve better scalability by exploiting application communication pattern and combining send/receive operations with RDMA operations. Our performance evaluation at the MPI level shows that for small messages, our RDMA-based design can reduce the latency by 24%, increase the bandwidth by over 104%, and reduce the host overhead by up to 22%. For large messages, performance is also improved because the time for transferring control messages is reduced. We have also shown that our new design benefits MPI collective communication and NAS Parallel Benchmarks.

CHAPTER 7

FAST AND SCALABLE MPI BROADCAST

In this part, we focus on one of the commonly used MPI collective functions : MPI_Bcast. This operation broadcasts a message to all the other nodes in a communication group. MPI_Bcast can be used alone or as building blocks for other collective operations such as MPI_Alltoall.

One of the notable feature of InfiniBand is that it supports hardware multicast. Thus, a message can be efficiently delivered to multiple receivers. Although they look similar, the semantics of hardware multicast in InfiniBand do not match with those of MPI_Bcast. For example, multicast in InfiniBand is supported only in Unreliable Datagram (UD) service and does not guarantee reliable message delivery. This leads to the following questions:

1. Can we take advantage of hardware multicast in InfiniBand to provide broadcast support in MPI?

2. How can we bridge the semantic gap of InfiniBand multicast and MPI_Bcast in an efficient and scalable manner?

In this work, we aim to provide answers to the above questions. To support MPI_Bcast, InfiniBand multicast lacks features such as reliability, in-order delivery and large message handling. We propose designing and using a substrate to enhance InfiniBand multicast by providing these features. This substrate is an integrated part of the MPI implementation and it exploits both multicast and point-to-point communication in InfiniBand.

Providing new features on top of InfiniBand multicast inevitably brings extra overhead. To achieve high performance and scalability, we have used two key design strategies. The first one is to remove the overhead from communication critical path so that it happens in the background. The second one is to balance and reduce this background overhead so that it is not a performance bottleneck in most cases. Based on the first strategy, we have proposed a sliding window based design which enables the root of MPI_Bcast to proceed without waiting for other nodes to send ACKs. Based on the second strategy, we have introduced a *co-root* scheme to balance background ACK traffic and various delayed ACK techniques to reduce the ACK traffic. We have also addressed many detailed design issues such as buffer management, efficient handling of out-of-order and duplicate messages, timeout and retransmission, flow control and RDMA based ACK communication.

7.1 MPI_Bcast Overview

MPI supports both point-to-point and collective communication functions. MPI_Bcast is a commonly used collective function in writing parallel applications. It broadcasts a message from a root process to other processes in a communication group, which is specified by an MPI communicator. In many cases, this communicator is MPI_COMM_WORLD, whose communication group includes all the processes participating in the MPI application.

MPI_Bcast is a blocking operation. For a root node, the operation does not return until the communication buffer can be reused. For a receiver node, the operation returns only after the broadcast data has been delivered into the receive buffer. However, it is not necessary that the operation returns only after the broadcast is finished at the root.

In many MPI implementations, MPI_Bcast is implemented with a tree-based algorithm, which exploits point-to-point communication operations. This approach is used in our MPI implementation over InfiniBand: MVAPICH [55, 53]. In the tree based approach, the number of hops to reach leaf nodes increases with the total number nodes (typically in a logarithmic manner). Therefore, MPI_Bcast latency also increases. In Figure 7.1, we show MPI_Bcast performance in MVAPICH using point-to-point communication. It can be seen that MPI_Bcast latency increases with the number of nodes.

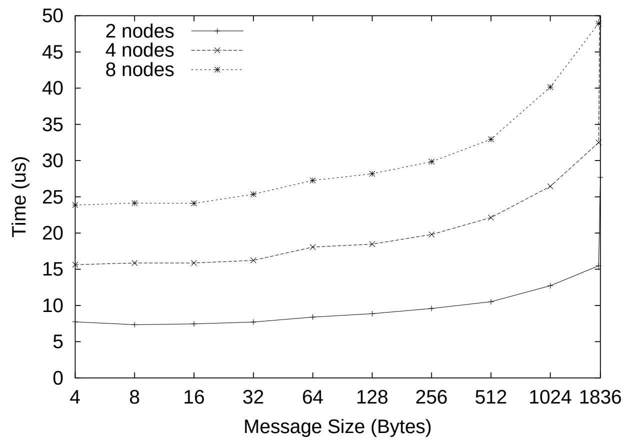

Figure 7.1: MPI_Bcast Latency in MVAPICH Using Point-to-Point Communication

Another drawback of tree based implementations is that if hosts are involved in intermediate nodes to forward broadcast messages, skew between different processes may significantly delay the forwarding [12]. This has adverse impact on the execution time of an application. Thus, the challenge is whether the hardware supported multicast scheme can alleviate the impact of process skew.

7.2 Designing MPI_Bcast with InfiniBand Multicast

There have been many studies about multicast and reliable multicast in the networking area [31, 23, 47]. A majority of the work done in this area focuses on networks based on TCP/IP protocol. Our work deals with implementing MPI_Bcast in InfiniBand. Compared with a general TCP/IP network, InfiniBand offers much higher communication performance and hardware supported multicast. Also, group membership in MPI is much more static than that in the dynamic environment of a TCP/IP network.

Recently, different collective operations in MPI have been studied on interconnects such as Virtual Interface Architecture (VIA) [30], Quadrics [79], Myrinet [103] and IBM SP [93]. Compared with these interconnects, InfiniBand provides new challenges and opportunities for implementing MPI collective operations. Our previous work [43] proposed an RDMA based scheme to implement efficient barrier operations over InfiniBand. In this work, we continue our work in this direction by presenting different broadcast designs while exploiting the hardware multicast support.

In the previous section, we have seen that MPI_Bcast implementations based on point-to-point communication are not scalable with respect to the number of processes. It also indicates that point-to-point implementations are susceptible to process skew. InfiniBand multicast provides a more scalable way of delivering a single message to multiple destinations. However, there are several major differences between InfiniBand multicast and MPI_Bcast:

1. InfiniBand multicast does not guarantee reliability, while in MPI, communication must be reliable.

2. Since InfiniBand multicast uses connectionless UD service, there is no guarantee regarding the ordering of multicast messages. However, MPI specifies that all collective operations must be matched according to the order they are initiated.

3. In InfiniBand UD service, the size of a message cannot exceed the MTU (Maximum Transfer Unit), which is typically 2K Bytes. MPI does not limit the message size in MPI_Bcast.

In other words, there exists a semantic gap between InfiniBand multicast and MPI_Bcast. This issue must be addressed to take advantage of hardware multicast in an MPI implementation. In this work, we propose a substrate which bridges this gap. As shown in Figure 7.2, this substrate sits on top of the underlying InfiniBand layer, exploiting multicast as well as other InfiniBand functionalities. It also interacts with other parts of the MPI implementation. To achieve high performance and scalability, we need to not only implement this substrate, but also do it in an efficient and scalable manner.

In designing the substrate, we need to address three issues: reliability, in-order delivery, and handling of large messages. Previous study [95] has shown that data sizes in MPI collective operations are typically quite small. Therefore, we will first concentrate on efficient handling of small messages. We will deal with large messages specifically at the end of this subsection.

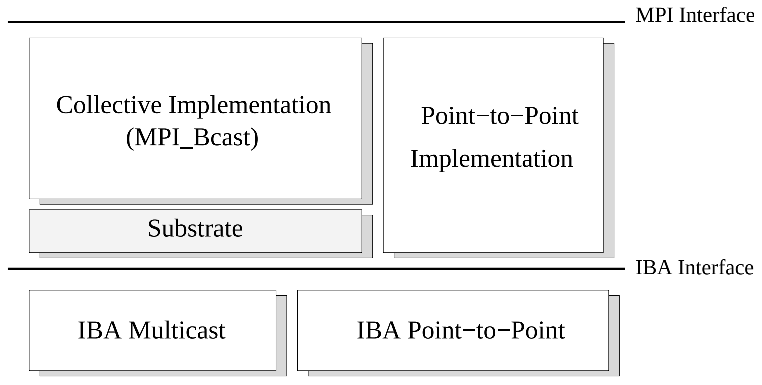
MPI Interface
Collective Implementation (MPI_Bcast)
Point−to−Point Implementation
Substrate
IBA Interface
IBA Multicast
IBA Point−to−Point

Figure 7.2: Bridging the Gap between InfiniBand Multicast and MPI_Bcast

In the following, we propose several designs used in the substrate. We first describe a basic design which is easy to understand and also straightforward to implement. Then we present several new designs which deal with performance and scalability issues of the basic design. Although reliability and ordering are two different issues, they can be addressed together. In the following discussions, we primarily focus on how to implement reliability. We have chosen ACK based approaches, in which delivery is confirmed by acknowledgments and message loss is handled by timeout/retransmission. In the proposed schemes, we also address how message ordering can be ensured for MPI_Bcast.

7.2.1 Basic Design

In the basic design, the root node of MPI_Bcast sends out a message using multicast and other nodes wait for this message. If the message is received, an ACK is sent back to the root node. The root blocks and waits for all ACKs to be received. If not all ACKs arrive within a certain period of time, it times out and retransmits the message using reliable point-to-point communication (using the RC service, as defined by the InfiniBand standard).

The basic design uses ACKs and timeout/retransmission to provide reliability. Two different broadcast messages from the same root are guaranteed to arrive in-order because the root node blocks for ACKs of the first message before it can send out the second one.

However, there are several major problems in this basic design. First, making the root block for all the ACKs significantly increases the overhead of the MPI_Bcast call at the root. Second, since all other nodes send back ACKs to a single root node, a hot spot is created at the root, which becomes a performance bottleneck when the total number of nodes in a system is large. This problem is also referred to as *ACK implosion* [80]. In the following subsections, we will address these problems.

7.2.2 Sliding-Window Based Design

Our basic design leads to poor performance because the root has to wait for all the ACKs to be received. MPI specifies that MPI_Bcast can return immediately after the buffer can be reused. Therefore, it is not necessary for the root to wait for all the ACKs to be received.

In order to alleviate this problem, we propose a solution which makes a copy of the user buffer. After the multicast operation is initiated, we can immediately return without waiting for all the ACKs to be received. To handle multiple outstanding MPI_Bcast initiated from a single root, we use a number of pre-allocated buffers at each root. These buffers are organized as a ring. A sliding-window based approach is used to manage these buffers, as shown in Figure 7.3. A buffer is consumed for each new MPI_Bcast operation. When all ACKs for this operation have arrived, this buffer can be freed and reused for other MPI_Bcast operations.

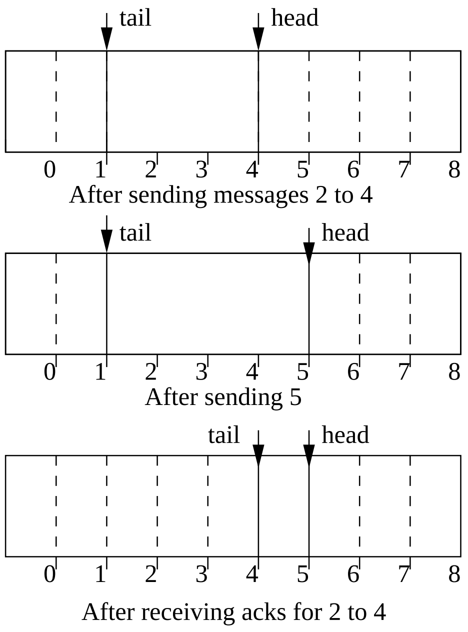

Figure 7.3: Sliding Window Buffer Management

Compared with the basic design, the sliding-window based design decouples ACK processing from the multicast. In other words, ACK processing is no longer done in the critical path of broadcast, but carried out in the "background". If the window size is sufficiently large such that all ACKs can arrive and be processed in time, MPI_Bcast will not block due to running out of buffers. As a result, the performance of MPI_Bcast can be significantly improved.

The window based design also has its drawbacks. First, the data in user buffers has to be copied to buffers in the window, which increases processing overhead. Fortunately, the typical size of MPI_Bcast is small and the copying overhead is negligible.

Another problem is that it consumes more buffer than the basic design. We can control the buffer space used by changing the total window size. The third issue is that this design does not solve the ACK implosion problem. Although ACK processing is now done in the background, it still happens that all ACKs arrive at the same root node. Therefore, the root can become a performance bottleneck in this design for large scale systems.

7.2.3 Avoiding ACK Implosion

To solve the ACK implosion problem, we should not let all the receivers send ACKs to a single root node. The basic idea to deal with this problem is to use a hierarchical structure for ACK collection and distribute the load to a number of nodes. One solution is to use a tree based structure to collect ACKs. In this approach, all nodes form a tree structure, with the root node being the root of the tree. Intermediate nodes are responsible for collecting ACKs for its children. After all ACKs have come from its children, an intermediate node sends an ACK to its parent node. The root node only needs to collect ACKs from its direct children instead of all other nodes.

The drawback of the tree based ACK collection is that it depends on intermediate nodes for ACK processing. Thus, ACK collection time depends on the communication progress of intermediate nodes. (A similar problem has been discussed in [12].) In a polling based MPI implementation such as MPICH, communication progress is only made within MPI function calls. Therefore, if an intermediate node is doing lengthy computation, ACK processing and forwarding could be delayed. The problem becomes even more serious when the tree has multiple levels. As a result, it is very hard to determine the timeout value for retransmission at the root. When ACK processing at intermediate nodes are delayed, the tree based ACK collection is prone to *false retransmission*, which is triggered by delayed ACKs instead of real message loss. To make matters worse, a single delayed ACK will result in the root node retransmitting the message to everyone in the same sub-tree, which can generate a lot of network traffic and increase the overhead of the root node.

To solve the ACK implosion problem and also to address problems with the tree based scheme, we propose a new ACK collection scheme called the *co-root scheme*. In this scheme, in addition to the root node, we select a subset of other nodes as *co-roots*. The remaining nodes are called *leaf nodes*. Each of the root and the co-roots is responsible for a group of leaf nodes. The basic idea is to guarantee that co-roots can get messages reliably and use them to help ACK processing. The co-root scheme is illustrated in Figure 7.4 and it consists of the following steps:

1. The root uses multicast to transfer the message to every other node.

2. The root does a small scale "broadcast" to all co-roots. The broadcast is done using reliable point-to-point communication. A tree based algorithm can be used, just like that in the current MPI implementation.

3. Each of the root and the co-roots collects ACKs from all other nodes in its sub-group. If timeout happens, the root or the appropriate co-root will do the retransmission.

Similar to the tree based ACK collection, the co-root scheme also uses a hierarchical structure to delegate ACK collection and processing to other nodes. They both aim to solve the ACK implosion problem. However, there are also major differences between them. The co-root scheme is a two-level hierarchy. After the message is

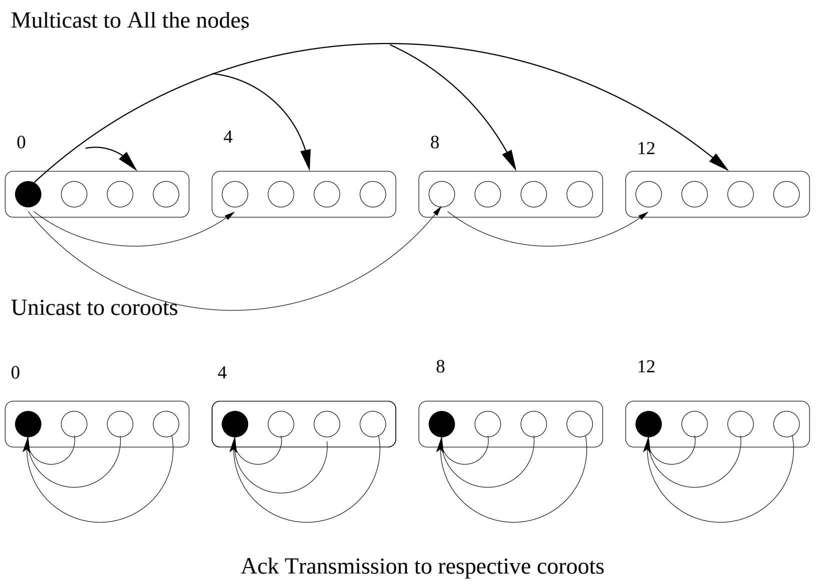
Multicast to All the nodes
0
4
8
12
Unicast to coroots
0
4
8
12
Ack Transmission to respective coroots

Figure 7.4: Co-Root Scheme

delivered to a co-root, the co-root essentially plays the same role as the root and ACK processing for its sub-group is completely decoupled from the root. In a tree based scheme, intermediate nodes are responsible for ACK collection and forwarding, while the root is responsible for ACK collection and retransmission. The ACK processing is not completely decoupled from the root because it has to handle all the retransmissions.

The co-root scheme has several advantages over a tree based scheme. Since co-roots now help with *both* ACK collection and retransmission, the load is more evenly distributed. The co-root scheme does not depend on the progress of intermediate nodes. As a result, it is easier to determine the timeout value for a given system size. The co-root scheme also results in fewer false retransmissions. (Note that false retransmission can still happen if an ACK from a leaf to its co-root is delayed.) Another advantage of the co-root scheme is that each co-root keeps information of all the leaf nodes in its sub-group. When an ACK is not received, retransmission is done only to that particular node. In a tree based scheme, the root can only track other nodes at the level of sub-trees. Therefore, retransmission must be done for all nodes in that sub-tree, which increases overhead and network traffic.

The co-root scheme also has its disadvantages. First, delivering the message reliably to every co-root introduces extra root processing overhead and network traffic. However, it should be noted that usually the co-root scheme does not increase latency of the broadcast. At any co-root, the broadcast can be completed when it receives either the multicast message or the "reliable broadcast" message. It does not have to wait for both messages. The second problem of the co-root scheme is that a copy of the message is duplicated at all co-roots. Therefore, it consumes more buffer space compared with a tree-based scheme. Another issue for co-root scheme is that we must carefully determine the number of co-roots (or the sub-group size). We address this issue (determining optimal number of co-roots) in Section 6.4.5.

7.2.4 Reducing ACK Traffic

ACK implosion avoiding schemes distribute ACK processing and retransmission tasks from the root to other nodes, but they do not reduce the total number of ACK messages. To improve utilization of the network resource and to avoid possible network congestion, it is also desirable to reduce ACK traffic.

Our basic idea of reducing ACK traffic is to send ACKs in a lazy manner. We propose two schemes:

1. *Piggybacking.* In this scheme, a node attaches the ACK with other messages instead of sending it as a separate message. If there is no message sending out

to the ACK destination after a certain period of time, an explicit ACK message is sent.

2. *Acknowledge every M broadcast messages.* Instead of sending an ACK for every broadcast message, we only send one ACK for every M broadcast messages. Timeout and explicit ACK messages are also used.

Both schemes can reduce the total amount of ACK traffic. The effectiveness of piggybacking is very dependent on the communication pattern of the application. In the best case, all ACKs can be attached with other messages. In the worst case, timeout happens and we have to send the ACK using an explicit message. However, even in this worst case, we can still possibly reduce ACK traffic. This is because we wait for the timeout before sending out the ACK messages. Therefore, if there are multiple broadcast messages received from the same root, they can be acknowledged using a single ACK message.

The second scheme effectively reduces the ACK traffic to 1/M of the original amount if there are many back-to-back broadcasts. However, one problem with the scheme is that after every M messages, the root will receive ACKs from all other nodes. This leads to similar situations as ACK implosion. To solve this problem, we introduce a technique called *skewed ACK*. In this technique, every node still acknowledges after receiving every M messages. However, they now do the ACK in a more independent way. For example, suppose there are n nodes in the broadcast group and every broadcast message has a sequence number B, then node i can generate an ACK based on the following condition: $B \mathbf{\ mod\ } M = i \mathbf{\ mod\ } M$.

We should note that schemes to avoid ACK implosion and to reduce ACK traffic are complementary. By combining both schemes, we can achieve even more benefit.

For example, the schemes proposed in this subsection can be used to reduce ACK traffic in the co-root scheme proposed in the previous subsection.

7.2.5 Dealing with Large Messages

In previous discussions, we dealt with small broadcast messages which can fit into a single buffer. (The buffer size is no larger than InfiniBand MTU.) Large messages can be divided into smaller chunks and sent out using multiple buffers. The techniques we have discussed previously are still applicable in this case. However, the copying cost may be significant because the message size is large.

Since large broadcast messages are relatively infrequent, an alternative way is to fall back on schemes based on point-to-point communication. The advantage of this approach is simplified design and implementation. Also the overhead of the root due to the copy can be eliminated because zero-copy point-to-point communication can be used for transferring the message.

7.3 Detailed Design Issues

In this part, we discuss some of the detailed design issues in our MPI_Bcast designs. These issues include buffer management, out-of-order and duplicate message handling, timeout and retransmission, flow control and RDMA based ACK communication.

7.3.1 Buffer Management

To ensure reliability of MPI_Bcast, we have to store broadcast messages in buffers until we can be sure that every other node has received this message. Therefore, buffer management is an important issue in our design.

For each node, a number of pre-allocated buffers are used for storing broadcast messages sent by this node. Since InfiniBand requires that communication buffers must be registered, we pre-register these buffers so as to save cost during communication. The buffers are organized as a ring and managed using a sliding window based algorithm. For each new broadcast, the message is copied to the buffer at the head of the window. For a buffer at the tail of the window, if we have collected all ACKs, we free this buffer by incrementing the tail pointer. For the co-root scheme, a window of buffers exist also in all co-roots and are managed in the same way.

One parameter we have to decide in buffer management is the window size. A larger window size means that the application can issue a large number of back-to-back broadcast without blocking because of delayed ACKs. However, using a large window size also consumes more buffer space. This parameter is best decided by the communication pattern of applications. Currently we use a static value for window size which can be changed at compile time.

One issue we have to deal with is what to do if we run out of buffers. In the current implementation, we treat this situation in the same way as timeout. Thus, we will retransmit the message to all nodes from which the ACK has not come.

7.3.2 Handling Out-of-Order and Duplicate Messages

Multicast messages in InfiniBand use Unreliable Datagram transport service, which does not maintain message order. Duplicate messages can also be sent to a receiver due to false retransmission or algorithms used in co-root scheme. These situations are handled by using sequence numbers attached with each broadcast message.

Each receiver maintains a counter which specifies the sequence number of the next broadcast message it is expecting. If the sequence number of the next message is equal to the counter, the message is processed and the counter is incremented. If the sequence number is larger than the counter, the processing is delayed and the message is put into a queue. If an arriving message is a duplicate, its sequence number is either less than the counter value or equal to the sequence number of one of the messages in the queue. In this case, the message is not processed but silently discarded.

7.3.3 Timeout and Retransmission

Whenever a root issues a multicast message, it sets a timeout value for this message. For the co-root scheme, all the co-roots also set a timeout value after receiving the message from the root. When we set or check the timeout value, the current time value is obtained by reading the time stamp counter register provided by the Intel Pentium architecture. This approach has very low overhead. To check if a timeout value has been reached, we use a polling based approach. Therefore, timeout and retransmission only happen inside MPI functions calls. An alternative is to use an interrupt based method. However, this approach is not used because it brings many race conditions and does not match well with the polling based implementation of MPICH.

There are many factors which affect the timeout value, such as multicast and point-to-point latency, process skew, window size at the root (or co-roots), the number of co-roots and the system size. Currently, we use a static value which can be changed at compile time. We plan to investigate these issues in future with the availability of large-scale InfiniBand clusters.

Retransmission is always done using reliable point-to-point communication. After retransmission, the message buffer can be freed because we are now sure that the message can arrive at the receiver. In certain retransmission cases, such as those when a large number of ACKs are not received, it may be more efficient to re-issue the multicast operation. However, we decided not to use this approach because it complicates the implementation and these cases are quite rare.

7.3.4 Flow Control

In UD service, a multicast send operation will consume one buffer at every receiver. If there are not enough buffers posted at the receiver, incoming messages may be dropped. The purpose of flow control is to keep the root from sending if the receivers have not posted enough receive buffers.

We use a credit-based scheme for flow control. During initialization, a number of buffers are pre-posted and each node has an array of credit values for every other node which are equal to the number of pre-posted buffers. After receiving a broadcast message, a node will decrement the credit count of all nodes because a multicast operation will consume buffers at all receivers. After the message is processed and the buffer is re-posted at a receiver, the credit count for this node should be incremented at other nodes. This information is transferred using piggybacking. Both point-to-point messages and multicast messages can carry piggybacked credit information.

7.3.5 RDMA Based ACK communication

In our previous study [55], we have shown that RDMA operations in InfiniBand provide better performance than send/receive operations. Another advantage of RDMA is that there is no descriptor posting or management overhead at the receiver.

To improve performance of ACK collection in MPI_Bcast, we have used RDMA write operations for ACK collections. To send back an ACK, a receiver issues an RDMA write to a memory location at the root (or its co-root). The root or co-root only needs to check the memory location in order to find out if an ACK has come. Since this check only involves memory read, it is very efficient.

7.4 Performance Evaluation

In this subsection, we evaluate performance of our MPI_Bcast designs based on InfiniBand multicast. We present results for several different designs and compare them with the original implementation in MVAPICH, which is a point-to-point implementation based on the binomial tree algorithm. We characterize broadcast performance using two micro-benchmarks: *latency* and *throughput*. We also show how process skew can affect different implementations. Since different ACK implosion avoiding techniques and ACK traffic reducing techniques may be combined, we can have different combinations for multicast based schemes.

We have chosen only a subset of all possible combinations in the performance evaluation. All schemes used in our tests and their abbreviations are as follows:

- Original: the original implementation based on point-to-point communication.
- Basic: the basic design.
- Window: sliding-window based design without ACK implosion avoiding or ACK traffic reduction.
- Co-root2: sliding-window based with one co-root.
- Aggregate10: sliding-window based with ACK for every 10 messages.

7.4.1 Latency Test

We define *broadcast latency* to be the time it takes for a broadcast message to reach every receiver. The test consists of a loop, in which an MPI_Bcast is issued from a root node and the receivers take turns to send back an acknowledgment using MPI_Send. The broadcast latency is derived from the time to finish each iteration and the MPI point-to-point latency.

Figure 7.5 shows the broadcast latency results for different designs. The buffer size in the multicast window is 2K bytes, which is equal to the MTU. However, because of the message header and other overhead (a portion of the buffer is used to store descriptor.), currently we can send a payload of up to 1836 bytes in a single buffer. We can see that our new implementations based on InfiniBand multicast performs significantly better than the original broadcast implementation based on point-to-point communication. In the broadcast latency test, most of the ACK processing is carried out in the background. Thus, all the multicast based designs shown in the figure perform comparably. For small messages, multicast based designs can perform up to 58% better than the original design. (Using the co-root scheme introduces around 2μs overhead for messages larger than 32 bytes. We are currently investigating this issue.)

Figure 7.6 compares one of the designs (Window) with the original design. We can see that although handling large messages requires fragmentation and reassembly of messages and extra copies, the performance can still be improved by using InfiniBand multicast for message sizes up to 32K bytes. For example, the improvements are 210% for 2K byte messages and 86% for 8K byte messages.

7.4.2 Throughput Test

We use *broadcast throughput* to measure how fast MPI_Bcast operations can be issued and finished. In this test, a number of back-to-back MPI_Bcast operations are issued from a root node. The throughput is simply the number of broadcast operations finished divided by the total time.

Figure 7.8 presents the throughput results for a number of different designs. We can see that the basic design performs the worst even though it uses InfiniBand multicast. This is because it always waits for all the ACKs before initiating the next broadcast. However, if we use a sliding-window based design, we can perform significantly better than the original design. By using ACK reducing technique, the performance can be further improved because the overhead to process ACKs is reduced. We can see that Aggregate10 scheme can perform up to 112% better than the original scheme in terms of throughput.

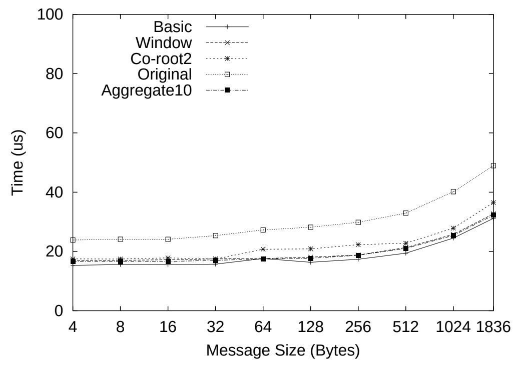

Figure 7.5: MPI_Bcast Latency for Small Messages Using Various Multicast Schemes(8 Nodes)

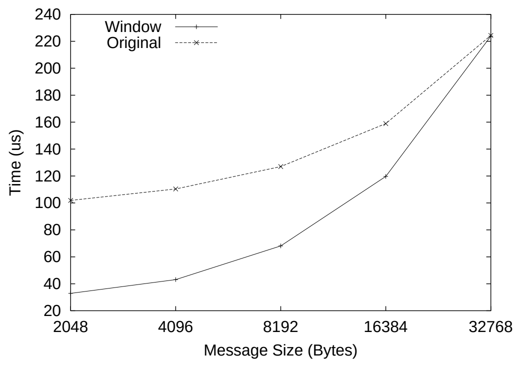

Figure 7.6: MPI_Bcast Latency for Large Messages (8 Nodes)

7.4.3 Impact of Process Skew

To measure the effect of process skew on broadcast performance, we use a test similar to that in [12]. The test consists of a loop, in which a barrier operation is performed before a broadcast. To emulate the effect of process skew, a random delay is inserted between the barrier and the broadcast for all the receivers. We then measure the average time spent in the MPI_Bcast function.

Figure 7.7 shows the impact of process skew on MPI_Bcast. Schemes Window and Original are chosen for comparison. We can see that multicast based scheme is not affected by process skew at all. This is because InfiniBand multicast is supported by hardware and does not depend on intermediate nodes to make progress. In contrast, the original design relies on intermediate nodes to forward broadcast messages. Therefore, as process skew increases, the receivers spend more time in MPI_Bcast. When the process skew is 400μs, the multicast based scheme performs 10 times better than the original design in the process skew test.

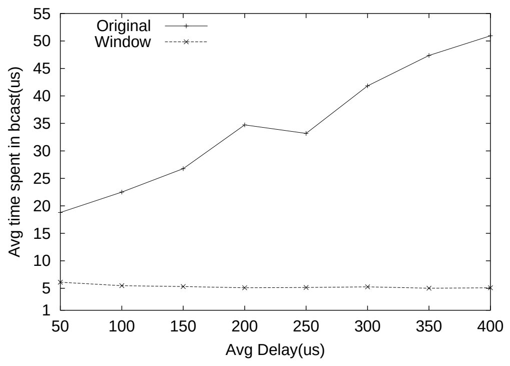

Figure 7.7: Impact of Process Skew on MPI_Bcast (8 Nodes)

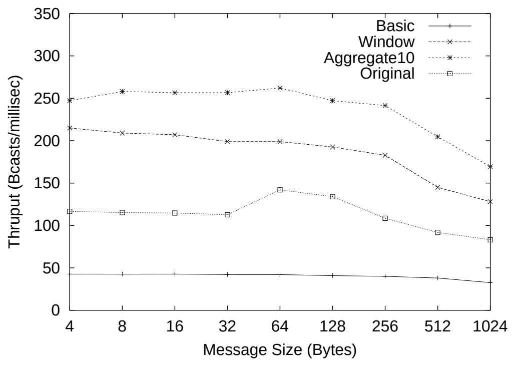

Figure 7.8: MPI_Bcast Throughput (8 Nodes)

7.5 Analytical Model

In this section, we use analytical models to characterize different MPI_Bcast implementations. Since our experiments were done in a relatively small testbed, these models help us to estimate the performance of different schemes in large scale systems. Here we present main results derived from our model. Model details and validation can be found in [49].

7.5.1 Modeling Broadcast Latency

Figure 7.9 shows the estimated results for the original implementation and the sliding-window based schemes in a 1024 node cluster. (In our model, the number of co-roots does not have significant impact on the results.) It should be noted that the results presented here are for *ideal* cases. For the original implementation, we do not take process skew into consideration. In the sliding-window based scheme, we assume that ACK process can be overlapped without adding extra overhead. Therefore, the results serve as a lower bound for both cases. From the figure we can see that in a 1024 node cluster where each node has similar configuration as those in our testbed, using InfiniBand multicast can improve MPI_Bcast performance significantly. For small message, the potential latency improvement can be as high as 3.86 times.

7.5.2 Determining the Optimal Number of Co-Roots

One of the important issues in the co-root scheme we proposed is to determine the number of co-roots for a given system. Since in the broadcast latency test, most of the processing overhead is in the background, different number of co-roots tend to give similar performance. Therefore, we used broadcast throughput to help us determine the optimal number of co-roots.

From Figure 7.10 we see that with 1024 nodes, performance degrades with too many or too few number of co-roots. This is because that if the number of co-roots is large, the root needs to spend a large amount of time to reliably deliver the message co-roots. If the number of co-roots is too small, then each co-root needs to spend a large amount of time processing ACKs because the sub-group size is large. From the figure, 128 co-roots are the best choice for 1024 nodes.

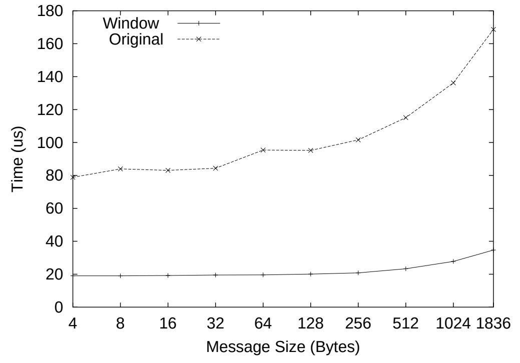

Figure 7.9: Estimated MPI_Bcast Latency for Small Messages (1024 Nodes)

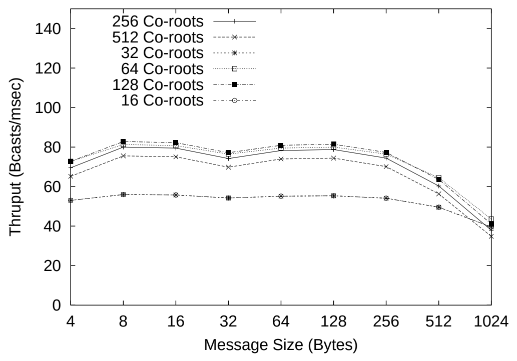

Figure 7.10: Estimated MPI_Bcast Throughput for Different Number of Co-Roots (1024 Nodes)

7.6 Summary

In this work, we described how to take advantage of hardware multicast in InfiniBand to implement MPI_Bcast operation in MPI. To support MPI_Bcast, we proposed a substrate on top of InfiniBand multicast which provides reliability, in-order delivery and handling of large messages. To improve performance of the substrate, we use sliding window based design which removes much of the processing from communication critical path. To further balance and reduce processing overhead, we proposed techniques such as the *co-root* scheme and *delayed ACK*.

Our performance evaluation on our 8-node cluster shows that our designs can improve MPI_Bcast latency up to 58% and throughput up to 112% compared with the current implementation. Our new designs also have much better tolerance to process skew. Our measurements show that the new designs outperform the current implementation by a factor of 10 when the average skew is 400μs on 8 nodes. To get more understanding of the performance of MPI_Bcast in large-scale clusters, we use analytical modeling to estimate the performance of different designs. Our results show that in a 1024 node cluster, our designs can perform 3.86 times better than the current design.

CHAPTER 8

SUPPORTING MULTIRAIL INFINIBAND CLUSTERS

One of the notable features of InfiniBand is its high bandwidth. Currently, InfiniBand 4x links support a peak bandwidth of 1GB/s in each direction. However, even with InfiniBand, network bandwidth can still become the performance bottleneck for some of today's most demanding applications. This is especially the case for clusters built with SMP machines, in which multiple processes may run on a single node and must share the node bandwidth.

One important way to overcome the bandwidth bottleneck is to use *multirail networks* [16]. The basic idea is to have multiple independent networks (rails) to connect nodes in a cluster. With multirail networks, communication traffic can be distributed to different rails. There are two ways of distributing communication traffic. In *multiplexing*[2], messages are sent through different rails in a round robin fashion. In *striping*, messages are divided into several chunks and sent out simultaneously using multiple rails. By using these techniques, the bandwidth bottleneck can be avoided or alleviated.

In this chapter, we present a detailed study of designing high performance multirail InfiniBand clusters. We discuss various ways of setting up multirail networks with

[2]Also called *reverse multiplexing* in the networking community.

InfiniBand and propose a unified MPI design that can support all these approaches. Our design achieves low overhead by taking advantage of RDMA operations in InfiniBand and integrating the multirail design with MPI communication protocols. Our design also features a very flexible architecture that supports different policies of using multiple rails. We have provided in-depth discussions of different policies and also proposed an adaptive striping policy that can dynamically change the striping parameters based on the current available bandwidth of different rails.

We have implemented our design and evaluated it using both microbenchmarks and applications using an 8-node InfiniBand testbed. Our performance results show that multirail networks can significantly improve MPI communication performance. With a two rail InfiniBand network, we have achieved almost twice the bandwidth and half the latency for large messages compared with the original MPI. The peak unidirectional bandwidth and bidirectional bandwidth we have achieved are 1723 MB/s and 1877 MB/s, respectively. Depending on the communication pattern, multirail MPI can significantly reduce communication time as well as running time for certain applications. We have also shown that for rails with different bandwidth, the adaptive striping scheme can achieve excellent performance without *a priori* knowledge of the bandwidth of each rail. It can even outperform static schemes with *a priori* knowledge of rail bandwidth in certain cases.

8.1 InfiniBand Multirail Network Configurations

InfiniBand multirail networks can be set up in different ways. In this section, we discuss three types of possible multirail network configurations and their respective benefits. In the first approach, multiple HCAs are used in each node. The second approach exploits multiple ports in a single HCA. Finally, we describe how to set up *virtual multirail networks* with only a single port by using the LID mask control (LMC) mechanism in InfiniBand.

8.1.1 Multiple HCAs

Although InfiniBand Architecture specifies 12x links, current InfiniBand HCAs in the market can support only up to 4x speed. A straightforward way to alleviate the bandwidth bottleneck is to use multiple HCAs in each node and connect them to the InfiniBand switch fabric. Through the support of communication software, users can take advantage of the aggregated bandwidth of all HCAs in each node without modifying applications. Another advantage of using multiple HCAs per node is that possible bandwidth bottlenecks in local I/O buses can also be avoided. For example, the PCI-X 133 MHz/64 bit bus (used by most 4x HCAs in the current market) can only support around 1 GB/s aggregated bandwidth. Although a 4x HCA has a peak aggregated bandwidth of 2 GB/s for both link directions, its performance is limited by the PCI-X bus. These problems can be alleviated by connecting multiple HCAs to different I/O buses in a system.

A multirail InfiniBand setup using multiple HCAs per node can connect each of HCAs in a node to a separate switch. If a larger switch is available, all HCAs can also be connected to this single physical network. Through the use of appropriate switch configurations and routing algorithms, using a single network can be equivalent to a multirail setup.

8.1.2 Multiple Ports

Currently, many InfiniBand HCAs in the market have multiple ports. For example, InfiniHost HCAs [60] from Mellanox have two ports in each card. Therefore, multirail InfiniBand networks can also be constructed by taking advantage of multiple ports in a single HCA. This approach can be very attractive because compared with using multiple HCAs, it only requires one HCA per node. Hence, the total cost of multirail networks can be significantly reduced.

However, as we have discussed, the local I/O bus can be the performance bottleneck in such a configuration because all ports of an HCA have to share the I/O bus. Hence, this approach will not achieve any performance benefit by using 4x HCAs with PCI-X buses. However, benefits can be achieved by using future HCAs that support PCI-X Double Data Rate (DDR) or Quad Data Rate (QDR) interfaces. Recently, PCI Express [76] has been introduced as the next generation local I/O interconnect. PCI Express uses a serial, point-to-point interface. It can deliver scalable bandwidth by using multiple lanes in each point-to-point link. For example, an 8x PCI Express link can achieve 2 GB/s bandwidth in each direction (4 GB/s total). Multiple port InfiniBand HCAs that support PCI Express are already available in the market [62]. Therefore, this approach can be very useful in constructing multirail networks using systems that have PCI Express interfaces.

8.1.3 Single Port with LID Mask Control (LMC)

In this subsection, we discuss another approach of setting up multirail InfiniBand networks which does not require multiple ports or HCAs for each node. The basic idea of this approach is to set up different paths between two ports on two nodes. By using appropriate routing algorithms, it is possible to make the paths independent of each other. Although a single network is used in this approach, we have multiple logical networks (or logical rails). If the logical networks are independent of each other, conceptually they are very similar to multirail networks. Therefore, we call this approach as *virtual multirail networks.*

In InfiniBand, each port has a *local identifier* (LID). Usually, a path is determined by the destination LID. Therefore, multiple LIDs need to be used in order to have different paths. To address this issue, InfiniBand provides a mechanism called *LID Mask Control* (LMC). Basically, LMC provides a way to associate multiple logical LIDs with a single physical port. Hence, multiple paths can be constructed by using LMC.

It should be noted that in virtual multirail networks, a port is shared by all the logical rails. Hence, if the port link bandwidth or the local I/O bus is the performance bottleneck, this approach cannot bring any performance benefit. It can only be used for fault tolerance in this case. However, if the performance bottleneck is inside the network, virtual multirail networks can improve communication performance by utilizing multiple paths.

8.2 Multirail MPI Design

In this section, we present various high level design issues involved in supporting multirail networks in MPI over InfiniBand. We first present the basic architecture of our design. After that, we discuss how we can have a unified design to support multirail networks using multiple HCAs, multiple ports, multiple connections for a single port, or any combination of the above. Then we describe how we can achieve low overhead by integrating our design with MPI and taking advantage of InfiniBand RDMA operations. One important component in our architecture is *Scheduling Policies*. In the last part of this section, we discuss several policies supported by our architecture and also present an *adaptive striping scheme* that can dynamically adjust striping parameters based on current system conditions.

8.2.1 Basic Architecture

The basic architecture of our design to support multirail networks is shown in Figure 8.1. We focus on the architecture of the sender side. In the figure, we can see that besides MPI Protocol Layer and InfiniBand Layer, our design consists of three important components: *Communication Scheduler*, *Scheduling Policies*, and *Completion Filter*.

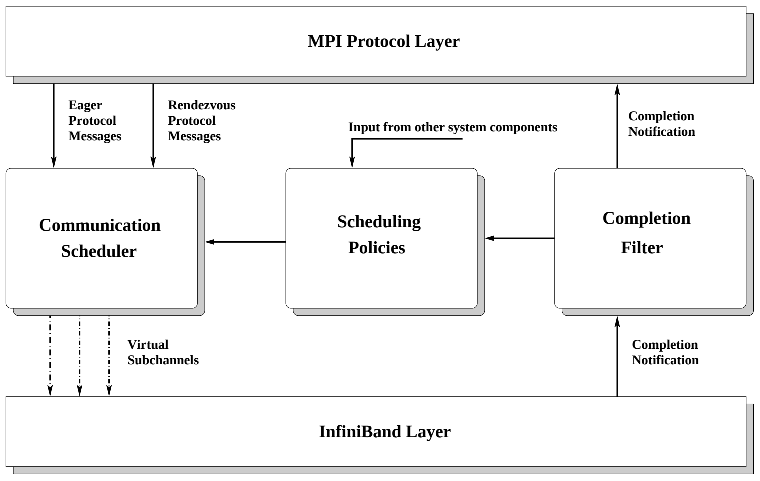

Figure 8.1: Basic Architecture of Multirail MPI Design

The Communication Scheduler is the central part of our design. Basically, it accepts protocol messages from the MPI Protocol Layer, and stripes (or multiplexes) them across multiple *virtual subchannels*. (Details of virtual subchannels will be described later.) In order to decide how to do striping or multiplexing, the Communication Scheduler uses information provided by the Scheduling Policies component. Scheduling Policies can be static schemes that are determined at initialization time. They can also be dynamic schemes that adjust themselves based on input from other components of the system.

Since a single message may be striped and sent as multiple messages through the InfiniBand Layer, we use the Completion Filter to filter completion notifications and to inform the MPI Protocol Layer about completions only when necessary. The Completion Filter can also gather information based on the completion notifications and use it as input to adjust dynamic scheduling policies.

8.2.2 Virtual Subchannel Abstraction

As we have discussed, multirail networks can be built by using multiple HCAs in a single node, or by using multiple ports in a single HCA. We have also seen that even with a single port, it is possible to achieve performance benefits by allowing multiple paths to be set up between two end-points. Therefore, it is desirable to have a single implementation to handle all these cases instead of dealing with them separately.

In MPI applications, every two processes can communicate with each other. This is implemented in many MPI designs by a data structure called *virtual channel* (or *virtual connection*). A virtual channel can be regarded as an abstract communication channel between two processes. It does not have to correspond to a physical connection of the underlying communication layer.

In this section, we use an enhanced virtual channel abstraction to provide a unified solution to support multiple HCAs, multiple ports, and multiple paths for a single port. In our design, a virtual channel can consist of multiple *virtual subchannels* (called subchannels later). Since our MPI implementation mainly takes advantage of the InfiniBand Reliable Connection (RC) service, each subchannel corresponds to a reliable connection at the InfiniBand Layer. At the virtual channel level, we maintain various data structures to coordinate all the subchannels.

It is easy to see how this enhanced abstraction can deal with all the multirail configurations we have discussed. In the case of each node having multiple HCAs, subchannels for a virtual channel correspond to connections that go through different HCAs. If we would like to use multiple ports of the HCAs, we can set up subchannels so that there is one connection for each port. Similarly, different subchannels/connections can be set up in a single port that follow different paths. Once all the connections are initialized, the same subchannel abstraction is used for communication in all cases. Therefore, there is essentially no difference for all the configurations except for the initialization phase. The subchannel abstraction can also easily deal with cases in which we have a combination of multiple HCAs, multiple ports, and multiple paths for a single port. This idea is further illustrated in Figure 8.2.

8.2.3 Integration with MPI protocols

In some MPI implementations, functionalities such as striping messages across multiple network interfaces are part of a messaging layer. This messaging layer provides an interface to upper layer software such as MPI. One advantage of this approach is high portability, as other upper layer software can also be benefited from multirail

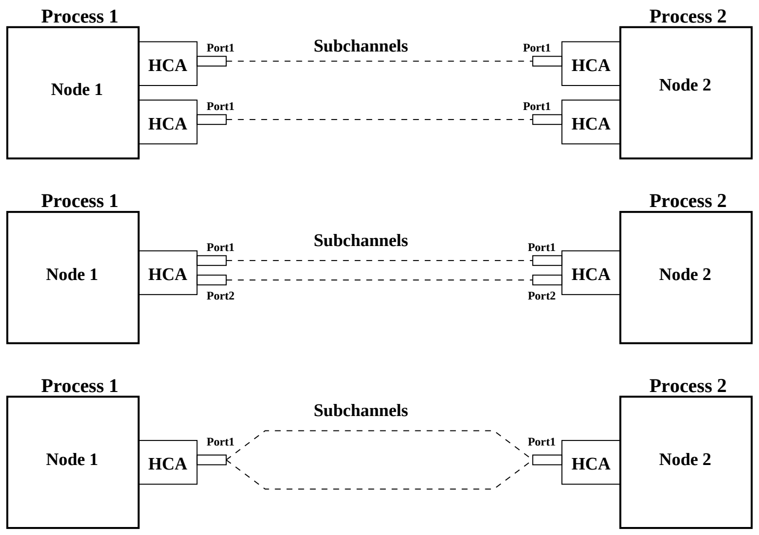

Figure 8.2: Virtual Subchannel Abstraction

networks. Our design is different because we have chosen to integrate these functionalities more tightly with the MPI communication protocols. Instead of focusing on portability, we aim to achieve high efficiency and flexibility in our implementation. Since multirail support is integrated with MPI protocols, we can specifically tailor its design to MPI to reduce overhead. This tightly coupled structure also gives us more flexibility in controlling how messages are striped or multiplexed in different MPI protocols.

One key design decision we have made is to allow message striping only for RDMA messages, although all messages, including RDMA and send/receive, can use multiplexing. This is not a serious restriction for MPI because MPI implementations over InfiniBand usually only use RDMA operations to transfer large messages. Send/receive operations are often used only for transferring small messages. By using striping with RDMA, there is almost no overhead to reassemble messages because data is directly put into the destination buffer. Zero-copy protocols in MPI, which usually take advantage of RDMA, can be supported in a straightforward manner.

As an example, let's take a look at the Eager and the Rendezvous protocols. In the Eager protocol, the data message can be sent using either RDMA or send/receive operations. However, since this message is small, striping is not necessary and only multiplexing is used. In the Rendezvous protocol, control messages are not striped. However, data messages can be striped since they can be very large.

8.2.4 Scheduling Policies

Different scheduling policies can be used by the Communication Scheduler to decide which subchannels to use for transferring messages. We categorize different policies into two classes: static schemes and dynamic schemes. In a static scheme, the policy and its parameters are determined at initialization time and stay unchanged during the execution of MPI applications. On the other hand, a dynamic scheme can switch between different policies or change its parameters.

In our design, scheduling policies can also be classified into multiplexing schemes and striping schemes. Multiplexing schemes are used for send/receive operations and RDMA operations with small data, in which messages are not striped. Striping schemes are used for large RDMA messages.

For multiplexing schemes, a simple solution is *binding*, in which only one subchannel is used for all messages. This scheme has the least overhead. It can take advantage of multiple subchannels if there are multiple processes in a single node. For utilizing multiple subchannels with a single process per node, schemes similar to Weighted Fair Queuing (WFQ) and Generalized Processor Scheduling (GPS) have been proposed in the networking area [2]. These schemes take into consideration the length of a message. In InfiniBand, the per operation cost usually dominates for small messages. Therefore, we choose to ignore the message size for small messages. As a result, simple *round robin* or *weighted round robin* schemes can be used for multiplexing. In some cases, different subchannels may have different latencies. This will result in many out-of-order messages for round robin schemes. A variation of round robin called *window based round robin* can be used to address this issue. In this scheme, a window size W is given and a subchannel is used to sent W messages before the Communication Scheduler switches to another subchannel. Since W consecutive messages travels the same subchannel, the number of out-of-order messages can be greatly reduced for subchannels with different latencies.

For striping schemes, the most important factor we need to consider is the bandwidth of each subchannel. It should be noted that we should consider *path bandwidth* instead of *link bandwidth*, although they can sometimes be the same depending on the switch configuration and the communication pattern. *Even striping* can be used for subchannels with equal bandwidth, while *weighted striping* can be used for subchannels with different bandwidths. Similar to multiplexing, *binding* can be used when there are multiple processes in a single node.

8.2.5 Adaptive Striping

As we have discussed in the previous subsection, it is important to take into consideration path bandwidth for striping schemes. A simple solution is to use *weighted* *striping* and set the weights of different subchannels to their respective link bandwidths. However, this method fails to address the following problems: First, sometimes information such as link bandwidth is not available directly to MPI implementations. Second, in some cases, bottlenecks in the network or switches may make the path bandwidth smaller than the link bandwidth. Finally, path bandwidth can also be affected by other ongoing communication. Therefore, it may change over time. A partial solution to these problems is to carry out small tests during the initialization phase of MPI applications to determine the path bandwidth. However, in addition to its high overhead (tests need to be done for every subchannel between every pair of nodes), it still fails to solve the last problem.

In this subsection, we propose a dynamic scheme for striping large messages. Our scheme, called *adaptive striping scheme*, is based on the weighted striping scheme. However, instead of using a set of fixed weights that are set at initialization time, we constantly monitor the progress of different stripes in each subchannel and exploit feedback information from the InfiniBand Layer to adjust the weights to their optimal values.

In designing the adaptive striping scheme, we assume the latencies of all subchannels are about the same and focus on their bandwidth. In order to achieve optimal performance for striping, a key insight is that the message must be striped in such a way that transmission of each stripe will finish at about the same time. This results in perfect load balancing and minimum message delivering time. Therefore, our scheme constantly monitors the time each stripe spent in each subchannel and use this information to adjust the weight so that striping distribution becomes more and more balanced and eventually reaches optimum. This feedback based control mechanism is illustrated in Figure 8.3.

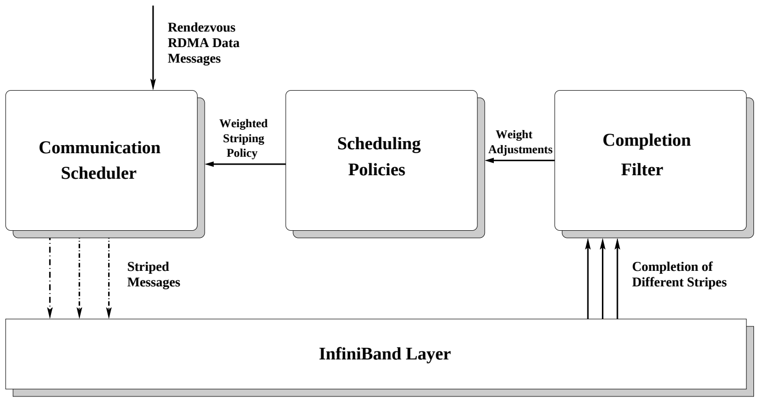

Figure 8.3: Feedback Loop in Adaptive Striping

In InfiniBand, a completion notification will be generated after each message is delivered to the destination and an acknowledgment is received. With the help of Completion Filter, the progress engine of our MPI implementation uses polling to check any new completion notification and take appropriate actions. In order to calculate the delivering time of each stripe, we first record the start time of each stripe when it is handed over to the InfiniBand Layer for transmission. When the delivery is finished, a completion notification will be generated by the InfiniBand Layer. The Completion Filter component will then record the finish time and derive the delivering time by subtracting the start time from it. After delivering times for all stripes of a message are collected, adjustment of weights is calculated and sent to the Scheduling Policies component to adjust the policy. Later, the Communication Scheduler will use the new policy for striping.

Next we will discuss the details of weight adjustment. Our basic idea is to have a fixed number of total weights and redistribute it based on feedback information obtained from different stripes of a single message. Suppose the total weight is W_{total}, the current weight of subchannel i is W_i, the path bandwidth of subchannel i is BW_i, the message size is S, and the stripe delivering time for subchannel i is t_i, we then have the following:

$$BW_i = \frac{S \cdot \frac{W_i}{W_{total}}}{t_i} = \frac{S \cdot W_i}{t_i \cdot W_{total}} \tag{8.1}$$

Since W_{total} and S are the same for all subchannels, we have the following:

$$BW_i \propto \frac{W_i}{t_i} \tag{8.2}$$

Therefore, new weight distributions can be done based on Equation 8.2. Suppose W_i' is the new weight for subchannel i, the following can be used to calculate W_i':

$$W_i' = W_{total} \cdot \frac{\frac{W_i}{t_i}}{\sum_{k \in subchannels} \frac{W_k}{t_k}} \tag{8.3}$$

In Equation 8.3, weights are completely redistributed based on the feedback information. To make our scheme more robust to fluctuations in the system, we can preserve part of the historical information. Suppose α is a constant between 0 and 1, we can have the following equation:

$$W_i' = (1 - \alpha) \cdot W_i + \alpha \cdot W_{total} \cdot \frac{\frac{W_i}{t_i}}{\sum_{k \in subchannels} \frac{W_k}{t_k}} \tag{8.4}$$

In our implementation, the start times of all stripes are almost the same and can be accurately measured. However, completion notification are generated by the

InfiniBand Layer asynchronously and we only record the finish time of a stripe as we have found its completion notification. Since MPI progress engine processing can be delayed due to application computation, we can only obtain an upper bound of the actual finish time and the resulting delivering t_i is also an upper bound. Therefore, one question is how accurately we can estimate the delivering time t_i for each subchannel. To address this question, we consider three cases:

1. Progress engine is not delayed. In this case, accurate delivering time can be obtained.

2. Progress engine is delayed and some of the delivering times are overestimated. Based on Equation 8.4, in this case, weight redistribution will not be optimal, but it will still improve performance compared with the original weight distribution.

3. Progress engine is delayed for a long time and we find all completion notifications at about the same time. Based on Equation 8.4, this will essentially result in no change in the weight distribution.

We can see that in no case will the redistribution result in worse performance than the original distribution. In practice, case 1 is the most common and accurate estimation can be expected most of the time.

8.3 Performance Evaluation

In this section, we evaluate the performance of our multirail MPI design over InfiniBand. Our evaluation consists of two parts. In the first part, we show the performance benefit we can achieve compared with the original MPI implementation.

In the second part, we provide an evaluation of our adaptive striping scheme. Due to the limitation of our testbed, we focus on multirail networks with multiple HCAs in the section.

8.3.1 Experimental Testbed

Our testbed consists of a cluster of 8 SuperMicro SUPER X5DL8-GG nodes with ServerWorks GC LE chipsets. Each node has dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, and PCI-X 64-bit 133 MHz bus. We have used InfiniHost MT23108 DualPort 4x HCAs from Mellanox. If both ports of an HCA are used, we can potentially achieve one way peak bandwidth of 2 GB/s. However, the PCI-X bus can only support around 1 GB/s maximum bandwidth. Therefore, for each node we have used two HCAs and only one port of each HCA is connected to the switch. The ServerWorks GC LE chipsets have two separate I/O bridges. To reduce the impact of I/O bus, the two HCAs are connected to PCI-X buses connected to different I/O bridges. All nodes are connected to a single Mellanox InfiniScale 24 port switch (MTS 2400), which supports all 24 ports running at full 4x speed. Therefore, our configuration is equivalent to a two-rail InfiniBand network built from multiple HCAs. The kernel version we used is Linux 2.4.22smp. The InfiniHost SDK version is 3.0.1 and HCA firmware version is 3.0.1. The Front Side Bus (FSB) of each node runs at 533MHz. The physical memory is 1 GB of PC2100 DDR-SDRAM.

8.3.2 Performance Benefits of Multirail Design

To evaluate the performance benefit of using multirail networks, we compare our new multirail MPI with our original MPI implementation. In the multirail MPI design, unless otherwise stated, even striping is used for large messages and round robin

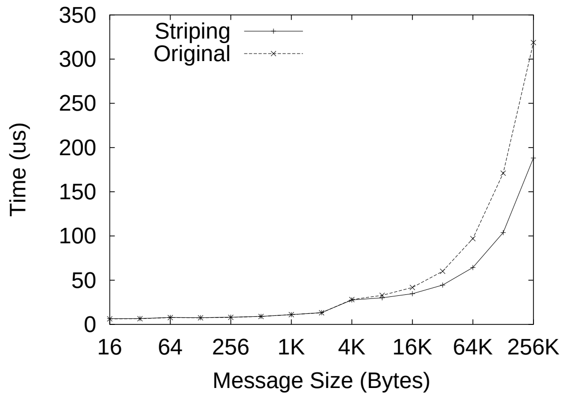

Figure 8.4: MPI Latency (UP mode)

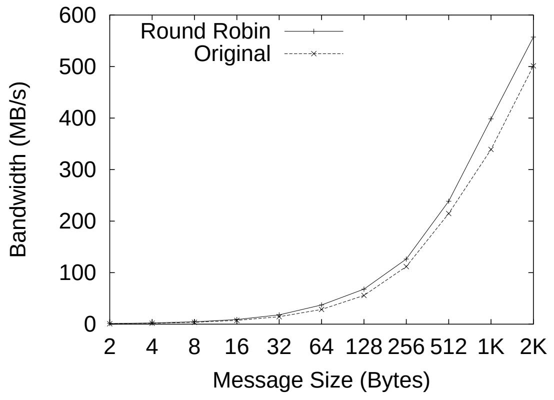

Figure 8.5: MPI Bandwidth (Small Messages, UP mode)

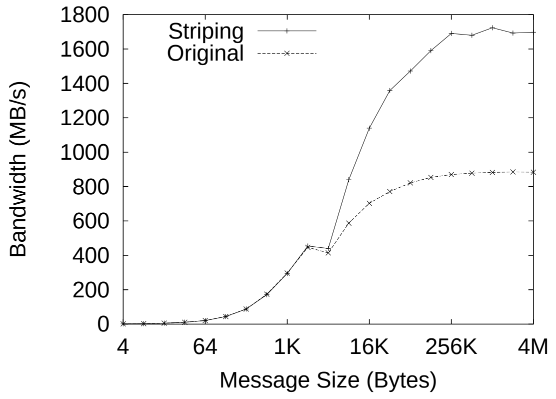

Figure 8.6: MPI Bandwidth (UP mode)

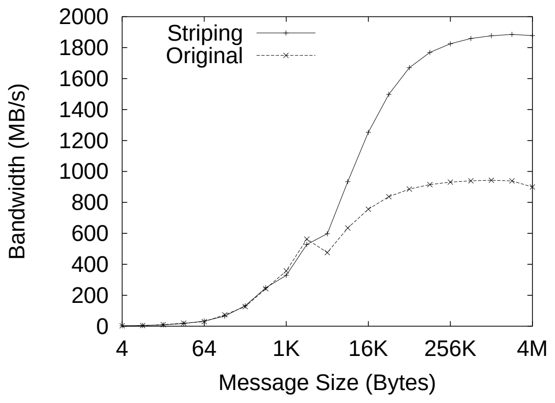

Figure 8.7: MPI Bidirectional Bandwidth (UP mode)

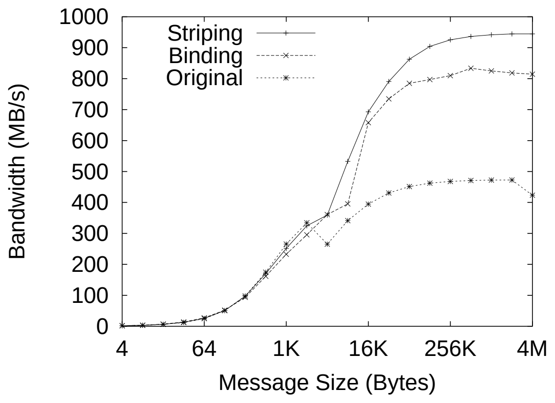

Figure 8.8: MPI Bandwidth (SMP mode)

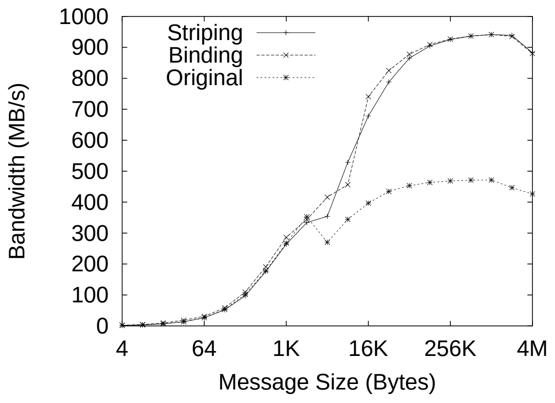

Figure 8.9: MPI Bidirectional Bandwidth (SMP mode)

scheme is used for small messages. We first present performance comparisons using micro-benchmarks, including latency, bandwidth and bi-directional bandwidth. We then present results for collective communication by using Pallas MPI benchmarks [75]. Finally, we carry out application level evaluation by using some of the NAS Parallel Benchmarks [68] and a visualization application. In many of the experiments, we have considered two cases: UP mode (each node running one process) and SMP mode (each node running two processes).

In Figures 8.4, 8.6 and 8.7, we show the latency, bandwidth and bidirectional bandwidth results in UP mode. We also show bandwidth results for small messages in Figure 8.5. (Note that in the x axis of the figures, unit K is an abbreviation for 2^{10} and M is an abbreviation for 2^{20}.) From Figure 8.4 we can see that for small messages, the original design and the multirail design perform comparably. The smallest latency is around 6 μs for both. However, as message size increases, the multirail design outperforms the original design. For large messages, it achieves about half the latency of the original design. In Figure 8.6, we can observe that multirail design can achieve significantly higher bandwidth. The peak bandwidth for the original design is around 884 MB/s. With the multirail design, we can achieve around 1723 MB/s bandwidth, which is almost twice the bandwidth obtained with the original design. Bidirectional bandwidth results in Figure 8.7 show a similar trend. The peak bidirectional bandwidth is around 943 MB/s for the original design and 1877 MB/s for the multirail design. In Figure 8.5 we can see that the round robin scheme can slightly improve bandwidth for small messages compared with the original scheme.

For Figures 8.8 and 8.9, we have used two processes on each node, each of them sending or receiving data from a process on the other node. It should be noted that in the bandwidth test, the two senders are on separate nodes. For the multirail design, we have shown results using both even striping policy and binding policy for large messages. Figure 8.8 shows that both striping and binding performs significantly better than the original design. We can also see that striping does better than binding. The reason is that striping can utilize both HCAs in both directions while binding only uses one direction in each HCA. Since in the bidirectional bandwidth test in SMP mode, both HCAs are utilized for both directions, striping and binding perform comparably, as can be seen from Figure 8.9.

In Figures 8.10, 8.11, 8.12 and 8.13 we show results for MPI_Bcast and MPI_Alltoall for 8 processes (UP mode) and 16 processes (SMP mode) using Pallas Benchmarks. The trend is very similar to what we have observed in previous tests. With multirail design, we can achieve significant performance improvement for large messages compared with the original design.

In Figures 8.14 and 8.15 we show application results. We have chosen the IS and FT applications (Class A and Class B) in the NAS Parallel Benchmarks because compared with other applications, they are more bandwidth-bound. We have also used a visualization application. This application is a modified version of the program described in [24]. We show performance numbers for both UP and SMP modes. However, due to the large data set size in the visualization application, we can only run it in UP mode.

From the figures we can see that multirail design results in significant reduction in communication time for all applications in both UP and SMP modes. For FT, the

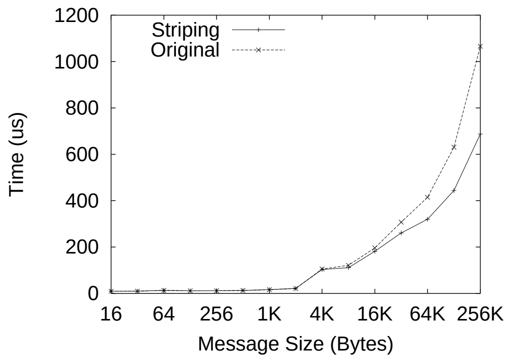

Figure 8.10: MPI_Bcast Latency (UP mode)

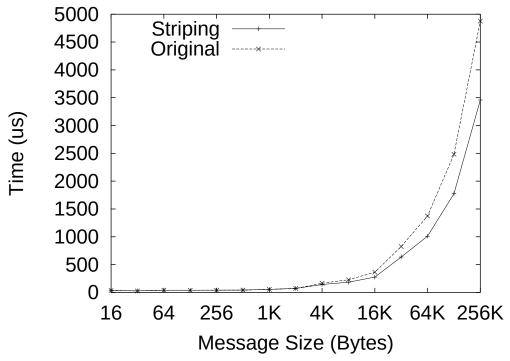

Figure 8.11: MPI_Alltoall Latency (UP mode)

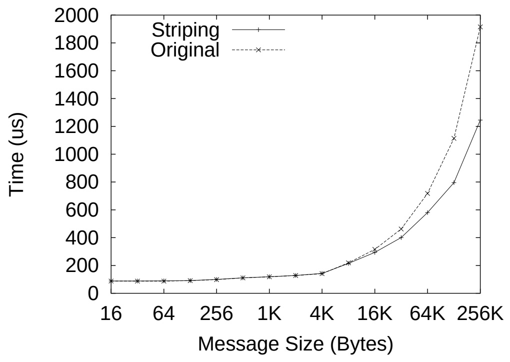

Figure 8.12: MPI_Bcast Latency (SMP mode)

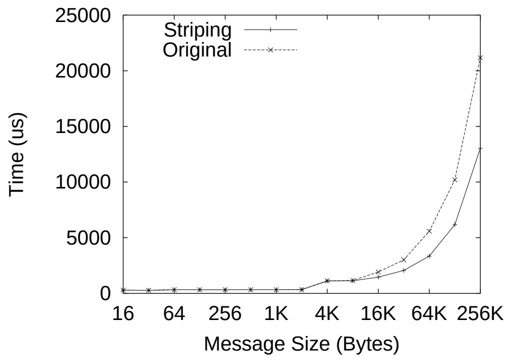

Figure 8.13: MPI_Alltoall Latency (SMP mode)

communication time is reduced almost by half. For IS, the communication time is reduce by up to 38%, which results in up to 22% reduction in application running time. For the visualization application, the communication time is reduced by 43% and the application running time is reduced by 16%. Overall, we can see that multirail design can bring significant performance improvement to bandwidth-bound applications.

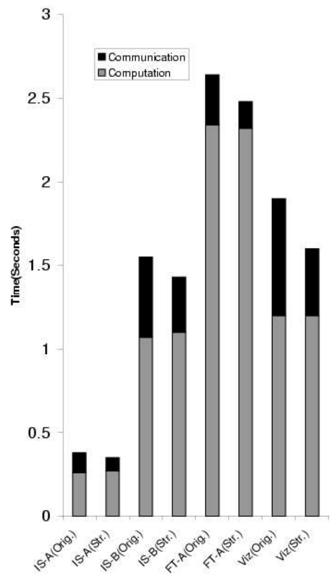

Figure 8.14: Application Results (8 processes, UP mode)

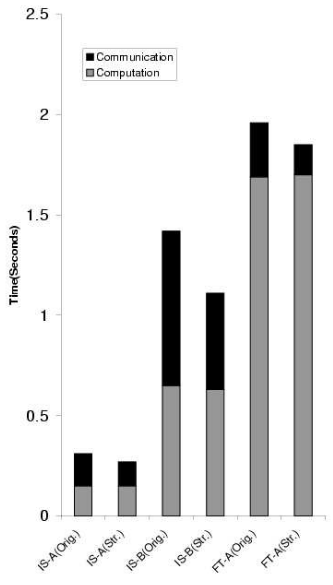

Figure 8.15: Application Results (16 processes, SMP mode)

8.3.3 Evaluating the Adaptive Striping Scheme

In this subsection, we show how our proposed adaptive striping scheme can provide good performance in cases each rail has different bandwidth. To simulate this environment, for most of our experiments, we have forced the second HCA on each node to run at 1x speed with a peak bandwidth of 250 MB/s. The first HCA on each node still operates at the normal 4x speed (1 GB/s peak bandwidth). Without *a priori* knowledge of this environment, our multirail MPI implementation will use even striping. With this knowledge, it will use weighted striping and set the weights to 4 and 1 respectively for each subchannel. We compare both of them with the adaptive striping scheme, which assigns equal weights to both subchannels initially. We focus on microbenchmarks and UP mode in this subsection.

Figures 8.16 and 8.17 show the latency and bandwidth results. We can see that the adaptive striping scheme significantly outperforms even striping and achieves comparable performance with weighted striping. In Figure 8.18, we show bidirectional bandwidth results for the three schemes. An important finding is that our adaptive scheme can significantly outperform weighted striping in this case. This is because in the test, the communication traffic is assigned to the two subchannels as 4:1 based on the link speed (4x vs. 1x). With bidirectional traffic, the aggregate link speeds would be 8x and 2x respectively for each subchannel. However, the PCI-X bus can only sustain a peak bandwidth of 1 GB/s, which is equivalent to 4x speed. Therefore, if we take into account the I/O bus bottleneck, the speed should be 4x and 2x for the two subchannels, respectively. Hence, the optimal weighted scheme should use 2:1 instead of 4:1. This also shows that even with *a priori* knowledge of link speed, static schemes may fail to achieve optimal performance because of impact from other system components and pattern of communication traffic. In contrast, the adaptive striping scheme can easily adjust the policy to achieve optimal striping.

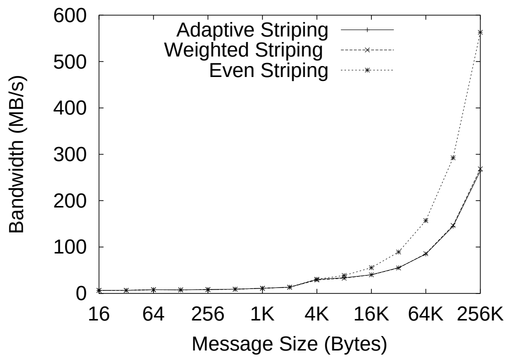

Figure 8.16: MPI Latency with Adaptive Striping (UP mode)

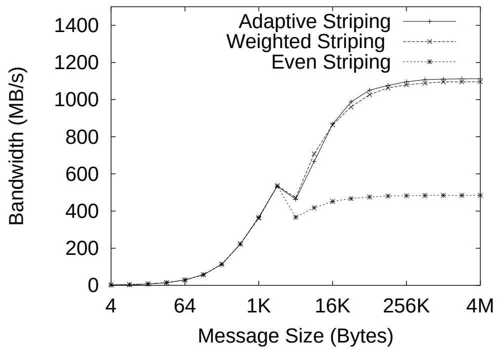

Figure 8.17: MPI Bandwidth with Adaptive Striping (UP mode)

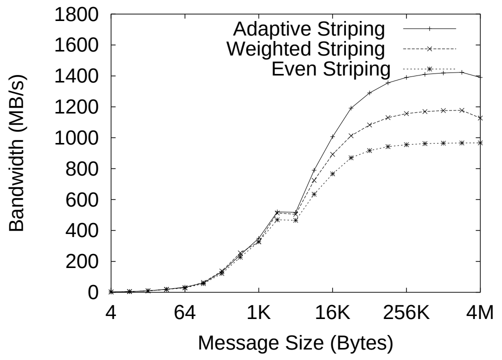

Figure 8.18: MPI Bidirectional Bandwidth with Adaptive Striping (UP mode)

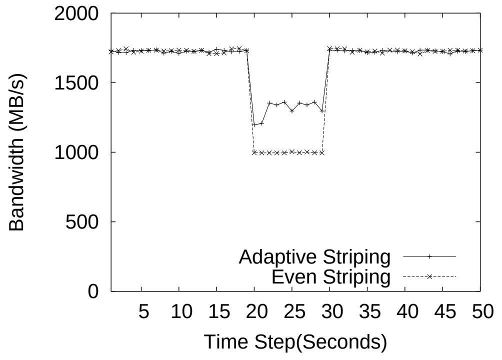

Figure 8.19: MPI Bandwidth with Adaptive Striping

In the following bandwidth tests, we let both HCAs operate at 4x speed. A bandwidth program runs on the two nodes and prints out the peak bandwidth results every one second. During the execution of the bandwidth program, we start another program on the two nodes which use the second HCA to transfer large messages. This program runs for around 10 seconds. We compare the adaptive striping scheme and even striping in Figure 8.19. We can see that at the beginning both schemes perform comparably. However, when the second program starts, one of the HCAs has to be shared by both programs. Hence, even striping is no longer optimal. As we can see, the adaptive scheme can achieve better performance by adjusting the weight of each subchannel accordingly. After the second program finishes, the adaptive striping scheme can again adjust the weights to achieve peak performance.

8.4 Related Work

Using multirail networks to build high performance clusters is discussed in [16]. The paper proposed different allocation schemes in order to eliminate conflicts at end points or I/O buses. However, the main interconnect focused in the paper was Quadrics [77] and the performance evaluation was done using simulation. In our work, we focus on software support at the end points to build InfiniBand multirail networks and present experimental performance data.

VMI2 [85] is a messaging layer developed by researchers at NCSA. An MPI implementation over VMI2 is also available [69]. VMI2 runs over multiple interconnects. [85] briefly mentions VMI2's ability to striping large messages across different network interconnects. Instead of using a separate messaging layer, our design has integrated the multirail support with MPI protocols.

LA-MPI [25] is an MPI implementation developed at Los Alamos National Labs. LA-MPI was designed with the ability to stripe message across several network paths. LA-MPI design includes a *path scheduler*, which determines which path a message will travel. This design bears some similarity with our approach. However, in this work, we focus on InfiniBand architecture and discuss different design issues and policies. We have also proposed an adaptive striping scheme.

IBM has supported using multirail networks for its SP switches and adapters [21]. SGI's Message Passing Toolkit [86] and Sun's MPI implementation over its SUN Fire Link [87] also support the ability of striping message across multiple links. However, details about how striping is implemented are not available in the literature. Recently, Myricom announced its message passing layers called *Myrinet Express* and *GM2* [67]. Both can stripe messages across two different ports on a single Myrinet NIC and overcome the limitation of Myrinet link bandwidth. GM2 is now available. However, Myrinet Express has not been released and its internal design are not yet available.

Striping in the network systems have been used for many years. [9] provides a survey of how striping is used at different layers in the network subsystems. Work done in [2] proposes an architecture to stripe packets across multiple links in order to achieve fair load sharing. Striping across multiple TCP/IP connections has also been studied in the literature. One example is PSockets [89]. PSockets presents to the application layer the same socket interface as that used in TCP/IP. It transparently stripes a message across multiple TCP/IP connections.

8.5 Summary

In this work, we present an in-depth study of designing high performance multirail InfiniBand clusters. We discuss various ways of setting up multirail networks with InfiniBand and propose a unified MPI design that can support all these approaches. By taking advantage of RDMA operations in InfiniBand and integrating the multirail design with MPI communication protocols, our design supports multirail networks with very low overhead. Our design also supports different policies of using multiple rails. Another contribution of this work is an adaptive striping scheme that can dynamically change the striping parameters based on the current available bandwidths of different rails.

We have implemented our design and carried out detailed performance evaluation. Our performance results show that the multirail MPI can significant improve MPI communication performance. With a two rail InfiniBand network, we can achieve almost twice the bandwidth and half the latency for large messages compared with the original MPI. The multirail MPI design can also significantly reduce communication time as well as running time for bandwidth-bound applications. We have also shown that the adaptive striping scheme can achieve excellent performance without *a priori* knowledge of the bandwidth of each rail.

CHAPTER 9

MPI PERFORMANCE EVALUATION FRAMEWORK

In this work, we present a comprehensive performance evaluation framework and compare the performance of our MPI with MPI implementations over Myrinet and Quadrics. The MPI implementations we use for Myrinet and Quadrics are those included in their respective software packages. For InfiniBand, we have used our RDMA-based MVAPICH [56, 55] implementation.

Our performance evaluation consists of two major parts. The first part consists of a set of MPI level micro-benchmarks. These benchmarks include traditional measurements such as latency, bandwidth and host overhead. In addition to those, we have also included the following micro-benchmarks: communication/computation overlap, buffer reuse, memory usage, intra-node communication and collective communication. The objective behind this extended micro-benchmark suite is to characterize different aspects of the MPI implementations and get more insights into their communication behavior.

The second part of the performance evaluation consists of application level benchmarks. We have used the NAS Parallel Benchmarks [68] and the sweep3D benchmark [33]. We not only present the overall performance results, but also relate application communication characteristics to the information we got from the micro-benchmarks. We use in-depth profiling of these applications to measure their characteristics. Using these profiled data and the results obtained from the micro-benchmarks, we analyze the impact of the following factors: overlap of computation and communication, buffer reuse, collective communication, memory usage, SMP performance and scalability with system sizes, and PCI-X bus. All the experiments were done with A1 InfiniHost cards and Intel compilers.

An interesting evaluation of current high performance networks was carried out in [6]. The authors used LogP model to evaluate a wide variety of interconnects at both the MPI level and the low level messaging software level. However, they did not include InfiniBand and the tests were done only at micro-benchmark level. The networks they studied were in different systems. We have done a performance evaluation in a single cluster for different interconnects, which makes it possible to compare them with minimum impact from other parts of the system.

Work done in [34] used both micro-benchmarks and the NAS Parallel Benchmarks to study the performance of Giganet and Myrinet on clusters of SMP servers. Our work follows a similar approach. However, we have greatly expanded the set of micro-benchmarks and studied the relationship between application communication characteristics and different performance aspects of MPI.

The LogP model was proposed in [18], and a study of application performance sensitivity to LogP parameters were carried out in [58]. In our micro-benchmarks, we

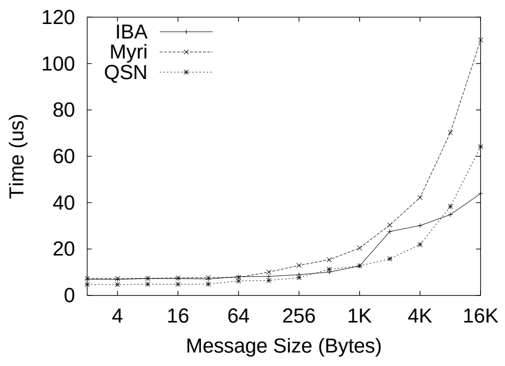

Figure 9.1: MPI Latency

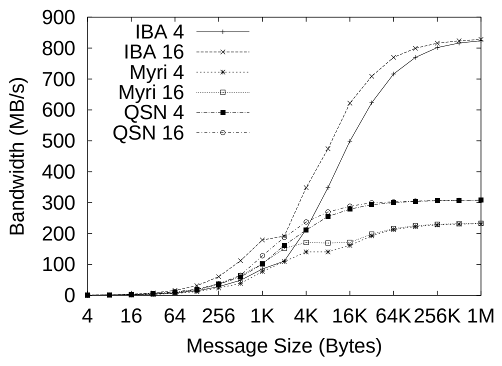

Figure 9.2: MPI Bandwidth

include not only measurements similar to those in the LogP model, but also include additional tests to characterize other performance aspects of MPI implementations.

There have also been many studies about communication patterns for parallel applications. Studies of the NAS Parallel Benchmarks have done in [99, 91]. Another excellent study on communication characteristics of large scale scientific applications was conducted in [95]. The focus of our work is to compare the three different MPI implementations. Therefore, we have used the communication pattern information to study the impact of different MPI implementations on application performance.

9.1 Micro-Benchmarks

To provide more insights into communication behavior of the three MPI implementations, we have designed a set of micro-benchmarks. They include basic measurements such as latency, bandwidth and host overhead. In addition, we use several micro-benchmarks to characterize the other aspects of an MPI implementation. The details for each micro-benchmark are presented below:

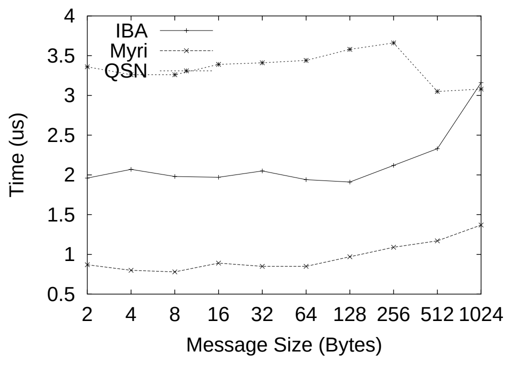

Figure 9.3: MPI Host Overhead

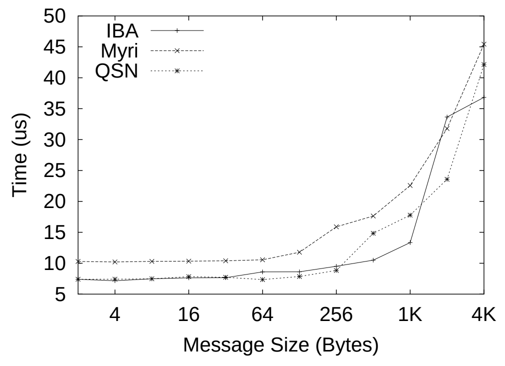

Figure 9.4: MPI Bi-Directional Latency

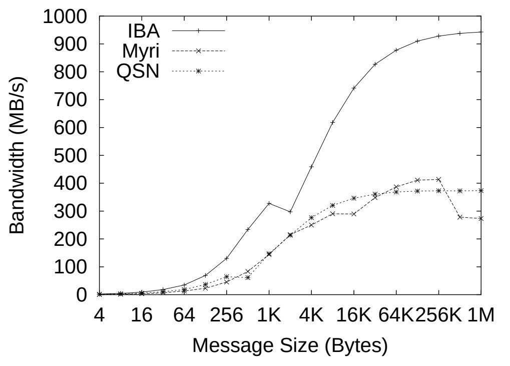

Figure 9.5: MPI Bi-Directional Bandwidth

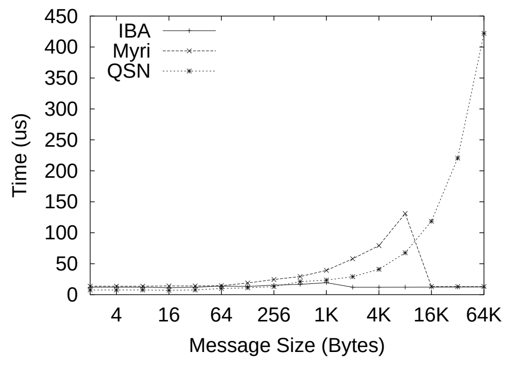

Figure 9.6: Overlap Potential

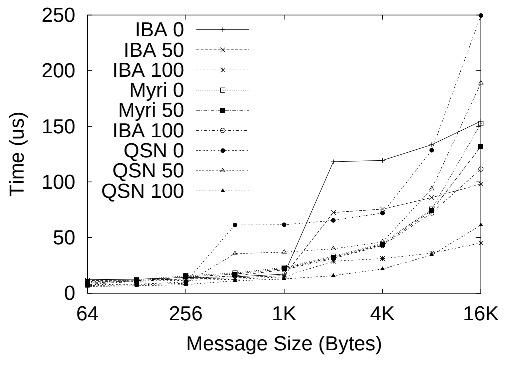

Figure 9.7: MPI Latency with Buffer Reuse

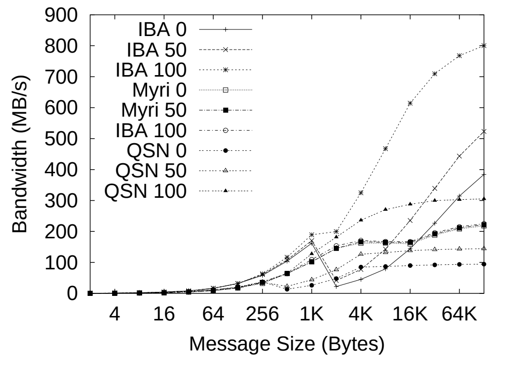

Figure 9.8: MPI Bandwidth with Buffer Reuse

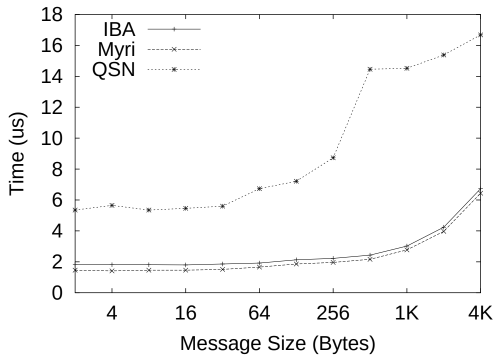

Figure 9.9: MPI Intra-Node Latency

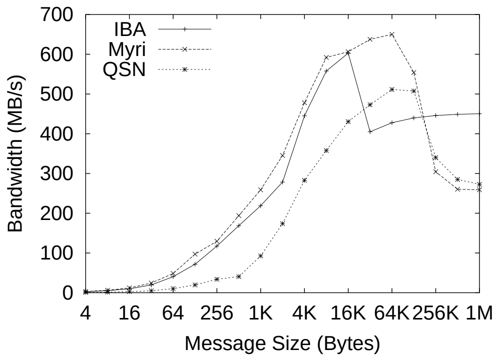

Figure 9.10: MPI Intra-Node Bandwidth

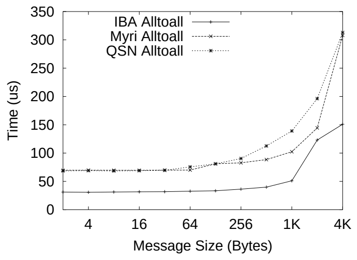

Figure 9.11: MPI Alltoall

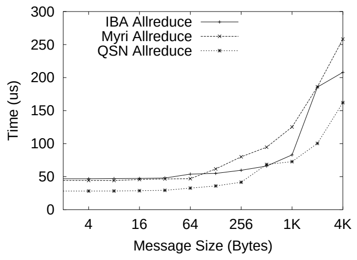

Figure 9.12: MPI Allreduce

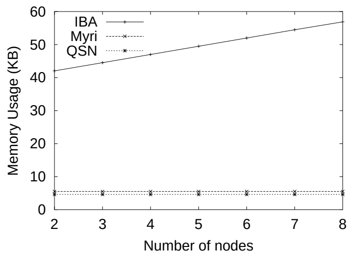

Figure 9.13: MPI Memory Consumption

9.1.1 Latency and Bandwidth

Figure 9.1 shows the MPI-level latency results. The test is conducted in a ping-pong fashion and the latency is derived from round-trip time. For small messages, Quadrics shows excellent latencies, which are under 5μs. The smallest latencies for InfiniBand and Myrinet are 6.8μs and 7.3μs, respectively. For large messages, InfiniBand has a clear advantage because of its higher bandwidth and PCI-X interface.

The bandwidth test is used to determine the maximum sustained data rate that can be achieved at the network level. Therefore, non-blocking MPI functions are used. In this test, a sender keeps sending back-to-back messages to the receiver until it has reached a pre-defined window size W. Then it waits for these messages to finish and sends out another W messages. Figure 9.2 shows the uni-directional bandwidth results for different W. For large messages and window size 16, InfiniBand can achieve bandwidth of over 830MB/s. The peak bandwidths for Quadrics and Myrinet are around 308MB/s and 233MB/s, respectively. The window size W also affects the bandwidth achieved, especially for small messages. For InfiniBand and Myrinet, their performance increases with the window size. Quadrics shows the same behavior for window size less than 16. However, its performance drops when the window size exceeds 16.

9.1.2 Host Overhead

Host overhead has been shown to have a significant impact on application performance [58]. Figure 9.3 presents the host overhead results for small messages. The overhead is obtained by measuring the time spent in communication. For Myrinet and InfiniBand, the overheads are around 1μs and 2μs, respectively. And their overheads increase slightly with the message size. Although Quadrics has better latency, it has higher overhead, which is over 3μs. Its overhead drops slightly after 256 bytes.

9.1.3 Bi-Directional Performance

Compared with uni-directional tests, bi-directional latency and bandwidth tests put more stress on the communication layer. Therefore they may be more helpful to us to understand the bottleneck in communication. The tests are carried out in a way similar to the uni-directional ones. The difference is that both sides send data simultaneously.

Figure 9.4 shows the bi-directional latency results. We can see that both Quadrics and Myrinet show worse performance compared with their uni-directional latencies, which are 4.8μs and 7.3μs, respectively. For small messages, their respective bi-directional latencies are 7.4μs and 10.3μs. The latency for MPI over InfiniBand is also 7.4μs. However, this number is only slightly worse than its uni-directional latency (6.8μs).

Figure 9.5 shows the bi-directional bandwidth results. The window size of the bandwidth tests is 16. From the figure we notice that InfiniBand bandwidth increases from 830MB/s uni-directional to 940MB/s. Then it is limited by the bandwidth of the PCI-X bus. Quadrics bandwidth improves from 308MB/s to 375MB/s. Myrinet shows even more improvement. Its peak bandwidth increases from 233MB/s to 413MB/s. However, Myrinet bandwidth drops to less than 300MB/s when the message size is larger than 256KB.

9.1.4 Communication/Computation Overlap

To achieve better performance at the application level, one of the techniques MPI programmers use is to overlap communication with computation. To measure the ability to overlap computation with communication of each MPI implementation, we have designed an overlapping test using MPI non-blocking functions. The test is based on the latency test. At the sender, we use non-blocking MPI functions to start receive and send operations. Then the program enters a computation loop. After that it waits for the send and receive operations to finish. We define the potential of overlapping to be the maximum time of the computation loop that does not increase the latency.

We present the overlapping potential results in Figure 9.6. We can see that for small messages, InfiniBand and Myrinet have better overlapping potential compared with Quadrics because of their higher latencies and lower host overheads. However, the amount of overlapping drops at a certain point and stays as a constant. For Quadrics, the overlapping potential increases steadily with the message size.

MPI implementations usually use eager protocol for small messages and rendezvous protocol for large messages. The rendezvous protocol needs a handshake between the sender and the receiver. For MPI over InfiniBand and Myrinet, this handshake needs host intervention. Therefore, their abilities for overlapping computation and communication are limited by the rendezvous protocol. MPI over Quadrics is able to make communication progress asynchronously by taking advantages of the programmable network interface card. Thus it shows much better overlapping potential for large messages.

9.1.5 Impact of Buffer Reuse

For interconnects using user-level mode communication, application buffer reuse patterns can have significant impact on performance. This is due to the following reasons:

- Interconnects such as InfiniBand and Myrinet require that communication buffers be registered before communication. Therefore user buffers need to be registered in order to achieve zero-copy communication. To reduce the number of registration and de-registration (events), MPI implementations usually use techniques similar to pin-down cache [32] to de-register buffers in a lazy fashion. Thus, application buffer reuse patterns directly affect the hit rate of pin-down cache.

- Modern high speed network interfaces such as those studied in this work usually have DMA engines to access host memory. An address translation mechanism is needed to translate user buffer virtual addresses to DMA addresses. Application buffer reuse patterns also affect the performance of this address translation process.

Our buffer reuse benchmark consists of N iterations of communication. We define a buffer reuse percentage R. For the N iterations of the test, N*R iterations will use the same buffer, while all other iterations will use completely different buffers. By changing buffer reuse percentage R, we can see how communication performance is affected by buffer reuse patterns. Figures 9.7 and 9.8 show the latency and bandwidth results for different buffer reuse percentage, respectively. We can see that all three MPI implementation are sensitive to the buffer reuse pattern. When the reuse percentage decreases, their performance drops significantly. For message sizes greater than 1KB, the InfiniBand latency suffers greatly when buffers are not reused. Quadrics also sees a steep rise in latency with lack of buffer reuse starting for all messages. Myrinet latency is not significantly affected until the message size reaches 16KB.

9.1.6 Intra-Node Communication

For SMP machines, it is possible to improve intra-node communication performance by taking advantage of shared memory mechanism. In this section, we present intra-node MPI performance for the three implementations. Figures 9.9 and 9.10 show the latency and bandwidth performance results. From the figures we can see that Quadrics does not perform well in SMP mode. Its intra-node latency is even higher than inter-node latency. The small message latencies for Myrinet and InfiniBand are about 1μs and 2μs, respectively. Bandwidth for both Myrinet and Quadrics drops for large messages because of cache thrashing. MPI over InfiniBand only uses shared memory for small messages (less than 16KB). Its peak bandwidth is around 600MB/s.

9.1.7 Collective Communication

MPI collective communications can be implemented by using point-to-point MPI functions. However, to achieve optimal performance, we can also implement them directly over the message passing layer. This is even more desirable when an interconnect has specially support for collective communications.

Two of the most frequently used MPI collective operations are MPI_Alltoall and MPI_Allreduce [95]. Figures 9.11 and 9.12 shows the performance of MPI_Alltoall and MPI_Allreduce for all three MPI implementations on 8 nodes. The Pallas MPI

Benchmarks [75] have been used for these tests. For MPI_Alltoall operations, InfiniBand performs much better than Quadrics and Myrinet, with a latency of 31μs for small messages compared with 67μs and 69μs for Quadrics and Myrinet, respectively. Quadrics achieves a latency of 28μs for small message MPI_Allreduce operations, which is better than Myrinet (44μs) and InfiniBand (46μs).

9.1.8 Memory Usage

One aspect of an MPI implementation often ignored by many micro-benchmarks is memory usage. The more memory allocated by the MPI implementation, the more likely it will adversely affect application performance. We run a simple MPI barrier program and measure the amount of memory it consumes. The memory data is obtained through the proc file systems in Linux.

The results are presented in Figure 9.13. We can see that MPI over Quadrics and Myrinet consume relatively small amount of memory, which does not increase with the number of nodes. Memory consumption for MPI over InfiniBand increases with the number of nodes. The reason for this increase is that the current implementation is built on top of InfiniBand Reliable Connection service. During initialization, a connection is set up between every two nodes and a certain amount of memory is reserved for each connection. Therefore, total memory consumption increases with the number of connections. This problem can be alleviated by using InfiniBand Reliable Datagram service or techniques like on-demand connection [100].

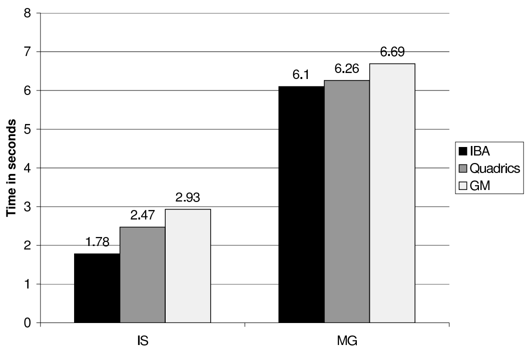

Figure 9.14: IS and MG on 8 Nodes

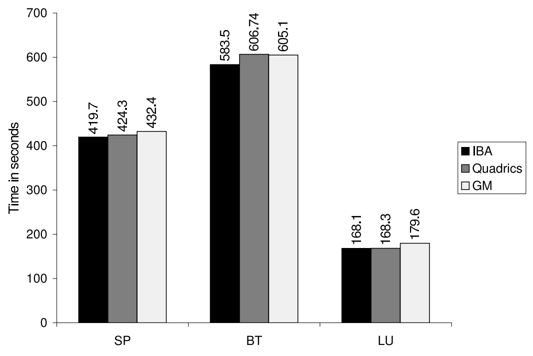

Figure 9.15: SP and BT on 4 Nodes and LU on 8 Nodes

9.2 Applications

In this section, we compare the three MPI implementations using the NAS Parallel Benchmarks [68] and the sweep3D [33] benchmark. Basic MPI performance parameters such as latency, bandwidth and overhead play an important role in determining application performance. However, depending on the application, other factors in MPI implementation such as computation/communication overlapping, collective communication, memory usage and buffer reuse can have great impact as well. To better understand the relationship between application performance and MPI implementations, we have done profiling for the applications under study. By relating application communication characteristics and different aspects of MPI implementations, we can get much more insights into the communication behavior of these applications. The profiling data is obtained through the MPICH logging interface [27]. We modified its source code to log more information such as buffer reuse patterns.

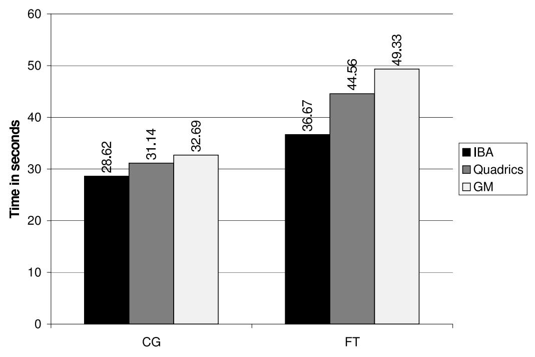

Figure 9.16: CG and FT on 8 Nodes

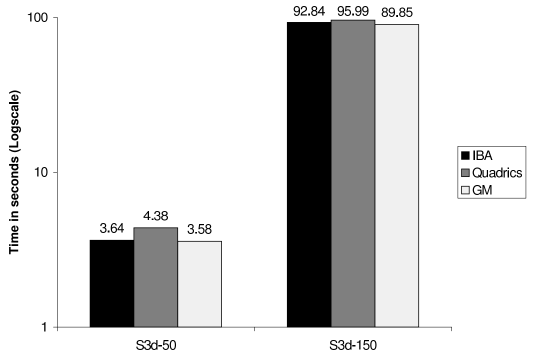

Figure 9.17: Sweep3D on 8 Nodes

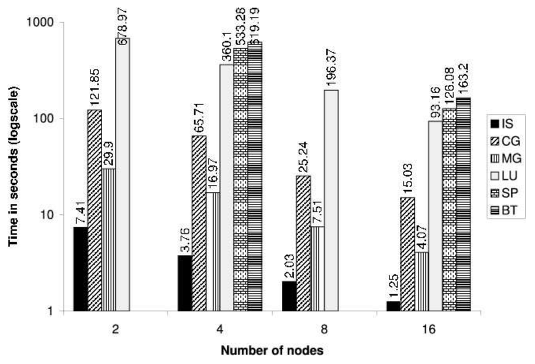

Figure 9.18: Scalability with System Sizes for a 16-Node System at Topspin

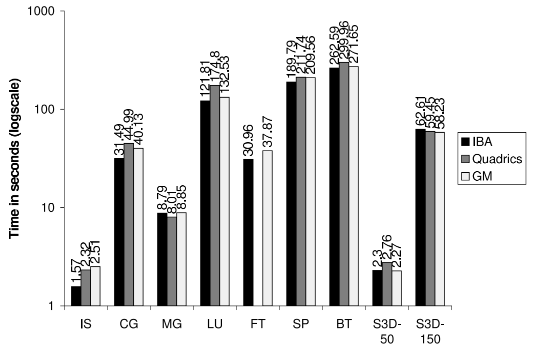

Figure 9.19: SMP Performance (16 Processes on 8 Nodes at OSU)

9.2.1 Application Performance Results

Figures 9.14, 9.15, 9.16 and 9.17 show the application running time for class B NAS parallel benchmarks and sweep3D. We use two input sizes for sweep3D: 50 and 150. We present 8 nodes results for IS, CG, MG, LU and FT. SP and BT require square number of nodes, therefore we only show results on 4 nodes for them. We can see that MPI over InfiniBand performs better than the other two implementations for all NAS benchmarks. The largest improvement comes from IS, which uses very large messages, as shown in Table 9.1. The much higher bandwidth of InfiniBand gives it a clear advantage. It performs 28% and 39% better than Quadrics and Myrinet, respectively. For other applications which use many large messages, such as FT and CG, InfiniBand also performs significantly better. For SP, BT and MG, MPI over InfiniBand also performs better than the other two. For applications that mostly use small messages, like LU, Quadrics and Myrinet performance is more comparable with InfiniBand.

Table 9.1: Message Size Distribution

Apps	<2K	2K-16K	16K-1M	>1M
IS	14	11	0	11
CG	16113	0	11856	0
MG	1607	630	3702	0
LU	100021	0	1008	0
FT	24	0	0	22
SP	9	0	9636	0
BT	9	0	4836	0
S3d-50	19236	0	0	0
S3d-150	28836	28800	0	0

For the sweep3D benchmarks, Quadrics performs worse than InfiniBand and Myrinet for input size 50. The three implementations perform comparably for input size 150.

9.2.2 Scalability with System Size

To study the scalability of the MPI implementations, we have measured application performance for 2, 4 and 8 processes in our 8 node cluster. (Due to the problem size, FT and sweep3D with input 150 do not run on 2 nodes.) We also measure performance of MPI over InfiniBand on a 16 node Topspin InfiniBand cluster [92], which is connected through a Topspin 360 24 port 4x InfiniBand switch. The HCAs are Topspin InfiniBand 4x HCAs. And the hosts are Microway dual 2.4GHz P4 Xeon systems with 2GB of memory based on a Tyan 2721-533 motherboard. The results are shown in Table 9.2 and Figure 9.18. We can observe that all three MPI implementations have good scalability, with some applications like MG and CG showing super-linear speedup. For IS, which uses very large messages, MPI over InfiniBand still shows almost linear speedup. However, Myrinet and Quadrics do not perform as well as InfiniBand for IS.

9.2.3 Impact of Computation/Communication Overlap

The effect of computation and communication overlap in real applications is difficult to characterize. As an approximation, we have collected information for non-blocking MPI calls in the applications. The results are shown in Table 9.3. (Average sizes are in bytes.) We can see that different applications use non-blocking MPI functions very differently. FT and sweep3D do not use them at all. MG, LU and CG only

Table 9.2: Scalability with System Sizes for Three Networks (Execution times are in seconds.)

Apps	**IBA**			**Myri**			**QSN**		
	2	4	8	2	4	8	2	4	8
IS	6.82	3.28	1.78	8.06	5.18	2.93	6.96	4.25	2.47
CG	132.195	73.91	28.62	136.34	75.33	32.69	133.59	76.21	31.14
MG	24.10	13.4	6.1	26.14	15.42	6.69	23.96	13.94	6.26
LU	667.76	320.93	168.13	718.06	338.8	179.67	658.45	323.74	168.36
FT	-	78.97	36.68	-	90.76	49.33	-	86.92	44.56
S3d-50	13.89	7.20	3.64	14.67	7.03	3.58	14.94	7.37	4.38
S3d-150	-	179.61	92.84	-	176.77	89.85	-	177.66	95.99

use non-blocking receive functions. SP and BT use both non-blocking send and non-blocking receive operations. We also noticed that the average sizes for non-blocking functions are very large. Therefore, it gives an advantage to MPI over Quadrics, which has better computation/communication overlap for large messages. As a result, for the applications with non-blocking operations, MPI over Quadrics performs more comparably with MPI over InfiniBand, as seen in the plots for SP in Figure 9.15.

Table 9.3: Non-Blocking MPI Calls

Apps	Isend		Irecv	
	# calls	Avg Size	# calls	Avg Size
IS	0	0	0	0
CG	0	0	13984	63591
MG	0	0	2922	270400
LU	0	0	508	311692
FT	0	0	0	0
SP	4818	263970	4818	263970
BT	2418	293108	2418	293108
S3d-50	0	0	0	0
S3d-150	0	0	0	0

9.2.4 Impact of Buffer Reuse

In Figures 9.7 and 9.8, we have shown that buffer reuse patterns have a significant impact on the performance of all the three MPI implementations. We define buffer reuse rate to be the percentage of accesses to previously used buffers. Table 9.4 shows buffer reuse rates and buffer reuse rates weighted by buffer sizes for all applications. One conclusion we can draw from the table is that in these applications, buffer reuse rates are very high. Therefore, although we have seen the MPI implementations can have different performance for different buffer reuse patterns, its impact to these applications is small. However, for other applications which have more dynamic memory usage patterns, this conclusion might not hold.

Table 9.4: Buffer Reuse Rate

Apps	**Buffer Reuse**	
	% Reuse	Wt % Reuse
IS	99.26	99.19
CG	99.99	99.99
MG	99.92	99.99
LU	99.59	99.99
FT	99.48	99.72
SP	99.78	99.99
BT	99.78	99.99
S3d-50	99.52	99.93
S3d-150	99.52	99.98

9.2.5 Impact of Other Factors

We have also studied the impact of other factors on application performance. These factors include: collective communication pattern, intra-node communication pattern and PCI (PCI-X) bus speed. Details can be found in [53].

9.3 Summary

In this work, we have presented a detailed performance study of MPI over InfiniBand, Myrinet and Quadrics, using both applications and micro-benchmarks. We have shown that MPI communication performance is affected by many factors. Therefore, to get more insights into different aspects of an MPI implementation, one has to go beyond simple micro-benchmarks such as latency and bandwidth. For example, we found that all the three MPI implementations are sensitive to buffer reuse patterns. We also found that MPI over Quadrics has better ability for overlapping computation and communication, and MPI over GM offers the best intra-node communication performance. None of these can be revealed by simple inter-node latency and bandwidth tests.

Our study also shows that although InfiniBand is relatively new in the HPC market, it is able to deliver very good performance at the MPI level. Our application results on the 8 node OSU cluster and 16 node Topspin cluster also show that InfiniBand has very good scalability.

CHAPTER 10

OPEN SOURCE SOFTWARE RELEASE AND ITS IMPACT

The work described in this dissertation has been integrated into our MVAPICH package [70] and released as open source software to the public. The current version of MVAPICH is 0.9.4. Our software supports different InfiniBand interfaces such as VAPI [61], IBAL [1] and OpenIB [73]. MVAPICH also supports different hardware architectures, including IA32, IA64, X86-64, EM64T and Apple G5.

Since its first release in November, 2002, our software has been adopted by more than 120 organizations (national laboratories, research centers, industry, and universities) world-wide to build high performance InfiniBand cluster systems. These clusters include both research testbeds and production systems. The software is also being distributed by almost every InfiniBand company with its software package. Our software has been used on some of the most powerful supercomputers in the world. For example, three of the top 500 supercomputers [88] (23rd edition) are powered by MVAPICH. They are 2200-processor Apple G5 cluster at Virginia Tech (ranked 3rd), 256-processor Intel Xeon cluster at Sandia National Lab (ranked 111th), and 512-processor AMD Opteron cluster at Los Alamos National Lab (ranked 116th). More information about our software release can be found at [70].

CHAPTER 11

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

The research in this dissertation has demonstrated the feasibility of developing high performance and scalable communication subsystems to support MPI by taking advantage of novel InfiniBand features. We have described how we can use InfiniBand features such as different communication semantics, multiple transport services, management infrastructure, hardware multicast, completion and event mechanism, and end-to-end flow control to improve performance and scalability in different MPI components in MPI point-to-point and collective communication. Our work has focused on efficient handling of MPI communication protocols, MPI flow control mechanisms, MPI collective communication, and MPI level support for multirail InfiniBand networks. We have also proposed an comprehensive framework for evaluating the performance and scalability of MPI implementations.

11.1 Summary of Research Contributions

The expected contributions of the research are as follows: First, our research is making impact on the cluster computing area. We have demonstrated that InfiniBand is a promising cluster interconnect for high-end computing clusters. Second, although we concentrate on MPI in this work, many of our research contributions are also directly applicable to communication subsystem design in other areas such as Distributed Shared Memory (DSM) systems [46, 72], parallel files systems [102, 101] and storage system networks [96, 83]. Therefore, our work will also have an impact on these areas. Finally, many of the salient features in InfiniBand are also present in other high-speed interconnects such as Myrinet, Quadrics and 10-Gigabit Ethernet. Therefore, some of our research results are also applicable to these interconnects and will contribute to developing high performance and scalable communication subsystem for them.

Below, we will describe our research contributions in detail.

11.1.1 MPI Communication Protocol handling over InfiniBand

Most MPI communication is handled by two important MPI protocols: Eager and Rendezvous. Therefore, it is very important to handle them in an efficient and scalable manner. The Eager protocol is often used for small messages. Thus, it is desirable to minimize message latency in this protocol. On the other hand, the Rendezvous protocol is usually used for large messages. Hence, we need to maximize delivered bandwidth.

Our basic MPI design described in Chapter 4 implements a zero-copy mechanism based on RDMA for the Rendezvous protocol. Our evaluation has shown that this design can achieve very good bandwidth for large messages. In fact, the MPI implementation can delivered the same peak bandwidth achieved at the hardware level. However, this basic implementation handles small data and control messages using

InfiniBand send/receive operations, which have higher overhead and lower performance than RDMA operations. As a result, message latency in the Eager protocol is not optimized.

Our RDMA based MPI design presented in Chapter 6 addressed this inefficiency in the basic design. In this design, we propose to use RDMA operations not only for large data messages, but also for small data messages in the Eager protocol and control messages in the Rendezvous protocol. The challenge of this design is how to take advantage of RDMA operations in InfiniBand, which are one-sided in nature, to meet the requirements of two-sided communication in MPI. We have proposed novel techniques such as persistent buffer association and RDMA polling set. By using these techniques, we not only delivered better performance, but also achieved good scalability. Our performance evaluation shows that it compares favorably with the basic design. With the latest hardware, this design can deliver less than 4.0 μs latency for small messages [51, 70]. This design also affected other MPI implementations [48, 69].

11.1.2 Flow Control in MPI over InfiniBand

Flow control is an important issue in communication sub-system design. Handling of flow control control affects not only the performance of MPI applications, but also their scalability.

Our work presented in Chapter 5 provides a detailed study of the flow control issues in designing MPI over InfiniBand. Two of the central issues in flow control are performance and scalability in terms of buffer usage. We have proposed three different flow control schemes (hardware-based, user-level static and user-level dynamic) and described their respective design issues. We have implemented all three schemes in our MPI implementation over InfiniBand and conducted performance evaluation using both micro-benchmarks and the NAS Parallel Benchmarks. Our performance analysis shows that in our testbed, most NAS applications only require a very small number of pre-posted buffers for every connection to achieve good performance. We also show that the user-level dynamic scheme can achieve both performance and buffer efficiency by adapting itself according to the application communication pattern. These results have significant impact in designing large-scale clusters (in the order of 1,000 to 10,000 nodes) with InfiniBand.

11.1.3 MPI Collective Communication over InfiniBand

Modern high performance applications require efficient and scalable collective communication operations. Currently, most collective operations are implemented based on point-to-point operations. In Chapter 7, we propose to use hardware multicast in InfiniBand to design fast and scalable broadcast operations in MPI. InfiniBand supports multicast with Unreliable Datagram (UD) transport service. This makes it hard to be directly used by an upper layer such as MPI. To bridge the semantic gap between MPI_Bcast and InfiniBand hardware multicast, we have designed and implemented a substrate on top of InfiniBand which provides functionalities such as reliability, in-order delivery and large message handling. By using a sliding-window based design, we improve MPI_Bcast latency by removing most of the overhead in the substrate out of the communication critical path. By using optimizations such as a new *co-root based scheme* and *delayed ACK*, we can further balance and reduce the overhead. We have also addressed many detailed design issues such as buffer management, efficient handling of out-of-order and duplicate messages, timeout and retransmission, flow control and RDMA based ACK communication.

Our performance evaluation shows that in an 8 node cluster testbed, hardware multicast based designs can improve MPI broadcast latency up to 58% and broadcast throughput up to 112%. The proposed solutions are also much more tolerant to process skew compared with the current point-to-point based implementation. We have also developed analytical model for our multicast based schemes and validated them with experimental numbers. Our analytical model shows that with the new designs, one can achieve MPI broadcast latency of small messages with 20.0μs and of one MTU size message (around 1836 bytes of data payload) with 40.0μs in a 1024 node cluster.

11.1.4 MPI Level Support for Multirail InfiniBand Networks

In the area of cluster computing, InfiniBand is becoming increasingly popular due to its open standard and high performance. However, even with InfiniBand, network bandwidth can still become the performance bottleneck for some of today's most demanding applications.

In Chapter 8, we study the problem of how to overcome the bandwidth bottleneck by using multirail networks. We present different ways of setting up multirail networks with InfiniBand and propose a unified MPI design that can support all these approaches. We have also discussed various important design issues and provided in-depth discussions of different policies of using multirail networks, including an adaptive striping scheme that can dynamically change the striping parameters based on current system condition.

We have implemented our design and evaluated it using both microbenchmarks and applications. Our performance results show that multirail networks can significant improve MPI communication performance. With a two rail InfiniBand cluster, we have achieved almost twice the bandwidth and half the latency for large messages compared with the original MPI. At the application level, the multirail MPI can significantly reduce communication time as well as running time depending on the communication pattern. We have also shown that the adaptive striping scheme can achieve excellent performance without *a priori* knowledge of the bandwidth of each rail. Since many large-scale, high-end clusters are built with multirail configurations, our design can be used in these systems to achieve better performance.

11.1.5 Performance Evaluation Framework

In Chapter 9, we present a comprehensive performance evaluation framework for MPI implementations. We compare our MVAPICH implementation with MPI implementations over two other popular high-speed interconnects: Myrinet [71, 66, 67] and Quadrics [81, 78].

Our performance evaluation consists of two major parts. The first part consists of a set of MPI level micro-benchmarks. These benchmarks include traditional measurements such as latency, bandwidth and host overhead. In addition to those, we have also included the following micro-benchmarks: communication/computation overlap, buffer reuse, memory usage, intra-node communication and collective communication. The objective behind this extended micro-benchmark suite is to characterize different aspects of the MPI implementations and get more insights into their communication behavior.

The second part of the performance evaluation consists of application level benchmarks. We have used the NAS Parallel Benchmarks [68] and the sweep3D benchmark [33]. We not only present the overall performance results, but also relate application communication characteristics to the information we got from the micro-benchmarks. We use in-depth profiling of these applications to measure their characteristics. Using these profiled data and the results obtained from the micro-benchmarks, we analyze the impact of the following factors: overlap of computation and communication, buffer reuse, collective communication, memory usage, SMP performance and scalability with system sizes.

The main contributions of this work are: First, we present a detailed performance study of MPI over InfiniBand, Myrinet and Quadrics, using both applications and micro-benchmarks. Second, we have shown that MPI communication performance is affected by many factors. Therefore, to get more insights into different aspects of an MPI implementation, one has to go beyond simple micro-benchmarks such as latency and bandwidth. Third, our results show that for 8-node clusters, InfiniBand can offer significant performance improvements for many bandwidth-bound applications compared with Myrinet and Quadrics.

It should be noted that our evaluation is done in the year 2003. Currently, Myrinet and Quadrics have released their next generation hardware which gives better performance. Similar, InfiniBand hardware is also improving. Thus, some of our conclusions might change. However, the same performance evaluation framework can be used to compare MPI implementations over the latest hardware.

11.2 Future Research Directions

The high performance and rich features provided by the InfiniBand Architecture make it an attractive interconnect for high performance computing. In this dissertation, we have demonstrated that it is possible to implement an efficient and scalable MPI layer over InfiniBand. We have discussed various design issues including MPI protocol handling, flow control, collective communication, multirail network support and MPI performance evaluation framework. However, there are still many interesting research topics to pursue in this area. Below we describe some of these future research topics:

Utilizing other InfiniBand features — In this dissertation, we have used many features provided by InfiniBand. But there are still more for us to explore. For example, InfiniBand recently introduced a feature called *shared receive queues*. By using this feature, it is possible to allow multiple connections to share a single buffer pool and thus achieve better buffer utilization and scalability. Another example is InfiniBand atomic operations. Although we have not used InfiniBand atomic operations in this dissertation, they are very important features and we have used them to design high-performance MPI one-sided communication in [40, 39]. However, it is possible to use atomic operations in two-sided communication also. For example, it is possible to use atomic operations in designing flow control mechanisms when shared receive queues are used.

InfiniBand also provides native QoS support. Researchers have studied various QoS issues over InfiniBand [38, 41, 3]. It is also an interesting research topic to use InfiniBand QoS mechanisms in implementing MPI. One possible scenario is to assign different service levels to different kinds of messages. For example, some control messages can be assigned a higher priority than other messages. The impact of different QoS policies on application performance is also a very important topic.

Multirail network support and collective communication — In Chapter 9, we have presented a unified MPI design to support different InfiniBand multirail networks. However, our design focused mostly on point-to-point communication. MPI collective communication usually has very regular communication pattern. Thus, it is possible to design multirail support more efficiently by directly considering the communication pattern in MPI collective communication operations. Multirail networks can not only improve communication bandwidth for operations such as MPI_Alltoall, but also avoid hot spots in one-to-all and all-to-one patterns which are common in collective communication.

MPI level support for fault tolerance over InfiniBand — In large scale systems, it is vital that the system can continue to function even in case of various hardware or software faults. Although, it is possible to mask some of these faults at a lower level, MPI level fault tolerance is still necessary in order to achieve overall reliability. In this dissertation, we have focused mostly on performance and scalability. It would be interesting to study how to design fault tolerant MPI over InfiniBand also. For example, it is possible to combine our multirail MPI design with reliability support to tolerate network failures. To deal with node failures, checkpointing/restarting is a very common technique. The high performance and various features offered by InfiniBand provide many opportunities to design efficient checkpointing, process migration and process restarting systems for MPI programs.

MPI support for 10-GigE networks — Recently, the next generation ethernet network – 10-Gigabit Ethernet has been introduced. Although still in its early stage of deployment, 10-GigE networks are already achieving quite high performance [35]. Although current they are very expensive, their price may drop dramatically when they are deployed in large scales. Traditionally, communication over ethernet is achieved by using TCP/IP protocols with a socket-based interface. This interface is not a very good match for high performance computing applications that use the MPI interface. Recently, an RDMA based interface (RDMA over IP) has been proposed for ethernet networks [84, 83]. This interface is quite similar to the interface provided by InfiniBand. Therefore, many of designs are also applicable in this case. On the other hand, RDMA over IP also provide some new features such as enhanced memory registration [14] as well as the opportunities to combine it with socket based communication. Therefore, further research is necessary to determine how to implement MPI efficiently for the next generation 10-GigE networks.

BIBLIOGRAPHY

[1] IBAL: InfiniBand Linux SourceForge Project. http://-infiniband.sourceforge.net/IAL/Access/IBAL.

[2] Hari Adiseshu, Guru M. Parulkar, and George Varghese. A reliable and scalable striping protocol. In *SIGCOMM*, pages 131–141, 1996.

[3] Francisco J. Alfaro, Jose L. Sanchez, Jose Duato, and Chita R. Das. A Strategy to Compute the Infiniband Arbitration Tables. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '02)*, April 2002.

[4] M. Banikazemi, V. Moorthy, L. Herger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for the IBM SP Switch-Connected NT Clusters. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 33–42, May 2000.

[5] Mohammad Banikazemi, Rama K. Govindaraju, Robert Blackmore, and Dhabaleswar K. Panda. MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 1081–1093, October 2001.

[6] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome, and Katherine Yelick. An evaluation of current high-performance networks. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, April 2003.

[7] Bhoedjang, Ruhl, and Bal. Efficient multicast on myrinet using link-level flow control. In *ICPP: 27th International Conference on Parallel Processing*, 1998.

[8] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro*, pages 21–28, Feb. 1995.

[9] C. Brendan, S. Traw, and Jonathan M. Smith. Striping within the network subsystem. *IEEE Network*, 9(4):22, 1995.

[10] 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.

[11] Ron Brightwell and Anthony Skjellum. MPICH on the T3D: A Case Study of High Performance Message Passing. In *1996 MPI Developers Conference*, July 1996.

[12] D. Buntinas, D. K. Panda, and R. Brightwell. Application-bypass broadcast in mpich over gm. In *International Symposium on Cluster Computing and the Grid (CCGRID '03)*, May 2003.

[13] Enrique V. Carrera, Srinath Rao, Liviu Iftode, and Ricardo Bianchini. User-Level Communication in Cluster-Based Servers. In *Proceedings of the Eighth Symposium on High-Performance Architecture (HPCA'02)*, pages 275–286, February 2002.

[14] Mallikarjun Chadalapaka, Hemal Shah, Uri Elzur, Patricia Thaler, and Michael Ko. A Study of iSCSI Extensions for RDMA (iSER). In *ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, August 2003.

[15] J. Chase, A. Gallatin, and K. Yocum. End System Optimizations for High-Speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[16] Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, and Leonid Gurvits. Using Multirail Networks in High-Performance Clusters. *Concurrency and Computation: Practice and Experience*, 15(7-8):625, 2003.

[17] Compaq, Intel, and Microsoft. VI Architecture Specification V1.0, December 1997.

[18] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[19] Rossen Dimitrov and Anthony Skjellum. An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. http://www.mpi-softtech.com/publications/, 1998.

[20] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.

[21] Abbas Farazdel, Gonzalo R. Archondo-Callao, Eva Hocks, Takaaki Sakachi, and Federico Vagnini. *IBM Red Book: Understanding and Using the SP Switch.* IBM, Poughkeepsie, NY, 1999.

[22] W. Feng, M. Gardner, M. Fisk, and E. Weigle. Automatic Flow-Control Adaptation for Enhancing Network Performance in Computational Grids. *Journal of Grid Computing*, 1(1):63–74, 2003.

[23] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[24] Jinzhu Gao and Han-Wei Shen. Parallel View-Dependent Isosurface Extraction Using Multi-Pass Occlusion Culling. In *Proceedings of 2001 IEEE Symposium in Parallel and Large Data Visualization and Graphics*, pages 67–74, October 2001.

[25] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnichand C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network failure tolerant message passing system for terascale clusters. In *16th Annual ACM International Conference on Supercomputing (ICS '02)*, June 2002.

[26] W. Gropp and E. Lusk. A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997.

[27] 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, 1996.

[28] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions.* MIT Press, Cambridge, MA, USA, 1998.

[29] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[30] Rinku Gupta, Vinod Tipparaju, Jarek Nieplocha, and Dhabaleswar K. Panda. Efficient Barrier using Remote Memory Operations on VIA-Based Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[31] H. Eriksson. Mbone: the multicast backbone. *Communications of the ACM*, August 1994.

[32] H. Tezuka and F. O'Carroll and A. Hori and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *In Proceedings of 12th International Parallel Processing Symposium*, pages 308–315, 1998.

[33] Adolfy Hoisie, Olaf Lubeck, Harvey Wasserman, Fabrizio Petrini, and Hank Alme. A General Predictive Performance Model for Wavefront Algorithms on Cluster of SMPs. In *ICPP 2000*, 2000.

[34] Jenwei Hsieh, Tau Leng, Victor Mashayekhi, and Reza Rooholamini. Architectural and performance evaluation of giganet and myrinet interconnects on clusters of small-scale SMP servers. In *Supercomputing*, 2000.

[35] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro*, 24(1):10–22, 2004.

[36] P. Husbands and J. C. Hoe. MPI-StarT: Delivering Network Performance to Numerical Applications. In *Proceedings of the Supercomputing*, 1998.

[37] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.1. http://www.infinibandta.org, November 2002.

[38] J. Pellissier. Providing Quality of Service over InfiniBand Architecture Fabrics. In *Hot Interconnect 8*, August 2000.

[39] W. Jiang, J. Liu, H. Jin, D. K. Panda, D. Buntinas, R. Thakur, and W. Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication over InfiniBand Clusters. In *Euro PVM/MPI Conference*, September 2004.

[40] W. Jiang, J. Liu, H. Jin, D. K. Panda, W. Gropp, and R. Thakur. High performance mpi-2 one-sided communication over infiniband. In *International Symposium on Cluster Computing and the Grid (CCGRID '04)*, April 2004.

[41] Eun Jung Kim, Ki Hwan Yum, Chita Das, Mazin Yousif, and Jose Duato. Performance enhancement techniques for infiniband architecture. In *International Symposium on High Performance Computer Architecture*, Feb. 2003.

[42] Sushmitha P. Kini. Efficient Collective Communication using RDMA and Multicast Operations for InfiniBand-Based Clusters. Master Thesis, The Ohio State University, June 2003.

[43] Sushmitha P. Kini, Jiuxing Liu, Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Fast and Scalable Barrier using RDMA and Multicast Mechanisms for InfiniBand-Based Clusters. In *EuroPVM/MPI*, Oct. 2003.

[44] Mario Lauria and Andrew Chien. MPI-FM: High performance MPI on workstation clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, 1997.

[45] Lawrence Berkeley National Laboratory. MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/index.html, August 2001.

[46] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–239, New York, NY, 1986. ACM Press.

[47] John C. Lin and Sanjoy Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, March 1996.

[48] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.

[49] J. Liu, A. Mamidala, and D. K. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.

[50] J. Liu and D. K. Panda. Implementing Efficient and Scalable Flow Control Schemes in MPI over InfiniBand. In *Proceedings of the 2004 Workshop on Communication Architecture for Clusters (CAC '04)*, April 2004.

[51] J. Liu, A. Vishnu, and D. K. Pand. Performance Evaluation of InfiniBand with PCI Express. In *Hot Interconnect 12*, August 2003.

[52] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, June 2004.

[53] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Pete Wyckoff, and Dhabaleswar K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *SuperComputing 2003 (SC '03)*, November 2003.

[54] Jiuxing Liu, Abhinav Vishnu, and Dhabaleswar K. Panda. Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation. In *SuperComputing 2004 (SC '04)*, November 2004, to be presented.

[55] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing (ICS '03)*, June 2003.

[56] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kinis, Darius Buntinas, Weikuan Yu, Balasubraman Chandrasekaran, Ranjit Noronha, Peter Wyckoff, and Dhabaleswar K. Panda. MPI over InfiniBand: Early Experiences. Technical Report, OSU-CISRC-10/02-TR25, Computer and Information Science, the Ohio State University.

[57] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.

[58] R. Martin, A. Vahdat, D. Culler, and T. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 152–159, 1997.

[59] Mathematics and Computer Science Division, Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich/, 2003.

[60] Mellanox Technologies. Mellanox InfiniBand InfiniHost MT23108 Adapters. http://www.mellanox.com, July 2002.

[61] Mellanox Technologies. Mellanox VAPI Interface, July 2002.

[62] Mellanox Technologies. Mellanox InfiniBand InfiniHost III Ex MT25208 Adapters. http://www.mellanox.com, February 2004.

[63] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[64] Jeffrey C. Mogul. TCP Offload Is a Dumb Idea whose Time Has Come. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.

[65] Myricom. GM Messaging Software. http://www.myri.com/scs/index.html.

[66] Myricom. MPICH-GM. http://www.myri.com/myrinet/performance/MPICH-GM/index.html.

[67] Myricom. Myrinet. http://www.myri.com/.

[68] NASA. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[69] NCSA. MPICH over VMI2 Interface. http://vmi.ncsa.uiuc.edu/.

[70] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI Layer. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html.

[71] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[72] Rajit M Noronha and D K Panda. Designing High Performance DSM Systems using InfiniBand Features. In *The 2004 International Workshop on Distributed Shared Memory on Clusters (DSM '04)*, April 2004.

[73] OpenIB Alliance. OpenIB InfiniBand Software. http://www.openib.org.

[74] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[75] Pallas. Pallas MPI Benchmarks. http://www.pallas.com/e/products/pmb/.

[76] PCI-SIG. PCI Express Architecture. http://www.pcisig.com.

[77] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, 2002.

[78] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[79] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware- and Software-Based Collective Communication on the Quadrics Network. In *IEEE International Symposium on Network Computing and Applications 2001 (NCA 2001)*, Boston, MA, February 2002.

[80] Sridhar Pingali, Don Towsley, and James F. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. In *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 221–230, New York, NY, USA, 1994. ACM Press.

[81] Quadrics. Quadrics, Ltd. http://www.quadrics.com.

[82] R. Gupta, P. Balaji, D. K. Panda, and J. Nieplocha. Efficient Collective Operations using Remote Memory Operations on VIA-Based Clusters. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[83] RDMA Consortium. iSCSI Extensions for RDMA (iSER) and Datamover Architecture for iSCSI (DA) Specifications, 2003.

[84] RDMA Consortium. iWARP Protocol Suite Specifications, 2003.

[85] S. Pakin and A. Pant. VMI 2.0: A Dynamically Reconfigurable Messaging Layer for Availability, Usability, and Management. In *SAN-1 Workshop (in conjunction with HPCA)*, Febuary 2002.

[86] SGI. SGI Message Passing Toolkit. http://www.sgi.com/software/mpt/overview.html.

[87] S. J. Sistare and C. J. Jackson. Ultra-High Performance Communication with MPI and the Sun Fire Link Interconnect. In *Proceedings of the Supercomputing*, 2002.

[88] TOP500 SUPERCOMPUTER SITES. 23rd Edition of TOP500 List. http://www.top500.org, November 2003.

[89] Harimath Sivakumar, Stuart Bailey, and Robert L. Grossman. PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *Supercomputing*, 2000.

[90] Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, and Jack Dongarra. *MPI–The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition.* The MIT Press, 1998.

[91] Ted Tabe and Quentin F. Stout. The use of the MPI communication library in the NAS parallel benchmarks. Technical Report CSE-TR-386-99, University of Michgan, 1999.

[92] Topspin Communications, Inc. Topspin Communications, Inc.

[93] V. Tipparaju, J. Nieplocha, D.K. Panda. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[94] Robert van de Geijn, David Payne, Lance Shuler, and Jerrell Watts. A Streetguide to Collective Communication and its Application. http://www.cs.utexas.edu/users/rvdg/pubs/streetguide.ps, Jan 1996.

[95] Jeffrey. S. Vetter and Frank Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '02)*, April 2002.

[96] Voltaire Inc. High Performance SAN Connectivity for InfiniBand Fabrics. http://www.voltaire.com/pdf/-storage_wp_final.pdf.

[97] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, pages 40–53, 1995.

[98] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.

[99] Frederick C. Wong, Richard P. Martin, Remzi H. Arpaci-Dusseau, and David E. Culler. Architectural requirements and scalability of the nas parallel benchmarks. In *In the Proceedings of Supercomputing'99*, 1999.

[100] Jiesheng Wu, Jiuxing Liu, Pete Wyckoff, and Dhabaleswar K. Panda. Impact of On-Demand Connection Management in MPI over VIA. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 152–159, September 2002.

[101] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Proceedings of the 2003 International Conference on Parallel Processing (ICPP 03)*, Oct. 2003.

[102] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Supporting Efficient Noncontiguous Access in PVFS over InfiniBand. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[103] Weikuan Yu, Darius Buntinas, and Dhabaleswar K. Panda. High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2. In *Int'l Conference on Parallel Processing, (ICPP 2003)*, Kaohsiung, Taiwan, October 2003.

[104] Yuanyuan Zhou, Angelos Bilas, Suresh Jagannathan, Cezary Dubnicki, James F. Philbin, and Kai Li. Expericences with VI Communication for Database Storage. In *Proceedings of International Symposium on Computer Architecture'02*, pages 257–268, 2002.