ENHANCING MPI WITH MODERN NETWORKING MECHANISMS IN CLUSTER INTERCONNECTS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Weikuan Yu, B.S., M.S.

* * * * *

The Ohio State University

2006

Dissertation Committee:

Professor Dhabaleswar K. Panda, Adviser

Professor Ponnuswamy Sadayappan

Professor Mario Lauria

Approved by

Adviser

Graduate Program in
Computer Science &
Engineering

ABSTRACT

Advances in CPU and networking technologies make it appealing to aggregate commodity compute nodes into ultra-scale clusters. But the performance achievable is highly dependent on how tightly their components are integrated together. The ever-increasing size of clusters and applications running over them leads to dramatic changes in the requirements. These include at least scalable resource management, fault tolerance process control, scalable collective communication, as well as high performance and scalable parallel IO.

Message Passing Interface (MPI) is the *de facto* standard for the development of parallel applications. There are many research efforts actively studying how to leverage the best performance of the underlying systems and present to the end applications. In this dissertation, we exploit various modern networking mechanisms from the contemporary interconnects and integrate them into MPI implementations to enhance their performance and scalability. In particular, we have leveraged the novel features available from InfiniBand, Quadrics and Myrinet to provide scalable startup, adaptive connection management, scalable collective operations, as well as high performance parallel IO. We have also designed a parallel Checkpoint/Restart framework to provide transparent fault tolerance to parallel applications.

Through this dissertation, we have demonstrated that modern networking mechanisms can be integrated into communication and IO subsystems for enhancing the scalability, performance and reliability of MPI implementations. Some of the research results have been incorporated into production MPI software releases such as MVAPICH/MVAPICH2 and

LA-MPI. This dissertation has showcased and shed light on where and how to enhance the design of parallel communication subsystems to meet the current and upcoming requirements of large-scale clusters, as well as high end computing environments in general.

Keywords: *InfiniBand, Myrinet, Quadrics, MPI, Parallel IO, RDMA*

To my wife, Juan Gao;

my kids, Wen-rui and Michael;

and my parents, Gongyuan and Zhixiu.

ACKNOWLEDGMENTS

I would like to thank my adviser Prof. D. K. Panda for culturing me as a researcher in the field of parallel and high performance computing science. I greatly appreciate his help, support and friendship during my graduate study. I am indebted for the time and efforts which he spent for my progress and appreciate the wisdom he shared with me.

I would also like to thank the other members of my dissertation committee, Prof. P. Sadayappan and Prof. M. Lauria, 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 Dr. Darius Buntinas, Dr. Pete Wyckoff and Piyush Shivam for their efforts and time in guiding me through the basics during the first year of my Ph.D. program.

I would also like to thank Dr. Rich Graham, Dr. David Daniel and Tim Woodall for their guidance and support during my summer intern at Los Alamos National Laboratory.

I am very fortunate to have worked with many current and previous members of NOWLAB: Jiuxing, Jiesheng, Pavan, Bala, Adam M., Sayantan, Shuang, Lei and Qi. I am grateful for their valuable comments, suggestions and critiques.

I am also thankful to other members of NOWLAB, especially Jin, Gopal, Amith, Sundeep, Karthik, Adam W., Wei, Matt and Prachi for their discussions on technical and non-technical issues, as well as their friendship.

Finally, I would like to thank my family: my wife, Juan Gao, my kids, Wen-rui and Michael, my sisters, Weiqiong and Weizhen, and my parents, Gongyuan and Zhixiu for their support, love, encouragement and also patience throughout my long journey of study over all these years.

VITA

1974	Born – Xiantao, Hubei Province, China, 433013
July 1995	B.S. – Genetics, Wuhan University, China
June 1998	M.S. – Molecular Cellular Biology, Chinese Academy of Science, Shanghai
August 2001	M.S. – Mol. Dev. & Cellular Biology, The Ohio State University, Ohio
December 2004	M.S. – Computer Science, Continued in Ph.D Track, The Ohio State University, Ohio
September 2004 – Present	Graduate Research Associate, The Ohio State University, Ohio
June 2004 – September 2004	Summer Intern, Los Alamos National Laboratory
June 1999 – June 2004	Graduate Teaching/Research Associate, The Ohio State University, Ohio

PUBLICATIONS

W. Yu, S. Sur, D. K. Panda, R. T. Aulwes and R. L. Graham. "High Performance Broadcast Support in LA-MPI over Quadrics". *Special Issue of The International Journal of High Performance Computer Applications*. March 2005.

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

Q. Gao, **W. Yu** and D. K. Panda. "Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand". *The 35th International Conference for Parallel Processing (ICPP'06)*, Columbus, OH, August 14-18, 2006.

S. Liang, **W. Yu** and D. K. Panda. "High Performance Block I/O for Global File System (GFS) with InfiniBand RDMA". *The 35th International Conference for Parallel Processing (ICPP'06)*, Columbus, OH, August 14-18, 2006.

W. Yu, R. Noronha, S. Liang and D. K. Panda. "Benefits of High Speed Interconnects to Cluster File Systems: A Case Study with Lustre". *International Workshop on Communication Architecture for Clusters (CAC '06), held together with IPDPS '06.*

W. Yu, Q. Gao and D. K. Panda. "Adaptive Connection Management for Scalable MPI over InfiniBand". *International Parallel and Distributed Processing Symposium (IPDPS '06).* Rhodes Island, Greece. April 2006.

W. Yu and D. K. Panda. "Benefits of Quadrics Scatter/Gather to PVFS2 Noncontiguous IO". *International Workshop on Storage Network Architecture and Parallel IO. Held with the 14th International Conference on Parallel Architecture and Compilation Techniques.* Sept 2005. St Louis, Missouri.

P. Balaji, W. Feng, Q. Gao, R. Noronha, **W. Yu** and D. K. Panda. "Head-to-TOE Comparison for High Performance Sockets over Protocol Offload Engines". *In the proceedings of IEEE International Conference on Cluster Computing (Cluster '05)*, September 2005, Boston, Massachusetts.

W. Yu, S. Liang and D. K. Panda. "High Performance Support of Parallel Virtual File System (PVFS2) over Quadrics". *The 19th ACM International Conference on Supercomputing.* June 2005. Cambridge, Massachusetts.

W. Yu, T.S. Woodall, R.L. Graham and D. K. Panda. "Design and Implementation of Open MPI over Quadrics/Elan4". *International Parallel and Distributed Processing Symposium (IPDPS '05).* Denver, Colorado. April 2005.

W. Yu, D. Buntinas, D. K. Panda. "Scalable, High Performance NIC-Based All-to-All Broadcast over Myrinet/GM". *International Conference on Cluster Computing (Cluster '04).* San Diego, CA. September 2004.

W. Yu, J. Wu, D. K. Panda. "Fast and Scalable Startup of MPI Programs In InfiniBand Clusters". *International Conference on High Performance Computing (HiPC '04).* Banglore, India. December 2004.

W. Yu, D. Buntinas, D. K. Panda and R. L. Graham. "Efficient and Scalable NIC-Based Barrier over Quadrics and Myrinet". *Workshop on Communication Architecture for Clusters (CAC '04). Held in conjunction with IPDPS 2004.* April 2004.

J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. P. Kini, **W. Yu**, D. Buntinas, P. Wyckoff and D. K. Panda. "Performance Comparison of MPI implementations over InfiniBand, Myrinet and Quadrics". *Int'l Conference on Supercomputing, (SC'03)*, Phoenix, Arizona. Nov 2003.

W. Yu, S. Sur, D. K. Panda, R. T. Aulwes and R. L. Graham. "High Performance Broadcast Support in LA-MPI over Quadrics". *In Los Alamos Computer Science Institute Symposium, (LACSI'03)*, Santa Fe, New Mexico, October 2003.

W. Yu, D. Buntinas and D. K. Panda. "High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2". *International Conference on Parallel Processing, (ICPP'03)*, Kaohsiung, Taiwan, October 2003.

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

FIELDS OF STUDY

Major Field: Computer Science

Studies in:

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

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Because of its excellent price/performance ratio [77], cluster environment has emerged as one of the main platforms for high end computing (HEC). Clusters of workstations have evolved into systems with tens of thousands of processors connected with high speed networks [4]. As shown in Figure 1.1, a typical cluster-based HEC environment consists of three different sets of nodes: front-end, computing cluster and storage cluster. Parallel applications are first launched from one of the front-end nodes and scheduled to a group of computing nodes. Many of these parallel applications demand large amounts of Input/Ouput (I/O) services. Their performance greatly relies on how well the I/O subsystem can sustain the demand for data requests.

Nowadays, CPU becomes more and more powerful. At the same time, more processors are able to be aggregated into a single box. Thus, in a HEC environment with tens of thousands of processors, the aggregated physical computing power can reach tens or hundreds of Tera-Flops (10^{15} FLoating point Operations Per Sec). Department of Energy (DOE) also has intiatives putting up several Peta-Flops (10^{18}) computing systems in the next five years. However, the performance that can be delivered to end applications, a.k.a efficiency, is highly dependent on how the nodes can be integrated together for the destined functionalities of their subsystems. Specifically, this performance efficiency is correlated to a spectrum of issues

including scalability, fault tolerance, collective communication, and parallel IO, etc. The achievable performance to end applications usually falls much below the peak performance of a cluster, i.e. the aggregated FLOPS and IO throughput from all participating nodes. Much of this performance gap can be attributed to the cost at different layers of programming libraries through which parallel applications access the computing and storage resources. These include the operating system, the TCP/IP protocol and parallel programming libraries such as MPI [52].

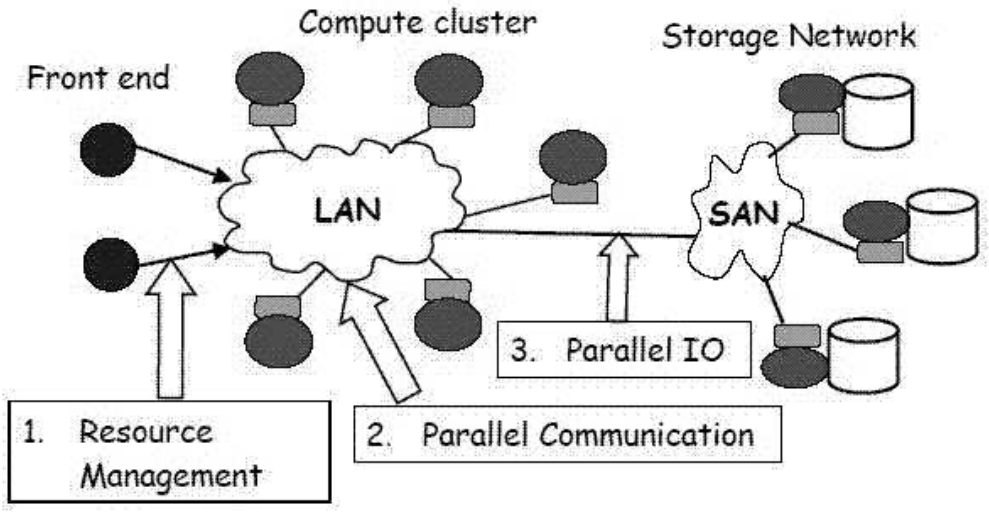


Figure 1.1: Cluster-Based High-End Computing Environment

Various strategies have been proposed and practically exploited to improve the communication performance. Some of them focus on optimizing the traditional TCP/IP communication path, while maintaining the interface to the user applications. Work, such as Container shipping [67], IO-Lite [65], TCP offloading engine, and KStream IO [47] fall into this category. Others propose new protocols that reduce or eliminate the kernel involvement from the communication critical path. These include research projects such as AM [86], VMMC [11]

and FM [66], as well as commercial packages that are delivered along with the interconnects, such as GM [59] and Elan libraries [73]. Using OS-bypass and zero-copy techniques, they provide high performance communication to the application layer. MPI [53] is one of the programming models that facilitate the development of parallel applications, and it can leverage the benefits of those high performance protocols. MPI specifies a set of interface for point-to-point communication, collective communication and noncontiguous communication. MPI-2 extends MPI with I/O, one-side communication, dynamic process management, various language bindings, as well as expanded collective interfaces. Efficiently implementing and optimizing MPI interfaces has been an active research field in both academia and industrial domains.

On the other hand, high end interconnect technologies, such as Myrinet [12], Quadrics [7] and InfiniBand [43], have provided modern networking mechanisms in software or hardware to support high performance communication. These include OS-bypass user-level communication, remote direct memory access (RDMA) [1], atomic network operations and hardware collective operations. Some of the interconnects provide programmable processors and a considerable amount of memory in the Network Interface Cards (NICs), e.g. Myrinet, Quadrics, and some of the new generations of 10Gigabit Ethernet interfaces. Communication subsystems built on top of these interconnects typically strive to expose the best performance of their modern networking mechanisms. For example, Quadrics [73] Elan and Myrinet GM (and newer MX) libraries [59] have offloaded part of MPI communication stack into the network interface and provide several hardware broadcast-based collective operations; MVAPICH/MVAPICH2 [60] leverages InfiniBand RDMA and hardware broadcast into the MPI ADI layer implementations. These studies have suggested their benefits on improving

communication performance, though only a few network mechanisms have already been exploited. In this dissertation, we present our studies on how to take advantage of modern networking mechanisms for enhancing MPI in modern cluster interconnects.

In the rest of this Chapter, we first provide an overview of MPI and relevant scalability issues. We then provide an overview of InfiniBand, Quadrics and Myrinet, and also describe modern networking mechanisms available from them. Following that, we present the problem statement and our research approaches. At the end, we provide an overview of this dissertation.

1.1. Overview of MPI

In the early 1990's, message passing was widely accepted as a promising paradigm for building scalable parallel applications on parallel computers. The Message Passing Interface (MPI) Forum was formed to standardize the core of message passing routines as the Message Passing Interface (MPI) [53] based on various existed message-passing systems at that time. MPI Forum has designed a MPI programming model with great portability, by carefully selecting the most attractive and essential features.

MPI [53] has since established itself as the *de facto* parallel programming standard. MPI-2 [54] provides extensions to the original MPI specification, including process creation and management, one-sided communication, expanded collective operations, additional language bindings, and MPI-IO. Various MPI-2 compliant implementations have been provided by commercial vendors and research projects, notable ones include, Sun-MPI [76], IBM-MPI [41], MPICH2 [57], LAM/MPI [22] and MVAPICH [60]. While MPI provides many low-level services, such as process management, IO, communication, debugging, timing, profiling, language bindings, etc, we focus on the components that are on the critical path of

MPI communication and IO in this dissertation. Figure 1.2 shows a generic diagram of these software components.

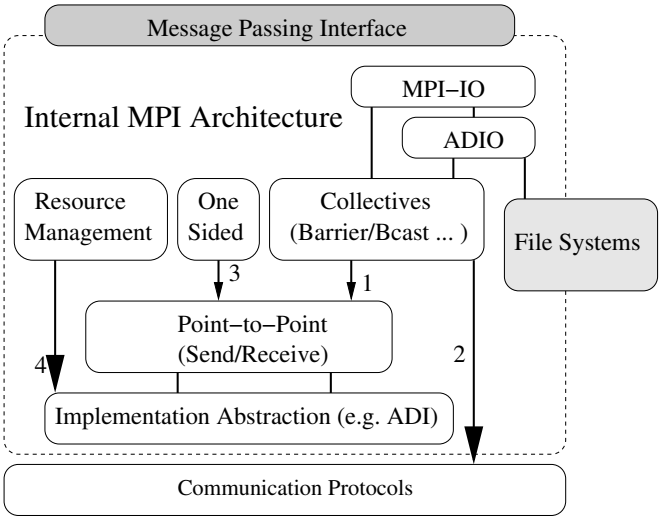


Figure 1.2: Typical Software Components of MPI Implementations

MPI basic functions are a set of send and receive operations, together referred to as point-to-point communication. A message is sent from the sender process using one of the send routines. It is then received by the receiver process using one of the receive routines. MPI enforces a matched, ordered message passing model. Typically, the standard blocking mode send/receive routines: MPI_Send and MPI_Recv, are used. Other communication modes such as *synchronous, buffered* and *ready* are also provided. MPI also provides a nonblocking communication mechanism. In this mechanism, nonblocking operations are posted to initiate the communication and return locally without waiting for the actual communication to complete.

1.1.1. MPI Collective Communication

Besides basic point-to-point communication, MPI provides a large set of routines to perform collective communication among a group of processes. An abstraction called *communicator* is used to represent process groups. In a collective operation, data is transmitted among all the processes identified by a communicator context. The barrier operation is provided to explicitly synchronize all the involving processes. It does not involve data communication. Other collective operations include broadcast, gather/scatter, reduce, as well as their variants. The semantics and requirements of the collective functions are simplified in several aspects. Notably, all collective operations are strictly blocking, but each process can complete as soon as its involvement is completed, i.e., without having to wait for the completion of other processes. In addition, no tags are provided to match collective operations and all collective operations must strictly be matched in order. These design choices allow easy usages for the application developers and efficient implementation for the MPI developers.

MPI implementations typically provide basic collective communication support layered on top of its point-to-point communication, as shown by the arrowhead 1 in Figure 1.2. Due to the significance of collective operations in parallel processing [85], a large amount of research has been done to improve their performance. This is often done by providing a lightweight collective implementation directly over the underlying network protocol, shown as the arrowhead 2 in Figure 1.2.

1.1.2. MPI-IO

MPI-IO defines a portable API for high performance parallel IO. It is designed to overcome limitations of POSIX compliant file systems to support high performance and portable

parallel IO. MPI-IO employs several abstractions: derived datatypes, file view and collective IO, to increase the expressiveness of its interface. It also supports asynchronous IO, strided accesses, and storage layout control, allowing the underlying file system to take advantage of this flexibility. MPI-IO also presents the IO services available from the underlying file system to the parallel application in the form of collective IO. As shown in Figure 1.2, it utilizes collective operations from an existing MPI implementation for easy collective communication support of collective IO. Currently, ROMIO provides a portable MPI-IO implementation using an abstract device IO interface (ADIO) [80]. It supports high performance MPI-IO using data-sieving and two-phase collective IO techniques [81].

1.1.3. MPI Scalability and Fault Tolerance

Ultra-scale systems with tens of thousands of processors [4] lead to dramatically different challenges and requirements, which include not only the traditional crave for low latency and high bandwidth but also the need for scalable startup/teardown and fault-tolerant process control. One of the major challenges in process management is the fast and scalable startup of large-scale applications [15, 39, 46, 34, 45]. As the number of processes scales it is also a challenge about how to establish and maintain the communication channels among all the processes. Furthermore, statistics tells us that the failure rate of a system grows exponentially with the number of the components. The applications running over ultra-scale systems become more error-prone, especially as the failure of any single component tends to cascade widely to other components because of the interaction and dependence between them. These issues become more pronounced while the scale of these systems increases.

1.2. Overview of InfiniBand, Quadrics and Myrinet

In this Section, we provide a brief overview of high speed interconnects including InfiniBand, Quadrics and Myrinet.

1.2.1. InfiniBand

The InfiniBand Architecture (IBA) [43] is an open specification designed for interconnecting compute nodes, IO nodes and devices in a system area network. As shown in Figure 1.3, it defines a communication architecture from the switch-based network fabric to transport layer communication interface for inter-processor communication with high bandwidth and low latency.

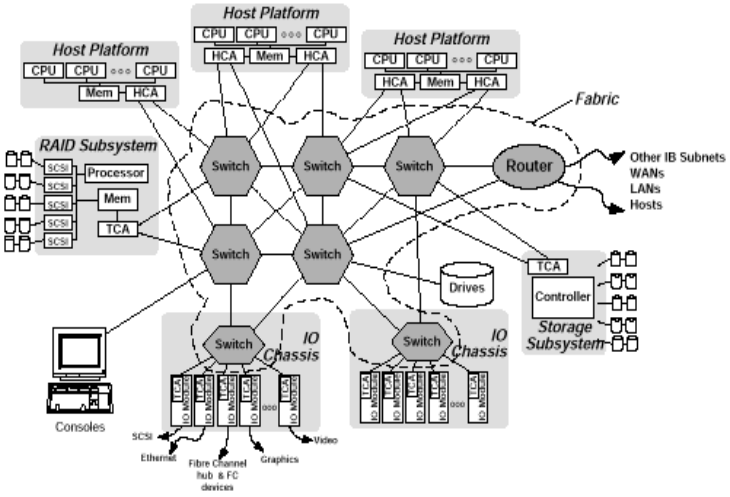


Figure 1.3: The Switch Fabric of InfiniBand Architecture (Courtesy InfiniBand Trade Association [43])

InfiniBand provides five types of transport services: Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC), Unreliable Datagram (UD), and raw datagram. It supports communications in both channel semantics with traditional send/receive operations and memory semantics with RDMA operations, which enables zero-copy data movement without host CPU involvement. Further details on RDMA are discussed in Section 1.3.

1.2.2. Quadrics

Quadrics network (QsNet) [71] provides low-latency, high-bandwidth clustering technology. In Quadrics clusters, computing nodes are connected using a quaternary fat-tree switch network. The interface between the processing nodes and switch network is provided by carefully designed network adaptors. The network adaptors and switches form the basic building blocks of Quadrics. They are named as Elan4 and Elite4, respectively. The most prominent features of Quadrics network include its ultra-low latency (0.9μs over Opteron cluster), highly scalable hardware broadcast, programmable thread processor, as well as an integral memory management unit (MMU) along with a TLB inside the Network Interface Card (NIC).

Quadrics interprocess communication is supported by two deferent models: Queue-based model (QDMA) and Remote Directed Message Access (RDMA) model. QDMA allows processes to post messages (up to 2KB) to a remote queue of other processes. Quadrics provides a communication architecture with several programming libraries. As shown in Figure 1.4, these include SHMEM, libelan (with Tport), libelan3 (Libelan4 for QsNet-II) and a kernel communication library. It also extends the underlying library with tagged message passing

(called Tagged Message Ports or Tport) and collective communication support. Tport interface is very similar to MPI [37]. The library `libelan` also provides a very useful *chained event* mechanism which allows one operation to be triggered upon the completion of another.

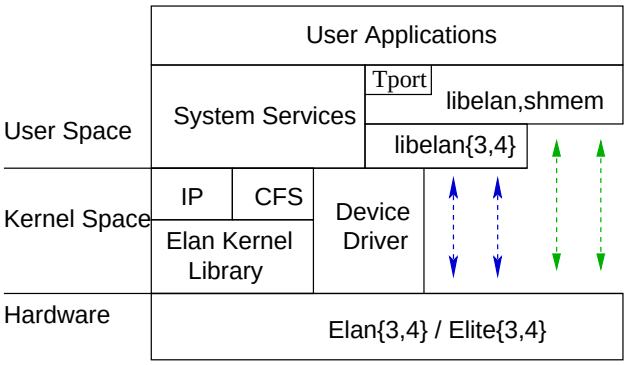


Figure 1.4: Quadrics Communication Architecture

1.2.3. Myrinet

Myrinet uses wormhole-routed crossbar switches to connect computing nodes provided by Myricom. As shown in Figure 1.5, a Myrinet network interface consists of four major components: a host DMA engine (hDMA) to access host memory, a send DMA engine (sDMA) to inject message into the network, a receive DMA engine (rDMA) to drain message from the network, a LANai processor and SDRAM memory (2 to 8MB). Latest cards also include a copy/CRC engine to speed up the memory copying and CRC generation (LANai XP and later), and support up to two ports (LANai XP-2). Each port supports full-duplex 2+2 Gbps link bandwidth.

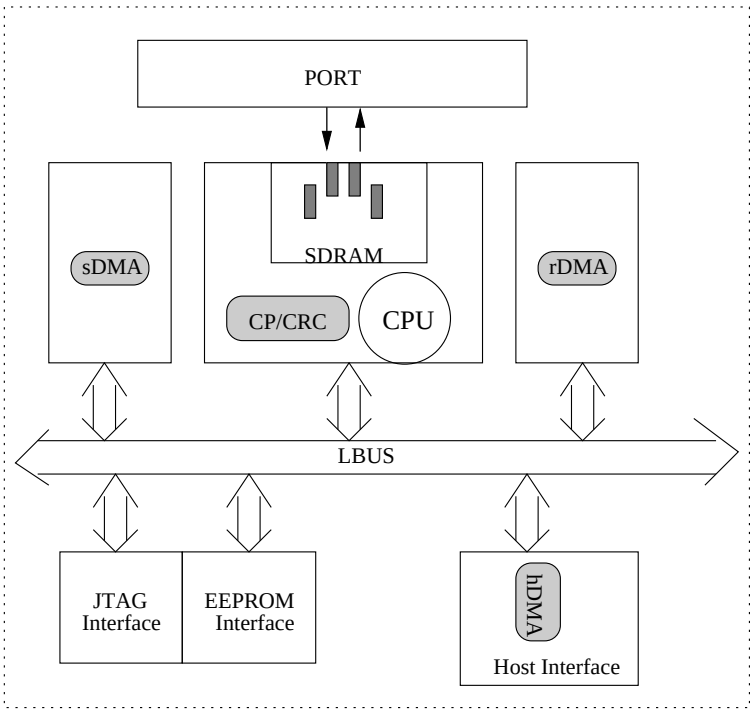


Figure 1.5: Myrinet LANai Network Interface

Two alternative low-level message-passing systems are supported over Myrinet: GM and MX. MX (Myrinet eXpress) is the new communication system for Myrinet. In this dissertation, we have used GM as our basis of study. GM is a user-level communication protocol that runs over the Myrinet [12] and provides a reliable ordered delivery of packets with low latency and high bandwidth. GM provides communications to user application between end-to-end communication points, called ports. All message passing must happen between registered memory on both sides. Besides its OS-bypass network communication, these prominent networking mechanisms are available over Myrinet: RDMA and programmable NIC processor. The programmable processor provided in the network interface lends a great degree of flexibility to develop experimental features with different design options [66, 78, 35, 8].

1.3. Network Mechanisms in Modern Cluster Interconnects

In this section, we describe some modern network mechanisms in these high speed cluster interconnects.

Remote Direct Memory Access – Remote Direct Memory Access (RDMA) is a networking mechanism that enables one process to directly access the memory from another process on a remote node, without incurring heavy load on the memory bus and host CPU processing overhead. RDMA is provided with special hardware on the NIC. It features zero-copy OS-bypass networking. Zero-copy means data is transferred directly to or from application memory. OS-bypass networking means that an application invokes RDMA operations from user space to the NIC without issuing a system call. This reduces the number of context switches compared to the traditional TCP/IP protocols. Two types of RDMA operations are available: RDMA write and RDMA read. RDMA write transfers data from the memory of the local process to the exposed destination memory at the remote process. RDMA read fetches data from the memory of a remote process to local destination memory.

Programmable Network Interface – Several modern interconnects provide programmable processors in their network interfaces [59, 73]. This increases the flexibility in the design of communication protocols. Coupled along with OS-bypass user-level communication, developers can choose to implement various portions of the software stack in the NIC. Some NIC cards also provide a significant amount of memory, from 2MB up to 128MB. For the data that needs to be immediately transmitted to the network, it is desirable to cache the data in the NIC memory for fast re-transmission [92, 69].

Hardware Atomic Operations Hardware atomic operations, such as *Read-Modify-Write*, *Compare-and-Swap* and *Test-and-Set*, can be utilized to synchronize the network nodes closely and provide stronger consistency guarantees. This is applicable to both the memory status and computation stages. Finer granularity of computation can be achieved using its potential of synchronization.

Hardware Collective Operations Some interconnects also provide hardware collective primitives in their networking hardware. For example, InfiniBand [43] and Quadrics [73] provide hardware broadcast/multicast. The emerging BlueGene/L supercomputer [38, 83] provides five different networks. Two of them are intended for scalable collective operations including broadcast, reduce and barrier. These hardware collective operations provide scalable mechanisms to synchronize and communicate data because the network switch combines/broadcasts the data from/to the destination nodes. Other collective operations can also take advantage of hardware collective operations by incorporating hardware collective support into their algorithms.

1.4. Problem Statement

Modern cluster interconnects provide new networking mechanisms in either software or hardware to support high performance communication. As described earlier, these include OS-bypass user-level communication, communication offloading, remote direct memory access (RDMA), atomic network operations, hardware collective operations, programmable network communication processor and a large amount of NIC resident memory (up to 128MB). A single interconnect may not provide all of these features. The benefits of networking mechanisms have been explored in supporting some types of communication, but not the others. For example, NIC programmability has been exploited in various ways to provide

offloaded communication processing and some collective operations (e.g., barrier, broadcast and reduce [84, 9, 17, 19, 16]), but they have not been exploited to support efficient parallel IO.

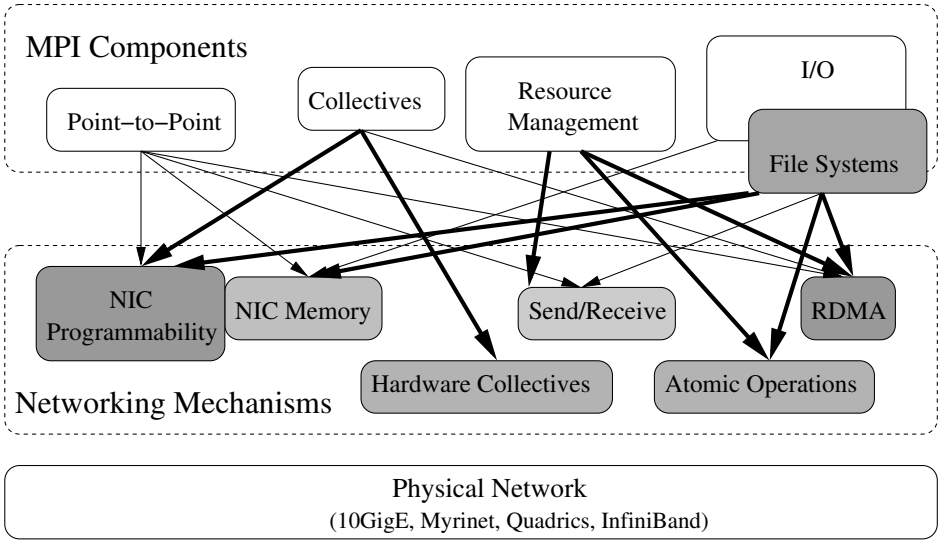


Figure 1.6: Problem Space for this Dissertation

Figure 1.6 shows the problem space for this dissertation. We have represented the intended research topics with bold arrowheads in the figure. In short, we aim to leverage new network mechanisms further to enhance MPI based parallel computing in the following three aspects: (a) resource management for scalable startup, connection management and fault tolerance process control; (b) collective communication for scalable MPI collective operations; (c) data movement for efficient and scalable parallel IO from computing cluster to storage clusters. We provide the detailed problem statements as follows:

- How to design mechanisms to support scalable startup of MPI programs over InfiniBand clusters? – The startup of a parallel program involves two steps. First, a parallel

job is launched by a process manager by creating a number of processes on a destined set of computing nodes. After that, these initiated processes require assistance from the process manager to set up peer-to-peer connections before starting parallel communication and computation. Both steps can be serious bottlenecks to job startup on very large scale platforms. It would be beneficial to analyze the startup bottlenecks and exploit the feasibility of leveraging networking mechanisms to relieve them.

- How to provide scalable connection management for MPI over ultra-scale clusters? – MPI assumes that all processes are logically connected and leaves the connection management specific issues to the lower device layer. It is important to provide a connection management policy on how to establish and maintain the communication channels among thousands of processes, especially for connection-oriented networks, such as InfiniBand and VIA. This is because an increasing number of connections can lead to prolonged startup time in creating queue pairs, exchanging connection information and establishing connections. Worse yet they can lead to heavy resource usage including the need of to maintain a large amount of connection information, which competes with the application data for physical memory, throttling parallel applications throughout their entire course of execution.

- How to provide process fault tolerance support to ultra-scale clusters? – Failure rates increase with increasing size of clusters. The applications running over large scale systems become more error-prone, especially as the failure of any single component tends to cascade widely to other components. On the other hand, few MPI implementations are designed with the fault tolerant support. Faults occurred during the execution time often abort the program and the program has to start from beginning.

For long running programs, this can waste a large amount of computing resources as all the computation done before the failure is lost. Thus it is desirable to provide fault tolerance for long running applications.

- How to support scalable and reliable end-to-end broadcast communication? – Broadcast is one of the important collective operations, which also can be used to implement other collective operations, such as allreduce, barrier and allgather. Quadrics network provides a hardware broadcast primitive that can be used to support efficient, reliable and scalable implementation of the MPI broadcast operation. However, compared to the raw hardware broadcast performance, existing broadcast implementations over Quadrics add more than 150% overhead to the basic hardware broadcast latency. Much of that overhead can be attributed to the design of the current broadcast algorithm, which enforces a synchronization before every broadcast operation. In addition, these implementations ignore the issue of end-to-end reliable data delivery, which is solely left to the Quadrics hardware. It remains to be investigated how one can add end-to-end reliability to the hardware broadcast without incurring much performance impact.

- How to use Myrinet programmable NIC processor to design efficient collective operation? – It has been shown that providing NIC-based collective operations is effective in reducing the host processor involvement [19], avoiding round-trip PCI bus traffic [19, 94], increasing the tolerance to process skew [16] and OS effects [55]. In collective operations, such as broadcast and all-to-all broadcast, a process can send the same message to many other processes. However, because of their dense communication pattern, offloading these collective operations to the NIC may impose a greater demand

on the NIC's limited memory and computation resources compared to other collective operations. It would be beneficial to explore the feasibility of designing efficient algorithms for collective operations, which can complement the research on existing NIC-based collective operations.

- How to enable high performance parallel IO over Quadrics Clusters? – The low-overhead high-bandwidth user-level communication provided by VI [99], Myrinet [64], and InfiniBand [90] has been utilized to parallelize I/O accesses to storage servers and increase the performance of parallel file systems. Some of these modern features, like RDMA, are exploited in other interconnects, e.g., Myrinet [64] and InfiniBand [90]. However, one of the leading interconnect technologies that supports many of the cutting-edge communication features, and provides great performance advantages, Quadrics Interconnects [73, 7], has not been leveraged to support scalable parallel IO at the user-level. It would be beneficial to provide this over Quadrics user-level communication to enable scalable parallel IO over Quadrics clusters.

1.5. Research Approaches

In this section, we present our general approaches in leveraging modern networking mechanisms to tackle the aforementioned issues.

1. **Designing scalable algorithms for efficient connection setup and adaptive connection management** – We have designed a scalable startup algorithm by pipelining the communication needed for exchanging connection information. Native InfiniBand user-level protocol is used instead of going over TCP/IP -based data transfer. We have also designed adaptive MPI connection management, utilizing the scalable

unreliable datagram (UD) transport services and InfiniBand Connection Management (IBCM) interface.

2. **Designing scalable collective operations with hardware broadcast and programmable NIC processors** – Hardware broadcast operations are designed to provide scalable collective communication primitives. We have designed a communication substrate to bridge the gap between the requirements of an end-to-end MPI implementation and the capability of Quadrics broadcast. This resulting implementation provides a scalable end-to-end reliable broadcast operation in LA-MPI. On the other hand, offloading the collective processing for collective operations can avoid much host CPU involvement and avoid round-trip IO bus transactions. However, it will add additional load to the less powerful NIC processors. We have exploited the capability of Myrinet programmable NIC processors to provide scalable collective operations by carefully designing NIC protocols for efficient data movement without adding too much load into the NIC.

3. **Designing efficient data movement for parallel IO with RDMA and chained event mechanisms** – We have leveraged the user-level protocol and RDMA capability Quadrics to provide scalable and high performance parallel IO. We focus on several issues for the utilization of Quadrics including: (a) constructing a client-server model over Quadrics at the user-level, (b) mapping an efficient PVFS2 transport layer over Quadrics, and (c) optimizing the performance of PVFS2 over Quadrics such as efficient non-contiguous communication support. An efficient zero-copy non-contiguous IO algorithm is also designed by chaining multiple RDMA operations with Quadrics chained-event mechanism.

1.6. Dissertation Overview

Our research studies can be divided into three different aspects. In Chapters 2, 3 and 4, we focus on scalable and fault tolerance MPI resource management issues. New algorithms for enhancing collective operations over Quadrics and Myrinet clusters are presented in Chapters 5 and 6, respectively. Chapter 7 covers our studies on enhancing MPI-IO performance.

In Chapter 2, we present our studies on scalable startup for InfiniBand clusters. We first characterize the startup of MPI programs in InfiniBand clusters and identify two startup scalability issues: serialized process initiation in the initiation phase and high communication overhead in the connection setup phase. To reduce the connection setup time, we have developed one approach with data reassembly to reduce data volume, and another with a bootstrap channel to parallelize the communication. In Chapter 3, we present adaptive connection management in handling the scalability problems faced by a single process in maintaining a large number of communication channels with peer processes. We have investigated two different adaptive connection management (ACM) algorithms: an on-demand algorithm that starts with no InfiniBand RC connections; and a partial static algorithm with only $2 * logN$ number of InfiniBand RC connections initially. We have designed and implemented both ACM algorithms in MVAPICH [60] to study their benefits. In Chapter 4, we present our design of a low-overhead, application-transparent checkpoint/restart framework. It uses a coordinated framework to save the current state of the whole MPI job to reliable storage, which allows users to perform rollback recovery if the application ever falls into some faulty state.

In Chapter 5, we present our studies in enhancing MPI collective communication with Quadrics hardware broadcast. A design of an end-to-end reliable broadcast using Quadrics hardware broadcast is provided. Experimental results have shown that it can achieve low

latency and high scalability while providing network-level fault tolerance. In Chapter 6, we present algorithms for scalable, high-performance NIC-based collective operations over Myrinet including barrier, broadcast and all-to-all broadcast. In designing these algorithms, we have come up with strategies for scalable, binomial tree-based group topology, efficient global buffer management, efficient communication processing with fast forwarding of packets, as well as reliability. Our results indicate that NIC-based collective operations can be designed with scalability, high performance and low CPU utilization. They are also shown to be beneficial for parallel processes to tolerate process skew.

In Chapter 7, we present our design and implementation of a Quadrics-capable version of a parallel file system (PVFS2) by overcoming Quadrics static communication model and providing an efficient transport layer over Quadrics. We have also designed an algorithm that supports zero-copy noncontiguous PVFS2 IO using a software scatter/gather mechanism over Quadrics. Their performance benefits have been investigated by comparing to the implementations of PVFS2 over other interconnects including Myrinet and InfiniBand.

Finally, in Chapter 8, we conclude the dissertation and present directions for future research.

CHAPTER 2

Scalable MPI Startup over InfiniBand Clusters

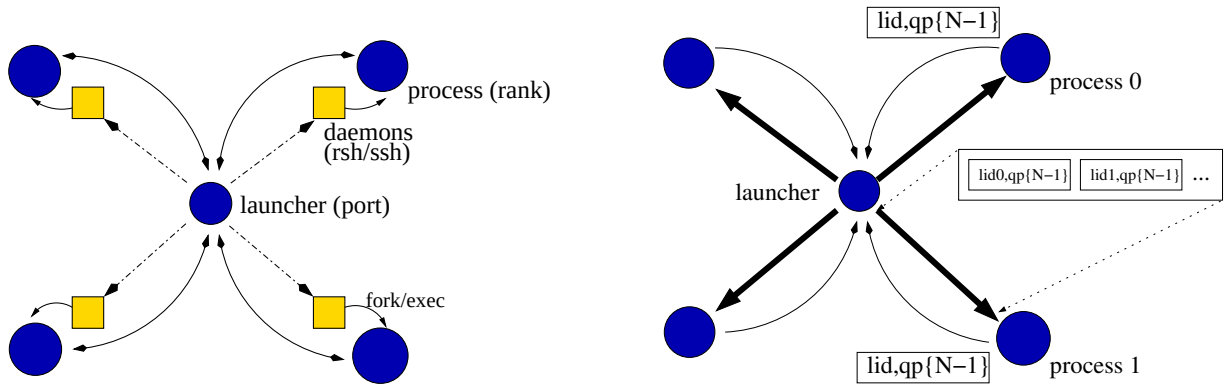
A parallel job is usually launched by a process manager, which is often referred to as the *process initiation phase*. These initiated processes usually require assistance from the process manager to set up peer-to-peer connections before starting communication and computation. This is referred to as the *connection setup phase*. We have taken on the challenge to support a scalable and high performance startup of MPI programs over InfiniBand clusters. With MVAPICH [60] as the platform of study, we have analyzed the startup bottlenecks. Accordingly, different approaches have been developed to speed up the connection setup phase, one with data reassembly at the process manager and another using pipelined all-to-all broadcast over a ring of InfiniBand queue pairs (referred to as a bootstrap channel). In addition, we have exploited a process management framework, Multi-Purpose Daemons (MPD) system to further speed up the startup. The bootstrap channel is also utilized to reduce the impact of TCP/IP-based communication including multiple process context switches and quadratically increased data volume over the MPD management ring. Over 128 processes, our work improves the startup time by more than 4 times. Scalability models derived from these results suggest that the improvement can be more than two orders of magnitude for the startup of 2048-process jobs.

The rest of this chapter is structured as follows. We first describe the challenge of scalable startup faced by parallel programs over InfiniBand and related work on process management. Then we describe the design of startup with different approaches to improve the connection setup time and the process initiation phase. Finally, we provide experiments and summarize this work.

2.1. Startup of MPI Applications using MVAPICH

MVAPICH [60] is a high performance implementation of MPI over InfiniBand. Its design is based on MPICH [37] and MVICH [49]. The implementation of MVAPICH utilized the Reliable Connection (RC) service for the communication between processes. The connection-oriented nature of IBA RC-based QPs requires each process to create at least one QP for every peer process. To form a fully connected network of N processes, a parallel application needs to create and connect at least $N \times (N-1)$ QPs during the initialization time. Note that it is possible to have these QPs be allocated and connected in an on-demand manner [89], which we have explored as a followup study and presented in Chapter 3. Another reason for the fully-connected connection model is its simplicity and robustness.

The startup of an MPI application using MVAPICH-0.9.1 can also be divided into two phases. As shown in Figure 2.1(a), an MPI application using MVAPICH is launched with a simple process launcher iterating over UNIX remote shell (rsh) or secure shell (ssh) to start individual processes. Each process connects back to the launcher via an exposed port. Except the rank of the process, each process has no global knowledge about the parallel program. In the second phase of connection setup, as shown in Figure 2.1(b), each process creates $N - 1$ QPs, one for each peer process, for an N-process application. Then, these processes exchange their local identifiers (LIDs) and corresponding QP identifiers (QP-IDs).



(a) The Initiation Phase

(b) The Connection Setup Phase

Figure 2.1: The Startup of MPI Applications in MVAPICH-0.9.1

Since each process is not connected to its peer processes, the data exchange has to rely on the connections that are created to the launcher in the first phase. The launcher collects data about LIDs and QP-IDs from each process, and then sends the combined data back to each process. Each process in turn sets up connections over InfiniBand with the received data. A parallel application with fully connected processes is then created.

2.2. The Scalability Problem

The startup paradigm described above is able to handle the startup of small scale parallel applications. However, as the size of an InfiniBand cluster goes to 100s–1000s, the limitation of this paradigm becomes pronounced. For example, launching a parallel application with 2000 processes may take tens of minutes. There are two main scalability bottlenecks, one in each phase. The first bottleneck is *rsh/ssh-based startup* in the process initiation phase. This process startup mechanism is simple and straightforward, but its performance is very poor on large systems. The second bottleneck is the communication overhead for exchanging

LIDs and QP-IDs in the connection setup phase. To launch an N-process MPI application, the launcher has to receive data containing $(N-1)$ QP-IDs from each process. Then it returns the combined data with $N \times (N-1)$ QP-IDs to each process. In total, the launcher has to communicate data in the amount of $O(N^3)$ for an N-process application. Each QP-ID is usually a four-byte integer, for a 1024-process application the launcher will receive about 4 MegaBytes data from and sends about 4 Gigabytes of data to every process. This communication typically goes through the management network which is normally Fast Ethernet or Gigabit Ethernet. This incurs significant communication overhead and can slow down the application startup.

2.3. Related Work on Job Startup and Process Management

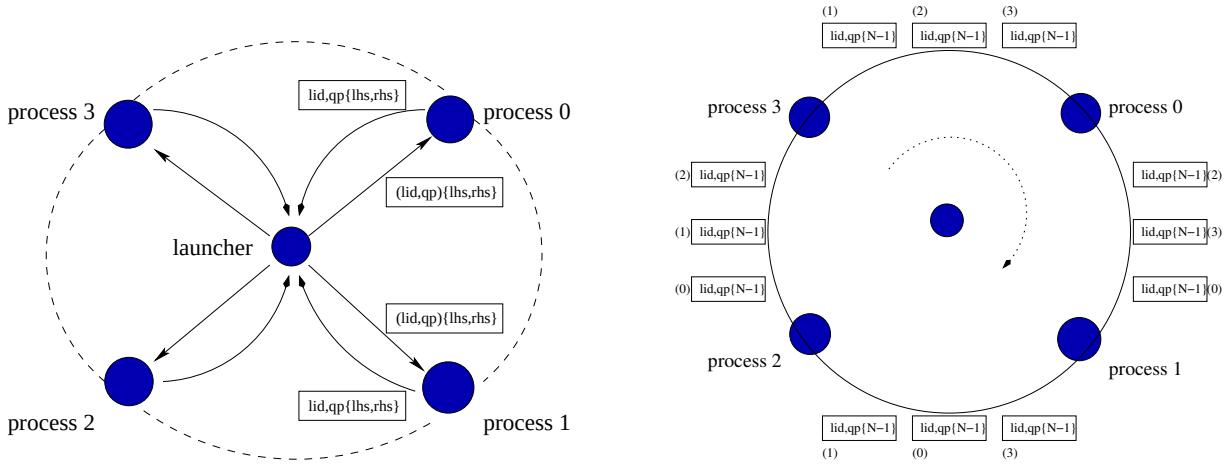
Numerous work have been done to provide resource management framework for collections of parallel processes, ranging from basic iterative rsh/ssh-based process launch in MVICH [49] to more sophisticated packages like MPD [23], Cplant [15], PBS [63], to name a few. Compared to the rsh/ssh-based iterative launch of processes, all these packages can provide more scalable startup and retain better monitoring and control of parallel programs. However, they typically lack efficient support for complete exchange of LIDs and QP-IDs as required by parallel programs over InfiniBand clusters. In this dissertation, we focus on providing an efficient support for the complete exchange of LIDs and QP-IDs, and applying such a scheme to one of these package, MPD, in order to obtain efficient process initiation support. We choose to study MPD [23] because it is one of the systems widely distributed along with MPICH [37] releases and has a large user base.

2.4. Efficient Connection Setup

As mentioned earlier, because the launcher has to collect, combine and broadcast QP IDs, the volume of these data scales up in the order of $O(N^3)$, which leads to prolonged connection setup time. One can consider two directions in order to reduce the connection setup time. The first direction is to reduce the volume of data to be communicated. The other direction is to parallelize communication for the exchange of QP IDs.

Approach 1: Reducing the Data Volume with Data Reassembly (DR) – To have processes fully connected over InfiniBand, each process needs to connect with another peer process via one QP. This means that each process needs to obtain $N-1$ QP IDs, one for each peer. That is to say, out of the combined data of $N \times (N-1)$ QP IDs in the launcher, each process only needs to receive $N-1$ QP IDs that is specifically targeted for itself. This requires a centralized component, i.e., the launcher, to collect and reassembly QP IDs. The biggest advantage of this data reassembly (DR) scheme is that the data volume exchanged can be reduced down to an order of $O(N^2)$. But there are several disadvantages associated with this scheme. First, the entire set of QP IDs need to be reassembled before sending them to each client process. This constitutes another performance/scalability bottleneck at the launcher. Second, the whole procedure of receive-reassembly-send is also serialized at the launcher.

Approach 2: Parallelizing Communication with a Bootstrap Channel (BC) – More insights can be gained on the possible parallelism with further examination of the startup. Essentially, what needs to be achieved at the startup time is an all-to-all personalized exchange of QP IDs, i.e., each process receives the specific QP IDs from other processes. In the original startup scheme as shown in Figure 2.1, the launcher performs a gather/broadcast to help the all-to-all broadcast of their QP data. On top of that, the DR



(a) Setup a Bootstrap Channel

(b) Ring-Based All-to-all Broadcast of QP Data

Figure 2.2: Parallelizing the Total Exchange of InfiniBand Queue Pair Data

scheme in Section 2.4 reassembles and "personalizes" QP data to reduce the data volume. Both do not exploit the parallelism of all-to-all personalized exchange. Algorithms that parallelize an all-to-all personalized exchange can be used here. These algorithms are usually based on a ring-, hypercube- or torus-based topology, which requires more connections to be provided among processes. With the initial star topology in the original startup scheme, providing these connections has to be done through the launcher. However, since a parallel algorithm can potentially overlap both sending and receiving QP data, it promises better scalability over clusters with larger sizes.

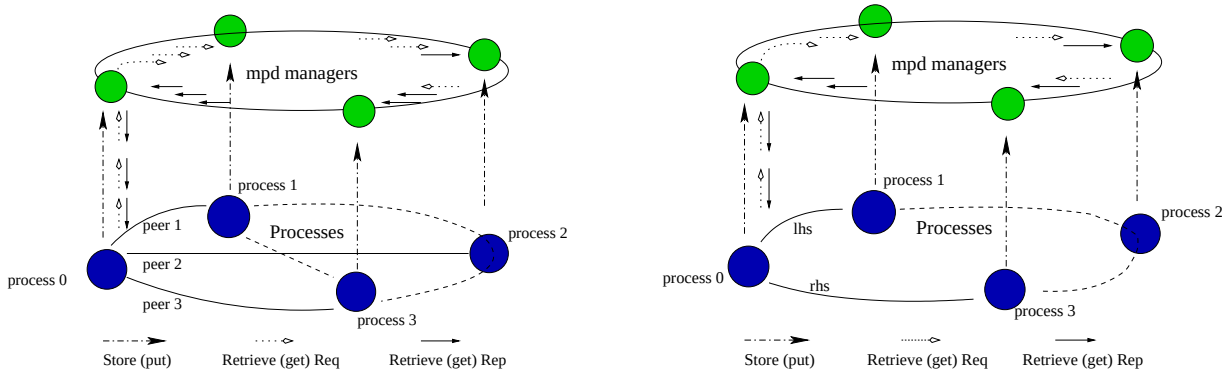
Among the three possible parallel topologies, the ring-based topology requires the least number of additional connections, i.e., 2 per process. This would minimize the impact of the ring setup time. Another design option is that which type of connections is possible. Either TCP/IP- or InfiniBand-based connections can be used. Since the communication over

InfiniBand is much faster than that over TCP/IP, we choose to use a ring of InfiniBand QPs as a further boost to the parallelized data exchange.

The second approach works as follows. First, each process creates two QPs for its left hand side (lhs) and right hand side (rhs) processes, respectively. We call these QPs *bootstrap QPs*. Second, the DR scheme mentioned in Section 2.4 is used to set up connections between these bootstrap QPs as shown in Figure 2.2(a). Thus, a ring of connections over InfiniBand is created, as shown by the dotted line in Figure 2.2(a). We refer to this ring as a *bootstrap channel* (BC). After this channel is set up, each process initiates a broadcast of its own QP IDs through the channel in the clockwise direction as shown in Figure 2.2(b) with four processes. Each process also forwards what it receives to its next process. In this scheme, we take advantage of both communication parallelism and high performance of InfiniBand QPs to reduce the communication overhead.

2.5. Fast Process Initiation with MPD

MPD [23] is designed to be a general process manager interface that provides the needed support for MPICH, from which MVAPICH is developed. It mainly provides fast startup of parallel applications and process control to the parallel jobs. MPD achieves its scalable startup by instantly spreading a job launch request across its ring of daemons, then launches one ring of manager and another ring of application processes in a parallel fashion. For processes to exchange individual information MPD system also exposes a BNR interface with a put/fence/get model. A process stores a {key,value} pair at its manager process, a part of the MPD database, then another process retrieves (gets) that value by providing the same key after a synchronization phase (fence).



(a) Exchange of Queue Pair IDs Over the Ring of Manager

(b) Setting up Bootstrap Channel within Processes

Figure 2.3: Improving the Scalability of MPD-Based Startup

Although this fast and parallelized process startup from MPD solves the process initiation problem, the significant volume of QP data still poses a great challenge to the MPD model. As shown in Figure 2.3(a), the database is distributed over the ring of manager processes when each process stores (puts) their process-specific data to its manager. To collect the data from every peer process, one process has to send a request and get the reply back for the target process. At the completion of these data exchanges, each process then sets up connections with all the peers, as shown with process 0 in Figure 2.3(a). Together, messages for the request and the reply make a complete round over the manager ring. For a parallel job with N processes, there are $N \times (N-1)$ message exchanges in total. Each of these messages is in the order of $O(N)$ bytes and has to go through the ring of manager processes. In addition, since application processes store and retrieve data through their corresponding manager processes at each node, process context switches are very frequent and they further

degrade the performance of ring-based communication. Furthermore, the message passing is over TCP/IP sockets, which delivers lower performance than InfiniBand-based connections.

There are different alternatives to overcome these limitations. One way of doing that is to replace the connections for the MPD manager ring with VAPI connections to provide fast communications. In addition, copies of QP data can be saved at each manager process as the first copy of QP data passes through the ring. Then further retrieve (get) requests can get the data from the local manager directly instead of the MPD manager ring. This approach will improve the communication time, however, the process context switches still exist between the application processes and manager processes. In addition, retrieve requests made before QP data reaches the local manager process still has to go through the manager ring. Last but not least, this approach necessitates a significant amount of instrumentation of MPD code and has only limited portability to InfiniBand-ready clusters.

Instead of exchanging all the QP data over the ring of MPD manager processes, we propose to exchange QP IDs over the bootstrap channel described in Section 2.4. Though setting up the bootstrap channel still needs help from the ring of manager processes. As shown in Figure 2.3(b), each process first creates and stores QP IDs for its left side (lhs) and right hand side (rhs) processes to the local manager. Then, from the database, they retrieve QP IDs for its left hand side and right hand side processes, and then set up InfiniBand connections. Eventually a ring of such connections are constructed and together form a bootstrap channel. This bootstrap channel is then utilized to perform a complete exchange of QP IDs. Since this bootstrap channel is provided within the application processes and over InfiniBand, this approach will not only provide fast communication and eliminate the process context switches, but also reduce the number of communications through each manager process.

2.6. Experimental Results of Scalable Startup

Our experiments were conducted on a 256-node cluster of 4GB DRAM dual-SMP 2.4GHz Xeon at the Ohio Supercomputing Center. For fast network discovery with data reassembly (DR) or the bootstrap channel (BC), we used ssh to launch the parallel processes. Performance comparisons were provided against MVAPICH 0.9.1 (Original). Since Networked File System (NFS) performance could be a big bottleneck in a large cluster and mask out the performance improvement of startup, all binary executable files were duplicated at local disks to eliminate its impact.

Number of Processes	4	8	16	32	64	128
Original (sec)	0.59	0.92	1.74	3.41	7.3	13.7
SSH-DR (sec)	0.58	0.94	1.69	3.37	6.77	13.45
SSH-BC (sec)	0.61	0.95	1.70	3.38	6.76	13.3
MPD-BC (sec)	0.61	0.63	0.64	0.84	1.58	3.10

Table 2.1: Comparisons of MVAPICH Startup Time with Different Approaches

Table 2.1 shows the startup time for parallel jobs of different number processes using different approaches. SSH-DR represents ssh-based startup with QP data assembly (DR) at the process launcher. SSH-BC represents ssh-based startup using the bootstrap channel (BC) to exchange QP IDs. MPD-BC represents MPD-based startup with a bootstrap channel for the exchange of QP IDs.

As the number of processes increases, both SSH-DR and SSH-BC reduce the startup time, compared to the original approach. This is because data reassembly can reduce the data volume by an order of $O(N)$ and the bootstrap channel can parallelize the communication

time. Note that the BC-based approach performs slightly worse than the the original and DR-based approach for small number of processes. This is due to the overhead from setting up the additional ring over InfiniBand. As the number of processes increases, the benefits become greater. Both SSH-BC and SSH-DR will be able to provide more scalable startup for a job with thousands of processes since they remove the major communication bottleneck imposed by potentially large volume of QP data. In contrast, the MPD-based approach with a bootstrap channel provides the most scalable startup. On one hand, MPD-BC provides efficient parallelized process initialization, compared to the ssh-based schemes. On the other hand, it also pipelines the QP data exchange over a ring of VAPI connections, hence this approach speeds up the connection setup phase. Compared to the original approach, the MPD-BC approach reduces the startup time for a 128-process job by more than 4 times.

2.7. Analytical Models and Evaluations for Large Clusters

As indicated by the results from Section 2.6, the benefits of the designed schemes will be more pronounced for parallel jobs with larger number of processes. Here we further analyze the performance of different startup schemes and provide parameterized models to gain insights about their scalability over large clusters. The total startup time $T_{startup}$ can be divided into the process initiation time and the connection setup time, denoted as T_{init} and T_{conn} respectively. Based on the scalability analysis, we use the following model to describe the startup time of the original scheme (Original), ssh-based scheme with data reassembly (SSH-DR) and the MPD-based scheme with the bootstrap channel (MPD-BC). Each of the models shows the time for the startup of N processes, and the last component describes the time for other overheads that are not quantified in the models, for example, process switching overhead.

Original: $T_{startup} = (O_0 * N) + (O_1 * N * (W_N + W_{N^2})) + O_2$

The process initiation phase time T_{init} scales linearly as the number of processes increases with ssh/rsh-based approaches, while during the connection setup there are $2N$ messages communicated over TCP/IP. Half of them are gathered by the launcher, each being in the order of $O(N)$ bytes; the other half are scattered by the launcher, each of $O(N^2)$ bytes .

SSH-DR: $T_{startup} = (D_0 * N) + (D_{comp} * N^3 + D_1 * 2N * W_N) + D_2$

The process initiation time T_{init} scales linearly with ssh/rsh. During the connection setup phase, the amount of computation scales in the order of $O(N^3)$ (the constant D_{comp} can be very small, being the time for extracting one QP Id), and there are 2*N message communicated over TCP/IP. Half of them are gathered by the launcher, each being in the order of $O(N)$ bytes; The other half are scattered by the launcher, each of them is only $O(N)$ bytes due to reassembly.

MPD-BC: $T_{startup} = (M_0 + N * W_{req}) + (M_{ch\_setup} * N + M_1 * N * W_N) + M_2$

The process initiation time T_{init} scales constantly using MPD, however there is a small fractional increase of communication time for the request message W_{req}. During the connection setup phase, the time to setup a bootstrap channel increases in the order of $O(N)$. Each process also handles N message in the pipeline, each in the order of $O(N)$ bytes.

Original: $T_{startup}$ (sec) $= (0.100 * N) + (10.5 * N * (W_N + W_{N^2})) + 0.12$

SSH-DR: $T_{startup}$ (sec) $= (0.100 * N) + (8.5e^{-9} * N^3 + 10.5 * N * W_N) + 0.12$

MPD-BC: $T_{startup}$ (sec) $= (0.20 + 0.0010 * N) + (0.0180 * N + 2.5 * N * W_N) + 0.30$

The above scalability models are parameterized based on our analytical modeling. As shown in Figure 2.4, the experiment results confirm the validity of these models for jobs with 4 to 128 processes. Figure 2.5 shows the scalability of different startup schemes when applying the same models to larger jobs from 4 to 2048 processes. Both SSH-DR and MPD-BC improves the scalability of job startup significantly. Note that MPD-BC scheme improves the startup time by about two orders of magnitudes for 2048-process jobs.

2.8. Scalable Startup Summary

We have presented our design for scalable startup of MPI programs in InfiniBand clusters. With MVAPICH as the platform of study, we have characterized the startup of MPI jobs into

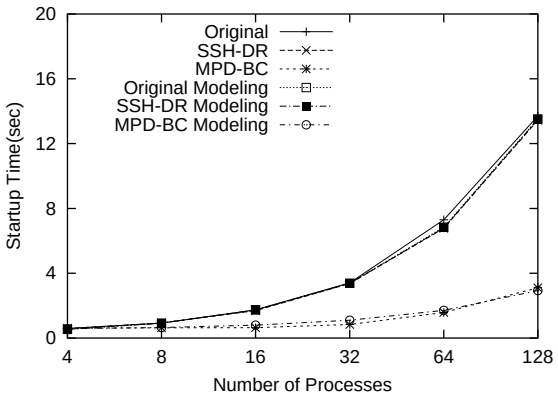


Figure 2.4: Performance Modeling of Different Startup Schemes

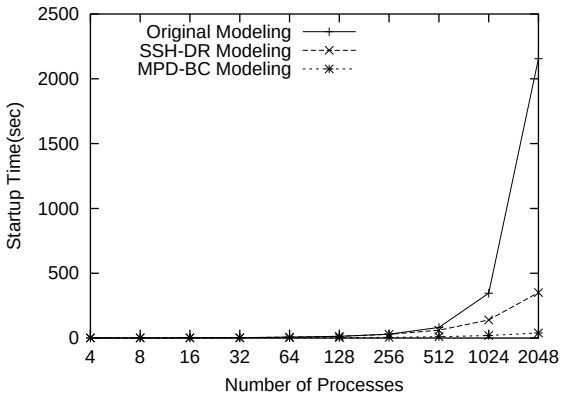


Figure 2.5: Scalability Comparisons of Different Startup Schemes

two phases: process initiation and connection setup. To speed up connection setup phase, we have developed two approaches, one with queue pair data reassembly at the launcher and the other with a bootstrap channel. In addition, we have exploited a process management framework, Multi-Purpose Daemons (MPD) system, to improve the process initiation phase. The performance limitations in the MPD's ring-based data exchange model, such as exponentially increased communication time and numerous process context switches, are eliminated by using the proposed bootstrap channel. We have implemented these schemes in MVAPICH [60] and the solutions are available from MVAPICH 0.9.2 onward. Our experimental results show that, for 128-process jobs, the startup time has been reduced by more than 4 times. We have also developed an analytical model to project the startup scalability. The extrapolated results suggest that the improvement can be more than two orders of magnitudes for the startup of 2048-process jobs with our startup scheme using ring-based parallelization.

CHAPTER 3

Adapative Connection Management for Scalable MPI over InfiniBand

MPI does not specify any connection model, but assumes that all processes are logically connected and leaves the connection management specific issues to the lower device layer. For connection-less interconnects, such as Quadrics [73] and Myrinet [58], an MPI process can start MPI communication without extra mechanisms to manage peer-to-peer connections. On top of InfiniBand, however, for any pair of processes to communicate over RC for its high performance and RDMA capability, a pair of RC queue pairs (QPs) must be created on each node with a connection established between them. To enable high performance RDMA fast path for small messages, additional RDMA receive buffers also need to be provided for each connection.

Several of the most commonly used MPI implementations over InfiniBand, such as MVAPICH [60] (up to 0.9.7 version), set up RC connections between every process pairs *a priori*. Because of its connection-oriented nature, every process needs to allocate a dedicated QP for each peer process. This leads to quadratically increased number of RC connections, e.g., 1024*1023 connections for a 1024-process MPI program. These number of connections in turn lead to prolonged startup time for the need of creating queue pairs, exchanging connection information and establishing connections. This also leads to heavy resource usage,

taking into account of the memory needed for all the QPs and their associated send and receive WQEs, as well as RDMA send and receive buffers. To make the matters worse, each QP also needs some number of send/receive buffers for taking advantage of fast RDMA for small messages.

Application	Number of Processes	Average Number of Distinct Destinations
sPPM	64	5.5
	1024	< 6
SMG2000	64	41.88
	1024	< 1023
Sphot	64	0.98
	1024	1
Sweep3D	64	3.5
	1024	< 4
Samrai 4	64	4.94
	1024	< 10
CG	64	6.36
	1024	< 11

Table 3.1: Average Number of Communicating Peers per Process in some applications (Courtesy of J. Vetter, et. al [85])

In fact, research on communication characteristics of parallel programs [85] indicates that not all pairs of MPI processes communicate among each other with equal frequency. Table 3.1 shows the average number of communicating peers per process in some scientific applications. The majority of process pairs do not communicate between each other. Thus, maintaining a fully-connected network not only leads to the aforementioned scalability problems, but also negatively affects the performance of the communicating processes. This is because the MPI program has to continuously check for potential messages coming from any process,

which drives the CPU away from attending to traffic of the most frequently communicated processes and destroys the memory cache locality that could be achieved thereof. There have been discussions in the IBA community to either provide only UD-based communication, or RC on-demand via an out-of-band asynchronous message channel. However, UD does not provide comparable performance as RC; the processing of out-of-band asynchronous messages will introduce the need of another thread that contends for CPU with the main thread. So these solutions can result in performance degradation. It remains to be systematically investigated what connection management algorithms can be provided for parallel programs over InfiniBand, and what are their performance and scalability implications.

In this dissertation, we take on the challenge of providing appropriate connection management for parallel programs over InfiniBand clusters. We propose adaptive connection management (ACM) to manage different types of InfiniBand transport services. The time to establish and tear town RC connections is dynamically decided based on communication statistics between the pair of processes. New RC connections are established through either an unreliable datagram-based mechanism or an InfiniBand connection Management-based mechanism. We have also studied strategies to overcome challenging issues, such as race conditions, message ordering and reliability, for the establishment new RC connections. The resulting ACM algorithms have been implemented in MVAPICH [60] to support parallel programs over InfiniBand. Our experimental data with NAS application benchmarks indicate that ACM can significantly reduce the average number of connections per process, and it is also beneficial in improving the process initiation time and reducing memory resource usage. Note, one of the side effects of ACM is that it moves connection establishment from the process initiation stage into the actual critical path of parallel communication. Our evaluation also indicates that ACM has little performance impact to microbenchmarks and

NAS scientific applications since only very few connections are established on the basis of frequent communication.

3.1. Related Work on Connection Management

There had been previous research efforts carried out to study the impact of connection management on the performance of parallel applications. Brightwell et. al. [14] analyzed the scalability limitations of VIA in supporting the CPlant runtime system as well as any high performance implementation of MPI. While not taking into account the impacts of the number of connections on the scalable usage of computation and memory resources to different connections, the authors argued that *on-demand connection management* could not be a good approach to increase the scalability of the MPI implementation by qualitative analysis. Wu et. al. [89] demonstrated that on-demand connection management for MPI implementations over VIA [29] could achieve comparable performance as the *static mechanism* with efficient design and implementation. Our work continues the research efforts of on-demand connection management [89] and scalable startup [97] to improve the scalability of MPI implementations over InfiniBand.

3.2. InfiniBand Connection Management

To support RC, a connection must be set up between two QPs before any communication. In the current InfiniBand SDK, each QP has a unique identifier, called *QP-ID*. This is usually an integer. For network identification, each HCA also has a unique a local identifier (*LID*). One way to establish a connection is to exchange the QP IDs and LIDs of a pair of QPs and then explicitly program the queue pair state transitions. Another way is to use InfiniBand connection management interface. In the IBA community, a new interface called RDMA

CMA (Connection Management Agent) has been proposed recently [62]. RDMA CMA over IBA is derived on top of IBCM, but provides an easy-to-use, portable, yet similar approach for connection establishment. It is currently available only in the OpenIB Gen2 stack [62]. We plan to study the benefits of RDMA CMA in our future work.

InfiniBand Communication Management (IBCM) encompasses the protocols and mechanisms used to establish, maintain, and release different InfiniBand transport services, such as RC, UC, and RD. Communication Managers (CMs) inside IBCM set up QPs (or end-to-end context for RD) upon calls to the IBCM interface. CMs communicate with each other and resolve the path to remote QPs through an address resolution protocol. There are two models to establish a connection: one is Active/Passive (also referred as client/server) model, the other Active/Active (or peer-to-peer) model. In the client/server model, the server side listens for connection requests with a service id; the client side initiates a connection request with a matching service ID. In the peer-to-peer model, both sides actively send connection requests to each other, and a connection is established if both requests contain matching service IDs. Compared to the peer-to-peer model, the client/server model is more mature in the current InfiniBand implementations, and is what we have studied in this dissertation.

Figure 3.1 shows the diagram of the client/server model of IBCM. The server begins to listen on a service ID. A client then creates a QP and initiates a request (REQ) to the server with a matching service ID. If the server can match the service ID, a callback function is called to create a QP, accept the client's request, and confirm the request with a reply (REP). When the client receives the server's reply indicating that its request is accepted, a client-side callback handler is called, within which the client confirms the establishment of a new connection back to the server via a RTU (ready-to-use) message. When the server receives RTU, the connection is then ready for communication. In the client/server model,

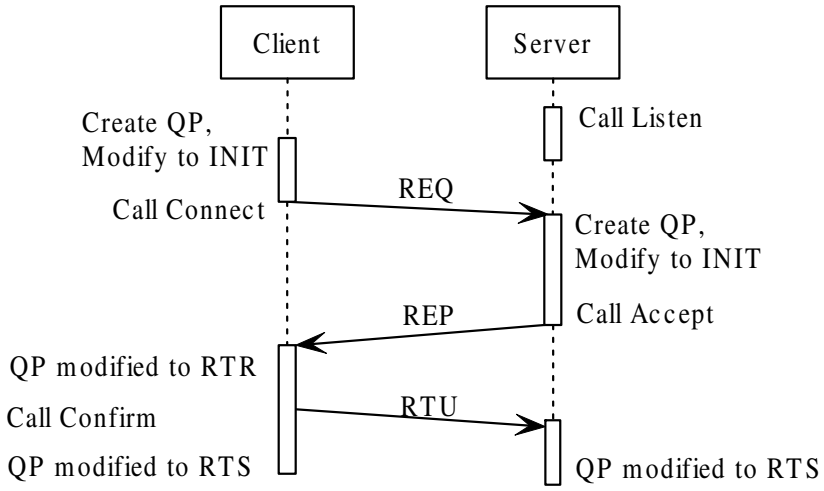


Figure 3.1: The Client/Server Model of IBCM

QPs are created in an INIT state and progressed through RTR (Ready-to-Receive) to RTS (Ready-to-Send) by CMs as shown in the figure.

3.3. Designing Adaptive Connection Management

Since different interconnects may have different software and hardware capabilities and exhibit different communication characteristics, the design of MPI [52] in general leaves interconnect specific issues to ADI (abstract device interface) implementations. For example, MPI does not specify any connection model, but assumes that all processes are logically connected. This does not lead to complications for connection-less interconnects, such as Quadrics [73] and Myrinet [58], on which MPI process can start MPI communication without extra care for managing peer-to-peer connections. InfiniBand [43], however, comes with a plethora of transport services, from the typical connection-less Unreliable Datagram (UD) to the high-performance connection-oriented Reliable Connection (RC). Different types of

transport services come with different performance qualities and different resource requirements. Within a parallel application spanning thousands of parallel processes, a process potentially needs to handle such resource requirements for a large number of connections depending on the number of peers it is communicating with. On top of this, a scalable MPI implementation also needs to satisfy the memory requirements from parallel applications. These complexities all need to be handled within rigid resource constraints. Therefore, when and how to enable what types of connections is a very important design issue for scalable and high performance MPI implementations.

To this purpose, we propose Adaptive Connection Management (ACM) to handle these complexities. There are two two different ACM algorithms: on-demand and partially static. In the on-demand algorithm, every process is launched without any RC connections; in the partially static algorithm, each process is initially launched with at most $2 * logN$ of RC connections to communicate with peers that have a rank distance of 2^N from it. These initial $2*logN$ RC connections are meant to capture the frequent communication patterns based on the common binary tree algorithms used in many MPI collective operations. The main design objective of ACM is to manage InfiniBand transport services in an adaptive manner according to the communication frequency and resource constraints of communicating processes. To this purpose, new RC connections are established only when a pair of processes have exhibited a frequent communication pattern. In this dissertation, this is decided when two processes have communicated more than 16 (an adjustable threshold) messages. Compared to the commonly used static connection management, ACM algorithms are designed to allow the best adaptivity, while the partially static algorithm also allows applications to pre-configure common communicating processes with RC connections at the startup time.

Figure 3.2 shows a diagram about the intended ACM functionalities in a typical MPI software stack. As shown in the figure, ACM works in parallel with the ADI's channel interface (CH2), which is in charge of the actual message communication functionalities. While the channel interface mainly provides appropriate channels to transport messages based on their sizes and destinations, ACM controls when to activate the transport services of different performance capabilities and maintains the communication statistics of these channels. Specifically, ACM manages the following information about transport services over InfiniBand.

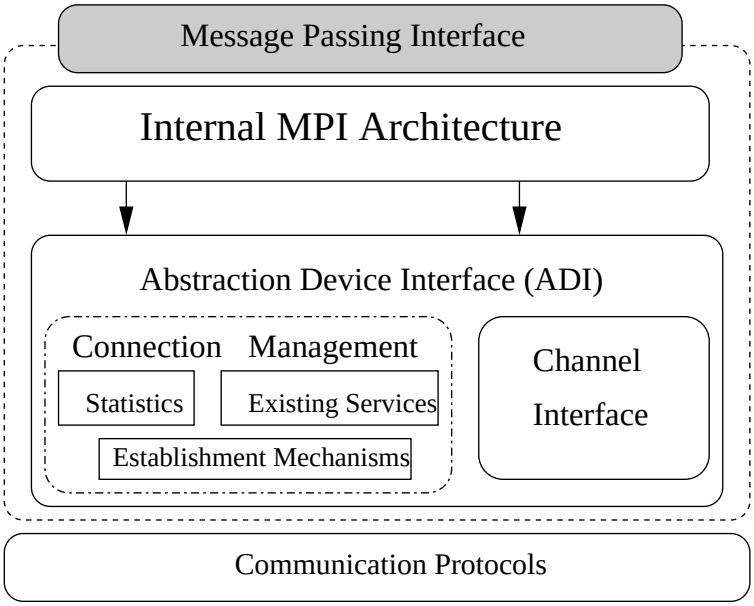


Figure 3.2: Adaptive Connection Management in MPI Software Stack

- **Existing transport services** – In ACM, each process maintains information about which transport services are available to reach peer processes. The most commonly

used InfiniBand transport services are UD and RC. All processes start with a UD queue pair, through which other processes can request the establishment of RC connections.

- **Resource allocation and communication statistics** – Messages can be transmitted over the existing transport services. The number of messages and the total message size are recorded as communication statistics, which determine when to set up new RC connections. Future work can also introduce mechanisms to dismantle RC connections when some processes are either quiescent or relatively less active for a certain amount of time.

- **Establishment mechanisms for new transport services** – Currently, we have exploited two different mechanisms for establishing new RC connections over InfiniBand: (1) Connection establishment via UD-based QP-ID exchange; and (2) IBCM-based RC connection establishment. In future, we plan to study the benefits of RDMA CMA for MPI connection management over the InfiniBand Gen2 stack [62].

On-demand and partially static ACM algorithms can use either connection establishment mechanisms to set up new RC connections. In this work, we intend to study combinations of ACM algorithms and connection establishment mechanisms to gain insights into the following questions:

1. How much can the adaptive connection management help on reducing the number of the connections for scientific applications?

2. What performance impact will the adaptive connection management have?

3. How much can the adaptive connection management help on reducing the process initiation time?

4. What benefits will the adaptive connection management have on the memory resource usage?

3.4. UD-Based Connection Establishment

Figure 3.3 shows the diagram of UD-based connection establishment. Upon frequent communication between Proc A and Proc B, Proc A sends a request for new connection to Proc B. Proc B responds with a reply to acknowledge the request. A new connection is established at the end of a three-way exchange of *request*, *reply* and *confirm* messages. The actual scenario is more complicated than what is shown in the diagram. We describe the detailed design issues as follows:

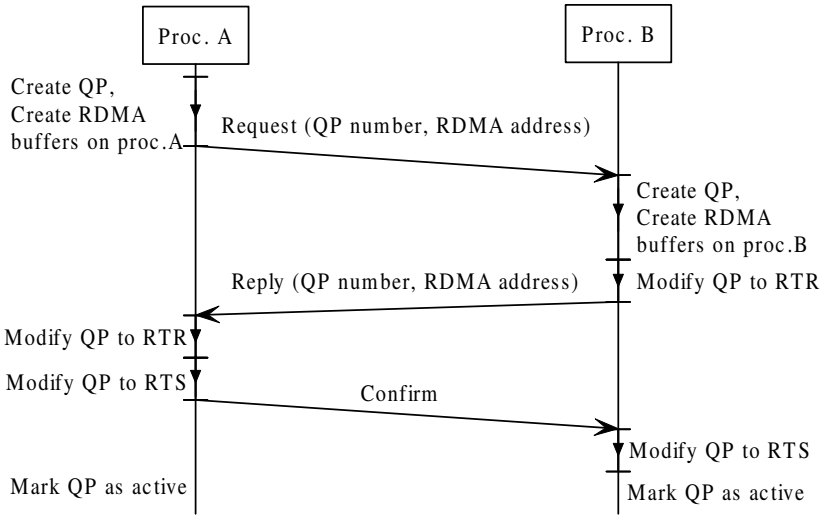


Figure 3.3: UD-Based Connection Establishment

- **Progress rules** – When using UD-based connection management, an MPI process has to handle extra UD messages for connection setup purposes. To provide a clean

and light-weight solution, we provide a dedicated completion queue for UD messages. However, this completion queue is being polled at a much reduced frequency compared to the completion queue for regular data messages. This is intended to reduce the extra burden on the main MPI process for attending to extra traffic.

- **Ordered reliable message delivery** – Any of three messages may get lost since they are sent over UD. We introduce timeout-based retransmission to handle such situations. However, this may also lead to duplicated requests. For this problem, a sequence number is introduced along with UD messages to avoid redundant requests.

- **Race conditions** – Race conditions between two requests can occur for the establishment of the same RC connection since Proc B may have sent out a connection request to Proc A at the same. Both Proc A and B are trying to set up a RC connection between them. To guard against race conditions, a process responds to a request with a positive acknowledgment only when it has not initiated a request or its rank is higher than the source rank contained in the request. Status flags are introduced to reflect the progression of connection state during the connection establishment.

3.5. IBCM-Based Connection Establishment

In the IBCM-based mechanism each process starts a new listening thread with a unique service ID, which is set to its rank plus a constant so that every process knows the service IDs of all other processes. When a process wants to establish a connection, it sends a request to the corresponding target. The procedure in general follows what we have described in Section 3.2. In particular, we describe the following design issues as follows.

Synchronization – To establish a RC connection via IBCM, some information such as source/destination ranks must be exchanged during the connection establishment phase.

And we also need to make sure that the receive descriptor be posted before the RC connection progresses to RTR. Thus, it would be more efficient if one can integrate the receive descriptor posting and RDMA receive buffers exchange into the process of IBCM connection establishment. To this purpose, we ensure that the server has prepared the receive descriptors and buffers before it replies back to the client. After the client receives the reply, we make sure that the client completes the same preparation before it confirms back to the server with a RTU (ready-to-use) message (Figure 3.1). Only until the server receives the expected RTU message and the client gets the correct local completion of RTU message, will the connection be marked as active on both sides. Both sides are then correctly synchronized on the state of the new RC connection and the connection is ready to use.

Race conditions – Each process has two possible activities for establishing new connections. One is the main MPI thread that may connect to a target process as a client, the other being the listening thread that functions as a CM server for incoming connection requests. It is critical to ensure that, at any time, one of them is driving the establishment of a new connection. Otherwise, both of them will fail for incorrect sharing of the same queue pair.

We describe our solution with two arbitrary processes, Proc A and B, the rank of A greater than B. When both of them simultaneously send a request to the other process, we let A act as a server to continue and have the CM server of B to reject the request from A. The client request from B continues to be processed by the CM server of A and finish the establishment of a new connection. In addition, to avoid an inconsistent connection state, before A accepts B's request, it also needs to wait until a reject notification from B is received. Structures related to connection progression are critical sections being protected by mutexes and they are used to avoid the race conditions between the main thread and the CM thread.

3.6. Performance Evaluation of Adaptive Connection Management

Our experiments were conducted on two clusters. One is a cluster of 8-node SuperMicro SUPER P4DL6, each with dual Intel Xeon 2.4GHz processors, 1GB DRAM, PCI-X 133MHz/64-bit bus. The other is a cluster of eight SuperMicro SUPER X5DL8-GG nodes: each with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, PCI-X 64-bit 133 MHz bus, 533MHz Front Side Bus (FSB) and a total of 2GB PC2100 DDR-SDRAM physical memory. The nodes are connected using the Mellanox InfiniScale 24 port switch MTS 2400. The original MVAPICH [60] release we used is 0.9.5 with patches up to 118. We evaluated the original static connection management of MVAPICH-0.9.5 (referred to as *Orig*) and the following combinations of ACM algorithms and connection establishment mechanisms (UD/IBCM): partially static ACM with UD (UD-PS), on-demand ACM with UD (UD-OD), and on-demand ACM with IBCM (CM-OD).

Average Number of Connections – One of the main benefits of adaptive connection management is to increase the scalability of MPI implementations in terms of scalable usage of RC connections over InfiniBand. Table 3.2 lists the average number of InfiniBand RC connections used in the NAS application benchmarks with different ACM configurations compared to the original static algorithm. With UD-OD, the number of RC connections for NAS benchmarks are in general less than the numbers reported before in [89]. This suggests that UD-OD can indeed eliminate all the connections that are either not communicating or very rarely. For example, the number of connections used in EP is 0, because no connections are established between the processes since the processes only communicate rarely with a few barrier operations for the purpose of synchronization.

Algorithm	SP	BT	MG	LU	IS	EP	CG
16 Processes							
Orig	15	15	15	15	15	15	15
UD-OD	6	6	5	3.6	15	0	2.7
UD-PS	9.5	9.5	7	7	15	7	7.8
32 Processes							
Orig	–	–	31	31	31	31	31
UD-OD	–	–	7	4.1	31	0	3.8
UD-PS	–	–	9.5	9	31	9	9.8

Table 3.2: Average Number of Connections in NAS Benchmarks

Process Initiation Time – Since adaptive connection management reduces the initial number of InfiniBand RC connections, the time needed for the establishment of these connections in the original static algorithm is no longer needed. We have investigated the benefits of our algorithms in terms of process initialization time. We first measured the initialization time using a ssh/rsh-based startup scheme and noticed that the variation in startup time is too high to obtain the portion of reduced initialization time. Instead we incorporated the ACM algorithms into a scalable MPD-based startup scheme and measured the initialization time.

Figure 7.2 shows process initialization time for 32-process programs over the 16-node cluster. Compared to the original algorithm, UD-OD can reduce process initialization time by 15-20%, while UD-PS can reduce the time by around 10%. The amount of reduction in initialization time is smaller for UD-PS because around $2 * logN$ connections need to be established at the beginning.

Reduction in Memory Usage – Another benefit of adaptive connection management is reducing the memory resource usage. Because the number of connections is tightly coupled to

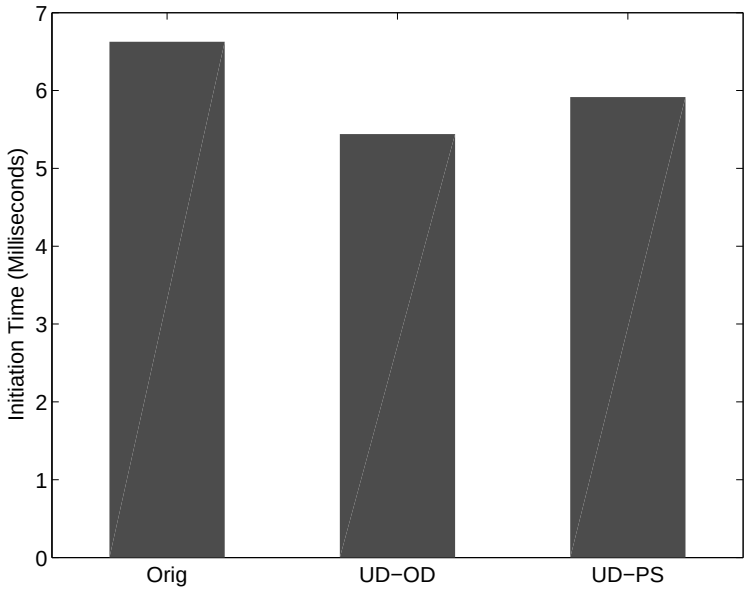


Figure 3.4: Initiation Time of Different Connection Management Algorithms

the communication behavior of parallel applications, it is not easy to decide an appropriate time to take a snapshot of memory usage. To gain insights into the memory usage, we measured the initial memory usage when the parallel processes first start up, i.e., at the end of `MPI_Init`.

Figure 3.5 shows the memory resource usage for a parallel program with varying number of processes over the 16-node cluster. Compared to the original, all ACM algorithms start with slightly higher memory usages, this is because the connection management algorithms introduced additional data structure such as a UD-queue pair and/or a CM server thread, which consumes slightly more memory. However, the original algorithm has a clearly faster increasing trend of memory usage compared to others. For a 32-process application, UD-OD can reduce memory usage by about 17%, while UD-PS can reduce the memory by about 12%. Again, because UD-PS has to set up around $2 * logN$ connections, the amount of memory

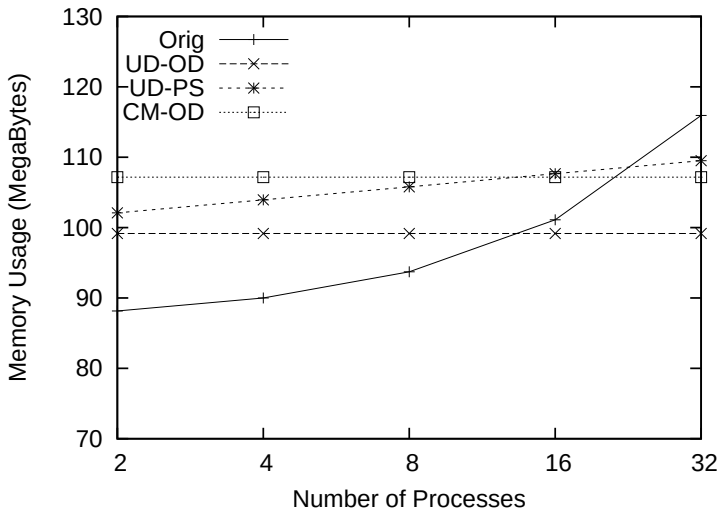


Figure 3.5: Memory Usage of Different Connection Management Algorithms

usage is higher than that of UD-OD. These results suggest ACM algorithms are beneficial in terms of memory resource usage. These benefits are expected to be more significant as system size increases.

Impact on Latency and Bandwidth – To find out the impact of ACM on the basic latency and bandwidth performance of MVAPICH, we have compared the performance of different algorithms with the original. As shown in Figures 3.6 and 3.7, UD-OD and UD-PS have negligible impacts on the latency and bandwidth performance. This suggests that our implementations are indeed light-weight and efficient. However, CM-OD causes degradation on latency and bandwidth. This is expected because the IBCM-based ACM introduces additional threads for managing connection requests and establish new connections.

Performance of NAS Parallel Benchmarks – The NAS suite consists of a set of programs, such as MG, CG, IS, LU, SP and BT. We compared the performance of these NAS programs over UD-PS and UD-OD to the original. Figures 3.8, 3.9, 3.10 and 3.11 show

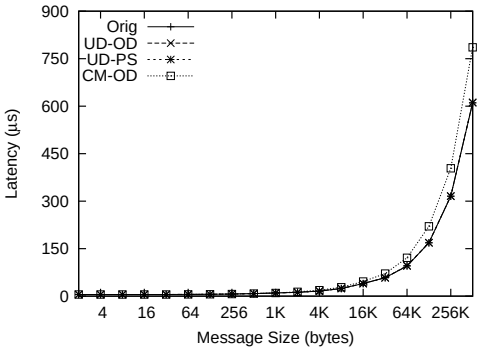


Figure 3.6: Latency of Different Connection Management Algorithms

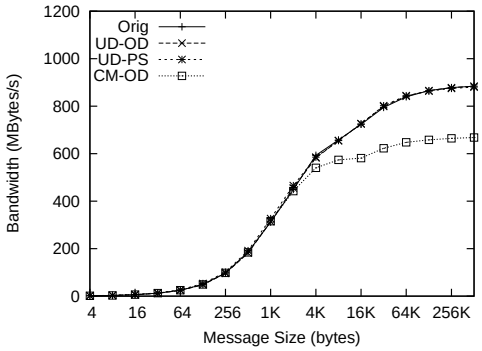


Figure 3.7: Bandwidth of Different Connection Management Algorithms

the performance of NAS programs with different program sizes and numbers of processes. Our testbed has 32 processors, the largest number of processes tested in CG, MG, IS is 32, while only 16 in SP and BT since they require a square number of processes. The performance results of NAS benchmarks indicate that the proposed algorithms have little performance impacts. Thus, the proposed algorithms can provide benefits in terms of scalability and memory resource usage while not having any impact on performance. We believe that the benefits of scalable resource usage would contribute to the performance improvement, which could become noticeable only with larger scale scientific applications. We plan to investigate into this issue further.

3.7. Summary of Adapative Connection Management

We have explored different connection management algorithms for parallel programs over InfiniBand clusters. We have exploited adaptive connection management to establish and maintain InfiniBand services based on communication frequency between a pair of processes. Two different mechanisms have been designed to establish new connections: an unreliable

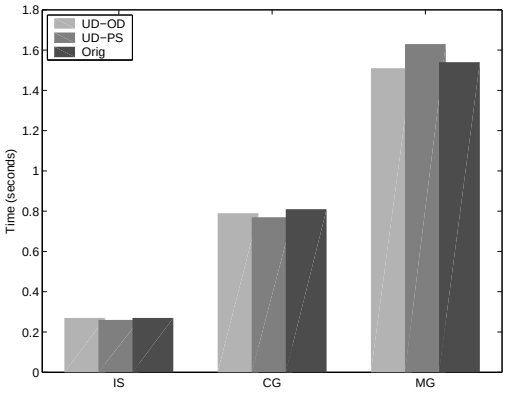


Figure 3.8: Performance of IS, CG, MG, Class A

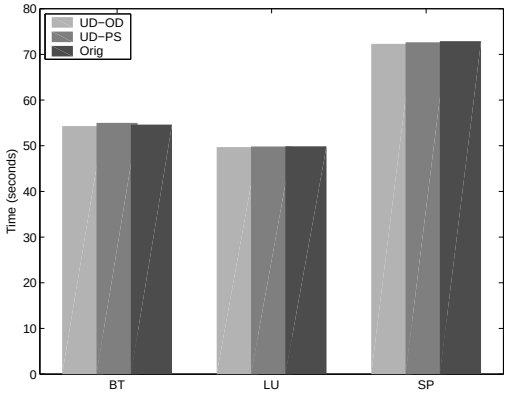


Figure 3.9: Performance of BT, LU, SP, Class A

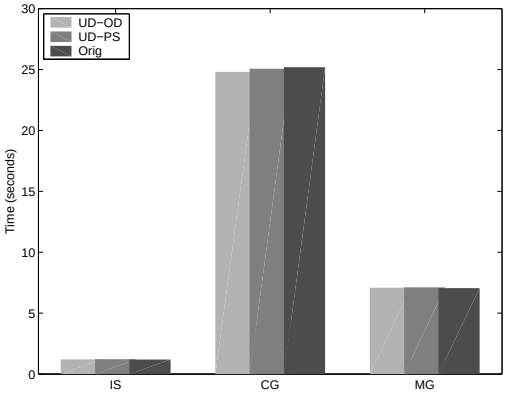


Figure 3.10: Performance of IS, CG, MG, Class B

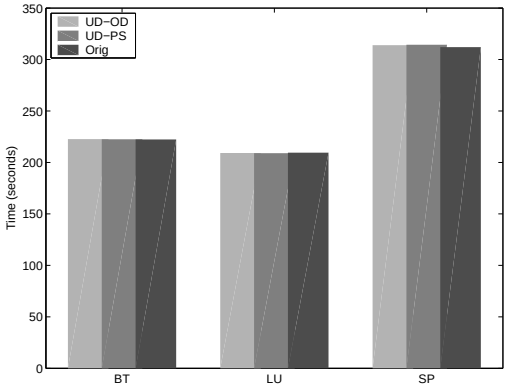


Figure 3.11: Performance of BT, LU, SP, Class B

datagram-based mechanism and an InfiniBand connection management-based mechanism. The resulting adaptive connection management algorithms have been implemented in MVAPICH to support parallel programs over InfiniBand. Our algorithms have been evaluated with respect to their abilities in reducing the process initiation time, the number of active connections, and the communication resource usage. Experimental evaluation with NAS application benchmarks indicates that our connection management algorithms can reduce the

average number of connections per process, the initial memory usage, as well as the startup time for the MPI program.

CHAPTER 4

Checkpoint/Restart for Fault Tolerant MPI over InfiniBand

Message Passing Interface (MPI)[53] has no specification about the fault tolerance support that a particular implementation must achieve. As a result, most MPI implementations are designed without any fault tolerance capability. Faults occurred during the execution time often abort the program and the program has to start from beginning. For long running programs, this can waste a large amount of computing resources as all the computation done before the failure is lost. To save the valuable computation resources, it is desirable that a parallel application can restart from some previous state before a failure and continue the execution. Checkpointing and rollback recovery is the most commonly used technique in fault recovery. Recent deployment of large-scale InfiniBand-based cluster systems makes it imperative to deploy checkpoint/restart support for the long-running MPI parallel programs so that they can recover from failures. However, it is still an open challenge to provide checkpoint/restart support for MPI programs over such InfiniBand clusters.

We have explored the existing checkpoint/restart efforts and utilized Berkeley Lab's Checkpoint/Restart(BLCR)[33] as a base of designing checkpoint/restart support for MPI programs over InfiniBand. Based on the capability of BLCR [33] to take snapshots of processes on a single node, we have designed a checkpoint/restart framework to orchestrate the

process of checkpointing and periodically take globally consistent snapshots of an entire MPI program.

We have implemented our design of checkpoint/restart in MVAPICH2 [60]. Our implementation of Checkpoint/restart-capable MVAPICH2 (MVAPICH2-CR) allows low-overhead, application-transparent checkpointing for MPI applications. To the best of our knowledge, this work is the first report of checkpoint/restart support for MPI over InfiniBand clusters in the literature.

4.1. Parallel Coordination of Checkpoint/Restart

While BLCR provides a way to checkpoint/restart an individual process, a process management framework must be designed to orchestrate the checkpointing and/or restart parallel applications. Here we first describe our checkpoint/restart framework and the functionalities of individual components. Then we sketch the steps involved during checkpoint/restart.

Multiple processes (referred as worker processes) together form a parallel application. For retaining management capabilities to processes, *stdio* and also resources within the computing nodes, a running MPI program involves at least two other components: the front-end process, a.k.a *MPI console*, and the management daemons, which create and help monitoring worker processes of an parallel application. The actual computating state of such application includes every process, as well as that of the console because it manages the important information on *stdio* and application status. There could be transient control messages across the management daemons, but they are not indispensable to the parallel application for the recovery purpose, thus can be skipped from the snapshot. So in order to checkpoint a parallel application, a parallel process management framework needs to provide functionalities to checkpoint console and worker processes. In addition, for coordination

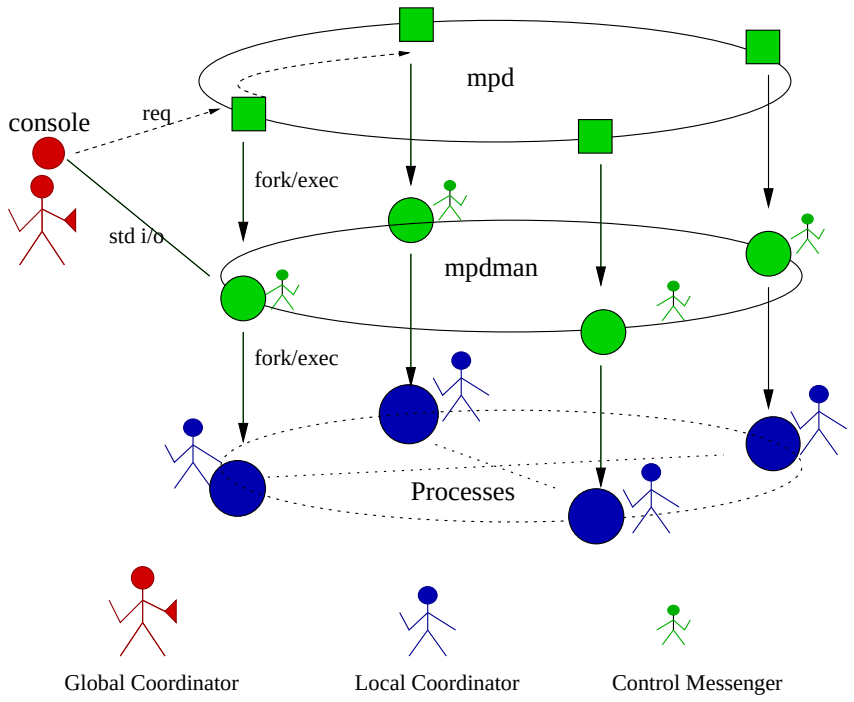


Figure 4.1: MPD-based Checkpoint/Restart Framework

purpose, control messages need to be communicated amongst the console, the management daemons and the worker processes. Based on this understanding we have introduced the global co-ordinator, local coordinators and control messenger to undertake these functionalities. Note that the management daemons can be from any process management framework, such as PBS and SLURM, even just rsh/ssh daemons at the simplest form. We have chosen to use MPD because it is the default process management framework distributed along with MVAPICH2.

Figure 4.1 shows the design of parallel checkpoint/restart components, along with two rings of MPD processes: mpd and mpdman. The global coordinator is a separate thread along with the MPI console process. Local coordinators are separate threads, one for each MPI processes. Control messengers are extended activities from mpdman processes.

- **Global Coordinator**: The global coordinator manages the checkpoint/restart procedure for an entire MPI job. It initiates checkpointing based on checkpoint/restart requests from a user signal or a system event (be it an error or faulty state). In addition, it triggers the checkpoint/restart of the console process in order to maintain the communication state of the console process.

- **Local Coordinator**: There is a local coordinator for each MPI process, which participates in the coordination initiated from the global coordinator. Each local coordinator is responsible for triggering the corresponding checkpoint/restart of the worker process. Its functionality includes the following: (a) communicating C/R requests from and replies to the global coordinator; (b) preparing the local MPI process for checkpoint/restart, by triggering the draining of communication state, and the open/close of the communication channels; (c) invoking BLCR [33] to perform the process of checkpoint/restart for a local worker process.

- **Control Messenger**: Control message manager provides an interface between the global coordinator and the local coordinator. It needs to be coupled with the process manager framework for the extended functionality of communicating C/R messages. In our design, we extend the functionality of MPD process manager, mpdman, to communicate C/R control messages.

Figure 4.2 depicts the state diagram of our checkpoint/restart framework. A checkpointing cycle involves five phases: initial synchronization, pre-checkpoint coordination, local checkpointing, post-checkpoint coordination, and restarting. These phases are described in detail below.

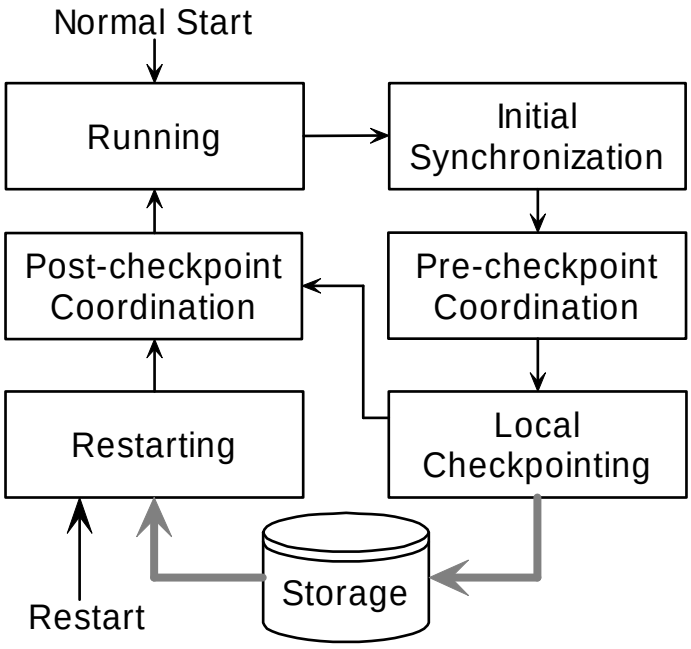


Figure 4.2: State Diagram for Checkpoint/Restart

Initial Synchronization All processes in a job synchronize with each other and prepare for pre-checkpoint coordination. First, the global C/R coordinator from the console process propagates a checkpoint request to all local C/R coordinators running in individual MPI processes. Then, upon the arrival of the request, local C/R coordinators takes over the control of communication channels from the main thread to avoid possible inconsistency of communication channels that might be caused by interleaved accesses from both C/R coordinator and main thread.

Pre-checkpoint Coordination The global and local coordinators work together to prepare all MPI processes and the front-end console for checkpointing to ensure global

consistency. During this phase, local coordinators must manage to drain the outstanding message inside the communication channels, i.e. over the network. This is to ensure the entire state of the parallel job can be captured from the compute nodes, that is to say, nothing left in the network.

Local Checkpointing Each coordinator invokes BLCR to take snapshots of the process it is responsible for. The checkpoint files are saved to stable secondary storage, either a local disk or a remote storage through network file system.

Post-checkpoint Coordination This phase is needed to resume the execution of the parallel application. The global and local coordinators cooperate together and reactivate communication channels, which involves rebuilding the low level network connections and processing outstanding send/receive messages.

Restarting This is an additional phase needed to recreate all processes from their individual checkpoint files when restarting a parallel job. The global coordinator first recreates the console and propagates the restart request to local coordinators, who in turn recreate all worker processes. Local C/R coordinators also reestablish the control channels with their worker processes to re-gain the process control and resume coordination with the global coordinator. At this point of time, the job is restarted from a state same as a previous state in local checkpointing phase. A post-checkpoint coordination phase then brings the entire application back to its running state.

4.2. Ensuring Consistency of Global State over InfiniBand

For the initial framework, we adopt a coordinated approach to take globally consistent, transparent snapshots of MPI applications. In this section, we explain how we transparently

suspend/reactivate the InfiniBand communication channel while preserving the channel consistency.

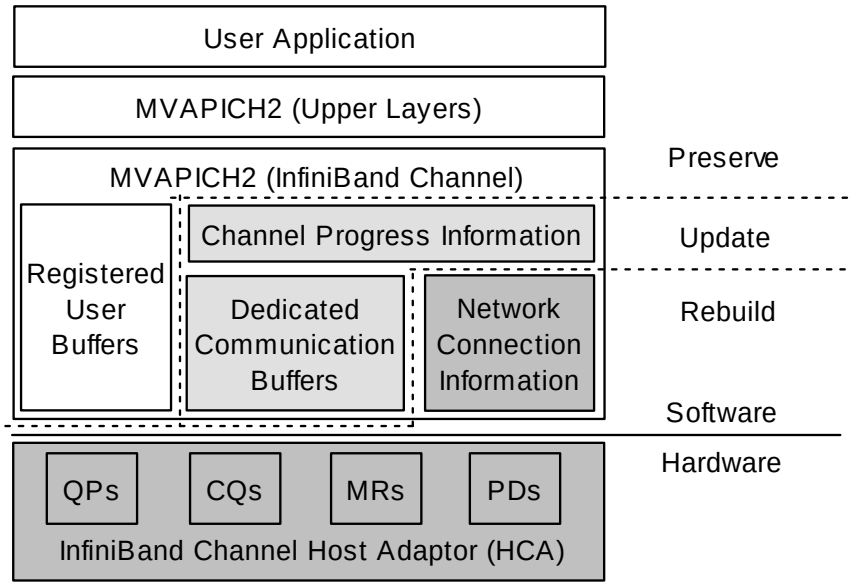


Figure 4.3: Consistency Ensurance of InfiniBand Channels for Checkpoint/Restart

The structure of InfiniBand communication channel in MVAPICH2 can be described by Figure 4.3. Below the MVAPICH2 InfiniBand channel is the InfiniBand Host Channel Adapter (HCA), which maintains the network connection context, such as Queue Pairs (QPs), Completion Queues (CQs), Memory Regions (MRs), and Protection Domains (PDs). MVAPICH2 InfiniBand channel state consists of four parts:

- **Network connection information** is the user-level data structures corresponding to the network connection context.
- **Dedicated communication buffers** are the registered buffers which can be directly accessed by HCA for sending/receiving small messages.

- **Channel progress information** is the data structures for book-keeping and flow control, such as pending requests, credits, etc.

- **Registered user buffers** are the memory allocated by user applications. These buffers are registered by communication channel to HCA for zero-copy transmission of large messages.

Among the four, network connection information can be discarded during checkpointing, and be rebuilt afterwards. Dedicated communication buffers, and channel progress information by and large remain the same. Registered user buffers need to be re-registered and the content of them need to be totally preserved.

In pre-checkpoint coordination phase, to suspend communication channels, channel managers first drain all the in-transit messages, which means that, upon synchronization, all the messages before that point must have been delivered and all the messages after that point must have not been posted to network. Two things need to be noted here: (a) the word 'messages' refer to the network level messages rather than MPI level messages, and one MPI level message may involve several network level messages, and (b) the synchronization points for different channels do not need to correspond to the same time point, and each channel can has its own synchronization point. Due to the First-In-First-Out (FIFO) nature of InfiniBand RC channel, this can be achieved by exchanging flagged control messages between each pair of channel managers. These control messages are exchanged to achieve synchronization for the channels. Once the channel manager detects the completion for all outgoing control messages and expected incoming messages, the channel can be successfully suspended. The channel manager then releases the underlying network connection. One key issue involved is when the channel manager should process the messages received before the control message, which are the drained in-transit messages. Because the communication channel is designed

to execute the transmission protocol chosen by upper layers in MPI library, processing an incoming message may cause sending a response message, which may lead to an infinite 'ping-pong' livelock condition. To avoid that, the channel manager has to either buffer the drained messages, or if possible, process these messages but hold on the response messages.

In post-checkpoint coordination phase, after rebuilding underlying network connections, the channel manager first updates the local communication channel as we described before, and then sends control messages to update the other side of the channel. The remote updating is to resolve the potential inconsistency introduced by invalid remote keys for RDMA operation. For example, the rendezvous protocol for transmitting large messages is implemented with RDMA write. To achieve high responsiveness and transparency, our design allows rendezvous protocol being interrupted by checkpointing. Therefore the remote keys cached in sender side for RDMA write will become invalid because of the re-registration on receiver side. Hence, the receive channel manager needs to obtain the refreshed memory keys from the source process.

4.3. Performance Evaluation of Initial Checkpoint/Restart Support

In this section, we describe experimental results and analyze the performance of our current implementation based on MVAPICH2-0.9.0. Our experiments are conducted on an InfiniBand cluster of 12 nodes. Each node is equipped with dual Intel Xeon 3.4GHz CPUs, 2GB memory and a Mellanox MT25208 PCI-Express InfiniBand HCA. The operating system used is Redhat AS4 with kernel 2.6.11. The filesystem we use is ext3 on top of local SATA disk.

Benchmark:	lu.C.8	bt.C.9	sp.C.9
Checkpoint File Size (MBs):	126	213	193

Table 4.1: Checkpoint File Size per Process

We evaluate the performance of our implementation using NAS parallel Benchmarks [87] and High-Performance Linpack (HPL) [5]. First, we analyze the overhead for taking checkpoints and restarting from checkpoints, and then we show the performance impact to applications for taking checkpoints periodically.

Overhead Analysis for Checkpoint/Restart – We have first analyzed the overhead for C/R in terms of checkpoint file size, checkpointing time, and restarting time. We choose BT, LU and SP from NAS Parallel Benchmarks and HPL Benchmarks.

Because checkpointing involves saving the current state of running processes into reliable storage, i.e. taking a full snapshot of the entire process address as a checkpoint file. Thus the file size is determined by the process memory working set, Table 4.1 shows the checkpoint file sizes per process for BT, LU and SP, class C, using 8 or 9 processes.

The time of checkpoint/restart is determined mainly by three factors: the time for synchronization; the time for store/retrieve the checkpoint file; and the time for suspending and resuming communication channels.

Figure 4.4 shows the time for checkpointing/restarting NAS benchmarks. It also provides the file accessing time for the checkpoint file for comparison. With the limited performance of underlying ext3 file systems over SATA disks on our system, we have observed that the file accessing time is the dominating factor. High performance parallel file system can also be used to store the checkpoint files for better performance. We plan to further investigate issues in this directions for speeding up the commitment of checkpoint files.

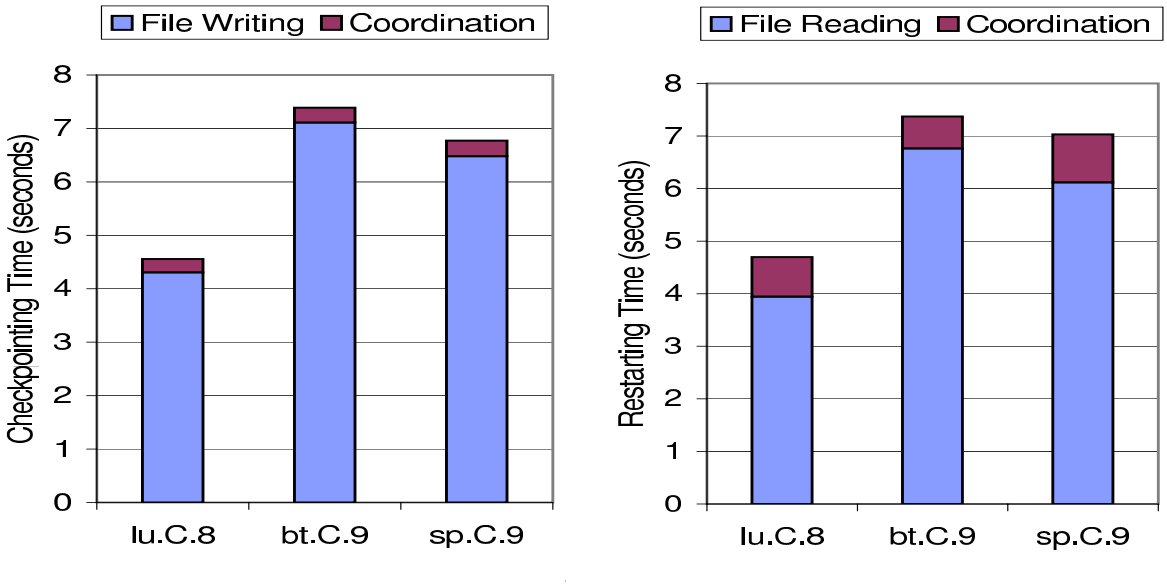


Figure 4.4: Overall Time for Checkpointing/Restarting NAS

To further analyze the coordination overhead, we have taken a time breakdown of the coordination time for individual phases. As shown in Figure 4.5, post-checkpoint coordination consumes most of the time. The reason is that post-checkpointing involves a relatively time-consuming component, the establishment of InfiniBand connections, which has been explored in our previous study [97]. The response time, which is the sum of initial synchronization time and pre-checkpoint coordination time, represents the delay from when the checkpoint request is isssued to the time when all MPI processes have drained all their network messages, reaching a stage for taking a global consistent checkpoint. For restarting, the post-checkpoint coordination consumes almost the same amount of time as that of checkpointing, but the major part of time is in restarting phase, during which all processes have to be reloaded from the checkpoint files.

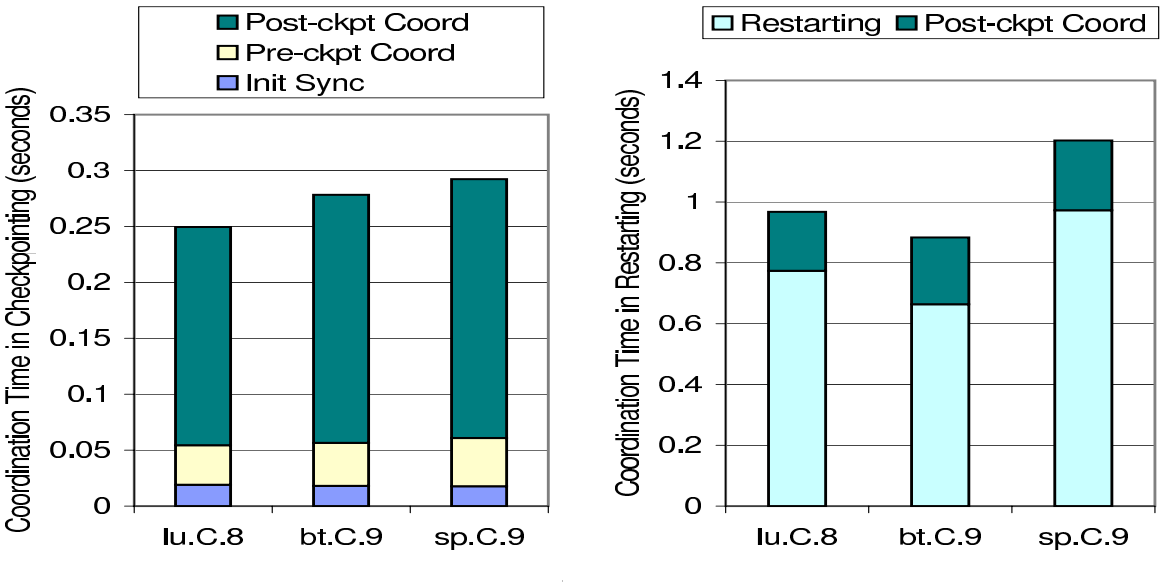


Figure 4.5: Coordination Time for Checkpointing/Restarting NAS

To evaluate the scalability of our design, we measure the average checkpointing time for HPL benchmark using 2, 4, 6, 8, 10, and 12 processes. In the experiment we choose the problem size to let HPL benchmark consume around 800MB memory for each process.

To improve the scalability, we adopt the technique of boot-strap channel described in [97] to reduce the InfiniBand connection establishment time from the order of $O(N^2)$ to $O(N)$, where N is the number of connections. As shown in Figure 4.6, post-checkpoint coordination time, is $O(N)$, the overall coordination time is also in the order of $O(N)$. To further improve the scalability of checkpoint/restart, we plan to utilize adaptive connection management model[95] to reduce the number of active InfiniBand connections.

Performance Impact for Checkpointing – we have conducted experiments to analyze the performance impact when taking checkpoints at different frequencies during the execution time of applications. We used LU, BT and SP from NAS benchmarks and HPL benchmark to simulate user applications.

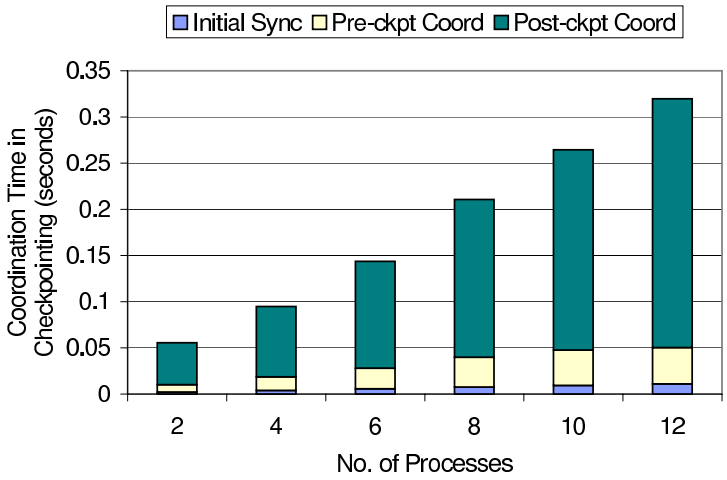


Figure 4.6: Coordination Time for Checkpointing HPL

As shown in Figure 4.7, the total running time of LU, BT and SP decreases as the checkpointing interval increases. The additional execution time caused by checkpointing matches the theoretical value: *checkpointing time* $\times$ *number of checkpoints*. Figure 4.8 shows the impact on calculated performance in GLFOPS of HPL benchmarks for 8 processes. The dominating part of the overhead for checkpointing is the file writing time. With a reasonable checkpointing interval, e.g. 4 minutes. the performance degradation appears to be negligible even with this overhead included.

4.4. Summary of Process Fault Tolerance with Checkpoint/Restart

We have presented our design of checkpoint/restart framework for MPI over InfiniBand. Our design enables application-transparent, coordinated checkpointing to save the state of the whole MPI program to reliable storage for future restart. We have evaluated our design using NAS benchmarks and HPL. Experimental results indicate that our design introduces very little checkpointing overhead.

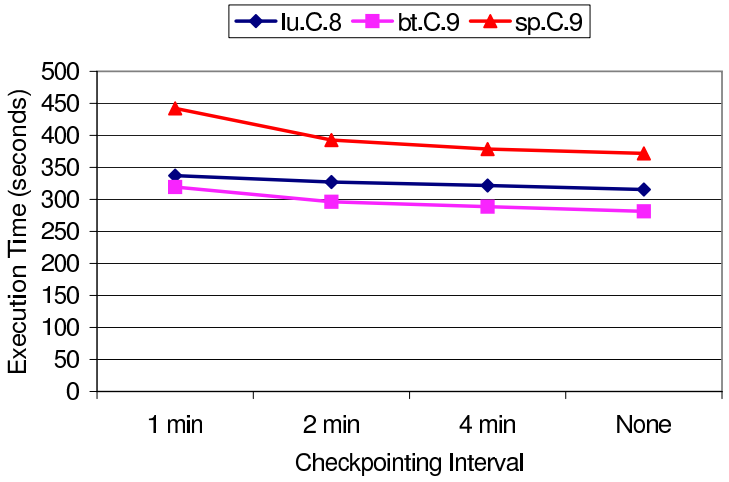


Figure 4.7: Performance Impact for Checkpointing NAS

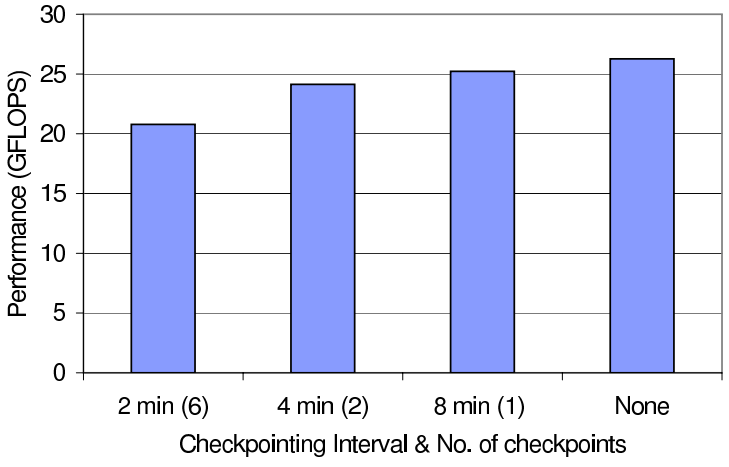


Figure 4.8: Performance Impact for Checkpointing HPL

CHAPTER 5

High Performance End-to-End Reliable Broadcast

Broadcast is an important collective operation, which can be used to implement other collective operations, such as allreduce and barrier. The current LA-MPI [36] implementation is generic, using point-to-point messaging and a spanning tree algorithm to implement the broadcast. Fault tolerance to data transmission errors is already provided by the point-to-point messaging layer. Here we describe how we use the hardware broadcast primitive provided by the Quadrics network to implement an efficient MPI broadcast function, while providing fault tolerance to data transmission errors. We first provide an overview of LA-MPI architecture. Then we propose a new end-to-end broadcast protocol that takes advantage of Quadrics hardware broadcast. Finally, we provide performance results.

5.1. LA-MPI Architecture

As shown in Figure 5.1, the implementation of LA-MPI has its MPI interface layered upon a set of User Level Messaging (ULM) interface primitives, which itself consists of two layers: the Memory and Message Layer (MML) and the Send and Receiver Layer (SRL) [36].

The Memory and Message Layer – The MML layer is composed of a memory manager, a set of network paths and a path scheduler. The memory manager controls all memory

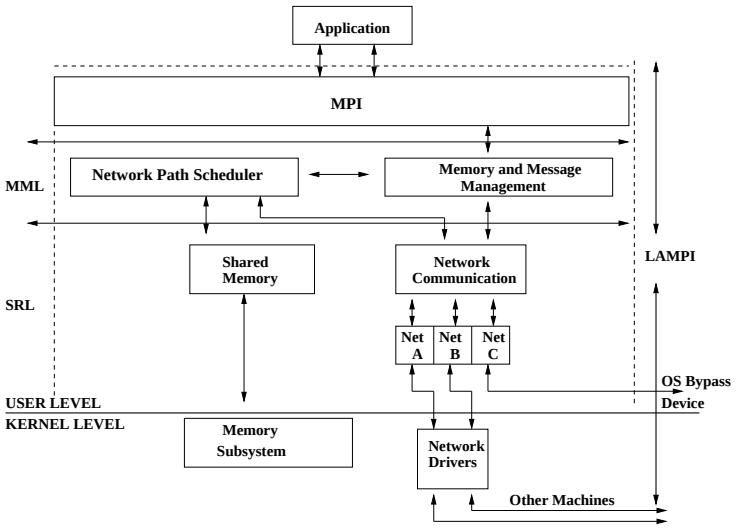


Figure 5.1: LA-MPI Architecture

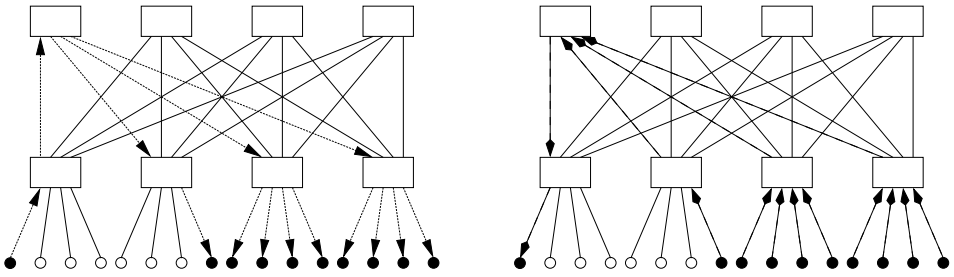
(physical and virtual), including the process private memory, shared memory, as well as "network memory", such as memory on the NIC. A network path is a homogeneous transport abstraction used to encapsulate the properties of different network devices and protocols. It controls access to one or more network interface cards (NICs), within a path there may be several independent "routes". The path scheduler "binds" a specific message between a pair of source and destination processes to a particular path. Message pipelining and reassembly are supported in LA-MPI at MML layer. Together all three components of the MML architecture provide support to MPI functionalities with network transmission-specific primitives (i.e., the SRL).

The Send and Receiver Layer – The Send and Receive Layer (SRL) is responsible for sending and receiving messages. It consists of multiple network path implementations and a highly optimized shared memory communication implementation. Messages that do not require the network (on-host) are simply copied through shared memory. Those that do

require the network (off-host) are handled by the Network Communication module, where the message fragments are sent via physical resources associated with the path, to which the message is bound. The SRL layer also supports message fragmentation and reassembly.

5.2. Quadrics Hardware Broadcast

An efficient, reliable and scalable hardware broadcast is also supported over QsNet [73]. As shown in Figure 5.2, a hardware broadcast packet takes a predetermined path to reach all the recipients. It is successfully delivered only when all the recipients send an acknowledgment. The top Elite switch in the path takes care of broadcasting the packets to and combining the acknowledgments from the recipients [70]. However, the hardware broadcast has its own restrictions. The destination addresses in a hardware broadcast have to be the same across all the processes being addressed. In addition, the processes being addressed must be located on contiguous nodes in order for the switch to perform the broadcast. The second release, QsNet-II, removes this restriction and supports hardware broadcast to non-contiguous nodes.



(a) Data Propagation (b) ACK Collection

Figure 5.2: Quadrics Hardware Broadcast

5.3. LA-MPI Communication Flow and its Broadcast Performance

LA-MPI implements an end-to-end reliable MPI library. Reliability is provided with sender-side retransmission by checking the list of unacknowledged messages every 5 seconds (adjustable at compile time). LA-MPI reliable transmission protocol is shown in Figure 5.3. The Sender and the receiver coordinate their activities as follows.

At the sender side:

1. Post a sender descriptor to send a message
2. Bind a send descriptor to a network path and generate fragment descriptors
3. Record transmission time and sequence numbers to the send descriptors
4. Send fragments along with the CRC/checksum
5. Check received ACK's and NACK's and update the list of fragment descriptors
6. Retransmit a fragment if it is timed out or a corresponding NACK is received

At the receiver side:

1. Post a receive descriptor to receive a message
2. Check the integrity of received fragments, and return ACK's or NACK's accordingly
3. Assemble the received fragments into messages and match with the receive descriptors

LA-MPI collective operations, such as MPI_Bcast, are layered on top of point-to-point operations. The broadcast operation over point-to-point operations uses a generic tree-based

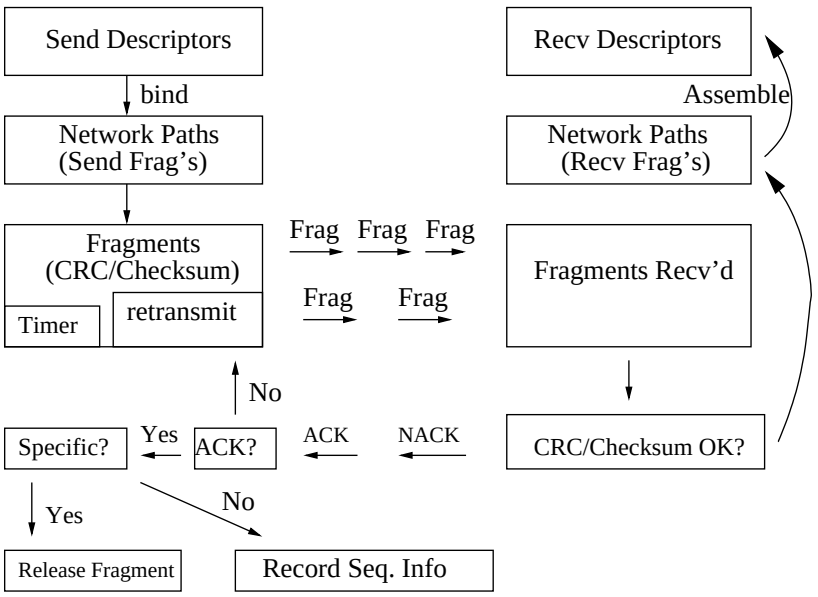


Figure 5.3: Communication Flow Path and Reliability Model in LA-MPI

algorithm for fast delivery. If the network path only provides point-to-point communication primitives, the tree-based algorithm (or its variants) can provide the best performance. However, one of LA-MPI supported interconnects, Quadrics, provides an efficient, reliable and scalable RDMA-based hardware broadcast. The broadcast algorithm implemented in libelan from Quadrics [73] provides a high performance, scalable broadcast operation by taking advantage of the hardware broadcast (shown in Figure 5.2). A tree-based broadcast algorithm is also provided in libelan. Our work focuses only on the algorithm on top of hardware-based broadcast. We refer to this algorithm as the libelan broadcast implementation unless specified otherwise. LA-MPI tree-based algorithm does not take advantage of Quadrics hardware broadcast. As shown in Figure 5.4, in our eight-node Quadrics cluster LA-MPI Broadcast operation is not performing as well as the MPICH broadcast operation. In addition, LA-MPI has a logarithmic scalability compared to the constant scalability of

the MPICH broadcast operation. Thus it is desirable to incorporate the hardware broadcast into LA-MPI's end-to-end reliability model for optimized broadcast performance.

5.4. End-to-End Reliability with Hardware Broadcast

End-to-end reliability for a broadcast operation implies a form of acknowledgments to be returned from the receivers to the sender. That adds an overhead to the base performance of the message passing activities. Therefore it seems to be conflicting goals in order to achieve end-to-end reliability while maintaining the maximum performance of message passing at the same time. More performance degradation is readily seen by the collectives, as exemplified by LA-MPI broadcast performance in Figure 5.3. It is important to characterize the hardware broadcast communication before any initiatives on exposing its maximum performance and providing end-to-end broadcast reliability.

Quadrics Hardware Broadcast Communication Flow – Quadrics hardware broadcast can be considered as a two-phase communication, up and down, as shown in Figure 5.2. In the up-phase, a packet first takes adaptive routing to a switch that can reach all the destination processes. Then in a down-phase it is replicated by the switch(es) to reach all the recipients. The hardware broadcast is also network reliable. The Elite switches in the path take care of broadcasting the packet to and combining the acknowledgments from recipients [70]. This hardware support produces a highly efficient and scalable hardware broadcast primitive. A communication library that does take advantage of this primitive can speed up the broadcast operation and also other collectives on top it. However, the hardware broadcast has its own limitations. The processes being addressed must have contiguous VPIDs in order for the switch to perform the broadcast (The next generation of Quadrics release, Quadrics-II, removes this limitation). In addition, the destination addresses in a hardware

broadcast have to be the same across all the recipient processes. And to make the hardware broadcast available to a general broadcast routine, a broadcast algorithm usually adds some amount of overhead on top of hardware broadcast.

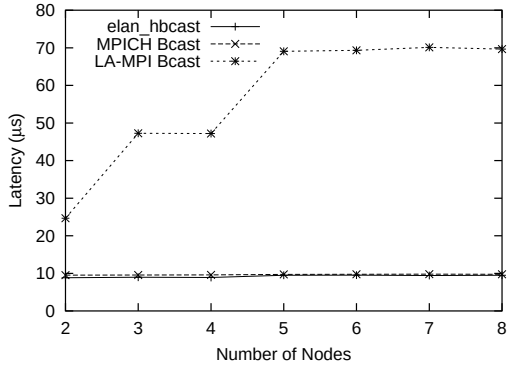


Figure 5.4: Performance Benefits of Hardware Broadcast

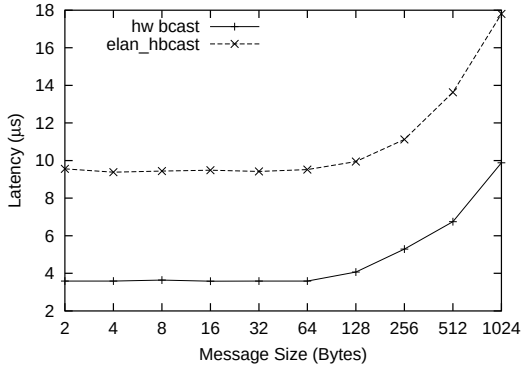


Figure 5.5: The Overhead of Utilizing Hardware Broadcast in libelan

Figure 5.5 shows the comparison between the raw performance of the hardware broadcast and the performance of the current broadcast operation implemented as elan_hbcast in libelan by Quadrics [73]. Again over an eight-node cluster, a raw broadcast latency is only 3.6us, but 9.5us with elan_hbcast for eight-byte message. So a significant amount of overhead is added to the hardware broadcast performance. A part of the overhead is related to the overhead of overcoming hardware broadcast limitations. However, a large portion of the overhead is from the algorithm itself as described in the following. In the case of Quadrics broadcast algorithm on top of the hardware broadcast, two sets of broadcast buffers and associated resources are provided, and used alternatively by the broadcast operations. Each broadcast operation consists of two steps: synchronization and message broadcast. To avoid the incoming broadcast messages clobbering the existing message with RDMA, the root process

starts the hardware broadcast only until it is notified of the completion of synchronization. Consecutive broadcast operations are then throttled by this inserted synchronization, which can lead to a lower broadcast throughput and higher overhead. Minimizing this synchronization overhead is critical for a broadcast algorithm to maximize the performance benefits of Quadrics hardware broadcast.

End-to-End Reliability – As the cost effectiveness of cluster computing continues to be appealing to the high end computing field, the average size of the cluster is ever increasing and the fault tolerance support that comes with the manufacturers are not adequate to guarantee error-free execution of an parallel application, given the length of a typical application run and the very large number of nodes. LA-MPI is an implementation of MPI library that provides end-to-end message passing reliability against possible I/O bus errors or networking errors [36]. Quadrics hardware broadcast is only network-level reliable. Simple means using another message for end-to-end reliability would lead to further degradation on the broadcast performance. It is beneficial if the reliability cost can be merged into the synchronization needed for Quadrics hardware broadcast or hidden from the critical path of broadcast communication.

In this work, we have taken the challenges on the aforementioned issues: maximizing performance benefits and providing end-to-end reliability for hardware broadcast. A broadcast protocol is proposed to reduce the synchronization cost and maximize the performance. A unified synchronization and reliability control is also exploited to reduce the performance penalty on adding end-to-end reliability. These issues are discussed in the following two sections.

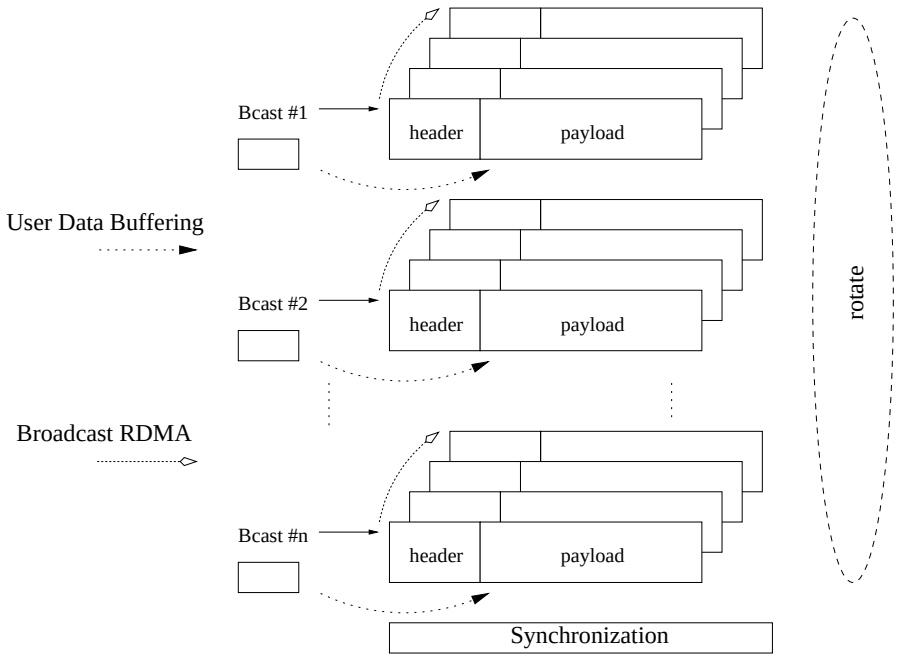


Figure 5.6: Proposed Broadcast Protocol

5.5. Proposed Broadcast Protocol

A general purpose MPI broadcast routine to take advantage of the hardware broadcast has to take into account the restrictions placed by the Quadrics hardware broadcast. One has to provide a global address space for the broadcast communication and overcome the non-contiguity of the receiver nodes. In this section, we propose a broadcast protocol and discuss the basic issues on buffer management and the mechanisms for broadcast transport on the global buffer management and transmission mechanism. In addition, the strategies adopted to design an efficient broadcast protocol are also discussed.

1. **Multiple Buffering**

   Hardware broadcast can only address a destination address that is global across a given set of receiving processes. This necessitates the use of global memory. In order to do

this, an identical elan3 address space across all processes is allocated and then mapped to the host virtual memory by each process. The consistency of the global memory has to be maintained for further broadcast operations. Address translation can have a big impact on the application performance [51]. To ease the buffer management and reduce the impact of address translation, the global memory is divided into an array of broadcast channels. Figure 5.6 shows the allocation of global broadcast channels. At the beginning of each channel, there is a reserved header field. Header/payload boundary can be dynamically determined. Channels with the same index from each process are grouped together. Broadcast operations will deliver the data to each process. This buffer allocation allows the potential advantage from double buffering, a technique to allow a transmission request to be initiated while another is still in progress. It also extends that by allowing multiple outstanding transmissions. Even if the hardware allows one transmission in progress, multiple buffering still can benefit from pipelining the broadcast communication. The management of multiple channels needs to be different from a sliding window protocol. In a typically slide window protocol a sender update the available windows when a notification about the used buffers is received. Since these broadcast channels must be preserved as global address space, one process cannot update its use of a channel unless a global decision is made across all the processes. Thus a global synchronization is inevitable before the reuse of any channel. Synchronization on these global buffer channels is discussed in detail later in the coming section.

2. **Non-blocking and Pipelining Transport**

For a broadcast operation to be performed over the hardware broadcast, it is first bound to a broadcast channel. The sender process fills the header and posts a broadcast

RDMA request down to the network interface. Tradeoffs can be made on whether to buffer the application message into the broadcast channel or broadcast directly from the application buffer. After receiving the broadcast message, a receiver process checks the header and copies the message into its user buffer. MPI broadcast operation can involve an arbitrary set of processes while Quadrics hardware broadcast can only address a set of processes with contiguous VPIDs. Thus the scope of this work focuses on how to expose the performance of the hardware broadcast over a set of contiguous processes to a higher application layer. Work in [28] has explored how to partition the processes into disjoint sets of processes with contiguous VPIDs, then use a tree-based algorithm for fast broadcasting of the data. Nonetheless, in order to make the hardware broadcast available to a set of processes with non-contiguous VPIDs, we use an iterative approach to perform multiple hardware broadcasts to deliver messages to a set of non-contiguous processes, one per subset of contiguous processes.

MPI broadcast operations are blocking in nature. As stated in MPI specification [56], the completion of a call indicates that the caller is free to access locations in the communication buffer. This specification does not mandate how the implementation shall meet the specification at the transport layer. Therefore one can design a collective algorithm that is non-blocking at the transport layer and it is also valid for an implementation of a broadcast operation to buffer the message and return the control to the application immediately so long as the message can be delivered later on. Of course, the implementation has to maintain the compliance to other specifications. With the provision of multiple broadcast channels, multiple broadcast operations can be invoked to the transport layer in a pipeline, and each is processed at different stages of communications. For example, while the receiver is copying a message out of the

broadcast channel, another message is being broadcasted in the network and a third is being buffered at the sender process. These broadcast operations can also be from different sender processes.

To avoid the copying overhead for large message, it is possible to use a form of *long* send, i.e., first send the header with a broadcast RDMA and then chain that with a second broadcast RDMA whose source data is located at the user buffer. However, this precludes the possibility of non-blocking broadcast at the transport layer as the call cannot be returned until the broadcast transport is done. Our earlier results [96] also indicate that there is a superlinear increase on the latency when the message size goes above the buffering threshold (16KB). So instead, large messages are split into multiple small fragments and their transmissions are pipelined at the transport layer. This maintains the benefits of non-blocking transport and also overlaps the buffering (and copying at the receiver side) of one message with the transmission of other messages.

5.6. Synchronization and Reliability

Since the data can be written to a remote process without its knowledge, we must ensure that a new message does not overwrite the previous message that is still in the global buffer. In addition, the broadcast memory must be preserved as global address space. One process cannot update its use of this memory unless a global decision is made across all the processes. Thus synchronization across these processes is needed before a broadcast operation reuses a global buffer.

1. **Synchronization**

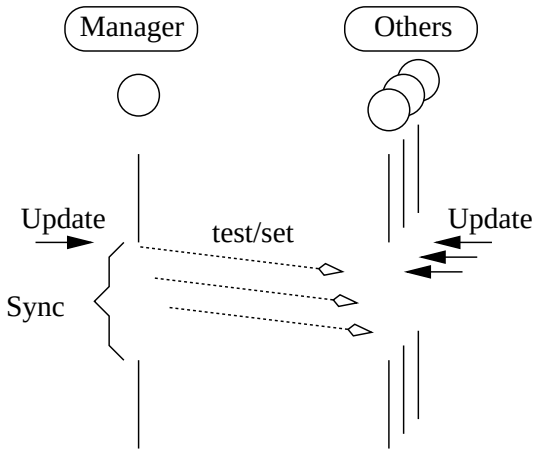


Figure 5.7: Synchronization with Atomic Test-and-Set

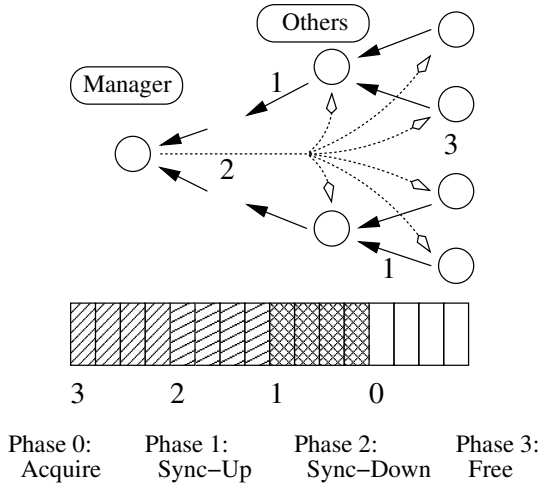


Figure 5.8: Tree-Based Synchronization with Split-Phases

One alternative for the synchronization is to use a tree-based algorithm to have a process collect all the acknowledgments and then update others with the combined results. Another is to use a special form of Quadrics network transactions, test-and-set, to perform a test on a global value [34, 73]. This transaction tests a global variable at all processes against a value and it optionally writes a new value into the global variable [70]. Using this transaction for the synchronization, one can have all processes write a number to a dedicated global variable, and wait for this variable to be updated. This synchronization is very efficient when all the processes reach the synchronization fairly close to each other. However, an exponential back-off scheme must be in place to avoid too much broadcast traffic incurred by the test-and-set transaction [70, 73] in the case of a set of processes unbalanced in their computation. As shown in Figure 5.7, the manager process has to keep polling on the global variable until all the processes have updated their value. Thus all processes must simultaneously participate

in synchronization and the time to complete the synchronization is determined by the process with the maximum skew.

Unlike strict requirement on all processes to participate in the synchronization with test-and-set atomic transaction and potential flooding traffic to the network, the tree-based algorithm is less intrusive to Quadrics network. As show in Figure 5.8, processes will form a tree to propagate their synchronization messages up to the manager. With Quadrics chained RDMA mechanism, these synchronization messages can be combined and propagated to the manager by the network interface without involving host processes or threads. In addition, with multiple broadcast channels and non-blocking transport of broadcast messages, the synchronization can be delayed until no more free broadcast channels are available. Thus the cost of synchronization can be amortized into multiple broadcast operations. With a provision of sufficiently large set of broadcast channels, the synchronization cost can be negligible. Besides, it does not generate polling broadcast traffic as needed by test-and-set.

Furthermore, a split-phase synchronization scheme can be adopted to completely free processes from explicit synchronization. As shown in Figure 5.8, broadcast channels are grouped into multiple sets. Once a process completes the use of one set of broadcast channels, it performs a RDMA to send the synchronization information up to the manager, referred to as sync-up phase. Chained RDMA's combine and propagate the messages to the manager. The manager checks the notifications it has received and, once a new set of broadcast channels are free, it broadcasts the synchronization conclusion down to all others. This is referred to as the sync-down phase. In addition, there are two sets of channels, one that is free to be used by upcoming broadcast operations, the other for which the manager has broadcasted its global decision and

to be marked as free by all processes. So at any given moment, there is one set of free channels being used, another set being reclaimed as free channels, a third set undergoing a sync-up phase for which the manager is being updated about the synchronization status, a fourth set in the sync-down phase for which the manager is broadcasting the combined conclusion on synchronization. Therefore no processes will ever be blocked in waiting for free broadcast channels provided that the time takes to use up one set of the free broadcast channels is more than the synchronization time for any of the four phases.

2. **Reliability and its Unified Control with Synchronization**

Unlike many MPI libraries that consider all underlying communication perfectly reliable, LA-MPI is designed to tolerate the failures of the PCI bus and the network [36]. These errors can be propagated and the effect of these errors can be amplified in long running parallel programs over terascale clusters, because of the sheer number of their components. To achieve reliable end-to-end message passing, LA-MPI optionally supports sender-side retransmission when the messages have exceeded their timeout periods for point-to-point messages. A similar sender side retransmission can be employed to achieve reliable broadcast.

Every broadcast message is transmitted along with an embedded 32-bit additive checksum or, optionally, 32-bit cyclic redundancy code (CRC). This checksum/CRC protects the message against network and I/O bus corruption. When a message is received, its checksum/CRC is validated before the recipient acknowledges its arrival. In the point-to-point cases, if the verification is successful, the recipients generate their positive

acknowledgments, ACK's. Otherwise, negative acknowledgments, NACK's are generated. However, explicit ACK or NACK messages for the reliability control could only increase the implementation overhead. Instead, the existing synchronization setup can be expanded and the reliability control message can be combined into the synchronization scheme. Figure 5.9 shows the proposed unified control of synchronization and reliability. The regular messages for the synchronization of broadcast channels are taken as the ACK's generated to parent processes and ultimately the manager. A separate network of chained RDMA is constructed to communicate NACK's. For fast detection of errors, each NACK is delivered to the manager directly instead of being propagated up. A NACK wins over all the received ACK's. The manger processes take immediate actions to notify all the other processes about a failure. Then all processes will be synchronized to perform the broadcast operation again. Multiple failures of the same broadcast operation will be taken as the hardware or network failure. A fail-over function is provided to detect this and re-bind all the outstanding broadcast operations to another broadcast operation on top of point-to-point operations. The fault tolerant point-to-point operations in LA-MPI can then ensure the completion of the outstanding broadcast operations by choosing another network path. A timestamp and the number of retransmission times are also recorded with every broadcast message. The loss of a message can be detected by the sender or the receiver when it has exceeded its timeout period.

5.7. Broadcast Implementation

As shown in Figure 5.10, we implemented the broadcast operation with added support in both the MML and the SRL in LA-MPI. To make use of the hardware broadcast, global

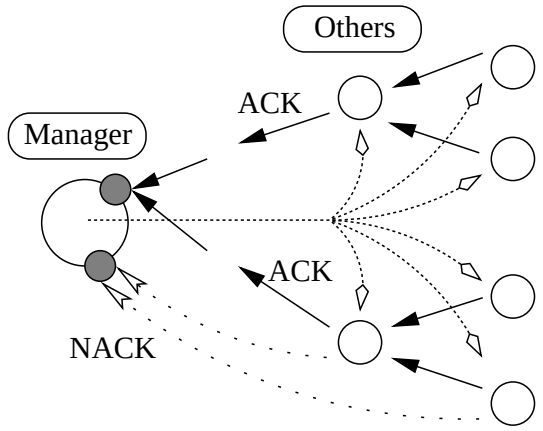


Figure 5.9: Unified Synchronization and Reliability Control

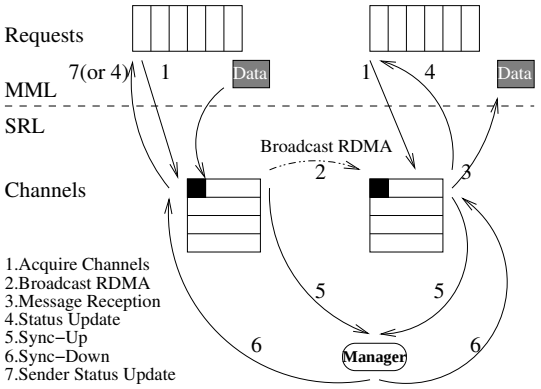


Figure 5.10: Flow Path of Broadcast Operation With Hardware Broadcast

memory is allocated and divided into a number of channels. As shown in Figure 5.10, in the MML for both the senders and receivers, a broadcast operation binds to a free channel. At the sender side, the sender generates CRC/checksum and creates a message header at the beginning of the channel. Messages $\leq$ 16KB are copied over into the channel as payload. Then this message is transmitted over the network with hardware broadcast. Messages $>$ 16KB are fragmented into separate small messages and broadcasted. At the receiver end, the message header is extracted and CRC/checksum checking is done to validate the message. If the received data is not corrupted, messages are delivered to the application buffer and the request status is updated as completed. The broadcast channel synchronization and reliability control are unified into a split-phase synchronization scheme as described earlier. When a message is corrupted or lost, the manager synchronizes the actions of all processes and the previous message is transmitted to complete the broadcast operation. Eventually, when a Quadrics network failure is suspected, the broadcast operation will fail over to the tree-based broadcast operation. And it will be bound to other network paths. Broadcast

within SMP nodes is also supported. One process on an SMP node first participates in the off-host communication that requires the network hardware broadcast. Once the off-host communication is completed, these processes perform an on-host communication with the other processes on the same node through the shared memory to share the data.

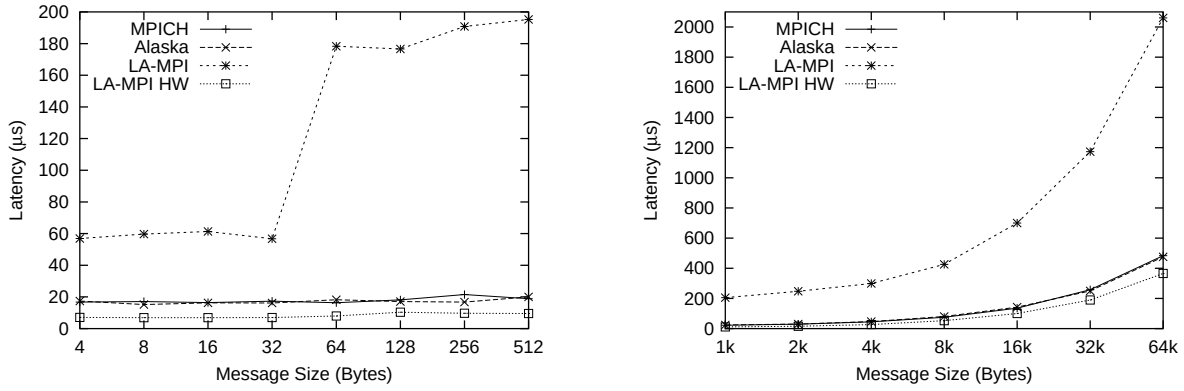
5.8. Performance Evaluation of End-to-End Reliable Broadcast

In this section, we describe the performance evaluation of the broadcast algorithm. We have evaluated the implementation on a TRU64 quad-1.25GHz alpha cluster, which is equipped with Quadrics interconnect, composed of a dimension four switch, Elite-256, and QM-400 cards. On the same system, we have also measured the performance of the broadcast implementation by Quadrics for MPICH [37], and HP's for Alaska MPI. We have used an eight-node cluster of quad-700MHz Pentium-III, in which an Elan3 QM-400 card is attached to each node and links to a quaternary fat tree switch of dimension two, Elite-16.

Our tests are performed by having the processes first warmed up with 20 broadcast operations. Then a sample size of 1000 broadcast operations are performed after a barrier synchronization. This is repeated for 100 samples, each again performed after a barrier. The time for performing each sample is recorded. With statistical analysis on these samples, we derive the average time for a broadcast operation as the latency. The same test is used to measure the performance of MPICH and Alaska MPI broadcast operation, which uses the hardware broadcast communication.

1. **Broadcast Latency**

   We measured the broadcast latency over 128 processes on a contiguous 32-node cluster. As shown in Figure 5.11, the new algorithm in LA-MPI (LA-MPI HW) significantly reduces broadcast latency, compared to the original algorithm. This is to be expected



(a) Small Messages

(b) Large Messages

Figure 5.11: Performance Comparison of Different Broadcast Algorithms

because the new broadcast algorithm takes advantage of the hardware broadcast. As also shown in Figure 5.11(a), our broadcast algorithm outperforms the algorithms implemented for MPICH by Quadrics [73] and HP's for Alaska MPI. This indicates that the new algorithm is able to take advantage of the hardware broadcast with much less overhead compared to the broadcast algorithm provided in libelan. These performance gains are due to the non-blocking and pipelining implementation of the broadcast operations. In addition, split-phase synchronization also contributes to the reduction on synchronization overhead.

2. **Broadcast Scalability**

Scalability is an important feature of any collective operation. To evaluate the scalability of the broadcast operation with different system sizes, we performed the same test with varying number of processes. Figure 5.12 shows the broadcast latency for an eight-byte message over different number of processes. Compared to the original

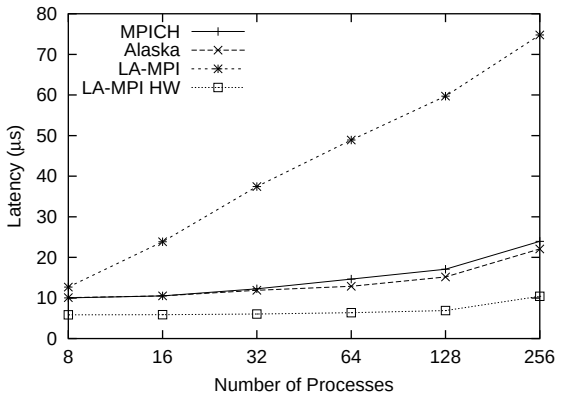


Figure 5.12: Broadcast Scalability with Different System Sizes

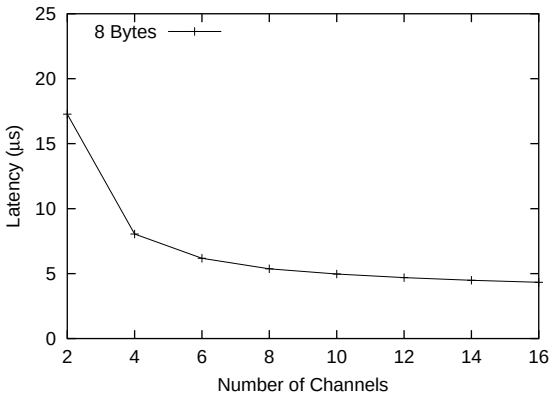
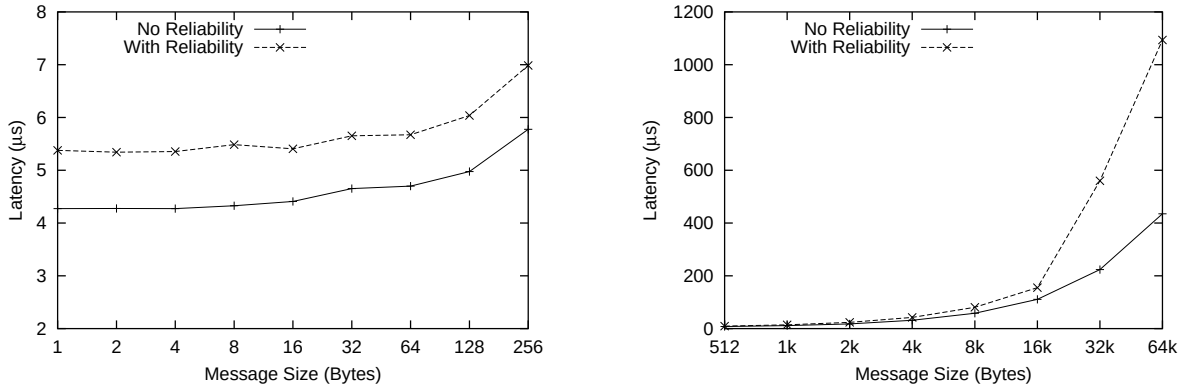


Figure 5.13: Impact of the Number of Broadcast Channels

generic algorithm in LA-MPI, the new broadcast algorithm maintains a scalability close to $O(1)$, up to 128 processes. When compared to MPICH or Alaska MPI, a better scalability is also observed. This indicates that the new broadcast algorithm is also able to provide a better scalability while taking performance advantage of the hardware broadcast.

3. **Cost of Reliability**

While the Quadrics [73] implementation for MPICH [37] and HP's for Alaska MPI leaves the issue of reliable data delivery to the Quadrics hardware, LA-MPI ensures end-to-end fault-tolerant message passing. This broadcast algorithm also allows the option to turn on the reliability. Under the reliable running mode, broadcast messages are retransmitted if any error occurs. The overhead of adding reliability can be measured by comparing the performance of LA-MPI under both the reliable mode and the unreliable mode. As shown in Figure 5.14(a) the reliability cost is about 1μs for messages $\leq$ 256 bytes. This indicates that the added reliability has a little impact



(a) Small Messages

(b) Large Messages

Figure 5.14: Cost of Reliability

to the overall broadcast performance. But it has a significant impact to large message broadcast operations, as shown in Figure 5.14(b). This is to be expected because the generation and validation of CRC/checksum for large messages takes a significant amount of time.

4. **Impact of the Number of Broadcast Channels**

Global synchronization is needed to maintain the consistency of the global buffers. With the provision of multiple broadcast channels, the synchronization cost can be amortized over a set of broadcast operations.

To gain insights into the performance of the broadcast algorithm, it is beneficial to find out the impact of providing different number of broadcast channels. Split-phase synchronization is disabled to isolate the synchronization overhead. We measured the broadcast latency of the broadcast operation with different number of channels over eight processes on an eight-node system. As shown in Figure 5.13, with two channels,

the broadcast latency is 17.1μs for eight-byte messages. The latency is reduced when more channels are provided. When sixteen channels are provided, this latency is reduced to 4.3μs. Thus the synchronization cost is reduced by 13μs as the frequency of synchronization is dropped to every sixteen broadcast operations. By using multiple buffering with sixteen channels, the cost is indeed amortized into multiple operations.

5.9. Summary of Scalable End-to-End Reliable Broadcast

We take on the challenge of incorporating Quadrics hardware broadcast communication into LA-MPI to provide an efficient broadcast operation. We then describe the benefits and limitations of the hardware broadcast communication and possible strategies to overcome them. Accordingly, a broadcast algorithm is designed and implemented with the best suitable strategies. Our evaluation shows that the new broadcast algorithm achieves significant performance benefits compared to the original generic broadcast algorithm in LA-MPI. It is also highly scalable as the system size increases. Moreover, it outperforms the broadcast algorithms implemented by Quadrics [73] MPICH, and HP's for Alaska MPI. Furthermore, instead of leaving the reliability of message passing to the Quadrics hardware, this new algorithm can ensure the end-to-end reliable data delivery.

CHAPTER 6

NIC-based Collective Operations over Myrinet/GM

Collective communication can consist up to 70% of the execution time of a scientific application [72]. High performance and scalable collective communication is an important factor to achieve good performance and increase the effective utilization of a high-end computing platform. The programmable processors in some modern interconnects, including Myrinet [12] and Quadrics [71], have been leveraged to offload communication processing and optimize collective communication and synchronization [10, 18, 19, 94, 55, 93]. Collective operations based on programmable Network Interface Cards (NICs) can reduce the host processor involvement [19, 94], avoid round-trip PCI bus traffic [19, 94], and increase the tolerance to process skew [16, 94] and operating system effects [55]. These benefits are beneficial to the performance of collective operations in terms of latency and bandwidth.

In this chapter, we present our studies on further enhancing collective operations using modern networking mechanisms over Myrinet. Through our studies, we have designed and implemented NIC-based barrier, broadcast and all-to-all broadcast over Myrinet/GM. Since they share some common design issues, we first describe these issues together. Then we discuss the design and performance evaluation of individual operations separately.

6.1. Myrinet Programmability and Myrinet Control Program

Myrinet provides a programmable processor in its network interface. That lends a great degree of flexibility to develop experimental protocols with different design options [66, 78, 35, 8]. New firmwares can be developed to support different protocols with different communication characteristics, or optimize the default firmware from Myrinet/GM, i.e., Myrinet Control Program (MCP) to facilitate certain communication patterns. In the second generation of the GM protocol, a data structure, the packet descriptor, is also introduced to describe every network packet. Inside this data structure, there is a callback handler, which allows the possibility of further actions when the previous action on the packet is completed. By using the Myrinet packet descriptor and its callback handler, one can easily have a packet queued again for transmission before it is freed. For example, a callback handler can be used to change the packet header and send a replica of the packet to another destination. This will be beneficial to collective operations, such as broadcast, in which a packet is repeatedly transmitted to different destinations.

6.2. Challenges in Designing NIC-Based Collective Operations

Collective operations usually involve complex algorithms, and dense communication pattern. This brings a variety of challenging issues including: (a) group topology management, (b) collective buffer management, (c) communication processing, (d) communication algorithm, (e) message reliability and deadlock. We discuss each of these issues in detail as follows.

1. **Group Topology Management**

Collective communication needs to maintain information about group topology. One topology may give better performance over another depending on the communication characteristics and also the desired performance metrics, latency or throughput. The performance of logical topology can be affected by the underlying hardware topology. Our intent is not to study the effects of hardware topology. We use Myrinet network default hardware topology, Clos network. For clusters with thousands of nodes, placing the entire group membership information in the NIC will have a large memory requirement. This requirement is even higher if the communication state for each peer NIC is to be maintained. Thus it is important to provide a scalable method to store and access group topology information and the associated group communication states.

2. **Collective Buffer Management**

   Processes in parallel applications can reach the same collective communication at different time. A message can arrive at the NIC before the host posts the receive buffer and it is discarded as unexpected. This message will not be retransmitted until the sender is timed out. To avoid dropping this type of immediately needed messages, a system buffer can be provided to accommodate them. This can improve the performance of collective communication and its tolerance to process skew [94].

3. **Communication Processing**

   The main objective of NIC-based collective process is to efficiently offload communication processing for collective messages from the host CPU to the NIC processor. By doing so, The host CPU can avoid polling on the intermediate steps of collective operations and the same data do not have to be pulled over the PCI bus multiple times. This means that, to send the same data to multiple destinations, the first and the only copy

should be injected to network multiple times. Other surrounding mechanisms, such as flow control and reliability, need to be modified or redesigned to fit together. Moreover, to achieve the best benefits, collective operations shall not be treated as algorithmic aggregations of multiple point-to-point operations. This is because point-to-point processing ensures the correct delivery of messages with a whole set of functionalities, such as, queuing, packetization, data reassembly, bookkeeping, and flow/error control, etc. Identifying and minimizing the redundancy of these functionalities in collective operations can simplify processing, decrease memory requirement and improve their scalability.

4. **Collective Algorithm**

   To achieve high performance, collective operations typically are carried out according to a predefined algorithm. Applicable algorithms are often restricted by the group's connectivity and topology. An ideal algorithm needs to avoid complex communication processing and reduce memory requirement at the NIC whenever possible. Thus it is important to choose a right algorithm for the correct collective operation.

5. **Message Reliability and Deadlock**

   GM employs a form of Go-back-N protocol to ensure ordered delivery between peer-to-peer communication end points, called ports. When a packet is not acknowledged within a timeout period, the sender NIC will retransmit the packet, as well as all the packets after it from the same port. Deadlock is another important aspect of concern for collective communication. This can occur if there is a cyclic dependency on some shared resources among multiple concurrent operations, and each of them demands resources at multiple destinations for its completion. With a small number of send

and receive tokens being recycled in the Myrinet Control Program (MCP), collective operations shall avoid any possible deadlock scenario that could starve the MCP.

6.3. Design of NIC-based Barrier

We have explored many of the challenging issues of NIC-based barrier over Myrinet/GM in an earlier work with GM-1.2.3 [19, 21]. We choose to create a separated protocol to process the barrier messages in this work. Figure 6.1 shows the comparisons between the earlier barrier scheme and the new scheme with separated processing for barrier messages.

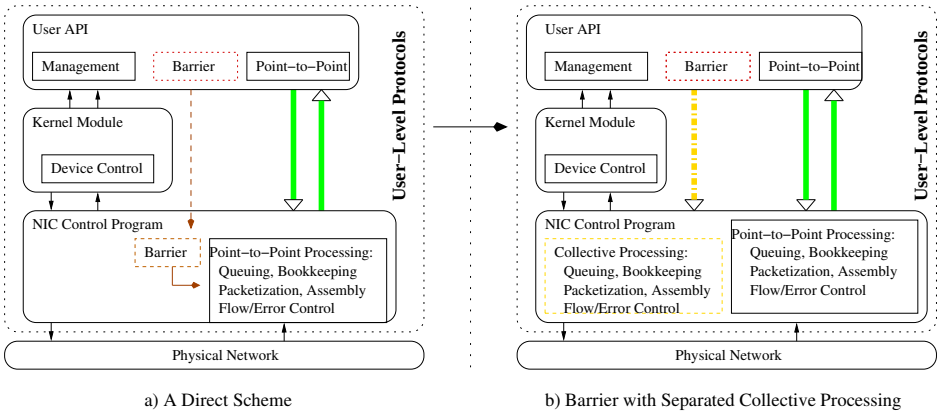


Figure 6.1: NIC-based Barrier with Separated Collective Processing

1. **Queuing the Barrier Operations**

   MCP processes the send tokens to different destinations in a round robin fashion. Send tokens to the same destination are processed in a FIFO manner. So the send tokens for barrier operations must go through multiple queues before their messages can be transmitted. This is enforced for the initial barrier message and also the barrier message that needs to be transmitted immediately when an earlier barrier message

arrives. It is rather inefficient to have the NIC-based barrier operations put up with so much waiting. We created a separate queue for each group of processes, and enqueued only one send token for every barrier operation. Then the barrier messages do not have to go through the queues for multiple destinations. With this approach, the send token for the current barrier operation is always located at the front of its queue. Both the initial barrier message and the ones that need to be triggered later no longer need to go through the queues for the corresponding destinations.

2. **Packetizing the Barrier Messages**

   Within MCP, to send any message, the sender NIC must wait for a send packet to become available and fill up the packet with data. So to complete a barrier operation, it is inevitable for the sender NIC to go through multiple rounds of allocating, filling up and releasing the send packets. Since all the information a barrier message needs to carry along is an integer, it is much more efficient if a static send packet can be utilized to transmit this integer and avoid going through multiple rounds of claiming/releasing the send packets.

   This static send packet can be very small since it only carries an integer. One can allocate an additional short send packet for each group of processes. However, there is a static send packet to each peer NIC in MCP, which is used for fast transmission of ACKs. We pad this static packet with an extra integer and utilize it in our implementation. With this approach, all the packetizing process (including packets claiming and releasing) for transmitting regular messages is avoided for the barrier messages.

3. **Bookkeeping and Error Control for Barrier Messages**

The Myrinet Control Program provides bookkeeping and error control for each packet that has been transmitted. This is to ensure the reliable delivery of packets. One acknowledgment must be returned by the receiver in order for the sender to release the bookkeeping entries, i.e., a send record in MCP. When a sender NIC fails to receive the ACK within a timeout period specified in the send record, it retransmits the packet. Besides creating multiple send records and keeping track of them, this also generates twice as many packets as the number of barrier messages. It is desirable to design a better way to provide the bookkeeping and error control for the barrier operations based on its collective nature.

For the bookkeeping purpose, we create only a send record for a barrier operation. Within the send record, a bit vector is provided to keep track of the list of barrier messages. When the barrier operation starts, a time-stamp is also created along with the send record. In addition, an approach called receiver-driven retransmission is provided to ensure reliable delivery of barrier messages. The receiver NICs of the barrier messages no longer need to return acknowledgments to the sender NICs. If any of the expected barrier messages is not received within the timeout period, a NACK will be generated from the receiver NIC to the corresponding sender NIC. The sender NIC will then retransmit the barrier message. Taken together, these enhancements ensure the reliable delivery with the minimal possible overhead and also reduce the number of total packets by half compared to the reliability scheme for the regular messages. Thus, it promises a more efficient solution for barrier operation.

6.4. Design of NIC-based Broadcast/Multicast

There are several design issues for the implementation of NIC-based broadcast: the sending of message replicas to multiple destinations, message forwarding at the intermediate NIC, reliability and in order delivery, deadlock, and construction of the spanning tree. For each of these issues, we describe design alternatives below and show how we choose the best alternative.

1. **Sending of Multiple Message Replicas**

   To send replicas of a message to multiple destinations, one can directly generate multiple send tokens and queue them to multiple destinations. An alternative is to use a callback handler as described in Section 6.1. A third way to do this is to change the header right after the transmit DMA engine is done transmitting the header and queue the packet again for transmission. The first approach performs the processing for each of the tokens, and it saves nothing more than the posting of multiple send events. The benefits of this is no more than 1μs, if any, since the host overhead over GM is less than 1μs. Both the second and third approach can save the repeated processing, but the third approach takes special care and demands good timing strategy in order to avoid clobbering the packet header before it is transmitted out. We implemented the second approach in our broadcast scheme. The benefits of the third approach could be more, but we decided to leave it for later research.

2. **Messages Forwarding**

   For a received message to be forwarded, we need to consider: 1) how to set up timeout and retransmission mechanisms, and 2) which replica of the message should be made available for the retransmission. As to the first issue, we create send records to record

the time the packets are forwarded. When the records are not acknowledged within the timeout period, retransmission of the packets is triggered. Since the intermediate NIC does not have a send token for this broadcast, one has to generate a token for the purpose of transmission. This can be done by grabbing a send token from the free send token pool, or by transforming the receive token into a send token. Using the former approach can probably lead to deadlock when the intermediate nodes are running out of send tokens. We take the second approach since it does not require additional resources at the NIC. The receive token is presumed to be available to receive any message. In this approach, the receive token is used for transferring the data to the host at the intermediate NIC, and is also used to retransmit the message when it is timed out. As to the second issue, a naive solution would be to keep the received packet available until all the children acknowledge the transmission. The problem with this approach is that the NIC receive buffer is a limited resource, and holding on to one or more receive buffer will slow down the receiver or even block the network. An alternative is to release the packet as the forwarding is done, and use the message replica in the host memory for retransmission. Since GM can only send and receive data from registered memory, this requires the host memory to be kept registered until all the children acknowledged that the packets are correctly received. We take the second alternative in our implementation.

3. **Reliability and In Order Delivery**

   To ensure ordered sending, GM employs a form of Go-back-N protocol to ensure ordered delivery between peer-to-peer communication end points, called ports. When a packet is not acknowledged within a timeout period, the sender NIC will retransmit the packet, as well as all the later packets from the same port. A reliable ordered broadcast

requires modification to the existing ordering scheme. Since each sender is involved with multiple receivers, the sending side must keep track of the ordering of packets in a one-to-many manner to all its children. A modified ordering scheme works as described below. Multicast *send token*s are queued by group. Each broadcast group has a unique group identifier. For each group, the NIC keeps tracks of: 1) a receive sequence number to record the sequence number for the packets received from its parent, 2) a send sequence number to record the packets that have been sent out, and 3) an array of sequence numbers to record the acknowledged sequence number from each child. A broadcast packet sent from one NIC to its children has the same sequence number and send record, ensuring ordered sending for the same group's broadcast packets. When an acknowledgment from one destination is received, the acknowledged sequence number for that destination is updated. If the record for a packet is timed out, the retransmission of the packet and the following ones will be performed only for the destinations which have not been acknowledged. A receiver only acknowledges the packets with expected sequence numbers for the desired group sequentially.

4. **Deadlock**

Deadlock is an important aspect of concern for any collective communication, which may occur if there is a cyclic dependence on using some shared resources among multiple concurrent operations. We take the following approaches to avoid the possibilities of the deadlock. First, we do not use any credit-based flow control, avoiding one source of deadlock. In addition, we provide a unique group identifier and a separate queue for each broadcast group with a sender, so that one group does not block the progress of another. The other possibility for a deadlock is when some nodes in multiple broadcast/broadcast operations form a cyclic parent-child relationship, in which all of them

are using its last receive token while requesting another to receive its message with a new receive token. Since the root node in a broadcast/broadcast operation only uses its send token, it will not be in such a cycle. To break a possible cycle among the rest of the nodes, we sort the list of destinations linearly by their network IDs before tree construction, and a child must have a network ID greater than its parent unless its parent is the root. Thus a deadlock on the use of receive token can not form under either situation. As long as receive tokens are available at the destinations, broadcast packets can be received by all the destinations. The responsibility of making receive tokens available to receive broadcast messages is left to client programs, the same way as is required to receive regular point-to-point messages.

5. **The Spanning Tree**

The tree topology is also important for broadcast performance. One tree topology may give better performance over another depending on the communication characteristics and also the desired performance metrics, latency or throughput. The performance of logical tree topology can be affected by the underlying hardware topology. Our intent is not to study the effects of hardware topology. The default hardware topology, Clos network, is used in our studies. The design issue we study here is *where* to generate the tree, since the NIC processor is typically rather slow to perform intensive computation. To better expose the potential of the NIC-based broadcast protocol, we use an algorithm similar to [17] for constructing an optimal tree in terms of latency. The optimality of such trees has been shown by Bar-Noy and Kipnis [6]. The basic idea of constructing an optimal tree is to have maximum number of nodes involved in sending at any time. In other words, we construct the tree such that a node will send to as many destinations as possible before the first destination it sent to becomes

ready to send out data to its own children. We compute the number of destinations a sender can send to before its first receiver can start sending as the ratio of: (a) the total amount of time for a node to send a message until the receiver receives it, and (b) the average time for the sender to send a message to one additional destination. The message delivery time is calculated as end-to-end latency. Different message lengths lead to different optimal tree topologies. Since the LANai processor is much slower compared to the host processor, we carried out the following division of labor in order to be efficient on tree construction: the host generates a spanning tree and inserts it into a group table stored in the NIC and the NIC is responsible for the protocol processing related to communication.

6.5. Design of NIC-based All-to-All Broadcast

We have explored the design challenges for scalable NIC-based all-to-all broadcast. Given the high demand of memory and computation resources of its dense communication pattern and the limited resources available at the NIC, NIC-based all-to-all broadcast algorithms need to minimize resource requirements. The following major design issues are considered in order to achieve good scalability and high performance.

1. **All-to-All Broadcast Group Topology Management**

   For clusters with thousands of nodes, placing the entire group membership information at the NIC incurs a large memory requirement, and this requirement is even greater if the communication state for each peer NIC is to be maintained. Thus it is important to provide a scalable method to store and access group topology information and the associated group communication states. Typically, collective operations over point-to-point links use spanning trees to cover all the nodes. One node in a spanning tree

communicates with its parent node and a limited number of child nodes. The binomial tree is one of the most commonly used topologies for collective communication. It provides two advantages over other topologies. First, it can be shared by barrier, broadcast (even with different roots), all-to-all broadcast, and other collective operations. Second, since the distance between any pair of directly communicating nodes is always some power of 2, updating the communication state of peers can be easily handled by the NIC processors using bit-shifting operations (typically, NICs are not equipped with FPU). Thus the binomial tree-based topology is a good choice for scalable group management. To achieve scalability, the entire group topology is managed distributively. Each NIC, i, maintains the information of NICs that are in the set $\{i \pm 2^j \bmod N : 0 \leq j < \log N\}$. So, each NIC only needs to maintain the topology and state information of $(2 \times \log N)$ NICs.

2. **Buffer Management for NIC-Based All-to-All Broadcast**

Processes in parallel applications can reach the same collective communication at different times. Packets can arrive at the NIC before the host even posts the receive buffer for the corresponding collective operation. In order to allow the NIC to still receive and forward the early arrived data packets, a system buffer must be present to accommodate these packets. This can avoid having to drop packets and can improve the performance of collective communication and its tolerance to process skew [94]. A common way of managing of collective buffers is to provide a global virtual memory [73, 96], which is divided into multiple channels to receive packets from multiple outstanding collective operations. The order in which these channels are used is globally synchronized across all the NICs when the collective protocol is initialized. At the end of each collective communication, the packets buffered in a global collective

channel (identified with a global collective sequence number) are copied to the application buffer if needed. Collective operations with large messages can be divided into multiple collective operations.

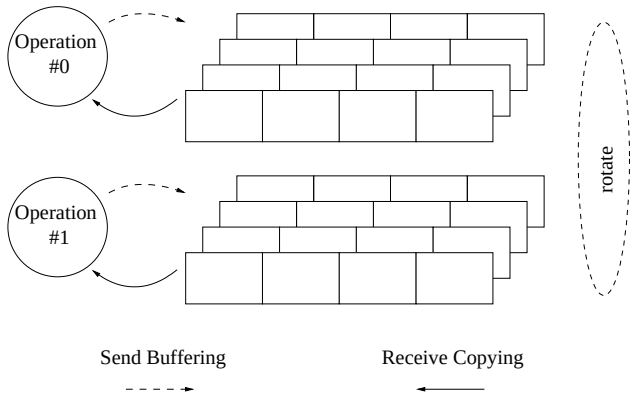


Figure 6.2: Buffer Management for NIC-Based All-to-All Broadcast

With the inherent synchronization of all-to-all broadcast, the completion of an all-to-all broadcast operation signifies that all the other nodes have at least reached the last all-to-all broadcast operation. When all-to-all broadcast requests are generated after the local completion of previous broadcast requests, the following two conditions are always ensured.

(a) At any moment, there is at most one outstanding all-to-all broadcast operation for which packets needs to be sent out.

(b) At any moment, there are at most two outstanding all-to-all broadcast operations for which packets need to be received.

These conditions together simplify the management of all-to-all broadcast buffer channels. At any moment, only two buffer channels need to be provided for outstanding

all-to-all broadcast operations. The use of these channels is inherently synchronized as operations rotate through them. Figure 6.2 shows the management of all-to-all broadcast buffers with two channels. Data to send can be buffered into the channel before it is actually sent to avoid memory registration. A message may be received in the buffer channel if it arrives before the corresponding request is posted, or if user applications choose to use the buffer channel and avoid the memory registration cost. At the end of the operation, the buffered data is then copied out the channel.

3. **Recursive Doubling Algorithm**

   In the recursive doubling algorithm [79], each process pairs with a peer process through bit operations and recursively doubles the exchanged message size at each step. It takes $\log N$ steps for a system size of N nodes. This algorithm is well suited for small messages because it can combine messages into a single packet, thus reducing the number of packets to be processed, and improving the communication performance. However, large packets cannot be efficiently buffered and combined at the NIC (due to slow NIC memory copy speed or insufficient NIC buffers); rather, this algorithm has to buffer the intermediate data by copying the received data, using DMA, to the host memory and then copying it back to the NIC for the next step of communication. It thus does not have the benefit of avoiding round-trip PCI bus traffic. We explore this algorithm in our design in order to minimize the latency and shed more light on the NIC communication processing.

4. **Concurrent Broadcasting Algorithm**

   The concurrent broadcasting algorithm broadcasts the data from each node to all the other nodes. Using the distributively maintained binomial tree topology, each NIC

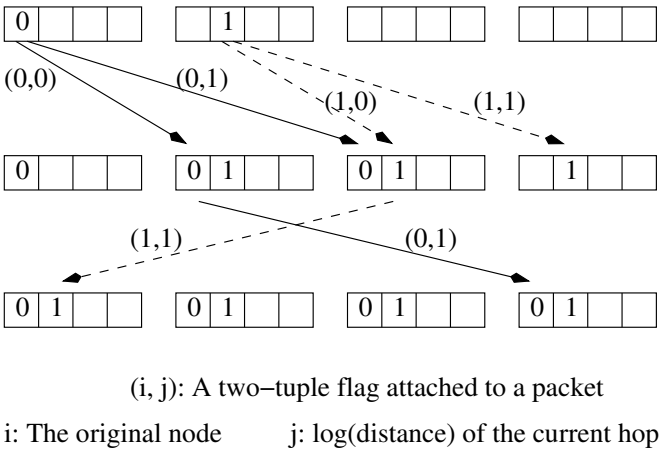


Figure 6.3: Concurrent Broadcasting Algorithm for NIC-Based All-to-All Broadcast (showing only two nodes broadcasting)

becomes a root for a different binomial spanning tree and broadcasts its packets to the other nodes. Having received or completed sending a data packet, a NIC needs to reuse the data packet for fast forwarding or re-sending. However, packets can come from any source, and the current NIC has to perform different roles for the broadcasting of different incoming packets. So the NIC has to decide whether the packet needs to be sent to a next destination and to which one. This process has to be done efficiently and with information from the packet. To this purpose, we have created a form of hopping packets to facilitate the traversal of a packet in its own broadcast spanning tree. Each packet is attached with a two-tuple flag to identify the broadcast spanning tree and the position of its traversal in the tree. As shown in Figure 6.3, in a flag (i, j), i denotes the rank of the original NIC for this packet, and j the log of the distance it has traversed for the current hop. The distance is measured as the difference between the ranks of NICs. The current NIC finds out the next destination of a hopping packet as $(|rank - i| + 2^j)$. If it is not less than the size of the group, then there is no need for the packet to make

any further hops. Figure 6.3 shows how the packets are broadcasted for an all-to-all broadcast operation with hopping packets (only two broadcast spanning trees for two NICs and the hopping packets are given, to avoid complicating the graph).

The all-to-all broadcasting algorithm involves numerous packet forwarding and re-sending and reduces much of the traffic over the PCI bus. However, each NIC has to maintain the communication state about the amount of data that has arrived from each peer NIC. This requirement has two disadvantages. First, for small-message all-to-all broadcast operations, it does not combine the data into larger packets. Thus, for a system size of N nodes, each NIC has to receive $(N-1)$ packets, and forward many received packets. So, for small-message all-to-all broadcast operations, the required processing time increases linearly with the system size, compared to $\log N$ packets with the recursive doubling algorithm. Second, the resource requirement for maintaining the communication state increases linearly with the number of peer NICs and can lead to a scalability constraint. To reduce this resource requirement, we use a status bit-vector to record the communication state. An additional integer can be used to count the number of packets arrived. This approach reduces both the memory requirement and the NIC processing time, because only when all expected packets have not arrived in time is this bit vector checked to find out the missing packets.

6.6. Results of NIC-Based Barrier

1. **Barrier Latency**

We tested the latency of our NIC-based barrier operations and compared it to the host-based barrier operations. Our tests were performed by having the processes execute consecutive barrier operations. To avoid any possible impact from the network topology

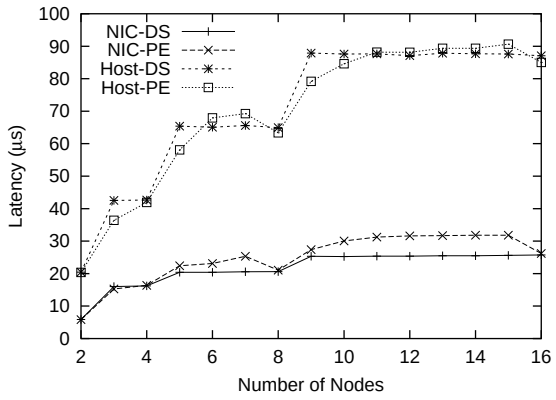


Figure 6.4: Performance Evaluation of NIC-based and Host-Based Barrier Operations with Myrinet LANai-9.1 Cards on a 16-node 700MHz cluster

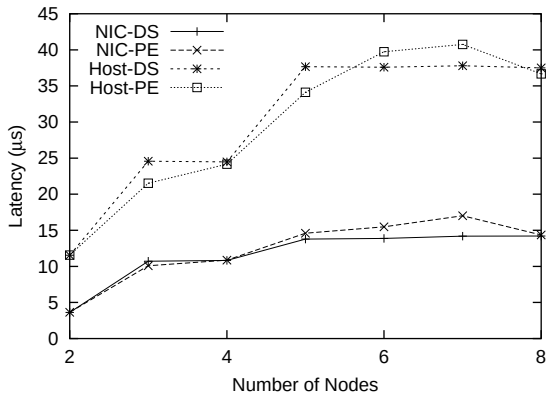


Figure 6.5: Performance Evaluation of NIC-based and Host-Based Barrier Operations with Myrinet LANai-XP Cards on an 8-node 2.4GHz cluster

and the allocation of nodes, our tests were performed with random permutation of the nodes. We observed only negligible variations in the performance results. The first 100 iterations were used to warm up the nodes. Then the average for the next 10,000 iterations was taken as the latency. We compared the performance for both the pairwise-exchange and dissemination algorithms.

Figure 6.4 shows the barrier latencies of NIC-based and host-based barriers for both algorithms over the 16-node quad-700MHz cluster with LANai 9.1 cards. With either pairwise-exchange (PE) or dissemination (DS) algorithm, the NIC-based barrier operations reduce the barrier latency, compared to the host-based barrier operations. The pairwise-exchange algorithm tends to have a larger latency over non-power of two number of nodes for the extra step it takes. Over this 16-node cluster, a barrier latency of 25.72μs is achieved with both algorithms. This is a 3.38 factor of improvement over host-based barrier operations. Using the direct NIC-based barrier scheme on the same

cluster, our earlier implementation [20, 21], achieved 1.86 factor of improvement using LANai 7.2 cards. The earlier work was done over GM-1.2.3 and not maintained as new versions of GM are released. We believe that the same amount of relative improvement (1.86) would have been achieved if the previous work was reimplemented over GM-2.0.3 since the NIC-base barrier is mainly dependent on the number of messages and processing steps to be performed. Although, direct comparisons are not available, the difference in the improvement factors over the common denominator (host-based barrier operations) suggests that our new scheme provides a large amount of relative benefits.

Figure 6.5 shows the barrier latencies of NIC-based and host-based barriers for both algorithms over the eight-node 2.4GHz Xeon cluster with LANai-XP cards. Similarly, the NIC-based barrier operation reduces the barrier latency compared to the host-based barrier operation. Over this eight node cluster, a barrier latency of 14.20μs is achieved with both algorithms. This is a 2.64 factor of improvement over the host-based implementation. The reason that the factor of improvement becomes smaller on this cluster is because this cluster has a much larger ratio of host CPU speed to NIC CPU speed and also a faster PCI-X bus. Thus the benefits from the reduced host involvement and I/O bus traffic are smaller.

2. **Scalability**

As the size of parallel system reaches thousands, it is important for parallel applications to be able to run over larger size systems and achieve corresponding parallel speedup.

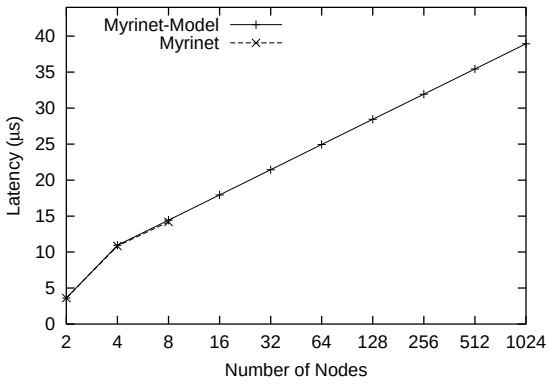


Figure 6.6: Modeling of the Barrier Scalability

This requires that the underlying programming models provide scalable communication, in particular, scalable collective operations. Thus it is important to find out how the NIC-based barrier operations can scale over larger size systems.

Since the NIC-based barrier operations with the dissemination algorithm exhibits a consistent behavior as the system size increases, we choose its performance pattern to model the scalability over different size systems. We formulate the latency for NIC-based barrier with the following equation.

$$T_{barrier} = T_{init} + (\lceil log_2\ N \rceil - 1) * T_{trig} + T_{adj}$$

In this equation, T_{init} is the average NIC-based barrier latency over two nodes, where each NIC only sends an initial barrier message for the entire barrier operation; T_{trig} is the average time for every other message the NIC needs to trigger when having received an earlier message; and T_{adj} is provided as the adjustment factor. The adjustment factor is needed to reflect the effects from other aspects of the NIC-based barrier, e.g., reduced PCI bus traffic and the overhead of bookkeeping. Through mathematical analysis, we have derived Myrinet NIC-based barrier latency as $T_{barrier} =$

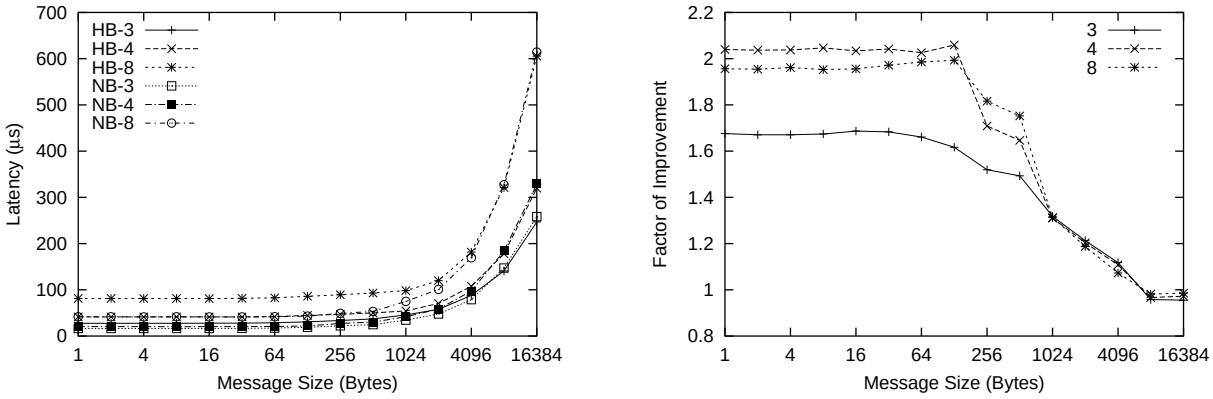
$3.60 + (\lceil log_2\, N \rceil - 1) * 3.50 + 3.84$ for 2.4GHz Xeon clusters with LANai-XP cards, and Quadrics NIC-based barrier latency as $T_{barrier} = 2.25 + (\lceil log_2\, N \rceil - 1) * 2.32 - 1.00$ for quad-700MHz clusters with Elan3 cards. As shown in Figure 6.6, the NIC-based barrier operations could achieve a barrier latency of 22.13μs and 38.94μs over a 1024-node Quadrics and Myrinet cluster of the same kinds, respectively. In addition, it indicates that the NIC-based barrier has potential for developing high performance and scalable communication subsystems for next generation clusters.

6.7. Results of NIC-Based Broadcast

Our modification to GM is done by leaving the code for other types of communications mostly unchanged. Our evaluation indicates that it has no noticeable impact on the performance of non-broadcast communications. Here we have discussed its performance results at both the GM-level and the MPI-level, as well as better tolerance to process skew of parallel programs.

1. **NIC-based multisend**

   We first evaluated the performance of the NIC-based multisend operation. Our tests were conducted by having the source node transmit a message to multiple destinations, and wait for an acknowledgment from the last destination. All destinations received the message from the source node, and none of them forwarded the message. The first 20 iterations were used to synchronize the nodes. Then the average for the next 10,000 iterations was taken as the latency. Figures 6.7(a) and 6.7(b) show the performance and the improvement of using the NIC-based multisend operation to transmit messages to 3, 4 and 8 destinations, compared to the same tests conducted using host-based multiple unicasts. For sending messages $\leq$ 128 bytes to 4 destinations, an improvement factor



(a) Latency

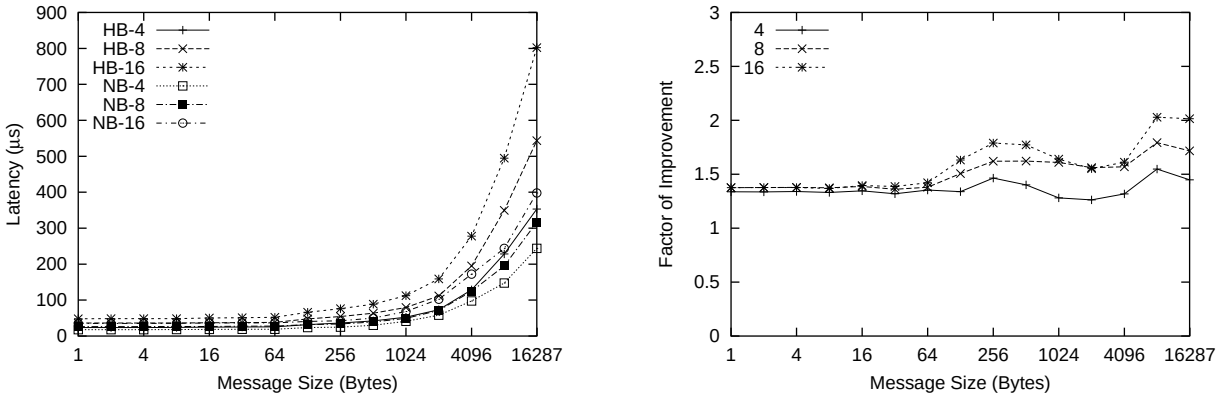
(b) The Performance Improvement

Figure 6.7: The performance of the NIC-based (NB) multisend operation, compared to Host-based (HB) multiple unicasts

up to 2.05 is achieved. This is due to the fact that the NIC-based broadcast was able to save repeated processing. As the message size gets larger, the improvement factor decreases and eventually levels off at a little below 1. This is to be expected because large message sizes leads to longer transmission time. With host-based multiple unicasts, the request processing is completely overlapped with the transmission of a previous queued packet, but there is still an overhead each time the packet header is changed with the NIC-based multisend.

2. **NIC-based Multicast**

We evaluated the performance of the broadcast with NIC-based forwarding using an optimal tree. Our tests were conducted by having the root initiate the NIC-based broadcast operation, and wait for an acknowledgment from one of the leaf nodes in the spanning tree. The first 20 iterations were used to synchronize the nodes. Then

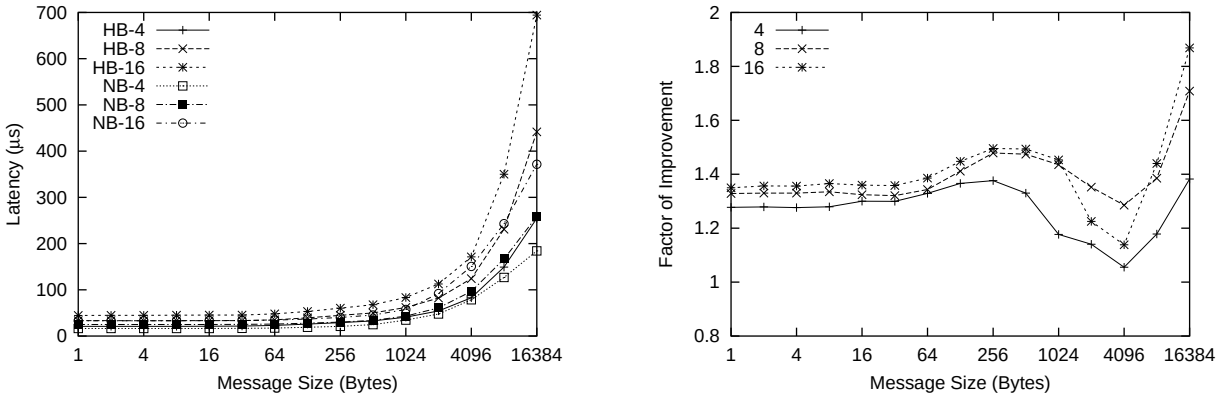


(a) Multicast Latency

(b) The Performance Improvement

Figure 6.8: The MPI-level performance of the NIC-based (NB) broadcast, compared to the host-based broadcast (HB), for 4, 8 and 16 node systems

10,000 iterations were timed to take the average latency. The same test was repeated with different leaf nodes returning the acknowledgment. The maximum from all the tests was taken as the broadcast latency. The traditional host-based broadcast was also evaluated in the same manner using the same version of GM as a comparison. Figures 6.9(a) and 6.9(b) show the performance of the NIC-based broadcast compared to the performance of host-based broadcast. For broadcasting messages $\leq$ 512 bytes on a 16 node system, the NIC-based broadcast achieves an improvement factor up to 1.48. Because multiple replicas of small messages can be sent out faster with the NIC-based broadcast, the optimal tree constructed for small messages has a larger average fan-out degree and so a shallower depth, compared to the same size binomial tree used in the traditional host-based broadcast. The average fan-out degree is the ratio as described in Section 6.4, and it imposes little impact on the latency. So the shallower depth reduces the broadcast latency significantly. As also shown in the figures, when



(a) Multicast Latency

(b) The Performance Improvement

Figure 6.9: The GM-level performance of the NIC-based (NB) broadcast, compared to the host-based broadcast (HB), for 4, 8 and 16 node systems

broadcasting a 16KB message on a 16 node system, the NIC-based broadcast achieves an improvement factor up to 1.86. This is due to the fact that, in the NIC-based broadcast, intermediate nodes do not have to wait for the arrival of the complete message to forward it. Thus the NIC-based broadcast achieves its performance benefits for the reduced intermediate host involvement and the capability of pipelining messages. Moreover, Figure 6.9(b) shows dips in the improvement factor curves when broadcasting 2KB and 4KB messages. The drop of improvement for these message is because these messages do not have the benefit for large multiple packet messages and also they do not have the benefit for small messages. The maximum packet size in GM is 4096 bytes, therefore $\leq$ 4096 byte messages do not benefit from message pipelining. On the other hand, since the NIC-based multisend does not have much improvement for these $\geq$ 1KB messages (See Figure 6.7(b)), the fan-out degree chosen in the optimal tree is about 1 and the shape of the resulted optimal tree is not significantly different from

the binomial tree used in the host-based approach. Therefore for these messages, the broadcast latency does not benefit much from the change of the spanning tree shape either. Taken together, the performance improvement is low for broadcasting these messages.

3. **MPI Level Broadcast**

   Since our modification to MPICH-GM only uses the NIC-based broadcast support for the eager mode message passing, the largest message that uses the NIC-based broadcast is the largest eager mode message, which is 16,287 bytes. We measured the broadcast latency at the MPI level in the same manner as that at the GM level. The maximum latency obtained was taken as the broadcast latency. Figures 6.8(a) and 6.8(b) show the latency performance and the improvement factor of the NIC-based broadcast at the MPI level, respectively. We observed an improvement factor of up to 2.02 for broadcasting 8KB messages over 16 node system. Also the trend of the performance improvements are similar to the trend at the GM-level (Figure 6.9(b)). However, when broadcasting 16,287 byte messages, there is a dip in the improvement factor curve. That is due to the larger cost of copying the data to their final locations. So the broadcast latency for a 16,287 byte message with the NIC-based broadcast is comparatively high, which leads to a lower improvement factor.

4. **Tolerance to Process Skew**

   Another major benefit of the NIC-based broadcast is the tolerance to process skew. Typically, with the blocking implementation of MPI_Bcast, the host CPU time, the time spent on performing the MPI_Bcast, becomes larger if a process is delayed at an intermediate node. In reality, all processes skew at random. Some processes call

MPI_Bcast before the root node does, and others do after the root node. The effects of the former can not be reduced by a broadcast operation, but those from the latter can be reduced if possible, because all the processes that have called MPI_Bcast inevitably have to wait for the root process. We evaluate the effects of the delayed processes, relative to the root processes, to the average host CPU time. We measure the average host CPU time to perform the MPI_Bcast with varying amount of process skew. All the processes are first synchronized with a MPI_Barrier. Then each process, except the root, chooses a random number between the negative half and the positive half of a maximum value as the amount of skew they have. The processes with a positive skew time perform computation for this amount of skew time before calling the MPI_Bcast operation. The average host CPU time from 5,000 iterations was plotted against the average process skew. The average skew is defined as the expected skew between two arbitrary non-root processes, which can be considered as the expected distance between two random distributed points on a given interval $[\,0,\, max\,]$, that is, $\frac{1}{3}max$ (See [74], page 118).

Figure 6.10 shows the average host CPU time for MPI_Bcast over 16 nodes with varying amount of average skew. Figure 6.10(a) shows the average host CPU time for broadcasting small messages (2, 4 and 8 bytes) over 16 nodes with varying amount of average skew. The NIC-based broadcast has much smaller host CPU time compared to the host-based broadcast. With a skew under 40μs, the host CPU time decreases using either approach. This is to be expected because a small amount of skew time can overlap with some of the message broadcasting time. When the skew goes beyond 40μs, the host CPU time increases with the host-based approach, while it decreases with the NIC-based approach. This is to be expected. As the skew increases, more

intermediate processes get delayed. With the host-based approach, more processes wait longer for their ancestors to call MPI_Bcast and forward the messages, which results in longer average host CPU time. In contrast, with the NIC-based approach, the delayed intermediate processes does not prevent their children from receiving the message and, on the other hand, their delay have more overlap with the message transmission time. which leads to less average host CPU time. Figure 6.10(b) shows that the improvement factor of the NIC-based approach over the host-based approach for small messages. With an average skew of 400μs, the NIC-based broadcast achieves an improvement factor up to 5.82. We also observed that the improvement factor becomes greater as the skew increases. When broadcasting large messages (2KB to 8KB), a similar trend of benefits on average host CPU time is also observed when comparing the NIC-based broadcast to the host-based broadcast. Figure 6.10(c) shows the similar trend of average host CPU time for large messages (2KB, 4KB and 8KB). Figure 6.10(d) shows that the factor of improvement of the NIC-based approach over the host-based approach for large messages. When there is no skew, the host CPU time is actually similar to the performance of MPI_Bcast at the MPI-level, as shown in Figure 6.8(b). As the skew increases, the benefits from the reduced skew become more pronounced and greater. For the larger cost of coping larger messages, the improvement factors for larger messages are smaller. With an average skew of 400μs, the NIC-based approach achieves an improvement factor up to 2.9 for 2KB messages, up to 2.22 for 4KB messages and up to 2.01 for 8KB messages.

We also evaluated the effect of process skew on the average host CPU time for different size systems. Figure 6.11 shows the factors of improvement on the host CPU time for broadcasting 4 byte and 4KB messages using the NIC-based broadcast compared to

the host-based broadcast, over systems of different sizes. For both sizes of messages, the improvement factor becomes greater as the system size increases for a fixed amount of process skew of 400μs. This suggests that a larger size system can benefit more from the NIC-based broadcast for the reduced effects of process skew.

6.8. Results of NIC-Based All-to-All Broadcast

The NIC-based all-to-all broadcast performance is measured as the time between when an all-to-all broadcast request is sent to the NIC and when its completion is detected through a receive event. A microbenchmark is used to measure the average time of 5,000 iterations after the first 20 warm-up iterations. The bandwidth is taken as the total number of bytes broadcast by each process divided by the time to perform an all-to-all broadcast operation. From the two typical host-based all-to-all broadcast algorithms, our experiments indicate that ring-based pipelining provides better performance than the does recursive doubling. Thus we choose the recursive doubling algorithm in our microbenchmark for measuring host-based all-to-all broadcast performance.

1. **Latency and Bandwidth**

   Figure 6.12 shows the latency comparisons between the host-based all-to-all broadcast operation and the NIC-based all-to-all broadcast with the concurrent broadcasting (NIC-CB) and recursive doubling (NIC-RD) algorithms. As shown in Figure 6.12(a), for small messages, NIC-RD provides the best performance, compared to NIC-CB or the host-based all-to-all broadcast. The reason is that the NIC-RD algorithm can enjoy the benefits of fast communication processing at the NIC, while, at the same time, it is able to combine small messages into larger packets and does not suffer from having to process a linear scaling number of packets from all the other processes, as is the

case for NIC-CB. In contrast, for large messages, the NIC-CB algorithm provides the best performance compared to NIC-RD or the host-based all-to-all broadcast. The reason is that the combined messages can no longer fit into a single MTU and they are still fragmented into packets at the NIC, so the recursive doubling algorithm no longer benefits from message combining, whereas NIC-CB still benefits from fast forwarding and retransmitting of the received (or transmission completed) packets. Both NIC-based algorithms perform better than the host-based all-to-all broadcast for large messages. Figure 6.12(b) shows the latency comparisons for large messages. NIC-based algorithms can improve performance by a factor of 3.

Figure 6.13 shows the bandwidth comparisons between the host-based and the NIC-based all-to-all broadcast algorithms. The bandwidth performance for the host-based all-to-all broadcast drops with multi-packet messages (greater than 4 KB). In contrast, the NIC-based all-to-all broadcast operations are able to sustain their bandwidth performance. Compared to NIC-RD, NIC-CB can achieve better bandwidth with the benefits from fast packets forwarding and retransmitting. Figure 6.14 shows the performance improvement factor of the NIC-based all-to-all broadcast algorithms. Over the 16-node cluster with LANai 9.1 cards, the concurrent broadcasting algorithm provides an improvement factor of up to 3.01 for large messages, and the recursive doubling algorithm provides an improvement factor of up to 1.54 for small messages.

2. **Scalability**

Figure 6.15 shows the scalability comparisons between the host-based and the NIC-based all-to-all broadcast algorithms with 4 byte small messages and 4 KB large messages. For small messages, NIC-RD provides the best scalability because it benefits

from offloaded communication processing and combining of small messages into larger packets. In contrast, NIC-CB performs the worst as the system size increases. The reason is that it does not combine small messages and hence the communication processing time for a large number of packets dominates over its benefits of message forwarding. These results are shown in Figure 6.15(a). For large messages, the NIC-CB algorithm provides the best scalability because it has the benefits the communication offloading and also the benefits of fast message forwarding. The other two algorithms do not have these features and have lower scalability, while the NIC-based all-to-all broadcast with recursive doubling still provides better scalability than the host-based all-to-all broadcast because of communication offloading. These results are shown in Figure 6.15(b).

3. **Host CPU Utilization**

One of the major benefits of NIC-based all-to-all broadcast is that it has low host CPU utilization. With host-based all-to-all broadcast, the host process must constantly poll for the arrival of messages and trigger the next step for the all-to-all broadcast operation. The host CPU is largely occupied during the all-to-all broadcast operation. Figure 6.16(a) shows the host CPU utilization of host-based all-to-all broadcast over 16 nodes. In contrast, with the NIC-based all-to-all broadcast, once the host process sends its request to the NIC, it is free to perform other useful computation. To determine the host CPU utilization for the NIC-based all-to-all broadcast, we measured the effective time that the host CPU spends on posting the all-to-all broadcast request to the NIC and processing the arrival of a all-to-all broadcast receive event. Figure 6.16(b) shows the average host CPU utilization for the NIC-based all-to-all broadcast operations with both the concurrent broadcasting and the recursive doubling algorithms over various

numbers of nodes. For the NIC-based all-to-all broadcast algorithms, the host CPU utilization is mostly constant over various numbers of nodes and different sizes of messages. The reason is that the time to post a send request to the NIC and the time to process a receive event does not vary with respect to the change of message size or system size. Note that for small messages the CPU utilization is higher. The reason is that for small messages the last-received data is DMAed into the receiver queue along with the event, to improve the overall performance. The host process thus has to perform an additional memory copy to obtain the message. Also note that user applications can perform nonblocking NIC-based all-to-all broadcast, where processes do not wait for the result from the NICs after they have posted the requests. Instead, these processes can perform useful computation that does not depend on the results, and they read the result from the NICs only when they need the data. Therefore, the NIC-based all-to-all broadcast allows user applications to achieve high utilization of the computation resources with its low CPU utilization.

6.9. Summary of NIC-Based Collective Operations over Myrinet

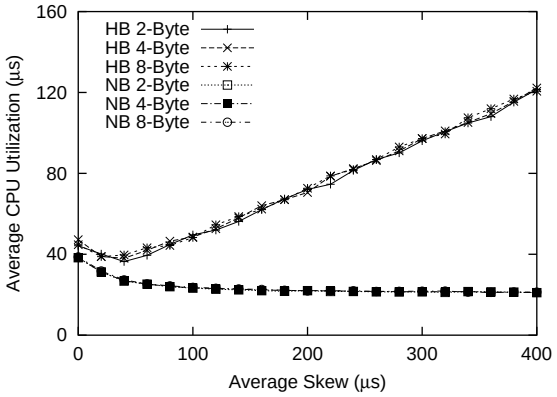
We have characterized general concepts and the benefits of the NIC-based barrier algorithms on top of point-to-point communication. We have then examined the communication processing for point-to-point operations, and pinpointed the relevant processing we can reduce for collective operations. Accordingly we have proposed a general scheme for an efficient NIC-based barrier operation over Myrinet. Our evaluation has also shown that, over a 16-node Myrinet cluster with LANai 9.1 cards, the NIC-based barrier operation achieves a barrier latency of 25.72us, which is a 3.38 factor of improvement compared to the host-based algorithm. Furthermore, our analytical model suggests that NIC-based barrier operations

could achieve a latency of 22.13μs and 38.94μs, respectively over a 1024-node Quadrics and Myrinet cluster.

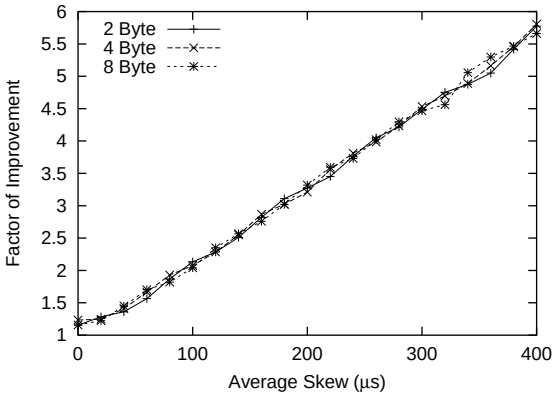
By using a NIC-based multisend mechanism, which can enable the transmission of multiple message replicas to different destinations, and a NIC-based forwarding mechanism, which allows intermediate NICs to forward the received packets without intermediate host involvement, we have designed an efficient and reliable NIC-based broadcast operation. At the GM-level, the NIC-based broadcast scheme provides an improvement factor up to 1.86 for 16KB messages and an improvement factor up to 1.48 for $\leq$ 512 byte messages over 16 nodes compared to the traditional host-based broadcast. At the MPI-level, the NIC-based broadcast achieves an improvement factor up to 2.02 for 8KB messages, and an improvement factor up to 1.78 for small messages $\leq$ 512 bytes over 16 nodes. In addition, at the MPI-level, the NIC-based broadcast was shown to have better tolerance to process skew.

We have also designed scalable, high-performance NIC-based all-to-all broadcast with concurrent broadcasting and recursive doubling algorithms. The resulting NIC-based operations have been implemented and incorporated into a NIC-based collective protocol [93] over Myrinet/GM. Compared to the host-based all-to-all broadcast, the NIC-based all-to-all broadcast operations improves all-to-all broadcast bandwidth over 16 nodes by a factor of 3. The NIC-based all-to-all broadcast with recursive doubling algorithm is more scalable for small messages, and the concurrent broadcasting algorithm more scalable for large messages. Furthermore, the NIC-based all-to-all broadcast operations have very low host CPU utilization, which allows user applications to achieve high CPU utilization when the operation is used in a nonblocking manner.

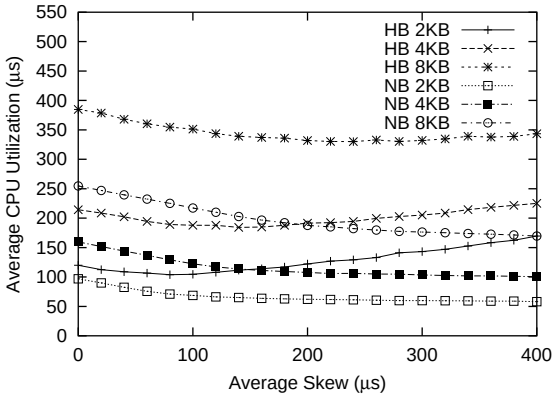
All the above NIC-based collective algorithms are designed to achieve their reliability and efficiency without using a centralized manager and requires minimum memory and processor resources at the NIC, which promises good scalability.



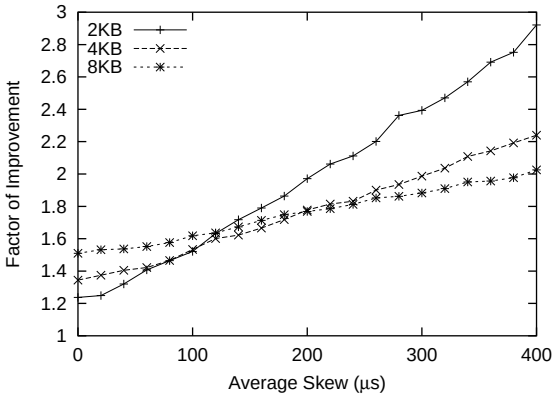
(a) Average Host CPU Time



(b) Improvement Factors



(c) Large Messages



(d) Improvement for Large Messages

Figure 6.10: Average host CPU time on performing MPI_Bcast under different amount of average skew with both the host-based approach (HB) and the NIC-based (NB) approach

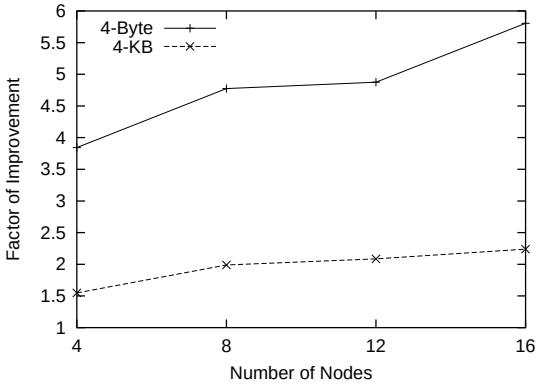
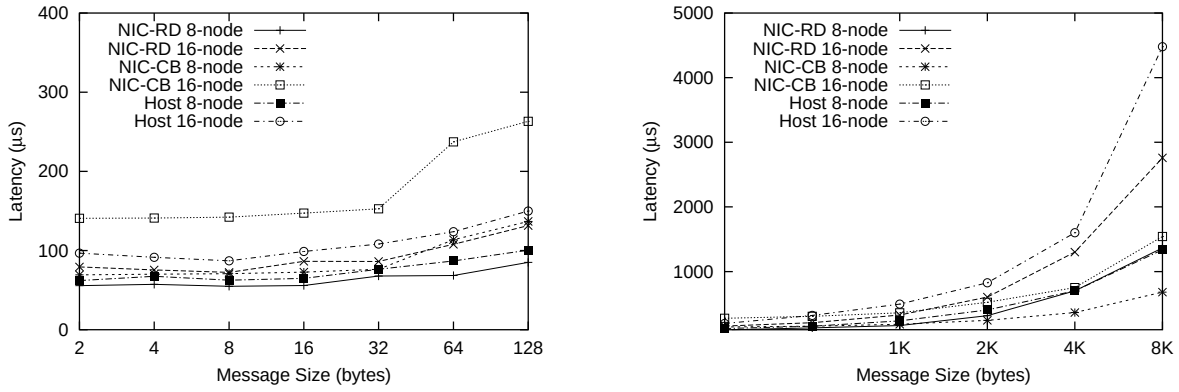


Figure 6.11: The effect of process skew for systems of different sizes



(a) Small Messages

(b) Large Messages

Figure 6.12: All-to-All Broadcast Latency Comparisons of NIC-Based Operations with the Concurrent-Broadcasting Algorithm (CB) and the Recursive-Doubling (RD) Algorithm to Host-Based operations

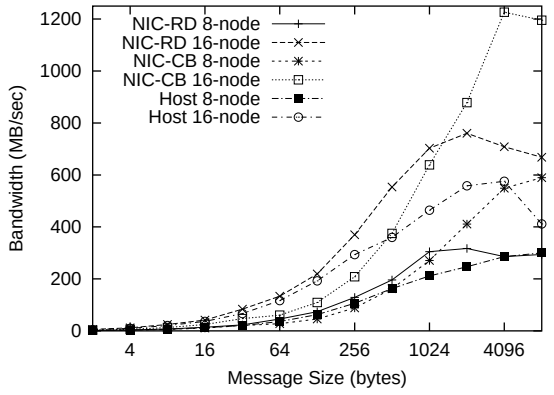


Figure 6.13: Bandwidth Comparisons of NIC-Based All-to-All Broadcast with the Concurrent Broadcasting Algorithm (CB) and the Recursive Doubling Algorithm (RD) to Host-Based All-to-All Broadcast

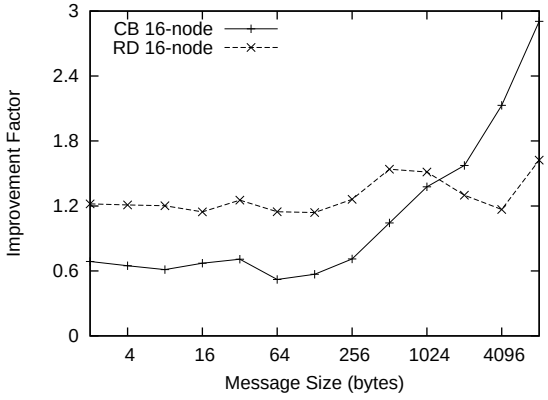
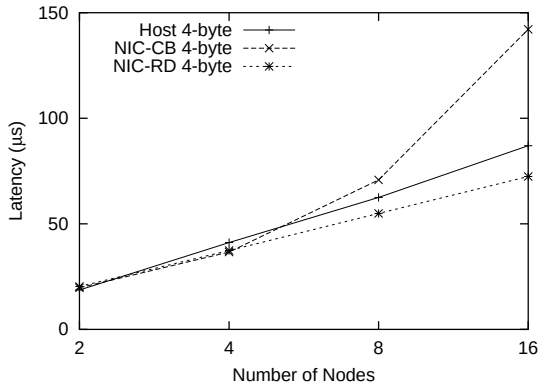
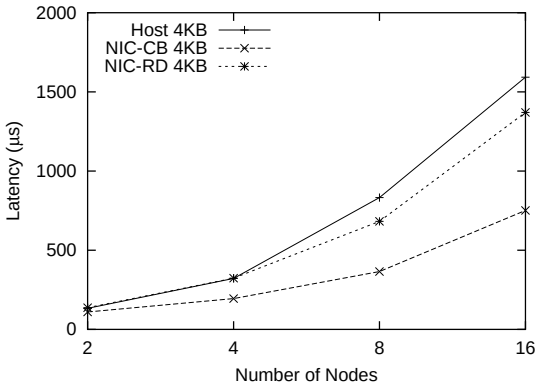


Figure 6.14: Improvement Factor for NIC-Based All-to-All Broadcast with the Concurrent Broadcasting Algorithm (CB) and the Recursive Doubling Algorithm (RD)

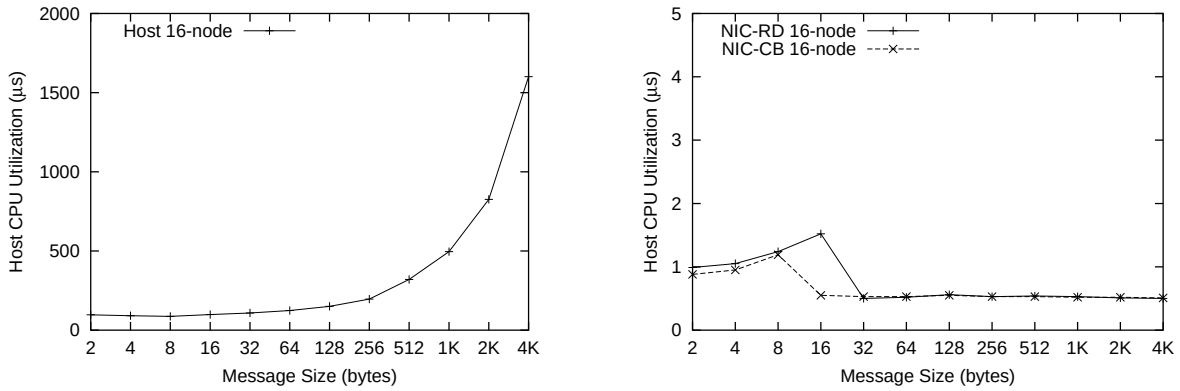


(a) Small Messages



(b) Large Messages

Figure 6.15: Scalability Comparisons of the NIC-Based All-to-All Broadcast with Concurrent Broadcasting Algorithm (CB) and Recursive Doubling Algorithm (RD) to the Host-Based All-to-All Broadcast



(a) Host-Based All-to-All Broadcast

(b) NIC-Based All-to-All Broadcast

Figure 6.16: Host CPU Utilization Comparison of the NIC-Based All-to-All Broadcast with the Concurrent Broadcasting Algorithm (CB) and the Recursive Doubling (RD) Algorithm to the Host-Based All-to-All Broadcast

CHAPTER 7

High Performance Parallel IO Support over Quadrics

The gap between computer processing power and disk throughput is becoming wider as the growth of the latter continuously lags behind that of the former [68]. Large I/O-intensive applications on ultra-scale clusters demand increasingly higher I/O throughput. Correspondingly, scalable parallel I/O needs to be available for these real world applications to perform well. Both commercial [42, 44, 30] and research projects [61, 40, 2] have been developed to provide parallel file systems for I/O accesses on such architectures. Among them, the Parallel Virtual File System 2 (PVFS2) [2] has been created with the intention of addressing the needs of next generation systems using low cost Linux clusters with commodity components.

On the other hand, high performance interconnect technologies such as Myrinet [12], InfiniBand [43], and Quadrics [7] not only have been deployed into large commodity component-based clusters to provide higher computing power, but also have been utilized in commodity storage systems to achieve scalable parallel I/O support. For example, the low-overhead high-bandwidth user-level communication provided by VI [99], Myrinet [64], and InfiniBand [90] has been utilized to parallelize I/O accesses to storage servers and increase the performance of parallel file systems. Quadrics Interconnects [73, 7] provides very low latency ($\leq 2\mu$s) and high bandwidth. It also supports many of the cutting-edge communication features, such as OS-bypass user-level communication, remote direct memory access (RDMA), as well as

hardware atomic and collective operations. Moreover, Quadrics network interface provides a programmable network co-processor, which offloads much of the communication processing down to the network interface and contributes greatly to its efficient point-to-point and collective communication. These salient features and their performance advantages of Quadrics have not been leveraged to support scalable parallel IO throughput at the user-level, though some of these modern features, like RDMA, are exploited in other interconnects, such as Myrinet [64] and InfiniBand [90]. Currently, some distributed file systems that exploit the advantages of Quadrics are developed on top of Quadrics kernel communication library, e.g., Lustre [27]. But this approach incurs higher network access overhead because the operating system is included in the communication path. In addition, as a distributed file system Lustre is designed to scale the aggregated bandwidth for accesses to files on different servers, while parallel file accesses from a single parallel job cannot directly take its maximum benefits. For example, concurrent writes from multiple processes in a single parallel job cannot benefit with Lustre. A typical platform may utilize a parallel file system such as PFS [44] to export scalable bandwidth to a single job by striping the data of a single parallel file system over multiple underlying file systems such as Lustre. However, the extra multiplexing process adds more to the cost in the path of IO accesses.

We have examined the feasibility of supporting parallel file systems with Quadrics user-level communication and RDMA operations. PVFS2 [2] is used as a parallel file system in this work. We first characterize the challenges of supporting PVFS2 on top of Quadrics interconnects, focusing on: (a) constructing a client-server model over Quadrics at the user-level, (b) mapping an efficient PVFS2 transport layer over Quadrics, and (c) optimizing the performance of PVFS2 over Quadrics such as efficient non-contiguous communication support. Accordingly, we implement PVFS2 over Quadrics by taking advantage of Quadrics

RDMA and event mechanisms. We then evaluate the implementation using PVFS2 and MPI-IO [54] benchmarks. The performance of our implementation is compared to that of PVFS2 over TCP. Quadrics IP implementation on top of its interconnect is used in the TCP implementation to avoid network differences. Our work demonstrates that: (a) a client/server process model necessary for file system communication is feasible with Quadrics interconnects; (b) the transport layer of a parallel file system can be efficiently layered on top of Quadrics; and (c) the performance of PVFS2 can be significantly improved with Quadrics user-level protocols and RDMA capabilities. Compared to a PVFS implementation over TCP/IP over Quadrics, our implementation increases the aggregated read performance of PVFS2 by 140%. It is also able to deliver significant performance improvement in terms of IO access to application benchmarks such as mpi-tile-io [75] and BT-IO [88]. To the best of our knowledge, this is the first work in the literature to report the design of a high performance parallel file system over Quadrics user-level communication protocols.

7.1. Related Work for Parallel IO over Quadrics

Previous research have studied the benefits of using user-level communication protocols to parallelize IO accesses to storage servers. Zhou et. al. [99] have studied the benefits of VIA networks in database storage. Wu et. al. [90] have described their work on InfiniBand over PVFS1 [64]. DeBergalis et. al. [31] have further described a file system, DAFS, built on top of networks with VIA-like semantics. Our work is designed for Quadrics Interconnects over PVFS2 [2].

Models to support client/server communication and provide generic abstractions for transport layer over different networks have been described in [98, 50, 25]. Our work explores

the ways to overcome Quadrics static process/communication model and optimize the transport protocols with Quadrics event mechanisms. Ching et. al [26] have implemented list IO in PVFS1 and evaluated its performance over TCP/IP. Wu et. al [91] have studied the benefits of leveraging InfiniBand hardware scatter/gather operations to optimize non-contiguous IO access in PVFS1. Our work exploits a communication mechanism with a single event chained to multiple RDMA to support zero-copy non-contiguous network IO over Quadrics.

7.2. Challenges in Designing PVFS2 over Quadrics/Elan4

Little is known about how to leverage Quadrics high performance user-level communication to support high performance parallel file system. In this section, we provide a brief overview of PVFS2 and discuss challenging issues in designing PVFS2 over Quadrics/Elan4.

7.2.1. Overview of PVFS2

PVFS2 [2] is the second generation parallel file system from the Parallel Virtual File System (PVFS) project team. It incorporates the design of the original PVFS [64] to provide parallel and aggregated I/O performance. A client/server architecture is designed in PVFS2. Both the server and client side libraries can reside completely in user space. Clients communicate with one of the servers for file data accesses, while the actual file IO is striped across a number of file servers. Metadata accesses can also be distributed across multiple servers. Storage spaces of PVFS2 are managed by and exported from individual servers using native file systems available on the local nodes. More information about PVFS2 can be found in [2].

7.2.2. Challenges for Enabling PVFS2 over Quadrics

PVFS2 provides a network abstraction layer to encapsulate all the functionalities needed for communication support. The resulting component is called Buffered Message Interface (BMI), which interacts with other components in the software architecture to support low-level IO accesses. Figure 7.1 shows a diagram of PVFS2 components on both the client side and the server side. As shown in the figure, BMI functionalities can be further classified into three categories: connection management between processes, message passing activities for interprocess communication (IPC) and the memory management needed for IPC. In particular, Quadrics user-level programming libraries has a unique design for running Higher Performance Computing (HPC) applications. All parallel jobs over Quadrics need to start from a static pool of application processes [73]. This is rather incompatible to the needs of file systems, which start servers first and deliver IO services to incoming clients. In addition, PVFS2 interprocess communication between servers and clients needs to be properly layered over Quadrics communication mechanisms to expose the best capability of Quadrics hardware. In this work, we take on the following issues to design PVFS2 over Quadrics: (a) constructing a client/server communication model in terms of connection management, (b) designing PVFS2 basic transport protocol to appropriate Quadrics communication mechanisms for message transmission, and (c) optimizing PVFS2 performance over Quadrics.

7.3. Designing a Client/Server Communication Model

As described in Section 7.2.1, PVFS2 is designed as a client/server architecture. In contrast, a parallel job over Quadrics libraries runs as a static pool of application processes [73]. All of these processes join or leave the Quadrics network in a synchronized manner. In addition, to facilitate this process, Quadrics requires a resource management framework such as

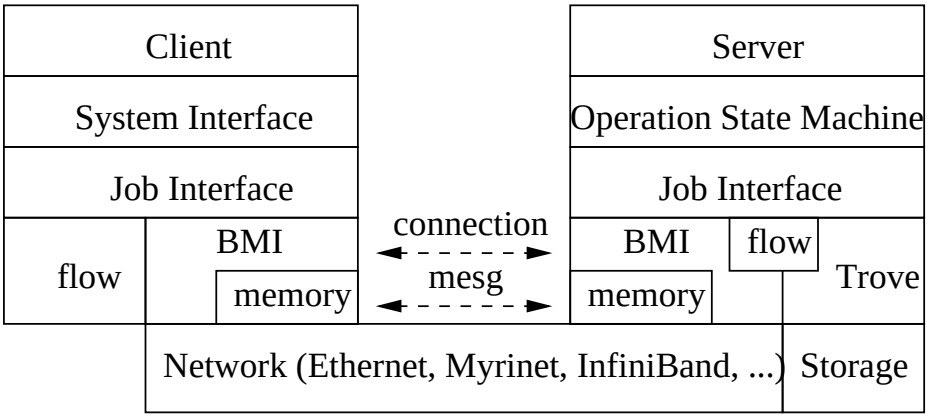


Figure 7.1: The Architecture of PVFS2 Components

RMS [73] to launch the parallel applications. To provide a PVFS2 client/server architecture over Quadrics, it is necessary to break the model of static process pool used in Quadrics parallel jobs and eliminate the need of a resource management framework.

7.3.1. Allocating a Dynamic Pool of Processes over Quadrics

Each process must acquire a unique Virtual Process ID (VPID) and use it as an identity for network addressing before the communication starts over Quadrics. VPID is an abstract representation of Quadrics capability, which describes the network node ID and context ID owned by a process, and the range of network nodes and the range of contexts all processes have. Typically, Quadrics utilizes RMS [73] to allocate appropriate capabilities for all application processes before launching a parallel job. The capabilities from all processes share the same range of network nodes and the same range of contexts. Together with the network node ID and a context ID, each process can determine its VPID based on the capability. In this way, a static pool of application processes is launched over Quadrics network.

To allocate a dynamic pool of processes over Quadrics, we change the aforementioned allocation scheme. First, we expand the range of nodes to include every node in the network. Second, a large range of contexts is provided on each node. Table 7.1 shows the format of Elan4 capability for all PVFS2 processes. On each node, the first context is dedicated to the server process, if present, and the rest of the contexts are left for the client processes. The VPID is needed to identify an elan4 process is calculated with this formula: $node\_id*(j-i+1)+(ctx-i)$. A client process obtains the corresponding parameters from the PVFS2 fstab entry as shown on the third row of Table 7.1. Clients connect to a server on a dynamic basis, and notify the server when they leave. Servers allocate communicating resources as new clients join in, and deallocate when they disconnect or timeout. There are no restrictions for processes to synchronize memory allocation and synchronized startup.

Setting	Value
Capability	$node\{0..N\}ctx\{i..j\}$
VPID	$node\_id*(j-i+1)+(ctx-i)$
fstab	elan4://server_id:server_ctx/pvfs2-fs

Table 7.1: Elan4 Capability Allocation for Dynamic Processes

7.3.2. Fast Connection Management

A process over Quadrics needs to know both the VPID and an exposed memory location of a remote process before sending a message. Parallel jobs built from default Quadrics libraries, typically use a global memory address to initiate communication because the memory allocation is synchronized and a global virtual memory [73] is available. Without the use

of a global memory, we design two different schemes for clients to initiate communication to PVFS2 servers over Quadrics/Elan4. Initially, a basic scheme utilizes TCP/IP-based socket. A server opens a known TCP port and polls for incoming communication requests from time to time. Clients connect to this known TCP port and establish a temporal connection to exchange VPID and memory addresses.

Because establishing and tearing down the connections between clients and servers is so common for file I/O services, it is desirable to design a fast connection management scheme to achieve scalable IO access. In another scheme, we use native communication over Quadrics for communication initiation. All servers start with a known node ID and context ID, which together determine the VPID according to the allocation scheme described earlier earlier in this section. A set of receive queue slots are also allocated, which start at a unified memory address across all the servers. Servers then poll on this receive queue for new connection requests (and also IO service), using a Queue-based Direct Memory Access model. The QDMA model is described in more detail in Section 7.4.1. Because this memory for the receive queue (a portion of the NIC memory mapped to the host address space) is allocated dynamically at run time in the current Quadrics implementation, one constraint here is that the server needs to report its address at the startup time. We pass this address to clients as an environmental parameter. Further investigation will study the feasibility and impact of mapping Quadrics NIC memory to a fixed memory space.

As shown in Figure 7.2, a client process that is initiating the connection with a server. The client obtains the VPID of the server based on the pvfs2 fstab file and the memory address of the server's receive queue through an environmental variable, `SERVER_ADDR`. Using the known memory address and the known VPID, a client can initiate a message to the server, which includes its own VPID and address of its exposed memory location. When a

connection is initiated, the corresponding network addressing information is recorded into a global address list. Lists to record all the outstanding operations are also created. This address information and associated resources are removed when a connection is finalized as if no connection has been established earlier.

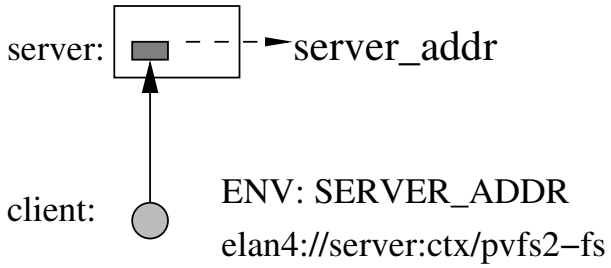


Figure 7.2: Connection Initiation over Native Elan4 Communication

7.4. Designing PVFS2 Basic Transport Layer over Quadrics/Elan4

All PVFS2 [2] network message transmission functionalities are included in the BMI interface [25]. Two models of message transmission, matched and unexpected, are included. All network operations are designed in a nonblocking manner to allow multiple of them in service concurrently. Several test APIs are specified for the completion of outstanding messages. Quadrics provides two basic interprocess communication models: Queue-based Direct Memory Access (QDMA) and Remote Direct Memory Access (RDMA) [73]. QDMA can only transmit messages up to 2KB. The other model, RDMA read/write, supports transmission of arbitrary messages over Quadrics network. Using these two models, the transport layer of PVFS2 over Quadrics/Elan4 is designed with two protocols, eager and rendezvous, to handle different size messages.

7.4.1. Short and Unexpected Messages with Eager Protocol

The QDMA model allows a process to check incoming QDMA messages posted by any process into its receive queue. An eager protocol is designed with this model to transmit short and unexpected messages. As mentioned in Section 7.3.2, this QDMA model is used in initiating dynamic client/server connection scheme with Quadrics native communication.

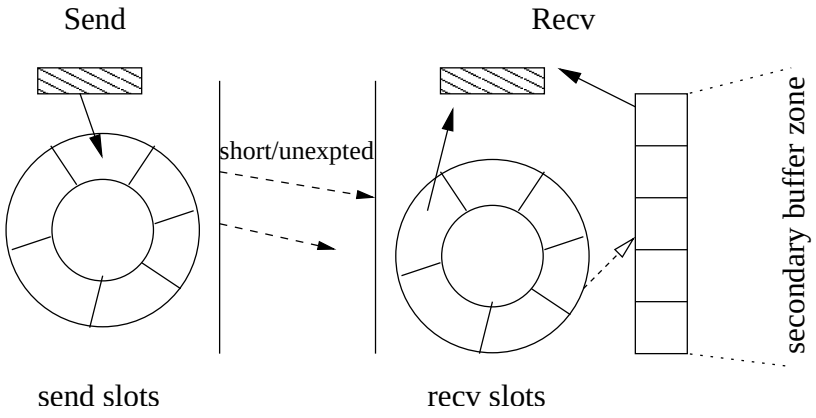


Figure 7.3: Eager Protocol for Short and Unexpected Messages

As shown in Figure 7.3, in the eager protocol, a number of sender buffers are allocated on the sender side to form a send queue, and a fixed number of receive queue slots are created on the receiver side to form a receive queue. In addition, a secondary receive buffer zone is created with another set of receive buffers. The number of receive buffers in the secondary zone can grow or shrink on an on-demand basis. In this eager protocol, a new message is first copied into a sender queue slot, sent over the network, and eventually received into a receive queue slot. If the message is an unexpected message, it is then copied into a receiver buffer immediately without waiting for a matching receive operation to be posted. The receive queue slot is then recycled to receive new messages. For a message that needs

to be matched, it remains in the receive queue slot until a matching receive operation is posted. This can save an extra message copy if the operations is posted in time. However, if the number of receive queue slots becomes low under various situations, these messages are copied into the receive buffers in the secondary buffer zone to free up receive slots for more incoming messages. When the messages are eventually matched, the receive buffers are also recycled into the secondary buffer zone. If there are relatively a large number of free receive buffers in the secondary zone, they are deallocated to reduce the memory usage.

7.4.2. Long Messages with *Rendezvous* Protocol

Quadrics RDMA (read/write) communication model can transmit arbitrary size messages [73]. A *rendezvous* protocol is designed with this model for long messages. Two schemes are proposed to take advantage of RDMA read and write, respectively. As shown in Figure 7.4 left diagram, RDMA write is utilized in the first scheme. A *rendezvous* message is first initiated from the sender to the receiver in both schemes. The receiver returns an acknowledgment to the sender when it detects a *rendezvous* message. The sender then sends the full message with a RDMA write operation. At the completion of RDMA write, a control fragment, typed as FIN, is sent to the receiver for the completion notification of the full message. The right diagram in Figure 7.4 shows the second scheme with RDMA read. When the *rendezvous* message arrives at the receiver, instead of returning an acknowledgment to the sender, the receiver initiates RDMA read operations to get the data. When these RDMA read operations complete, a different control message, typed as FIN_ACK, is sent to the sender, both for acknowledging the arrival of the earlier *rendezvous* fragment and notifying the completion of the whole message.

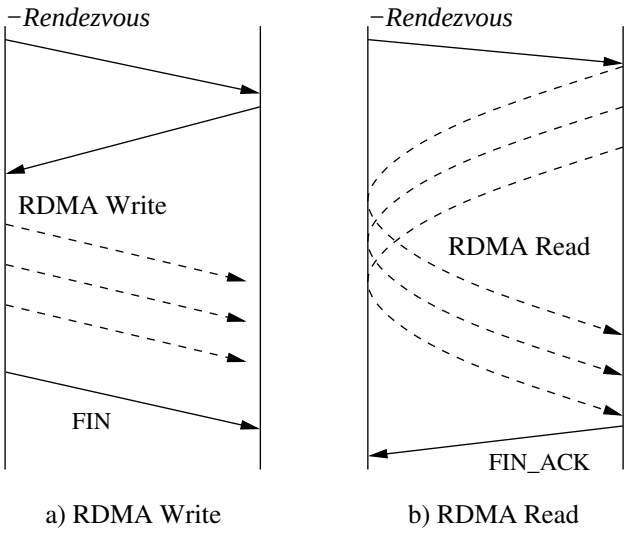


Figure 7.4: *Rendezvous* Protocol for Long Messages

7.5. Optimizing the Performance of PVFS2 over Quadrics

To improve the basic design discussed in Section 7.4, we have explored several further design issues including zero-copy non-contiguous network IO access, adaptive *rendezvous* message transfer with RDMA read/write and optimization on completion notification.

7.5.1. Adaptive *Rendezvous* with RDMA Read and RDMA Write

As discussed in Section 7.4.2, RDMA read and write are both utilized in the *rendezvous* protocol. This achieves zero-copy transmission of long messages. File systems, such as DAFS [31], also take advantage of similar RDMA-based message transmission. Typically the server decides to use RDMA read or write based on whether the client is performing a read or write operation: a read operation is implemented as RDMA write from the server, and a write operation as a RDMA read. Thus a server process can be potentially overloaded with a large number of outstanding RDMA operations, which can lead to suboptimal

performance due to the bandwidth drop-off [13]. Therefore a basic throttling mechanism is needed to control the number of concurrent outstanding RDMA operations. We introduce an adaptive throttling mechanism to regulate the number of RDMA operations at any given time. Under a light load, the sender initiates a long message operation to the receiver, which in turn pulls the message from the sender through RDMA read. If either side has a large number of outstanding RDMA operations, the receiver gathers this knowledge from its local communicate state and the information passed from the sender in the initial *rendezvous* packet. When the receiver is heavily loaded, it notifies the sender accordingly and the sender completes the operation through RDMA write. For the reason of fairness to multiple clients, if both the sender and the receiver are heavily loaded, the priority is given to the server and have the client carry out the RDMA operations. It does not matter whether it is a sender or a receiver. Table 7.2 provides a sample receiver's decision table for the RDMA *rendezvous* protocol.

load: sender <> receiver	loaded server?	RDMA
greater	yes	write
greater	no	read
equal	yes	write
equal	no	read
less	yes	write
less	no	read

Table 7.2: Receiver's Decision Table for Adaptive RDMA *Rendezvous* Protocol

7.5.2. Optimizing Completion Notification

Event mechanisms that enable both local and remote completion notification are available in Quadrics communication models. In particular, this mechanism can be used to enable notification of message completion along with RDMA read/write operations. In the *rendezvous* protocol, so long as the control information contained in the last control message is available to the remote process, the completion of a full message can be safely notified through an enabled remote event. We have designed this as an optimization to the *rendezvous* protocol. A sender process allocates a completion event and encodes the address of this event in the first *rendezvous* message. When the receiver pulls the message via RDMA read, it also triggers a remote event to the sender using the provided event address. Similarly, in the case of RDMA write, the receiver provides the address of such an event in its acknowledgment to the sender. The receiver detects the completion of a full message through the remote event triggered by a RDMA write operation. In either case, both sides notice the completion of data transmission without the need of an extra control message.

7.6. Designing Zero-Copy Quadrics Scatter/Gather for PVFS2 List IO

Noncontiguous IO access is the main access pattern in scientific applications. Thakur et. al. [82] noted that it is important to achieve high performance MPI-IO with native noncontiguous access support in file systems. PVFS2 provides list IO interface to support such noncontiguous IO accesses. Figure 7.5 shows an example of noncontiguous IO with PVFS2. In PVFS2 list IO, communication between clients and servers over noncontiguous memory regions are supported over list IO so long as the combined destination memory is larger than the combined source memory. List IO can be built on top of interconnects with native

scatter/gather communication support, otherwise, it often resorts to memory packing and unpacking for converting noncontiguous memory fragments to contiguous memory. An alternative is to perform multiple send and receive operations. This can lead to more processing and more communication in small data chunks, resulting in performance degradation.

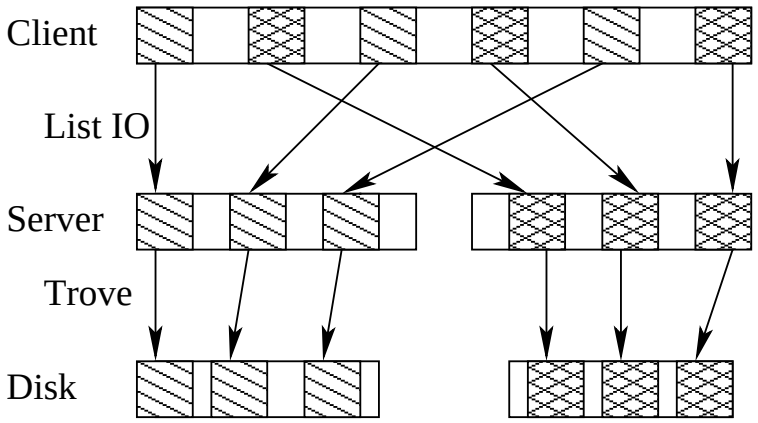


Figure 7.5: An Example of PVFS2 List IO

There is a unique chain DMA mechanism over Quadrics. In this mechanism, one or more DMA operations can be configured as chained operations with a single NIC-based event. When the event is fired, all the DMA operations will be posted to Quadrics DMA engine. Based on this mechanism, the default Quadrics software release provides noncontiguous communication operations in the form of *elan_putv* and *elan_getv*. However, these operations are specifically designed for the shared memory programming model (SHMEM) over Quadrics. The final placement of the data still requires a memory copy from the global memory to the application destination memory.

To support zero-copy PVFS2 list IO, we propose a software zero-copy scatter/gather mechanism with a single event chained to multiple RDMA operations. Figure 7.6 shows

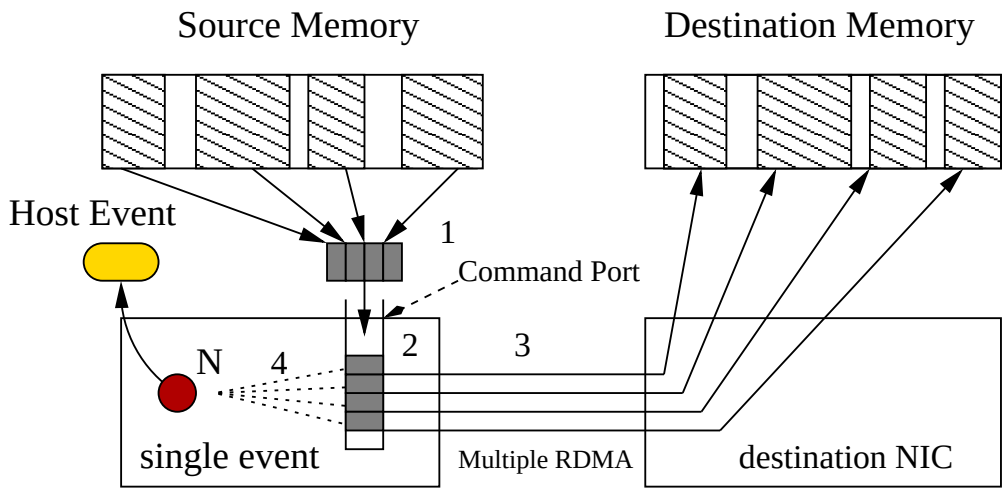


Figure 7.6: Zero-Copy Noncontiguous Communication with RDMA and Chained Event

a diagram about how it can be used to support PVFS2 list IO with RDMA read and/or write. As the communication on either side could be noncontiguous, a message first needs to be exchanged for information about the list of memory address/length pairs. The receiver can decide to fetch all data through RDMA read, or it can inform the sender to push the data using RDMA write. With either RDMA read or write, the number of required contiguous RDMA operations, N, needs to be decided first. Then the same number of RDMA descriptors are constructed in the host memory and written together into the Quadrics Elan4 command port (a command queue to the NIC formed by a memory-mapped user accessible NIC memory) through programmed IO. An Elan4 event is created to wait on the completion of N RDMA operations. As this event is triggered, the completion of list IO operation is detected through a host-side event. Over Quadrics, a separate message or an event can be chained to this Elan4 event and notify the remote process. Note that, with this design, multiple RDMA operations are issued without calling extra sender or receiver routines. The

data is communicated in a zero-copy fashion, directly from the source memory regions to the destination memory regions.

7.7. Implementation

With the design of client/server connection model and the transport layer over Quadrics communication mechanisms, we have implemented PVFS2 over Quadrics/Elan4. The implementation is based on the recent release of PVFS2-1.1-pre1. Due to the compatibility issue of PVFS2 and Quadrics RedHat Linux kernel distribution, we have utilized a patched stock kernel linux-2.4.26-4.23qsnet. Our implementation is layered on top of Quadrics library, libelan4, and completely resides in the user space. We include the following choices in our implementation: short messages are transmitted in the eager protocol along with the chained control message; long messages are transmitted through the adaptive *rendezvous* protocol using zero-copy RDMA read and write; zero-copy non-contiguous IO is supported using multiple RDMA and a single chained event; a throttling mechanism is enforced to regulate the number of concurrent RDMA read and write operations.

7.8. Performance Evaluation of Parallel IO over Quadrics

In this section, we describe the performance evaluation of our implementation of PVFS2 over Quadrics/Elan4. The experiments were conducted on a cluster of eight SuperMicro SUPER X5DL8-GG nodes: each with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, PCI-X 64-bit 133 MHz bus, 533MHz Front Side Bus (FSB) and a total of 2GB PC2100 DDR-SDRAM physical memory. All eight nodes are connected to a QsNet^{II} network [73, 7], with a dimension one quaternary fat-tree [32] QS-8A switch and eight Elan4 QM-500 cards. Each node has a 40GB, 7200 RPM, ATA/100 hard disk Western Digital WD400JB. The operating

system is RedHat 9.0 Linux. To minimize the impact in network capacity, we used the TCP implementation of PVFS2 as a comparison. Quadrics provides an IP implementation on top of its kernel communication library.

7.8.1. Performance Comparisons of Different Communication Operations

Table 7.3 shows the comparisons of the latency and bandwidth between TCP/IP over Quadrics and Quadrics native communication operations, including QDMA and RDMA read/write. Quadrics IP implementation is often referred to as EIP based on the name of its Ethernet module. The performance of TCP stream over Quadrics is obtained using the netperf [3] benchmark. The performance of Quadrics native operations is obtained using microbenchmark programs, `pgping` and `qping`, available from standard Quadrics releases [73].

Operations	Latency	Bandwidth
TCP/EIP	23.92μs	482.26MB/s
Quadrics RDMA/Write	1.93μs	910.1MB/s
Quadrics RDMA/Read	3.19μs	911.1MB/s
Quadrics QDMA	3.02 μs	368.2MB/s

Table 7.3: Network Performance over Quadrics

As shown in the table, Quadrics native operations provide better performance in terms of both latency and bandwidth compared to the performance of TCP over Quadrics. Moreover, the host CPU has less involvement in the communication processing when using Quadrics RDMA operations because of its zero-copy message delivery. More CPU cycles can be used

to handle computation in other components and contribute to better overall file system performance. To demonstrate the potential and effectiveness of leveraging Quadrics capabilities, we focus on the following aspects: the performance of bandwidth-bound data transfer operations, the performance of the latency-bound management operations, and the performance benefits to application benchmarks, such as MPI-Tile-IO [75] and BT-IO [88].

7.8.2. Performance of Data Transfer Operations

To evaluate the data transfer performance of PVFS2 file system, we have used a parallel program that iteratively performs the following operations: create a new PVFS2 file, concurrently write data blocks to disjoint regions of the file, flush the data, concurrently read the same data blocks back from the file, and then remove the file. MPI collective operations are used to synchronize application processes before and after each I/O operation. In our program, each process writes and then reads a contiguous 4MB block of data at disjoint offsets of a common file based on its rank in the MPI job. At the end of each iteration, the average time to perform the read/write operations among all processes is computed and recorded. Seven iterations are performed, and the lowest and highest values are discarded. Finally, the average values from the remaining iterations are taken as the performance for the read and write operations.

We have divided the eight-node cluster into two groups: servers and clients. Up to four nodes are configured as PVFS2 servers, and the remaining nodes are running as clients. Experimental results are labeled as *N*S for a configuration with N servers. Figure 7.7 shows the read performance of PVFS2 over Elan4 compared to the PVFS2 over TCP. PVFS2 over Elan4 improves the aggregated read bandwidth by more than 140% compared to that of PVFS2 over TCP. This suggests that the read performance of PVFS2 is much limited by the

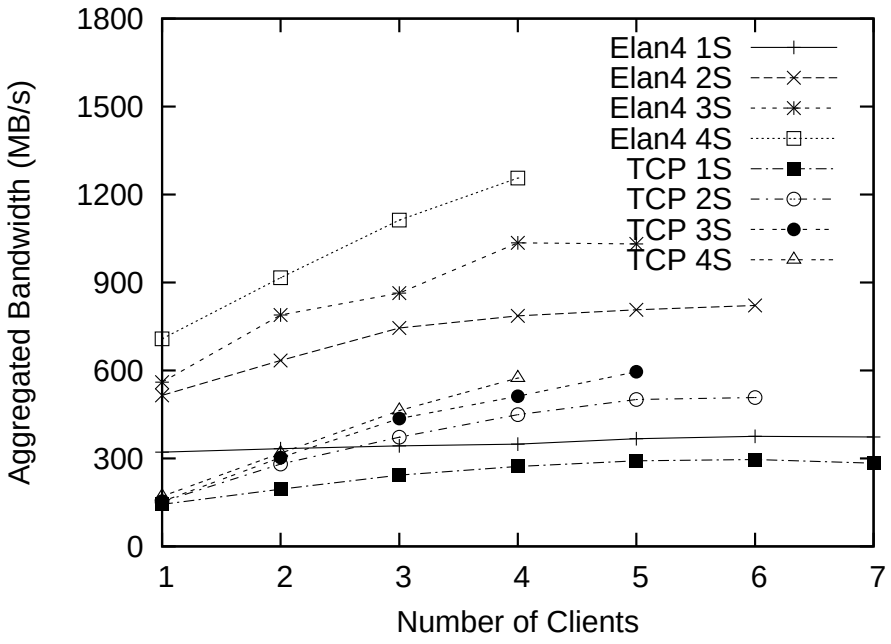


Figure 7.7: Performance Comparisons of PVFS2 Concurrent Read

network communication and can significantly benefit from the improvement in the network performance.

We have also performed experiments to evaluate the write performance of PVFS2/Elan4. We have observed less than 10% performance improvement compared to PVFS2/TCP (data not shown). This is because the network bandwidth of both Elan4 and TCP are more than 350MB/s, which is much higher than the performance of the local IDE disk in the order of 40MB/s. Instead, we have used a memory-resident file system, ramfs, to avoid the bottleneck of disk access. This is shown in Figure7.8. With varying numbers of clients concurrently writing to the file system, PVFS2 over Elan4 improves the aggregated write bandwidth by up to 82% compared to that of PVFS2 over TCP. This suggests that PVFS2

write bandwidth can also benefit from Quadrics communication mechanisms, though it is relatively less bounded by the network communication compared to the read performance.

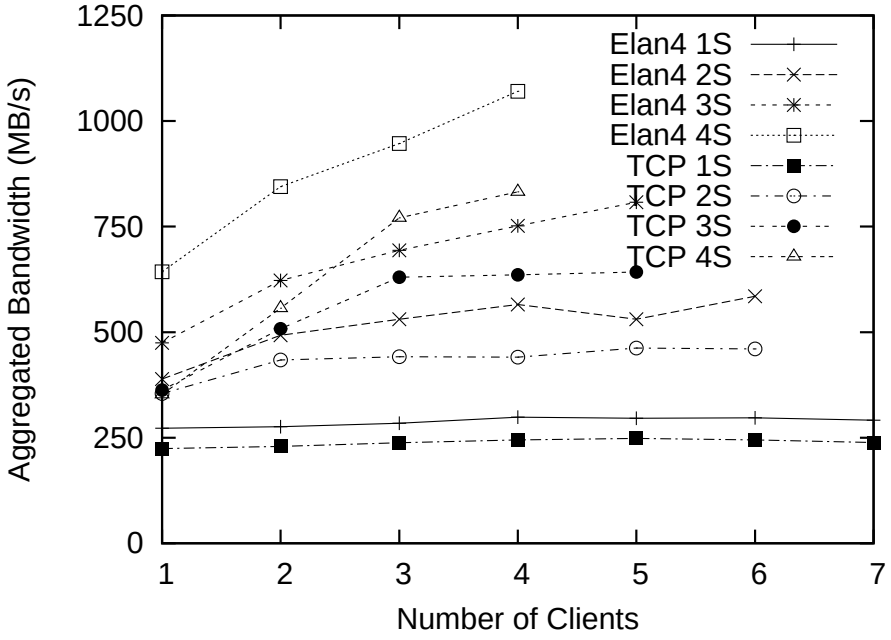


Figure 7.8: Performance Comparisons of PVFS2 Concurrent Write

7.8.3. Performance of Management Operations

PVFS2 parallel file system is designed to provide scalable parallel IO operations that match MPI-IO semantics. For example, management operations, such as `MPI_File_open` and `MPI_File_set_size`, are shown to be very scalable in [48]. These management operations typically do not involve massive data transfer. To evaluate the benefits of Quadrics low latency communication to these management operations, we have performed the following experiments using a microbenchmark program available in the PVFS2 distribution.

No. of clients	TCP	Elan4
Create (milliseconds)		
1	28.114	27.669
2	28.401	28.248
3	28.875	28.750
4	28.892	28.710
5	29.481	29.123
6	29.611	29.410
Resize (milliseconds)		
1	0.192	0.141
2	0.248	0.187
3	0.330	0.201
4	0.274	0.180
5	0.331	0.226
6	0.338	0.213

Table 7.4: Comparison of the Scalability of Management Operations

With the eight-node cluster, a PVFS2 file system is configured with two servers, both act as metadata and IO servers. The first experiment measures the average time to create a file using collective `MPI_File_open` with different numbers of clients. The second experiment measures the average time to perform a resize operation using collective `MPI_File_set_size` with different numbers of clients. As shown in Table 7.4, our PVFS2 implementation over Elan4 improves the time to resize a file by as much as 125μs (37%) for up to 6 clients. However, the improvement on the time to create a file is just marginal compared to the total time. This is because the time in allocating the storage spaces at the PVFS2 server for the new file, though small, still dominates over the communication between the client and the server. On the other hand, once the file is created, the time for the operations that update the file metadata, as represented by the resize operation, can be reduced by the PVFS2

implementation over Elan4. Therefore PVFS2 implementation over Elan4 is also beneficial to the scalability of MPI-IO management operations.

7.8.4. Performance of MPI-Tile-IO

MPI-Tile-IO [75] is a tile reading MPI-IO application. It tests the performance of tiled access to a two dimensional dense dataset, simulating the type of workload that exists in some visualization applications and numerical applications. Four of eight nodes are used as server nodes and the other four as client nodes running MPI-tile-IO processes. Each process renders a 2×2 array of displays, each with 1024×768 pixels. The size of each element is 32 bytes, leading to a file size of 96MB.

PVFS2 provides two different modes for its IO servers: trovesync and notrovesync. The former is the default mode in which IO servers perform `fsync` operations to flush its underlying file system buffer cache; the latter allows the IO servers to take the cache effects of the local file system for better performance. We have evaluated both the read and write performance of mpi-tile-io over PVFS2/Elan4 under both modes. As shown in Figure 7.9, compared to PVFS2/TCP, PVFS2/Elan4 improves MPI-Tile-IO write bandwidth by 170% with server side caching effects (under notrovesync mode, W/N), and 12% without caching effects (under trovesync mode, W/T). On the other hand, MPI-Tile-IO read bandwidth is improved by about 240% with or without server side caching effects. These results indicate our implementation is indeed able to leverage the performance benefits of Quadrics mechanisms into PVFS2: when the server disk access is a bottleneck, it improves the write performance with its zero-copy user-level communication which competes less with the disk access for CPU time; when the server disk access is not a primary bottleneck, it improves both the read and write bandwidth significantly.

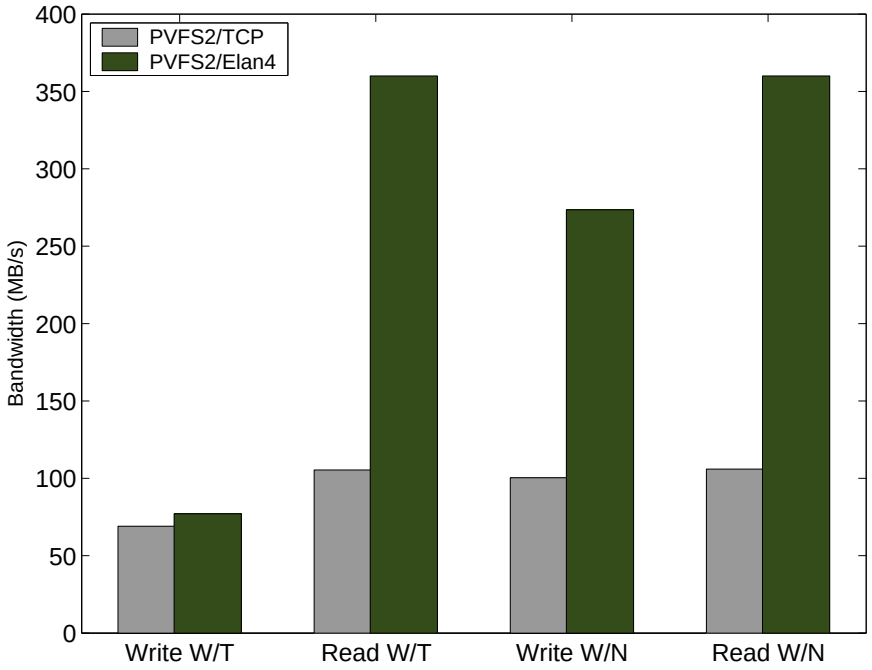


Figure 7.9: Performance of MPI-Tile-IO Benchmark

7.8.5. Benefits of Zero-copy Scatter/Gather

To evaluate the benefits fo zero-copy scatter/gather support, we also have compared the performance of PVFS2 over Quadrics with an implementation of PVFS2 over InfiniBand from the release PVFS2-1.1.0. MPI-Tile-IO achieves less aggregated read and write bandwidth over InfiniBand (labeled as IB-LIO), compared to our Quadrics-based PVFS2 implementation, with or without zero-copy scatter/gather. This is because of two reasons. First, memory registration is needed over InfiniBand for communication and the current implementation of PVFS2 over InfiniBand does not take advantage of memory registration cache to save registration costs. In contrast, memory registration is not needed over Quadrics with its NIC-based MMU. The other reason is that PVFS2/IB utilizes InfiniBand RDMA

write-gather and read-scatter mechanisms for non-contiguous IO. These RDMA-based scatter/gather mechanisms over InfiniBand can only avoid one local memory copy. On the remote side, memory packing/unpacking is still needed. So it does not support true zero-copy list IO for PVFS2. As shown in Figure 7.10, with Quadrics scatter/gather support, MPI-Tile-IO write bandwidth can be improved by 66%. On the other hand, MPI-Tile-IO read bandwidth can be improved by up to 113%. These results indicate zero-copy scatter/gather support can bring significant benefits to noncontiguous IO of a parallel file system. It would be desirable to exploit ways to integrate InfiniBand scatter/gather capabilities into PVFS2 while overcoming the concern for memory registration.

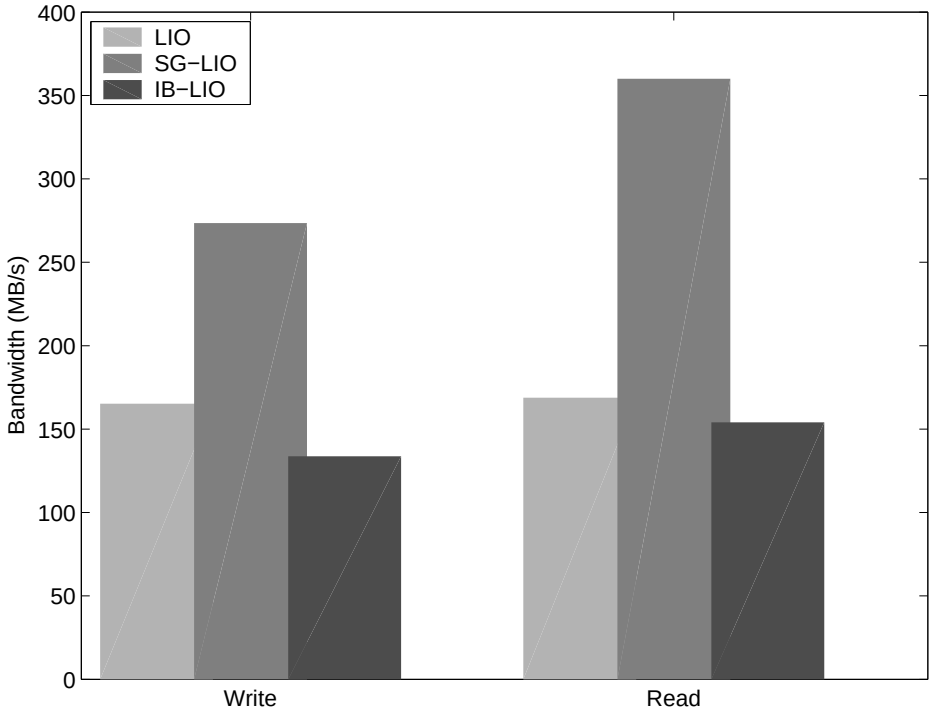


Figure 7.10: Benefits of Zero-Copy Scatter/Gather to MPI-Tile-IO

7.8.6. Performance of NAS BT-IO

The BT-IO benchmarks are developed at NASA Ames Research Center based on the Block-Tridiagonal problem of the NAS Parallel Benchmark suite. These benchmarks test the speed of parallel IO capability of high performance computing applications. The entire data set undergoes complex decomposition and partition, eventually distributed among many processes, more details available in [88]. We have used four of eight nodes used as server nodes and the other four as client nodes. The BT-IO problem size class A is evaluated. Table 7.5 shows the BT-IO performance of PVFS2/Elan4 and PVFS2/TCP, along with the performance of BT benchmark without IO access. Compared to pure BT benchmark, the BT-IO benchmark has only 2.12 seconds extra IO time when accessing a PVFS2/Elan4 file system, but 5.38 seconds IO time when accessing a PVFS2/TCP file system. With PVFS2/Elan4, the IO time of BT-IO is reduced by 60% compared to PVFS2/TCP.

Type	Duration	IO Time
No IO	61.71	−−
BT/IO Elan4	63.83	2.12
BT/IO TCP	67.09	5.38

Table 7.5: Performance of BT-IO Benchmark (seconds)

7.9. Summary of Parallel IO over Quadrics

We have examined the feasibility of designing a parallel file system over Quadrics [73] to take advantage of its user-level communication and RDMA operations. PVFS2 [2] is used as the parallel file system platform in this work. The challenging issues in supporting PVFS2 on

top of Quadrics interconnects are identified. Accordingly, strategies have been designed to overcome these challenges, such as constructing a client-server connection model, designing the PVFS2 transport layer over Quadrics RDMA read and write, and providing efficient non-contiguous network IO support. The performance of our implementation is compared to that of PVFS2/TCP over Quadrics IP implementation. Our experimental results indicate that: the performance of PVFS2 can be significantly improved with Quadrics user-level protocols and RDMA capabilities. Compared to a PVFS2 implementation over TCP/IP over Quadrics, our implementation improves the aggregated read performance by more than 140%. It is also able to deliver significant performance improvement in terms of IO access to application benchmarks such as mpi-tile-io [75] and BT-IO [88]. To the best of our knowledge, this is the first high performance design and implementation of a user-level parallel file system, PVFS2, over Quadrics interconnects.

CHAPTER 8

Conclusions and Future Research Directions

In this dissertation, we have addressed the criticality of leveraging modern networking mechanisms for the benefits of cluster-based high-end computing environments. We have shown that the HEC component functionalities, such as resource management, parallel communication, and parallel IO, are highly dependent on the performance of the underlying interconnects. Different state-of-the-art mechanisms such as OS-bypass user-level protocols, RDMA, hardware broadcast, hardware atomic operations and NIC-based programmable processors over contemporary interconnects can be utilized to enhance the performance of different components of a HEC environment.

By designing a mechanism for parallel and pipelined connection setup using InfiniBand user-level protocol, we have demonstrated that the startup scalability of MPI programs can be significantly improved. On top of that, we have presented an adaptive connection management (ACM) scheme that utilizes native InfiniBand communication management (IBCM) and transport services, and set up new InfiniBand connections based on communication statistics. ACM is shown to provide further benefits to the startup scalability besides its contribution to scalable resource usage. In addition, we have designed an initial checkpoint/restart framework

High performance and scalable MPI collective operations are also important to HEC environments. We have demonstrated Quadrics hardware broadcast can be leveraged to provide high performance, highly scalable and end-to-end reliable broadcast support to LA-MPI. We have also demonstrated that, over Myrinet, NIC-based algorithms can be designed for scalable, high-performance collective operations including barrier, broadcast and all-to-all broadcast by carefully offloading light-weight communication processing to NIC processors, reducing the redundant data copies across IO buses, and minimizing the host CPU involvement. These algorithms are also shown to augment the tolerance of parallel jobs to process skew in a large-scale environment.

In view of the lack of parallel IO support over Quadrics, we have presented our design and implementation of a Quadrics-capable version of a parallel file system (PVFS2). We have presented our design of zero-copy noncontiguous PVFS2 IO using a software scatter/gather mechanism over Quadrics. By comparing to similar studies on other interconnects, we have shown the integration of Quadrics RDMA capability and zero-copy noncontiguous IO mechanism is able to enable high performance parallel IO, whose benefits are demonstrated for both benchmarks and applications.

8.1. Summary of Research Contributions

We summarize our research results and contributions in this dissertation.

8.1.1. Scalable Startup for MPI Programs over InfiniBand Clusters

With MVAPICH as the platform of study, we have characterized the startup of MPI jobs into two phases: process initiation and connection setup. To speed up connection setup phase, we have developed two approaches, one with queue pair data reassembly at the launcher and the other with a bootstrap channel. In addition, we have exploited a process

management framework, Multi-Purpose Daemons (MPD) system, to improve the process initiation phase. The performance limitations in the MPD's ring-based data exchange model, such as exponentially increased communication time and numerous process context switches, are eliminated by using the proposed bootstrap channel. We have implemented these schemes in MVAPICH [60]. Our experimental results show that, for 128-process jobs, the startup time has been reduced by more than 4 times. We have also developed an analytical model to project the scalability of the startup schemes. The derived models suggest that the improvement can be more than two orders of magnitudes for the startup of 2048-process jobs with the MPD-BC startup scheme. The enhanced startup scheme has been integrated into MVAPICH 0.9.2 release onward.

8.1.2. Adaptive Connection Management for Scalable MPI over InfiniBand

We have explored different connection management algorithms for parallel programs over InfiniBand clusters. We have introduced adaptive connection management to establish and maintain InfiniBand services based on communication frequency between a pair of processes. Two different mechanisms have been designed to establish new connections: an unreliable datagram-based mechanism and an InfiniBand connection management-based mechanism. The resulting adaptive connection management algorithms have been implemented in MVAPICH to support parallel programs over InfiniBand. Our algorithms have been evaluated with respect to their abilities in reducing the process initiation time, the number of active connections, and the communication resource usage. Experimental evaluation with NAS application benchmarks indicates that our connection management algorithms can significantly reduce the average number of connections per process.

8.1.3. Transparent Checkpoint/Restart Support for MPI over InfiniBand

We have presented our design of checkpoint/restart framework for MPI over InfiniBand. Our design enables application-transparent, coordinated checkpointing to save the state of the whole MPI program into checkpoints stored in reliable storage for future restart. We evaluated our design using NAS benchmarks and HPL. Experimental results indicate that our design impose a low overhead for checkpointing.

8.1.4. High Performance End-to-End Broadcast for LA-MPI over Quadrics

We have incorporated Quadrics hardware broadcast communication into LA-MPI to provide an efficient broadcast operation. We then describe the benefits and limitations of the hardware broadcast communication and possible strategies to overcome them. Accordingly, a broadcast algorithm is designed and implemented with the best suitable strategies. Our evaluation shows that the new broadcast algorithm achieves significant performance benefits compared to the original generic broadcast algorithm in LA-MPI. It is also highly scalable as the system size increases. Moreover, it outperforms the broadcast algorithms implemented by Quadrics [73] MPICH, and HP's for Alaska MPI. Furthermore, instead of leaving the reliability of message passing to the Quadrics hardware, this new algorithm can ensure the end-to-end reliable data delivery.

8.1.5. Scalable NIC-based Collective Communication over Myrinet

We have examined the communication processing for point-to-point operations, and pinpointed the relevant processing A NIC-based multisend primitive is designed to enable the

transmission of multiple message replicas to different destinations; and a NIC-based forwarding primitive is provided to intermediate NICs to forward the received packets without intermediate host involvement. We have carefully examined the pros and cons of dividing the functionalities between the NIC and the host processors. Pair-wise exhange and dissemination algorithms are designed for barrier operation. Concurrent broadcasting and recursive doubling algorithms are proposed for all-to-all broadcast. Optimal spanning tree-based forwarding is designed for broadcast operations. Our results indicate that these algorithms have achieved their goals in offloading communication processing, reducing IO buses transactions, minimizing host CPU involvement, as well as increasing process skew tolerance.

8.1.6. High Performance Parallel IO over Quadrics

We have demonstrated that it is feasible of designing a parallel file system over Quadrics [73] to take advantage of its user-level communication and RDMA operations. PVFS2 [2] is used as the parallel file system platform in this work. We have designed approaches to constructing a client-server connection model, providing an efficient PVFS2 transport layer over Quadrics RDMA read and write, and achieving zero-copy non-contiguous network IO support. To the best of our knowledge, this is the first high performance design and implementation of a user-level parallel file system, PVFS2, over Quadrics interconnects. Experimental results indicate that: the performance of PVFS2 can be significantly improved with Quadrics user-level protocols and RDMA capabilities for both IO benchmarks and real applications.

8.2. Future Research Directions

MPI Resource Management The increasing size of the clusters and the demand of ultra-scale problem size will continue pushing the edge of high end computing. So the problem of managing limited resource within or accessible to a single process for the needs

of ever-increasing number of peer processes will continue to be present in the following angles: scalable startup and teardown of a parallel job, dynamic migration and concentration processes for network locality within a cluster; fault tolerant managing of MPI communication at all levels. Scalable startup and teardown can be further pursed in the angle of hypercube-based process startup and pipeplined image distribution. With the large scale clusters be populated within multiple parallel jobs or a single parallel job sparsely spreaded across distant corners of a cluster, it could be desirable to investigate how the process/node distribution pattern can affect the parallel communication performance of a single job or the aggregated productivity of an entire cluster. Process migration and concentration efforts will be indispensable to this study. In addition, fault tolerant MPI resource management needs to enable error recovery at all-levels incuding packet delivery, message retransmission, connection re-establishment, process reinitiation and even reconstruction of MPI communicator group. With network technologies continue to provide interconnects with lower error rates and high bandwidth, providing error recovery on a per-packet or per-message basis will beome less desirable, rather counter-productive methods, especially taking into account of the scale of memory and CPU resources that will be needed for recording and locating the individual bookkeeping entries. Future research shall pay more attention on managing clustered recovery of message transmissions and grouped recovery of threads, processes and MPI communicators.

MPI Collective Communication Numerous studies have been accumulated in literature for enhancing the performance of collective communication and synchronization. However, there is relatively less study on how the applications can benefit from optimized collective operations. So some of the future research directions would be to investigate:

(a) how the applications can be better designed for leveraging the benefits of enhanced collective operations; (b) how the collective operations can be better designed for the diverse communication pattern of applications; (c) what additional mechanism can be leveraged for further enhancements of collective operations. For both (a) and (b), it is our belief that process skew is one of main reasons that applications are not able to benefit much from enhanced collective operations. Further research efforts shall be put on providing better synchronization tools for parallel applications, such as global interrupt (a way to interrupt all processes in an instant manner for concurrent task processing), as well as on reducing the inherent synchronization of collective operations.

Scalable MPI-IO In the same manner, the scale of clusters will pose new challenges on the IO angle in terms of how to support tens of thousands clients whose IO requests fall on the same IO device, the same partition, the same directory or even the same block of the same file. Added to the fact is that many of the random IO pattern could be non-aligned, non-contiguous, small IO requests. Further efforts need to be put forth on (a) adaptive distribution of parallel prefetch and writeback; and (b) zero-copy IO data placement [24]. While data sieving and two-phase IO techniques are successful in re-ordering and re-organizing the IO distribution within a parallel job, few research has done on how to achieve adaptive distribution of parallel IO prefetch and writeback, for example, how to establish and correlate parallel prefetch/writeback with application IO request pattern and how to have clients cooperatively performing prefetch and writeback of better overlapped IO and communication. Due the depth of IO processing stacks, true zero-copy data placement from source to destination storage (or just memory) is still a challenging issue for many disk and networked file systems.

Research in this direction shall shed light on how the current file system can be better designed for zero-copy data placement.

BIBLIOGRAPHY

[1] RDMA Consortium. `http://www.rdmaconsortium.org/home`.

[2] The Parallel Virtual File System, version 2. http://www.pvfs.org/pvfs2.

[3] The Public Netperf Homepage. http://www.netperf.org/netperf/NetperfPage.html.

[4] TOP 500 Supercomputers. http://www.top500.org/.

[5] A. Petitet and R. C. Whaley and J. Dongarra and A. Cleary. http://www.netlib.org/benchmark/hpl/.

[6] Amotz Bar-Noy and Shlomo Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *SPAA*, 1992.

[7] J. Beecroft, D. Addison, F. Petrini, and Moray McLaren. QsNet-II: An Interconnect for Supercomputing Applications. In *the Proceedings of Hot Chips '03*, Stanford, CA, August 2003.

[8] R.A.F. Bhoedjang, T. Ruhl, and H.E. Bal. LFC: A Communication Substrate for Myrinet. In *Proceedings of the Fourth Annual Conference of the Advanced School for Computing and Imaging*, pages 31–37, June 1998.

[9] Raoul A.F. Bhoedjang, Tim Ruhl, and Henri E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *27th International Conference on Parallel Processing*, 1998.

[10] Raoul A.F. Bhoedjang, Tim Ruhl, and Henri E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *27th International Conference on Parallel Processing*, 1998.

[11] 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.

[12] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[13] Dan Bonachea, Christian Bell, Paul Hargrove, and Mike Welcome. GASNet 2: An Alternative High-Performance Communication Interface, November 2004.

[14] R. Brightwell and A. Maccabe. Scalability limitations of via-based technologies in supporting mpi. March 2000.

[15] Ron Brightwell and Lee Ann Fisk. Scalable parallel application launch on Cplant. In *Proceedings of Supercomputing, 2001*, Denver, Colorado, November 2001.

[16] D. Buntinas and D. K. Panda. NIC-Based Reduction in Myrinet Clusters: Is It Beneficial? In *SAN-02 Workshop (in conjunction with HPCA)*, Feb. 2003.

[17] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan. Broadcast/Multicast over Myrinet Using NIC-Assisted Multidestination Messages. In *CANPC*, 2000.

[18] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan. Broadcast/Multicast over Myrinet Using NIC-Assisted Multidestination Messages. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 115–129, 2000.

[19] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-Level Barrier over Myrinet/GM. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2001.

[20] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-Level Barrier over Myrinet/GM. In *IPDPS*, 2001.

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

[22] G. Burns, R. Daoud, and J. Vaigl. LAM: an Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, Toronto, Canada, 1994.

[23] Ralph Butler, William Gropp, and Ewing Lusk. Components and interfaces of a process management system for parallel programs. *Parallel Computing*, 27(11):1417–1429, 2001.

[24] Brent Callaghan and Tom Talpey. NFS Direct Data Placement. http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-nfsdirect-02.txt.

[25] Philip H. Carns, Walter B. Ligon III, Robert Ross, and Pete Wyckoff. BMI: A Network Abstraction Layer for Parallel I/O, 2004.

[26] A. Ching, A. Choudhary, W.-K. Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the IEEE International Conference on Cluster Computing*, Chicago, IL, September 2002.

[27] Cluster File System, Inc. Lustre: A Scalable, High Performance File System. http://www.lustre.org/docs.html.

[28] Salvador Coll, Jose Duato, FAbrizio Petrini, and Franscisco J. Mora. Scalable Hardware-Based Multicast Trees. In *Proceedings of Supercomputing '03*, November 2003.

[29] Compaq, Intel, and Microsoft. The Virtual Interface Architecture (VIA) Specification. available at http://www.viarch.org.

[30] Abbie Matthews David Nagle, Denis Serenyi. The Panasas ActiveScale Storage Cluster – Delivering Scalable High Bandwidth Storage. In *Proceedings of Supercomputing '04*, November 2004.

[31] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. The Direct Access File System. In *Proceedings of Second USENIX Conference on File and Storage Technologies (FAST '03)*, 2003.

[32] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. The IEEE Computer Society Press, 1997.

[33] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Berkeley Lab, 2002.

[34] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-Fast Resource Management. In *Proceedings of the Supercomputing '02*, Baltimore, MD, November 2002.

[35] Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. In *1999 USENIX Technical Conference (Freenix Track)*, June 1999.

[36] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A Network-Failure-tolerant Message-Passing system for Terascale Clusters. In *Proceedings of the 2002 International Conference on Supercomputing*, June 2002.

[37] 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.

[38] Manish Gupta. Challenges in Developing Scalable Scalable Software for BlueGene/L. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002.

[39] Eric Hendriks. Bproc: The beowulf distributed process space. In *Proceedings of the International Conference on Supercomputing*, New York, New York, June 2002.

[40] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, Spain, July 1995. ACM Press.

[41] IBM. Parallel Environment for AIX 5L V4.1.1 MPI Programming Guide. http://publib.boulder.ibm.com/clresctr/, 2004.

[42] IBM Corp. IBM AIX Parallel I/O File System: Installation, Administration, and Use. Document Number SH34-6065-01, August 1995.

[43] Infiniband Trade Association. http://www.infinibandta.org.

[44] Intel Scalable Systems Division. Paragon System User's Guide, May 1995.

[45] Morris Jette and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In *Proceedings of the International Conference on Linux Clusters*, San Jose, CA, June 2003.

[46] Avi Kavas, David Er-El, and Dror G. Feitelson. Using Multicast to Pre-Load Jobs on the ParPar Cluster. *Parallel Computing*, 27(3):315–327, 2001.

[47] Jiantao Kong and Karsten Schwan. Kstreams: Kernel support for efficient end-to-end data streaming, 2004.

[48] Rob Latham, Rob Ross, and Rajeev Thakur. The impact of file systems on mpi-io scalability. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2004)*, pages 87–96, September 2004.

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

[50] J. Liu, M. Banikazemi, B. Abali, and D. K. Panda. A Portable Client/Server Communication Middleware over SANs: Design and Performance Evaluation with InfiniBand. In *In SAN-02 Workshop (in conjunction with HPCA)*, February 2003.

[51] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. P. Kini, P. Wyckoff, and D. K. Panda. Micro-Benchmark Level Performance Comparison of High-Speed Cluster Interconnects. In *Proceedings of Hot Interconnects 10*, August 2003.

[52] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications*, 8(3–4):159–416, 1994.

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

[54] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[55] Adam Moody, Juan Fernandez, Fabrizio Petrini, and Dhabaleswar Panda. Scalable NIC-based Reduction on Large-Scale Clusters. In *SC '03*, November 2003.

[56] MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, November 1993.

[57] MPICH2, Argonne. http://www-unix.mcs.anl.gov/mpi/mpich2/.

[58] Myricom. Myrinet Software and Customer Support. http://www.myri.com/scs/GM/doc/, 2003.

[59] Myricom Corporations. The GM Message Passing Systems.

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

[61] Nils Nieuwejaar and David Kotz. The Galley Parallel File System. *Parallel Computing*, (4):447–476, June 1997.

[62] Open Infiniband Alliance. http://www.openib.org.

[63] OpenPBS Documentation. `http://www.openpbs.org/docs.html`, 2004.

[64] P. H. Carns and W. B. Ligon III and R. B. Ross and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

[65] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: A unified i/o buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000.

[66] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing '95*, 1995.

[67] Joseph Pasquale, Eric W. Anderson, and Keith Muller. Container shipping: Operating system support for i/o-intensive applications. *IEEE Computer*, 27(3):84–93, 1994.

[68] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, Chicago, IL, 1988.

[69] Gang Peng, Srikant Sharma, and Tzi cker Chiueh. Network-Centric Buffer Cache Organization. In *Proceedings of The 25th IEEE International Conference on Distributed Computing Systems*, May 2005.

[70] 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.

[71] 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.

[72] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *SC '03*, November 2003.

[73] Quadrics Supercomputers World, Ltd. Quadrics Documentation Collection. http://www.quadrics.com/onlinedocs/Linux/html/index.html.

[74] John A. Rice. *Mathematical Statics and Data Analysis*. Duxbury Press, Belmont, California, 1995.

[75] Rob B. Ross. Parallel i/o benchmarking consortium. http://www-unix.mcs.anl.gov/rross/pio-benchmark/html/.

[76] 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.

[77] T. Sterling, D. Becker, M. Warren, T. Cwik, J. Salmon, and B. Nitzberg. An assessment of beowulfclass computing for nasa requirements: Initial findings from the first nasa workshop on beowulf-class clustered computing, 1998.

[78] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An operating system coordinated high performance communication library. In *HPCN Europe*, pages 708–717, 1997.

[79] Rajeev Thakur and William Gropp. Improving the Performance of Collective Operations in MPICH. In *Proceedings of EuroPVM/MPI '03*, September 2003.

[80] Rajeev Thakur, William Gropp, and Ewing Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, Oct 1996.

[81] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, February 1999.

[82] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[83] The BlueGene/L Team. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the Supercomputing '02*, Baltimore, MD, November 2002.

[84] Kees Verstoep, Koen Langendoen, and Henri E. Bal. Efficient Reliable Multicast on Myrinet. In *ICPP*, 1996.

[85] Jeffrey. S. Vetter and Frank Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *IPDPS*, April 2002.

[86] 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.

[87] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Proceedings of Supercomputing*, 1999.

[88] Parkson Wong and Rob F. Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division.

[89] Jiesheng Wu, Jiuxing Liu, Pete Wyckoff, and Dhabaleswar K. Panda. Impact of On-Demand Connection Management in MPI over VIA. In *Proceedings of the International Conference on Cluster Com puting*, 2002.

[90] Jiesheng Wu, Pete Wychoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Proceedings of the International Conference on Parallel Processing '03*, Kaohsiung, Taiwan, October 2003.

[91] Jiesheng Wu, Pete Wychoff, and Dhabaleswar K. Panda. Supporting Efficient Non-contiguous Access in PVFS over InfiniBand. In *Proceedings of Cluster Computing '03*, Hong Kong, December 2004.

[92] Hyong youb Kim, Vijay S. Pai, and Scott Rixner. Increasing web server throughput with network interface data caching. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[93] Weikuan Yu, Darius Buntinas, Rich L. Graham, and Dhabaleswar K. Panda. Efficient and Scalable Barrier over Quadrics and Myrinet with a New NIC-Based Collective Message Passing Protocol. In *Workshop on Communication Architecture for Clusters,*

in Conjunction with International Parallel and Distributed Processing Symposium '04, April 2004.

[94] Weikuan Yu, Darius Buntinas, and Dhabaleswar K. Panda. High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2. In *Proceedings of the International Conference on Parallel Processing '03*, October 2003.

[95] Weikuan Yu, Qi Gao, and Dhabaleswar K. Panda. Adaptive Connection Management for Scalable MPI over InfiniBand. In *International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006.

[96] Weikuan Yu, Sayantan Sur, Dhabaleswar K. Panda, Rob T. Aulwes, and Rich L. Graham. High Performance Broadcast Support in LA-MPI over Quadrics. In *Los Alamos Computer Science Institute Symposium*, October 2003.

[97] Weikuan Yu, Jiesheng Wu, and Dhabaleswar K. Panda. Fast and Scalable Startup of MPI Programs In InfiniBand Clusters. In *Proceedings of the International Conference on High Performance Computing '04*, Banglore, India, December 2004.

[98] Rumi Zahir. Lustre Storage Networking Transport Layer. http://www.lustre.org/docs.html.

[99] Yuanyuan Zhou, Angelos Bilas, Suresh Jagannathan, Cezary Dubnicki, James F. Philbin, and Kai Li. Experiences with VI Communication for Database Storage. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 257–268. IEEE Computer Society, 2002.