Design and Evaluation of Efficient Collective Communications on Modern Interconnects and Multi-core Clusters

by

Ying Qian

A thesis submitted to the Department of Electrical and Computer Engineering

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

(January, 2010)

Abstract

Two driving forces behind high-performance clusters are the availability of modern interconnects and the advent of multi-core systems. As multi-core clusters become commonplace, where each core will run at least one process with multiple intra-node and inter-node connections to several other processes, there will be immense pressure on the interconnection network and its communication system software.

Many parallel scientific applications use *Message Passing Interface* (MPI) collective communications intensively. Therefore, efficient and scalable implementation of MPI collective operations is critical to the performance of applications running on clusters. In this dissertation, I propose and evaluate a number of efficient collective communication algorithms that utilize the modern features of Quadrics and InfiniBand interconnects as well as the availability of multiple cores on emerging clusters.

To overcome bandwidth limitations and to enhance fault tolerance, using multiple independent networks known as multi-rail networks is very promising. Quadrics multi-rail QsNet^{II} network is constructed using multiple network interface cards (NICs) per node, where each NIC is connected to a rail. I design and evaluate a number of *Remote Direct Memory Access* (RDMA) based multi-port collective operations on multi-rail QsNet^{II} network. I also extend the gather and allgather algorithms to be shared memory aware for small to medium messages. The algorithms prove to be much more efficient than the native Quadrics MPI implementation.

ConnectX is the newest generation of InfiniBand host channel adapters from Mellanox Technologies. I provide evidence that ConnectX achieves scalable performance

for simultaneous communication over multiple connections. Utilizing this ability of ConnectX cards, I propose a number of RDMA based multi-connection and multi-core aware allgather algorithms at the MPI level. My algorithms are devised to target different message sizes, and the performance results show that they outperform the native MVAPICH implementation.

Recent studies show that MPI processes in real applications could arrive at an MPI collective operation at different times. This imbalanced process arrival pattern can significantly affect the performance of the collective communication operation. Therefore, design and efficient implementation of collectives under different process arrival patterns is critical to the performance of scientific applications running on modern clusters. I propose novel RDMA-based process arrival pattern aware alltoall and allgather for different message sizes over InfiniBand clusters. I also extend the algorithms to be shared memory aware for small to medium messages under process arrival patterns. The performance results indicate that the proposed algorithms outperform the native MVAPICH implementation as well as other non-process arrival pattern aware algorithms when processes arrive at different times.

Acknowledgements

I would like to thank the guidance and support of my supervisor Professor Ahmad Afsahi. Without him, this dissertation would have never been possible. I would like to acknowledge the financial support from the Natural Science and Engineering Research Council of Canada (NSERC) and Ontario Graduate Scholarship (OGS). I am indebted to the Department of Electrical and Computer Engineering for awarding me Teaching Assistantship.

I would like to thank my friends at the Parallel Processing Research Laboratory, Reza Zamani, Ryan E. Grant, and Mohammad J. Rashti for their great help. I also would like to especially thank my friends, HaoJie Ji, Qian Wu, and Helen Zhou for their kind support.

Lastly, special thanks to my parents for their love and support during my study.

Table of Contents

List of Figures

List of Tables

Glossary

CA	Channel Adapter
CQ	Completion Queues
HCA	Host Channel Adapter
HPC	High-Performance Computing
IBA	InfiniBand Architecture
iWARP	Internet Wide Area RDMA Protocol
MIF	Maximum Imbalanced Factor
MPI	Message Passing Interface
NICs	Network Interface Cards
NUMA	Non-Uniform Memory Access
OFED	OpenFabrics Enterprise Distribution
PAP	Process Arrival Pattern
PGAS	Partitioned Global Address Space
PIO	Programmed Input/Output
QDMA	Queue-based Directed Message Access
QP	Queue Pair
RC	Reliable Connection
RD	Raw Datagram
RD	Reliable Datagram
RDMA	Remote Direct Memory Access
SMP	Symmetric Multiprocessors
SRQ	Shared Receive Queue
TCA	Target Channel Adapter
Tports	Tagged Message Ports
UC	Unreliable Connection
UD	Unreliable Datagram
VPI	Virtual Protocol Interconnect
VPID	Virtual Process ID

WQE	Work Queue Element
WQR	Work Queue Request
XRC	eXtended Reliable Connection

Chapter 1: Introduction

With the introduction of high-speed networks, the trend in the *high-performance computing* (HPC) community is to use network-based computing systems such as clusters of multiprocessors to achieve high performance. *Symmetric multiprocessors* (SMP) and *non-uniform memory access* (NUMA) clusters are the predominant platforms for HPC due to their cost-performance effectiveness [86]. SMP and NUMA nodes are traditionally equipped with multiple single-core processors. However, industry has recently adopted the development of chip multiprocessors, or multi-cores, for general-purpose applications. With the emergence of such multi-core clusters, each core will run at least one process with multiple intra-node and inter-node connections to several other processes. This will put immense pressure on the interconnection network and its communication system software. Therefore, researchers in academia and industry have been working on improving the performance of communication subsystems through advancements in high-speed networking technologies and messaging layers [4, 6, 7, 16, 35, 37, 90].

1.1 Message Passing and Collective Communications

Most scientific applications running on HPC use the *Message Passing Interface* (MPI) [45] as the parallel programming paradigm of choice. In MPI, data transfer is done using explicit communications. MPI provides two kinds of communications: point-to-point and collective. Previous studies of application usage show that the performance of collective communications is critical to HPC applications, including those in [26, 56, 90]. The study in [50] shows that some applications spend more than eighty percent of their overall communication time in collective operations. They also use a large number of

collective operations, some with very large payloads and some with small payloads below 2KB [33]. As such, collective communication has been an active area of research [2, 22, 27, 39, 41, 54, 69, 70, 74, 76, 80, 91]. This trend is still ongoing, especially with the availability of contemporary interconnects and emerging architectures.

Collective communications involve a group of MPI processes. They can be categorized into one-to-all, all-to-one, alltoall, and synchronization primitives. There are two services in the one-to-all category: *broadcast* and *scatter*. In broadcast, the same message is delivered from a root process to all other processes. In scatter, which is also called personalized broadcast, the sending process delivers different messages to all other processes. *Gather*, an all-to-one operation, is the reverse of scatter. In gather, messages from different processes are gathered by the root process. In alltoall communications, all processes exchange data with each other. In *alltoall broadcast*, also called *allgather*, each process sends the same data to all other processes. In *alltoall personalized exchange*, also called *alltoall*, each process has a different data for every other processes. *Barrier*, a synchronization operation, is a global control operation that synchronizes across all processes in the group.

1.2 Modern Interconnects and User-level Messaging

Currently, several switch-based interconnects provide low latency and high bandwidth for HPC. Myrinet [48], Quadrics [65], InfiniBand [31], and iWARP Ethernet [68] are the most famous interconnects available today. They use the user-level messaging layers GM and MX [81], Elan4lib [65], and *OpenFabrics Enterprise Distribution* (OFED) [82] respectively. These messaging layers provide protected user-level access to the network interface. The user-level network protocols offered by these

high-speed interconnects are designed to bypass the operating system, and to thereby reduce the end-to-end latencies and lower the CPU utilization. These interconnects have several new features to provide better performance. They all support one-sided communications at the user level known as *Remote Direct Memory Access* (RDMA) Write, RDMA Read and RDMA Atomic operations. RDMA is a one-sided operation that allows a process to access the memory of a remote process. RDMA has been used in enhancing the performance of point-to-point and collective communications in MPI [29, 36, 42, 51, 67], to name a few, as well as other parallel programming paradigms such as *Partitioned Global Address Space* (PGAS) languages [23]. RDMA has also been used to design communication subsystems for databases and file systems [92] and to implement web servers on clusters [12].

1.3 Problem Statement

With the availability of fast interconnects and the advent of multi-core technology, there is a dire demand for improving the performance of communication subsystems in modern clusters in general, and collectives in particular. In essence, this work tries to address the following questions:

(1) How can we effectively utilize the new features of modern interconnects in the design of collectives?

(2) How can we devise collectives that could benefit from the emerging SMP/multi-core nodes?

(3) How can we adapt existing collective algorithms on such architectures?

(4) How can we design collectives under different process arrival patterns?

This dissertation seeks to understand the underlying architectures of the systems and the contemporary networks, and to efficiently utilize them to improve the performance of collective communications. It tackles a number of challenges including efficient data transfer mechanisms for intra-node and inter-node communications, efficient algorithms and protocols for different message sizes, algorithm adaptations for SMP and multi-core systems, process skew time, and scalability. Specifically, features such as RDMA, multi-rail communication, hardware broadcast, shared memory intra-node vs. inter-node message passing, multi-core awareness, multi-connection capability, and process arrival pattern awareness have been used to propose novel algorithms and protocols.

1.4 Contributions

In this dissertation, I have proposed several contributions to the efficient design and implementation of collective communications, as follows:

- **RDMA-based Multi-port Collectives on Multi-rail Quadrics QsNet^{II}**

To overcome bandwidth limitations and to enhance fault tolerance, using multiple independent networks known as multi-rail networks is very promising. Quadrics multi-rail QsNet^{II} network [65] is constructed using multiple *network interface cards* (NICs) per node, each NIC connecting to a rail. However, this feature has been available only for point-to-point communications. This dissertation contributes by designing and implementing RDMA-based multi-port algorithms for scatter, gather, allgather, and alltoall collectives on top of multi-rail Quadrics QsNet^{II} systems.

Performance results indicate that my multi-port collectives implementations are better than the native implementation except for messages less than 512 bytes. The proposed multi-port gather gains an improvement of up to 2.15 for 4KB message. The

RDMA-based allgather is better than the Quadrics implementation in the *elan_gather()* for messages larger than 2KB. The allgather Direct algorithm gains an improvement of up to 1.49 for 32KB messages over the native *elan_gahter()*. My multi-port Direct alltoall algorithm is also much better than the native *elan_alltoall()* for medium and large size messages, with up to a factor of 2.26 improvement for 2KB messages. It should be noted that the native library, both in Quadrics [55] and MVAPCH [47], implements well-known collective communication algorithms.

- **RDMA-based and Shared Memory Aware Multi-port Gather and Allgather on Multi-rail Quadrics QsNet^{II}**

Multi-rail communication improves the performance for medium to large messages. Shared memory communication improves the performance for short messages. However, simply replacing network communication with shared memory transactions for intra-node communication will not enhance the collective performance much. This dissertation proposes three optimized SMP aware allgather algorithms for short to medium message sizes, *SMP-aware Gather and Broadcast* algorithm and *SMP-aware Direct and Bruck* [10] algorithm. In the *SMP-aware Gather and Broadcast* algorithm, I first do an SMP-aware gather algorithm across all processes in the system and then broadcast the gathered data to all processes. In the *SMP-aware Direct and Bruck* algorithm, I adapt the traditional multi-port *Direct* and *Bruck* allgather algorithms to SMP clusters by performing them across the SMP nodes rather than processes. Shared memory gather and broadcast operations are used within the nodes.

Compared to the native implementation in QsNet^{II}, the *SMP-aware Gather and Broadcast* algorithm is the best algorithm for up to 256-byte messages. For short to

medium size messages (512B to 8KB), the *SMP-aware Bruck* algorithm outperforms all other algorithms. An improvement of up to 1.96 for 4KB message can be observed using the *SMP-aware Bruck* algorithm. For medium to large messages (16KB to 1MB), the RDMA-based *Direct* algorithm is superior among all algorithms, gaining an improvement of up to 1.49 for 32KB messages.

- **Multi-connection and Multi-core Aware Allgather on InfiniBand Clusters**

This dissertation provides evidence that the latest ConnectX InfiniBand cards [44] achieve scalable performance for simultaneous communication over multiple connections (up to a certain number of connections). Utilizing this, I propose a number of RDMA based multi-connection and multi-core aware allgather algorithms. Specifically, the proposed algorithms are *Single-group Multi-connection Aware*, *Multi-group Gather-based Multi-connection Aware*, and *Multi-group Multi-connection Aware* allgather algorithms. The *Single-group Multi-connection Aware* algorithm is a multi-connection extension of the proposed SMP-aware algorithm targeted at small to medium messages. Designed for small messages, the *Multi-group Gather-based Multi-connection Aware* allgather algorithm takes advantage of the availability of multiple cores to distribute the CPU processing load. Finally, to further utilize the multi-connection capability of the InfiniBand network, I propose the *Multi-group Multi-connection Aware* allgather algorithm for medium to large message sizes.

My algorithms are devised to target different message sizes and the performance results show that they outperform the native MVAPICH implementation for up to 128KB messages. The gather-based algorithm has the best performance for very small messages up to 32 bytes. The single-group multi-connection aware algorithm outperforms all other

algorithms from 64 bytes up to 2KB. From 4KB to 64KB, the 2-group multi-connection-aware algorithm performs the best.

- **Process Arrival Pattern Aware Collectives on Multi-core InfiniBand Clusters**

I provide evidence, and confirm previous observations, that MPI processes in real applications could arrive at an MPI collective operation at different times. This imbalanced process arrival pattern can significantly affect the performance of the collective operation and consequently the application itself. Therefore, its efficient implementation under different process arrival patterns is critical to the performance of scientific applications running on modern clusters. This dissertation proposes RDMA-based and process arrival pattern aware allgather and alltoall collectives on top of InfiniBand. To boost the performance for small messages, the process arrival pattern aware allgather and alltoall algorithms are enhanced with shared memory awareness.

The process arrival pattern aware alltoall algorithms are better than their non process arrival pattern aware counterparts for all message sizes. My algorithms are also superior to the native MVAPICH, with an improvement factor of 3.1 at 8KB for the *Direct* algorithm and 3.5 at 4B for *shared memory Direct algorithm*, with a *maximum imbalanced factor* (MIF) of 32. With a larger MIF of 512, the improvements are 1.5 and 1.2, respectively. The *process arrival pattern aware allgather Direct* algorithms are also better than their non-process arrival pattern aware counterparts for most of the message sizes. The proposed allgather algorithms gain an improvement factor of 3.1 at 8KB compared to MVAPICH for RDMA-based version and 2.5 times at 1B for shared memory aware version, with MIF equal to 32. With a larger MIF of 512, 1.3 and 1.2 times improvement are achieved respectively.

1.5 Dissertation Outline

The rest of this dissertation is organized as follows. In Chapter 2, I provide the background material for this study. I will discuss the message passing programming paradigm and then introduce MPI collective communication primitives in detail. I will also introduce two popular high-performance interconnects and their user-level messaging protocols. The communication and port modeling used in this dissertation are provided at the end of Chapter 2. In Chapter 3, I will present high performance RDMA-based multi-port collectives on multi-rail QsNet^{II}. RDMA-based and SMP-aware multi-port allgather on multi-rail QsNet^{II} SMP clusters will be introduced in Chapter 4. In Chapter 5, I design and implement multi-connection aware collectives on InfiniBand clusters. In Chapter 6, I take the process arrival pattern into consideration for devising collectives, and propose process arrival pattern aware alltoall and allgather algorithms. Finally I conclude the dissertation in Chapter 7.

Chapter 2: Background

In this chapter, I will summarize the background material that is related to this dissertation. In the past couple of decades, several parallel machines with different architectures have been built as viable platforms for high-performance computing, such as SMPs, NUMAs, distributed shared memory, and clusters of multiprocessors. However, with the availability of high-speed networks, the HPC community has adopted network-based computing clusters as cost-effective platforms to achieve high performance. This trend has been accelerated by the advent and use of multi-core processors in high-performance clusters, shown in Figure 2.1. According to the ranking in June, 2009, more than 82% of top 500 supercomputing sites are clusters [86]. Not to mention, clusters are also increasingly used in the low-to-medium end of spectrum, as well as in data centers, financial institutions, etc.

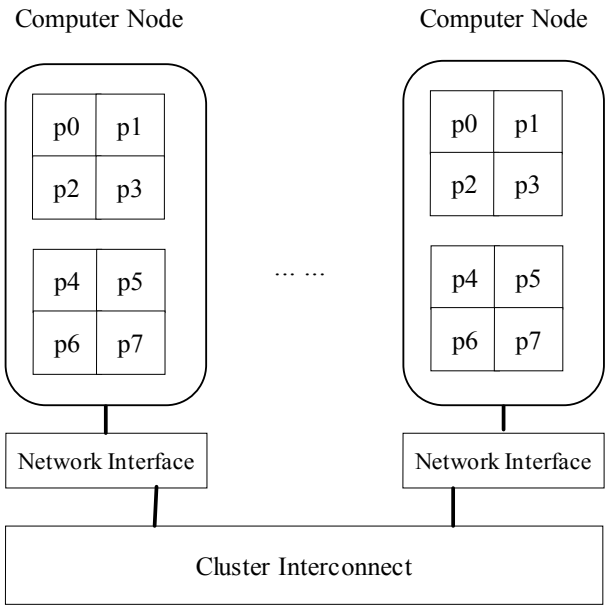

Figure 2.1 A typical multi-core cluster.

Interconnection networks are critical in achieving high performance. Currently, there are several switch-based modern interconnects that provide low latency and high bandwidth. The most famous and leading products include Myrinet [48], InfiniBand [31], Quadrics [55], and Internet Wide Area RDMA Protocol (iWARP) Ethernet [68]. High-speed interconnects offer their own user-level messaging layers to replace the traditional costly TCP/IP protocol stack. The user-level network protocols offered by these high-speed interconnects are designed to bypass the operating system and to directly access the network hardware, thereby reducing the end to end latencies. These user-level network protocols move some of the services normally provided by the kernel into the user-level. Bypassing the operating system, the user-level protocols avoid the costs associated with switching to privileged mode. GM and MX [81] Elan4lib [65] and OpenFabrics [82] and the Deep Computing Messaging Framework [37], are the user-level protocols offered by Myrinet, Quadrics and InfiniBand and iWARP Ethernet interconnects, and IBM Blue Gene/P machines [30], respectively.

The processes in parallel applications running on clusters mostly communicate with each other by explicit message passing through the interconnection network. MPI [45] is the de facto standard for parallel programming on clusters. Figure 2.2 shows the layers of abstraction for messaging layers for high-performance networks. MPI functions as a communication middleware providing the parallel programming environment to the application layer. It hides the details of the underlying network hardware and also the user-level network protocol. MPI is the critical component bridging the gap between network hardware and the user application. Therefore, it is important to have a high-performance and scalable MPI design.

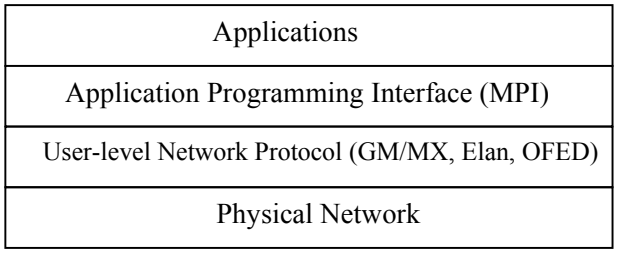

Figure 2.2 Layers of abstraction.

MPI provides different kinds of communication services to the application: point-to-point, one-sided, and collective communications. In collective communications, a group of processes are involved in a collective communication operation. Previous profiling studies of applications show that applications spend more than eighty percent of the overall communication time in collective operations [50]. Therefore, performance of collective communications becomes critical to HPC. The objective of this research is therefore to design and evaluate efficient collective communication algorithms on emerging multi-core/SMP clusters with their modern high-performance interconnection networks.

In Section 2.1, I will explain the MPI library and its communication services. In Section 2.2, the Quadrics and InfiniBand networks are introduced along with their user-level messaging layers. Section 2.3 describes the port modeling and the communication cost modeling used to analyze the performance of the collective communications.

2.1 Message Passing Interface

The most commonly used programming model for clusters is MPI [45]. An advantage of the MPI programming model is that the user has complete control over data distribution and process synchronization, which can provide optimal data locality and

workflow distribution. It is also portable as MPI programs can run on distributed-memory multicomputers, shared memory multiprocessors, and clusters.

MPI specifies an *Application Programming Interface* to provide different kinds of communications, including point-to-point communications, collective communications, and one-sided communications. One has to bear in mind that one-sided communication in MPI-2 is at the application level to mimic the essence of shared memory programming on clusters, while RDMA is a feature at the network-level. For the sake of this dissertation, we only discuss point-to-point and collective communications. Point-to-point communication is covered because many MPI distributions implement their collectives using explicit MPI point-to-point operations. This of course incurs a lot of overhead as compared to the RDMA-based communications, which is one of the focuses of this dissertation.

2.1.1 Point-to-point Communications

Point-to-point communication is the basic communication mechanism used in transmitting data between a pair of processes in MPI. The source process initiates the communication by calling an *MPI_Send()* function, and the destination process receives this message by issuing an *MPI_Recv()* function. A message consists of two parts: the actual message payload, and the message envelope that helps route the data. The message envelope consists of source, destination, a tag field and the communicator. The tag field can be used by the program to distinguish different types of messages. A communicator specifies the communication context for a communication operation. It should be mentioned that there are a number of modes available for MPI point-to-point communication including the *standard, synchronous, buffered*, and *ready* modes.

MPI implementations such as MPICH2 [46], MVAPICH [47], and OpenMPI [24] treat small and large messages differently. An *Eager* protocol is used to eagerly transfer small messages to the receiver to avoid extra overhead of pre-negotiation. For large-size messages, a *Rendezvous* protocol is used in which a negotiation phase makes the receiver ready to receive the message data from the sender. After the data transfer, a finalization packet is sent by the sender to inform the receiver that the data is placed in its appropriate application buffer. Researchers have proposed different techniques to boost the performance of Eager and Rendezvous protocols in RDMA-based interconnects. Sur et al. [77] proposed an RDMA Read-based Rendezvous method. Rashti and Afsahi [67] proposed a speculative MPI Rendezvous protocol to effectively improve communication progress and consequently the overlap ability. Recently, Small and Yuan [75] refined the Rendezvous protocol for medium and large messages using three customized protocols.

2.1.2 Collective Communications

MPI offers a number of collective communication operations, where a group of processes are involved in the operation. Collective communication operations involve global control and global data movement. *MPI_Barrier()* is a global control operation that synchronizes all processes in the group. Global data movement operations include *MPI_Bcast(), MPI_Scatter(), MPI_Gather(), MPI_Allgather(), MPI_Alltoall(), MPI_Reduce(), MPI_Allreduce(),* and variants of them. Some collective operations are shown in Figure 2.3. The actual implementation may differ from those shown. Such collectives can be formally defined as follows, where p is the number of processes in the group:

- *MPI_Bcast()*: a process, the root, sends the same message to all other

processes of the group.

- *MPI_Scatter()*: the root process sends a different message to every other process in the group. This operation is also known as one-to-all personalized communication.
- *MPI_Gather()*: it is the reverse of *MPI_Scatter()*. The root gathers data from every other process. A gather operation is different from an all-to-one *MPI_Reduce()* in that it does not involve any combination or reduction of data.
- *MPI_Allgather()*: in this operation, each process sends the same message to all other processes. It is also called alltoall broadcast. This data intensive operation is heavily used in matrix multiplication kernels.
- *MPI_Alltoall()*: In this operation, each process sends a different message to every other processes. This operation is used in a variety of parallel algorithms such as fast fourier transform, matrix transpose, sample sort, and some parallel database join operation.
- *MPI_Reduce()*: combines the data received from every other process, using the operation, "*", and returns the combined value to the root process. The operation, "*", can be *add*, *maximum*, *minimum*, etc.
- *MPI_Allreduce()*: the result of *MPI_Reduce()* is returned to all processes in the group.

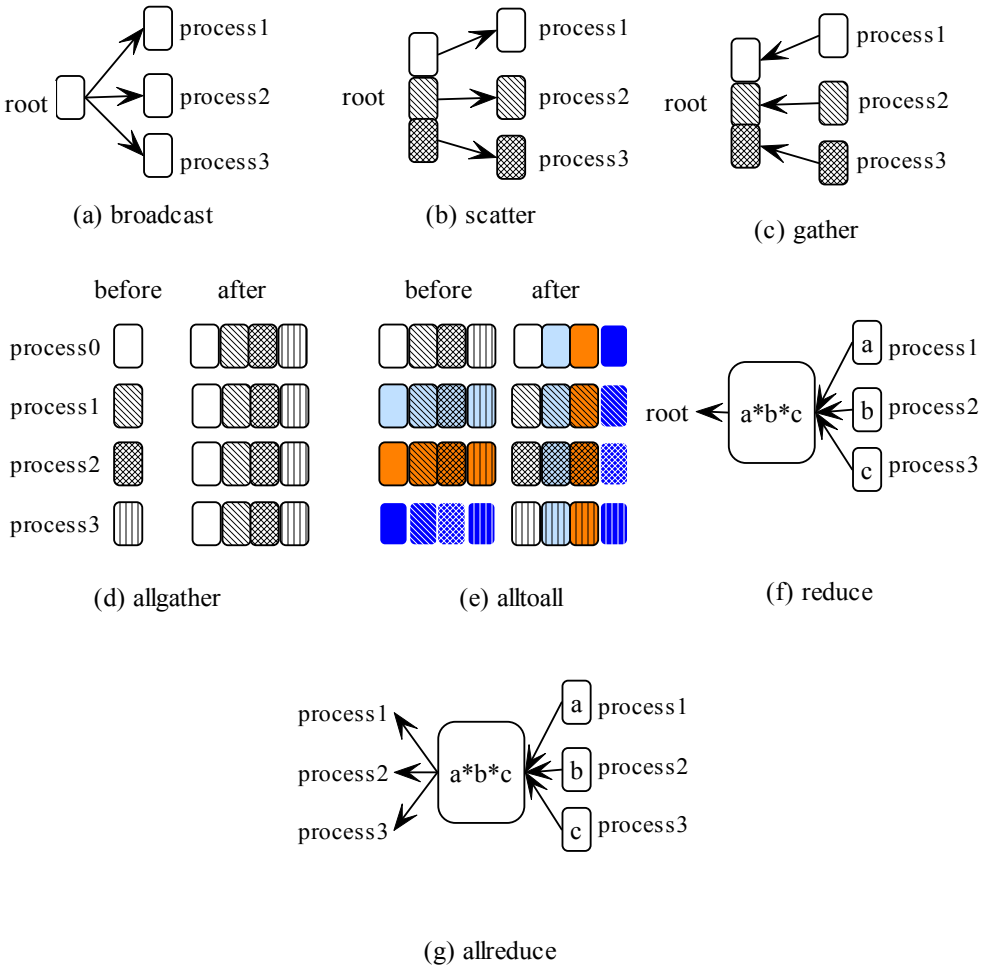

Figure 2.3 Some collective communication operations.

2.2 High-Performance Interconnects

To have a high-performance cluster computer system, the interconnection network that connects the nodes of the system plays a crucial role in performance. This dissertation considers Quadrics QsNet^{II} [6], and InfiniBand [31] as representatives of a proprietary and an open standard interconnect, respectively. It should be mentioned that these two interconnects substantially differ from each other. Brightwell et al. [9] discussed some of the differences between the earlier versions of these interconnects. In the following, I will discuss both interconnects in detail.

2.2.1 Quadrics QsNet^{II}

Quadrics QsNet^{II} is a butterfly bi-directional multistage interconnection network with 4x4 switches, which can be viewed as a quaternary fat-tree [6]. The network supports hardware broadcast and barrier, and multiple NICs per node. The latest generation, QsNet^{II}, has two building blocks, a low-latency high-bandwidth communication switch called Elite4 and a programmable network interface called Elan4 [6]. Elite switches are connected in a fat tree topology, shown in Figure 2.4 permitting 4096 nodes in the system. Quadrics switch uses a full crossbar connection and supports wormhole routing.

In addition to generating and accepting packets to and from the network, the Elan4 provides substantial local processing power to the host processor to implement high-level message passing protocols such as MPI. An embedded user-programmable I/O processor on Elan4 is used to offload asynchronous protocol handling tasks.

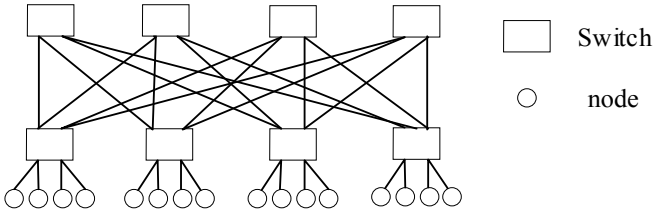

Figure 2.4 Quaternary fat tree structure for 2 dimensions.

2.2.2 Elan4lib and Elanlib

Quadrics provides libelan and libelan4 libraries [54], on top of its Elan4 network as shown in Figure 2.5. *Elan4lib* [65] provides the lowest-level, user-space programming interface to the network. Elanlib is a higher-level machine independent communication library to provide low-level accesses. It provides a global virtual address space by integrating the address space of individual nodes.

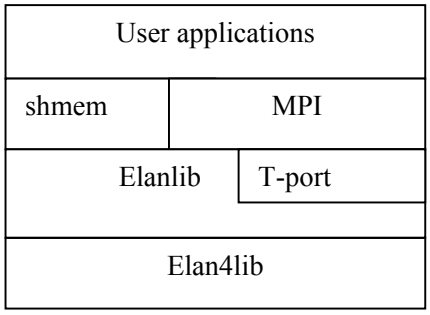

Figure 2.5 Quadrics programming libraries.

Under these default programming libraries, each parallel job first acquires a job-wise capability, it is then allocated a *virtual process ID* (VPID). The communication between the processes is supported by two different models: *Queue-based Directed Message Access* (QDMA) and RDMA. QDMA allows processes to post messages (up to 2KB) to a remote queue of another process. RMDA give processes direct access to remote memory exposed by other processes. QsNet^{II} provides efficient and protected access to a global virtual memory using RDMA operations.

A general-purpose synchronization mechanism based on events stored in memory is provided so that the completion of RDMA operations can be reported. The event mechanism allows one operation to be triggered upon the completion of the other operations. This event can be utilized to provide fast and asynchronous progress of back-to-back operations.

The Quadrics library also provides basic mechanisms for point-to-point message passing, called *Tagged Message Ports* (Tports) [65]. Unlike GM [81] and OFED [82], the QsNet^{II} does not require the communication buffers to be pinned. Elanlib also supports multi-rail point-to-point communications. It also supports hardware broadcast and barrier collective operations.

SHMEM [73] is a message passing library very similar to MPI. It was originally developed for the Cray T3E series of vector computers. SHMEM uses active messaging, where a source process reads from and writes onto a target process's memory directly, without any need for the target processor's cooperation. This allows for very low latencies and high bandwidth for inter-processor communications.

2.2.3 InfiniBand Architecture

InfiniBand Architecture (IBA) [31] is proposed as a generic interconnect for inter-process communication and I/O. In this section, I will introduce the InfiniBand architecture and its features, including the communication semantics provided and the associated transport services. This section will show how InfiniBand differs from the Quadrics network.

InfiniBand Architecture Overview

InfiniBand is an open-standard interconenct. IBA defines a System Area Network to connect multiple platforms. In the InfiniBand network, processing nodes and I/O nodes are connected to the fabric by *Host Channel Adapters* (HCAs) and *Target Channel Adapters* (TCAs) respectively, as shown in Figure 2.6. IB Verbs specify the semantic interface between HCA and consumers. A *Channel Adapter* (CA) that is installed in processor nodes and I/O units generates and consumes packets as well as initiating DMA operations. It connects to the host through the PCI-X or PCI-Express bus.

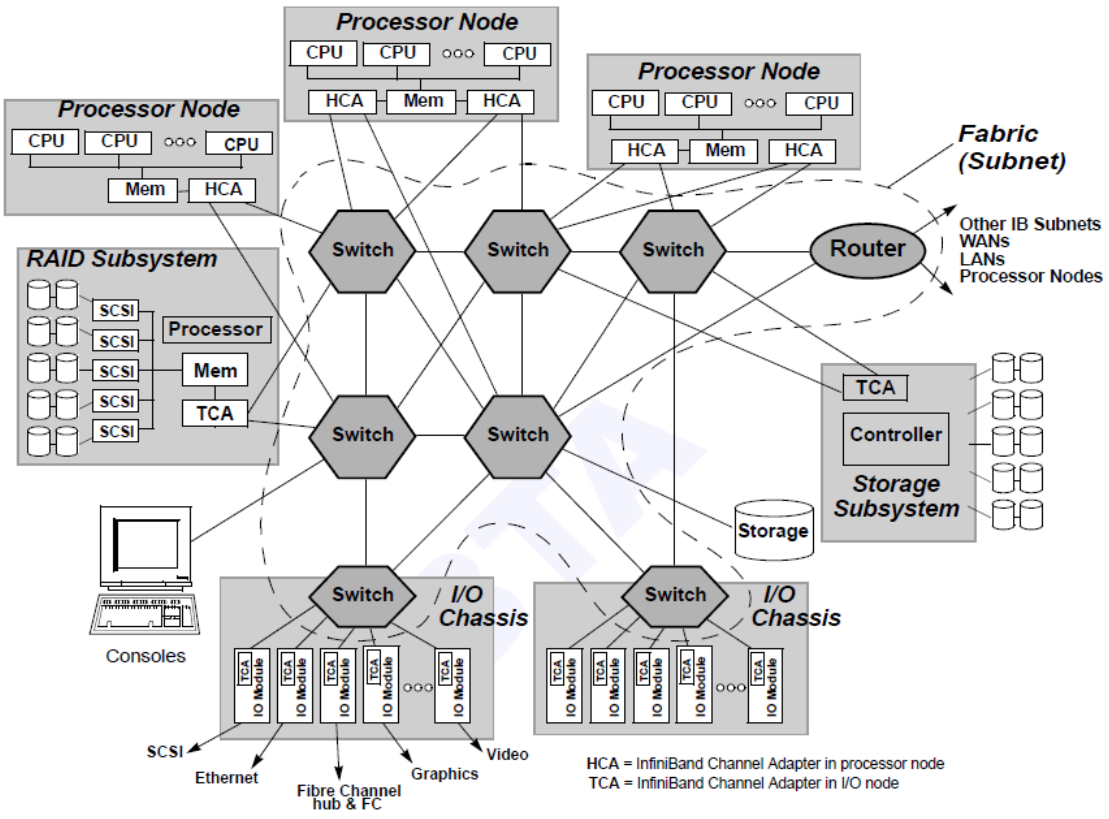

Figure 2.6 IBA System Area Network [31].

InfiniBand Protocol Stack

The communication in InfiniBand is based on the concept of the *Queue Pair* (QP) [31], which serves as a virtual communication port. The structure of IBA communication stack is shown in Figure 2.7. Each QP has two queues: a *send queue* and a *receive queue*. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where the received data is to be stored. Two QPs on different nodes can be connected to each other by a logical bi-directional communication channel. An application can have multiple QPs. Communication operations are described by *Work Queue Requests* (WQR), which are then submitted to the queue pairs. Once submitted, a WQR becomes a *Work Queue Element* (WQE). The completion of communication requests is reported through *Completion Queues* (CQ).

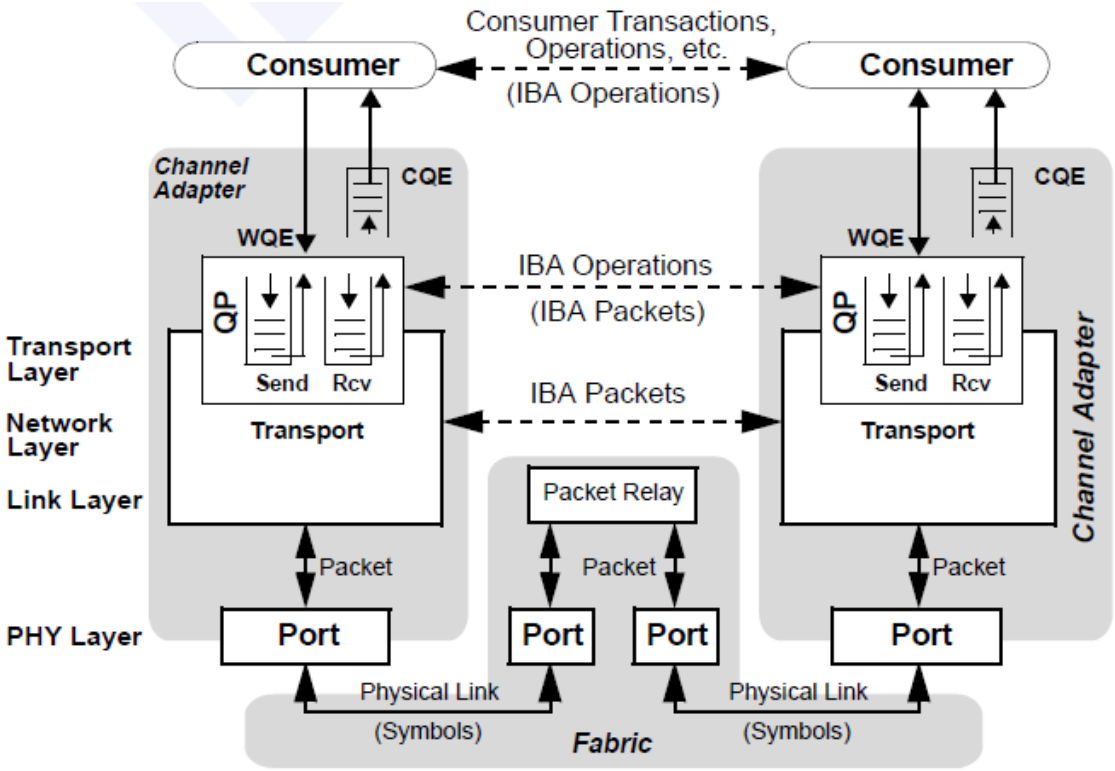

Figure 2.7 IBA communication stacks [31].

InfiniBand Transport Services

IBA provides five transport services: *Reliable Connection* (RC), *Unreliable Connection* (UC), *Reliable Datagram* (RD), *Unreliable Datagram* (UD), and *Raw Datagram* (RD). The current generation IBA adapters support RC, UC, and RD modes. Both RC and UC are connection-oriented services, which require the consumer to initiate a communication establishment procedure (a connection) with the target node before any data transfer can take place. Both RC and RD are acknowledged services, which means both transport protocols guarantee that all data is delivered in order. This dissertation is concerned about the RC mode where RDMA is supported.

Datagram services including RD and UD are connectionless. They allow the consumer of the QP to communicate with any unreliable datagram QP on any node.

Similarly, the receive operation accepts incoming messages from any unreliable datagram QP on any node. The RD is more like a data link service that allows a QP to send and receive raw datagram messages. It is connectionless and unreliable. RD has two types of raw datagram (EtherType and IPv6).

When a receiver handles incoming receives on a given QP, RC or UD, the Consumer must post the number of receive WQRs. It is difficult when the Consumer cannot predict the incoming rate on a given QP. To address this problem, the *Shared Receive Queue* (SRQ) concept, on the other hand, allows a set of receive queues to draw from a common pool of receive WQE - the shared receive queue. The SRQ contains WQEs that can be used to receive incoming data on any RC or UD QP that is associated with the SRQ.

The latest InfiniBand network cards from Mellanox Technologies [44] introduce support for a new InfiniBand transport service: *eXtended Reliable Connection* (XRC). The XRC transport attempts to give the same feature set of RC while providing additional scalability for multi-core clusters. [36] In RC, each process is required to have a connection to each other process in the cluster for full connectivity. Instead, XRC allows a single process to have only one connection per destination node. Given this capability to reduce the number of required connections in RC mode, the connection memory required can be potentially reduced by a factor equal to the number of cores per node.

InfiniBand Channel and Memory Semantics

IBA offers both channel semantics and memory semantics for communications. Channel semantics is also called Send/Receive. In this semantics, the message only specifies the destination's QP without naming the memory space of the destination. On the other hand, in the memory semantics the initiating consumer directly writes or read

to/from the virtual memory space of a remote node. The remote node is not involved in the data transfer. I will be only using the memory semantic in this dissertation, which includes RDMA Read, RDMA Write and Atomic operations.

RDMA Read reads from a virtually contiguous buffer on a remote node and writes the data to a local memory buffer. RDMA Write writes a virtually contiguous buffer onto a remote node. The virtually contiguous buffer can gather from a list of local buffer segments. The RDMA Atomic operation is a combined Read, Modify, and Write atomic operation on a remote 64 bit word.

2.2.4 OFED

OpenFabrics Enterprise distribution (OFED) [82] is a high-performance server and storage connectivity software for RDMA and transport offload hardware solutions. The OFED, maintained by OpenFabrics Alliance [82], collaborates the development and testing by all major InfiniBand and iWARP Ethernet vendors.

Verbs is one of the core InfiniBand modules and an abstract description of functionalities of a HCA. It provides infrastructure for kernel/user communication, handles memory pinning, pass most operations on to device-specific driver and provides direct path to the HCA driver. Mellanox Technologies OFED is based on the InfiniBand verbs layer. It is a single *Virtual Protocol Interconnect* (VPI) [90] software stack based on the OFED Linux stack, which supports all Mellanox network adapters.

Three communication operations are provided in OFED: send/receive, RDMA operations and Atomic. Both reliable connection and unreliable datagram services have been implemented on HCAs. Similar to GM, memory buffers must be registered with HCA before being used. For parallel applications, OFED offers MVAPICH [47] MPI

implementation from Ohio State University (OSU). Existing designs of MPI over InfiniBand use send/receive operations for small data messages and control messages, and RDMA operations for large data messages.

2.3 Communication Modeling

To design collective communication algorithms on different systems, there is a need to have a cost model to estimate the lower bounds for the latency and bandwidth cost of collective communication operations. In addition, design and performance of collective operations are influenced by the network system characteristics, including the port modeling.

2.3.1 Hockney's Model

Hockney has proposed a simple model for point-to-point communication operations, [25] as in Equation 2.1:

$$t = t_0 + \frac{m}{r_\infty} \qquad (2.1)$$

where r_∞ is the *asymptotic bandwidth*, which is the maximum achievable bandwidth. t_o is the startup time, and m/r_∞ represents the transmission time in sending an m-byte message through a network with bandwidth r_∞.

Hockney's model has some advantages. Firstly, it is a simple model that is a linear function of the message size m. Secondly, r_∞ and t_o represent two fundamental quantities of the network. Thirdly, it is architecture-independent. It can be applied to networks with different architectures by changing the value of parameters. While there are other models available such as the postal model [5], LogP model [19], LogGP [1],

and PlogP [34]. Hockney's model is sufficient for the study of the algorithms in chapter 3 of this dissertation, as we are not concerned with congestion in the network.

2.3.2 Port Modeling

In a direct network, each node has a bi-directional link to all other nodes. The port model of a system refers to the number of links that can be used at the same time. If each node can only send and receive messages over one of its links at a time, this is called a *one-port* communication. In an *all-port* system, a node can send and receive data over all the links at the same time. If the number of links that can be used at once is greater than one but less than the number of links available, the port modeling is called *k-port*.

2.4 Summary

In this chapter, I surveyed the related background for this dissertation. I discussed the different components of high-performance clusters including MPI parallel programming paradigm, the high-performance interconnects and their user-level messaging layers. I also introduced Hockney's communication modeling and port modeling. In the following chapters, I will propose and implement different algorithms and techniques to improve the performance of collective communication over Quadrics and InfiniBand SMP/multi-core clusters.

Chapter 3: RDMA-based Multi-port Collectives on Multi-rail QsNet^{II} Clusters

Network bandwidth usually becomes the performance bottleneck for today's most demanding applications [16, 38]. Recently, a new technique has been emerging that uses multiple independent networks/rails or multi-port NICs to overcome bandwidth limitations and enhance fault tolerance. Existing examples include native multi-rail support on Quadrics and dual-port NICs in InfiniBand and Myrinet. Quadrics QsNet^{II} uses multiple NICs per node to construct a multi-rail cluster network, in which the *i*-th NIC connects to the *i*-th rail.

There are two basic ways in distributing the traffic over multiple rails. One is to use *multiplexing*, where messages are transferred over different rails in a round robin fashion. The other method is called *message striping*, where messages are divided in multiple chunks and sent over multiple rails simultaneously. Quadrics has a native support for a simple even message striping over multi-rail QsNet^{II} networks only for large point-to-point messages through its Elan put and get, SHMEM put and get, and Tports send/receive functions. However, it does not support multi-rail collectives. In this chapter, I devise and evaluate multi-port collective communications on Multi-rail Quadrics QsNet^{II} networks [58, 60].

3.1 Related Work

Study of collective communication operations has been an active area of research. Thakur and his colleagues discussed recent collective algorithms used in MPICH [80]. They have shown some algorithms perform better depending on the message size and the number of processes. In [88], Vadhiyar et al. introduced the idea of automatically tuned

collectives in which collective communications are tuned for a given system by conducting a series of experiments on the system. Both works are implemented based on MPI point-to-point communications. The authors in [57] analyzed the performance of collective communication operations under different communication cost models.

Petrini, et al. described how they improved the effective performance of ASCI Q supercomputer interconnected with a Quadrics QsNet. [56] A number of papers have been reported on the use of RDMA in the design and implementation of collectives on modern networks. Roweth and his colleagues studied how different collective algorithms have been devised and implemented on $QsNet^{II}$ [70, 71]. Sur, et al. proposed efficient RDMA-based alltoall broadcast and alltoall personalized exchange for InfiniBand clusters [76, 78]. In [84], Tipparaju and Nieplocha used the concurrency available in modern networks to optimize *MPI_Allgather()* on InfiniBand and $QsNet^{II}$. All the above research is done under single-rail systems, while this chapter adapts multi-port algorithms over multi-rail Quadrics $QsNet^{II}$ networks.

On multi-rail systems, Coll and his associates [16] did a comprehensive simulation study on static and dynamic allocation schemes for multi-rail systems. The authors in [38] designed an MPI-level multi-rail InfiniBand clusters. However, their work addressed only point-to-point communications. On multi-port collectives, Chan et al. [15] redesigned and re-implemented a number of multi-port MPI collectives for IBM Blue Gene/L using MPI point-to-point communications, and not RDMA as studied in this chapter. Recently, the New Madeleine communication library [3] is designed for multi-rail message transfers across multiple heterogeneous high-performance networks.

3.2 Experimental Framework

The experiments were conducted on a 4-node dedicated SMP cluster interconnected with two QM500-B Quadrics QsNet^{II} NICs per node, and two QS8A-AA QsNet^{II} E-series 8-way switches. The QM500-B PCI-X network adapter for Quadrics QsNet^{II} [65] uses Elan 4 network processor and has 64 Mbytes onboard DDR-SDRAM memory.

Each node is a Dell PowerEdge 6650 that has four 1.4 GHz Intel Xeon MP Processors with 256KB unified L2 cache, 512KB unified L3 cache, and 2GB of DDR-SDRAM on a 400 MHz Front Side Bus. Each NIC is inserted in a 64-bit, 100 MHz PCI-X slot. The operating system is the Vanilla kernel version 2.6.9. The Quadrics software installed is the latest "Hawk" release with the kernel patch qsnetp2, kernel module 5.10.5qsnet, QsNet Library 1.5.9-1, and QsNet^{II} Library 2.2.11-2. Test codes were launched by the *pdsh* [53] task launching tool, version 2.6.1. The MPI implementation is the Quadrics MPI, version MPI.1.24-49.intel81.

3.3 Motivation

In this section, I will perform a feasibility study of the potential performance that could be gained using multi-port message striping in the algorithms on a multi-rail system. My intention in this section is to show while point-to-point messages benefit from message striping, only a couple (Elan and MPI) collectives that are currently implemented on top of point-to-point Tports or *elan_put()* will gain from multi-rail striping.

In the following, I first present the performance of Elan *put* and *get*, Tports send/receive, as well as MPI point-to-point under single-rail and dual-rail QsNet^{II} on my platform (SHMEM *put* and *get* also stripe large messages). I will then demonstrate the

performance of Elan collectives, and the *MPI_Scatter()* that does not have any Elan counterpart. Please note that the Elan collectives are directly used by MPI collectives.

The point-to-point experimentation is done with the uni-directional, bi-directional, and both-way traffics. In the uni-directional bandwidth test, the sender transmits a message repeatedly to the receiver, and then waits for the last message to be acknowledged. The bi-directional test is the ping-pong test where the sender sends a message and the receiver upon receiving the message immediately replies with a message of the same size. This is repeated a sufficient number of times to eliminate the transient conditions of the network. In the both-way test, both the sender and receiver send data simultaneously. This test puts more pressure on the communication subsystem and the PCI-X bus.

3.3.1 Elan RDMA Performance

Figure 3.1 presents the bandwidth performance of the RDMA Write using the *pgping* micro-benchmark available in the Elan Library. It is evident that the bandwidth is doubled in the dual-rail system. The both-way single-rail and dual-rail *elan_put()* bandwidths are 670MB/s and 1332 MB/s, respectively. The bandwidth for *elan_get()* is almost the same as *elan_put()* in each case (not shown).

The Elan RDMA Write short message latency does not change much between single-rail and dual-rail. The latency varies between 2 μs to 2.77 μs for a 4-byte message. The *elan_get()* short message latency is slightly larger than the RDMA write. That is why I decided to use *elan_put()* in the design and implementations of collectives.

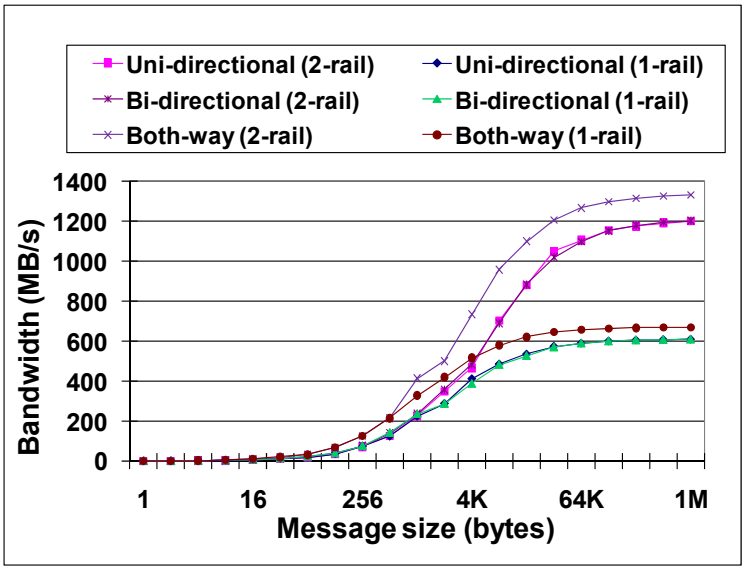

Figure 3.1 Elan RDMA Write performance.

3.3.2 Tports Performance

Figure 3.2 shows the Tports bandwidth. Tests are done using the *tping* micro-benchmark (except for the uni-directional case, where I wrote my own code). Like the Elan RDMA, the dual-rail Tports bandwidth outperforms the single-rail bandwidth in each case. The single-rail T-ports bandwidth is roughly the same as RDMA bandwidth. However, dual-rail bandwidth falls short of RDMA. The short message latency is slightly larger than the RDMA.

3.3.3 MPI Send/Receive Performance

Figure 3.3 compares the MPI bandwidth under different cases. Unlike the both-way, the uni-directional and bi-directional MPI bandwidths for dual-rail are almost doubled. This shows that the MPI point-to-point implementation over Tports mostly benefit from striping in the dual-rail QsNet^{II}. The short message MPI latency is close to that of the T-ports (not shown here).

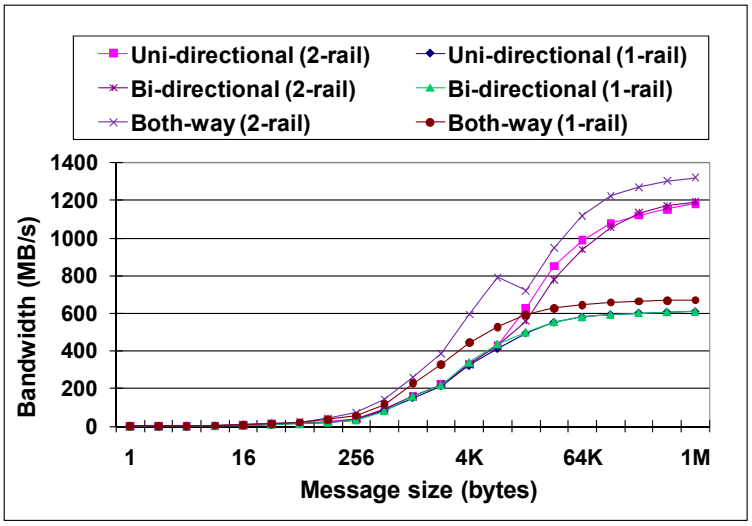

Figure 3.2 T-port send/receive performance.

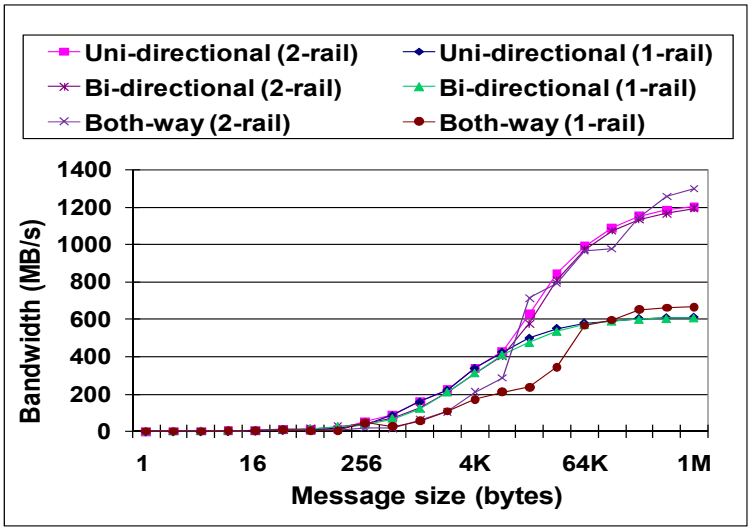

Figure 3.3 MPI send/receive performance.

3.3.4 Collective Performance

Figure 3.4 depicts the aggregate bandwidth for the Elan hardware and software broadcasts, gather, allgather, and alltoall, as well as *MPI_Scatter()*. For the Elan collectives, I have used the *gping* micro-benchmark in the Elan Library. For the MPI collectives, I have written my own code to measure their performance. From the results, except for the gather, all other Elan collectives do not benefit from the dual-rail QsNet^{II}.

It should be mentioned that both gather and allgather use single-port algorithms in the Elan library. *MPI_Scatter()* is implemented on top of Tports, so it achieves larger bandwidth under dual-rail.

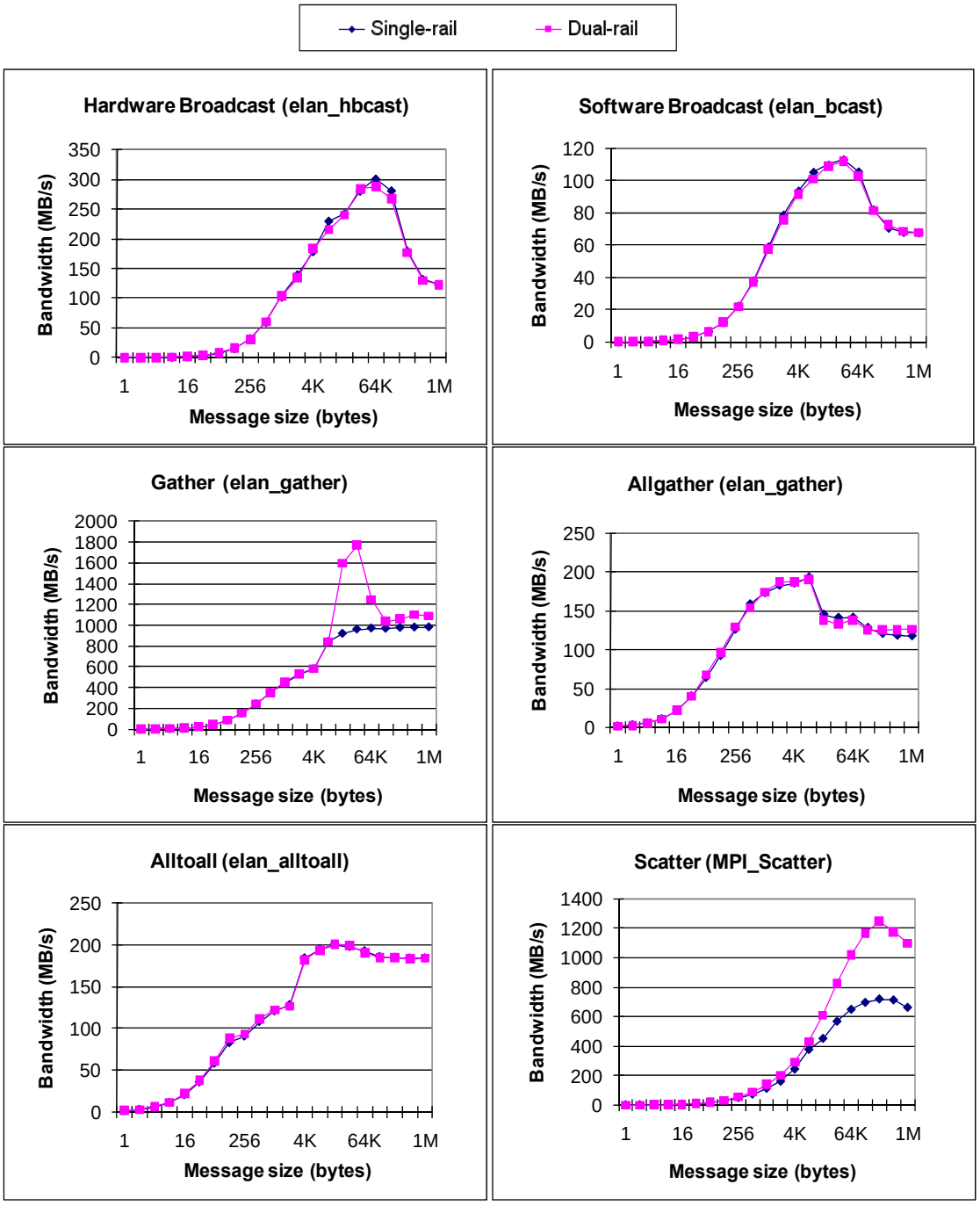

Figure 3.4 Elan collectives and *MPI_Scatter()* bandwidth on dual-rail QsNet^{II}.

Efficient implementation of collective operations is one of the keys to the performance of parallel applications. Given the multi-rail performance offered at the Elan and Tports levels, excellent opportunities exist for devising efficient collectives for such systems. Basically, there are two ways to improve the performance of collectives on multi-rail systems. One is to implement single-port collective communication algorithms that gain multi-rail striping from the underlying communication subsystem. This is the approach currently used for *MPI_Scatter()*. However, this will only improve the performance for large messages. The second approach that I propose is to design and implement multi-port algorithms for multi-rail systems that also benefit from the striping feature supported by QsNet^{II}. In this regard, I have used some known multi-port algorithms [10] and implemented them on the dual-rail QsNet^{II} network directly at the Elan level using RDMA Write.

3.4 Collective Algorithms

In this section, I provide an overview of some well-known algorithms for scatter, gather, allgather, and alltoall personalized exchange, and adapt them on top of Quadrics QsNet^{II} networks. In the following discussion, N is the number of processors (or processes) and k is the number of ports in the multi-port algorithms (equal to the number of available rails). In the k-port (or multi-port) modeling, each process has the ability to simultaneously send and receive k messages on its k links. The assumption is that the communication between any pair of processes has the same cost, and the number of processes is a power of $(k + 1)$. Otherwise, dummy processes can be assumed to exist until the next power of $(k + 1)$, and the algorithms apply with little or no performance loss.

3.4.1 Scatter

The *spanning binomial tree* algorithm [43] can be extended for *k*-port modeling. In this algorithm, the scattering process sends *k* messages of length *N/(k* + 1*)* each to its *k* children. Therefore, there are (*k* + 1) processes having *N/(k* + 1*)* different messages. These processes, at step 2, send one *(k* + 1*)*-th of their initial message to each of their immediate *k* children. This process continues and all processes are informed after $\lceil \log_{k+1} N \rceil$ communication steps. Using Hockney's model [25], the total communication time, *T*, is:

$$T = (t_s \times \log_{k+1} N) + (l_m \times \tau) \sum_{i=1}^{\log_{k+1} N} (k+1)^{(\log_{k+1} N)-i}$$
$$T = (t_s \times \log_{k+1} N) + \frac{N-1}{k}(l_m \times \tau) \qquad (3.1)$$

where t_s is the message startup time, l_m is the message size in bytes, and τ is the time to transfer one byte.

The above algorithm has a logarithmic number of steps, therefore suitable for short messages. Another algorithm, for large messages, is the *Direct* algorithm, which is the extension of *sequential tree* algorithm for *k*-port modeling. At each step, the source process sends its *k* different messages to *k* other processes. There are a total of (*N* -1)/*k* communication steps. Therefore, the total communication time, *T*, is:

$$T = \frac{N-1}{k} \times (t_s + l_m \times \tau) \qquad (3.2)$$

3.4.2 Gather

Gather is the exact reverse of scatter and so the same spanning binomial tree algorithm extended for *k*-port modeling can be used. However, the communication starts

from the leaf processes and messages are combined in the intermediate processes until it reaches the root. The total communication cost is the same as Equation (3.1).

3.4.3 Allgather

I provide an overview of three well-known allgather algorithms: *Direct, Standard Exchange* [8], and *Bruck* [10]. The Direct algorithm is used for medium to large messages. Standard Exchange is targeted for short to medium size messages, while the Bruck algorithm typically performs better for short messages.

Direct Allgather Algorithm: The *Direct* allgather algorithm is the extension of *sequential tree* algorithm for *k*-port modeling and suitable for medium to large messages. In each step, each process sends its own message to *k* other processes in a wrap-around fashion. There are a total of $\left\lceil \frac{N-1}{k} \right\rceil$ communication steps. Using Hockney's model, the total communication cost, *T*, is:

$$T = \left\lceil \frac{N-1}{k} \right\rceil \times (t_s + l_m \times \tau) \qquad (3.3)$$

Standard Exchange Allgather Algorithm: The *Standard Exchange* allgather algorithm [8] is the extension of *Recursive Doubling* algorithm [80] for *k*-port modeling, and works for power of (*k*+1) processes. It should generally perform well for short and medium size messages. In the *k*-port Standard Exchange algorithm, processes are divided into $N/(k+1)$ groups of (*k*+1) processes each. Processes are grouped as (0, 1, …, *k*), (*k*+1, *k*+2, …, 2(*k*+1)-1), …, (*N* - (*k*+1), *N* - (*k*+1)+1, …, *N* - 1). In step 1, all processes within a group exchange their messages using *k*-port. At the end of this step, each process has (*k*+1) messages. In step 2, process *p* exchanges all its messages with processes (p + (*k*+1)) *mod N*, (p + 2(*k*+1)) *mod N*, …, (p + *k*(*k*+1)) *mod N*. At the end of this step, each

process has $(k+1)^2$ messages. This continues to the step $log_{k+1}\ N$. At each step i of this algorithm, each process sends messages of size $(k+1)^{i-1}$ to k other processes. Note this algorithm needs correction steps when the number of processes is not a power of $(k+1)$. Figure 3.5 illustrates the 2-port Standard Exchange algorithm for nine processes. The total communication cost, T, is:

$$T = t_s \times \log_{k+1} N + \frac{N-1}{k}(l_m \times \tau) \qquad (3.4)$$

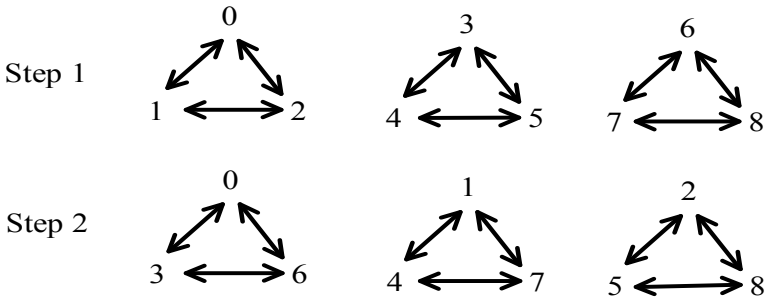

Figure 3.5 Standard Exchange algorithm for 9 processes under 2-port modeling.

Bruck Allgather Algorithm: The *Bruck* allgather algorithm [8] works on any number of processes, and is proposed to improve the performance for small messages. Figure 3.6 illustrates the 2-port Bruck algorithm for nine processes. The allgather operation among N processes can be represented as a sequence of process-memory configurations. Each process has an N-block output buffer. Initially, local data is placed at the top of the output buffer.

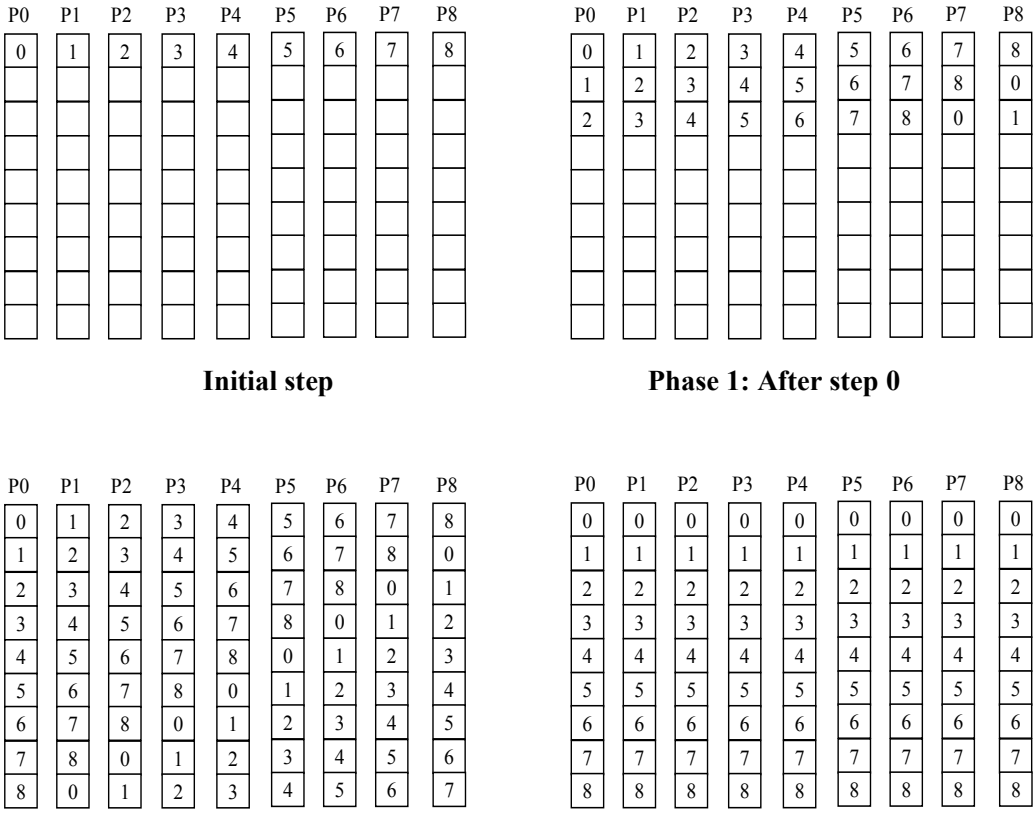

Figure 3.6 Bruck allgather algorithm for 9 processes under 2-port modeling.

The algorithm consists of two phases. Phase 1 has $\lfloor \log_{k+1} N \rfloor$ steps. In each step i of phase 1, process p sends all its data to processes $(p - (k+1)^i)$, $(p - 2(k+1)^i)$, …, $(p - k(k+1)^i)$, and stores the data it receives from processes $(p + (k+1)^i)$, $(p + 2(k+1)^i)$, …, $(p + k(k+1)^i)$ at the end of the data it already has. An additional step is required if N is not a power of $(k+1)$, where each process sends the first $(N - (k+1)^{\lfloor \log_{k+1} N \rfloor})$ blocks from the top of its output buffer to the destination processes and appends the received data to the end of its current data. The second phase consists of a single round local memory shift. The total communication cost is the same as Equation (3.4).

3.4.4 Alltoall Personalized Exchange

I also provide an overview of three well-known algorithms for alltoall Personalized Exchange: *Direct, Standard Exchange* [8], and *Bruck* [10].

Direct Alltoall Personalized Exchange Algorithm: A lower bound for alltoall personalized exchange time is $(N-1)/k$ since each process must receive $N-1$ different messages and it can only receive at most k messages at a time. A simple algorithm is based on the extension of the Direct algorithm for k-port modeling. The processes are arranged in a virtual ring. That is, at step i, process p sends its message to processes $(p + (i-1)k + 1) \bmod N$, $(p + (i-1)k + 2) \bmod N$, …, $(p + ik) \bmod N$. The modulus operation avoids sending messages to a single destination. The communication cost is the same as Equation (3.3).

Standard Exchange Alltoall Personalized Exchange Algorithm: The Standard Exchange algorithm [8] for alltoall personalized exchange has the same grouping as the Standard Exchange algorithm for allgather. However, each node sends $N/(k+1)$ message at a time. The total communication cost, T, is the same as Equation (3.3).

Bruck Alltoall Personalized Exchange Algorithm: The Bruck algorithm [10] of alltoall personalized exchange operation among N processors can be represented as a sequence of processor-memory configurations. Each processor-memory configuration has N columns of N blocks each. Columns i represents the processor P_i , and the block j represents the data j to be sent to processor P_j.

Bruck algorithm [10] for alltoall personalized exchange operation consists of three phases. The first and the third phases only require local memory movement in each processor. The first phase does a local copy and shift of the data blocks such that the data block to be sent by each processor to itself is at the top of the column. In each

communication step *j* of the second phase, process *i* sends to rank (*i* + *kj*) (with wrap-around) all those data blocks whose *j*th bit is 1, receives data from rank (*i* - *kj*), and stores the incoming data into blocks whose *j*th bit is 1 (overwriting the data which was just sent). All communications are independent, so *k* communications can be combined together under *k*-port modeling. The final phase does a local inverse shift of the blocks to place the data in the right order. This algorithm also takes $\lceil \log_{k+1} N \rceil$ steps communications. The total communication cost is same as Equation (3.3). Figure 3.7 shows the example of this algorithm with four communication steps on nine nodes.

3.5 Implementation Issues

The algorithms are devised based on two-port *put*-based algorithms, where a sending process has direct control in sending messages simultaneously over the two rails using the *elan_doput()* function. When a message is larger than a threshold (1KB), even message striping is used over the two rails. When a message is sent, the sending process uses the *elan_wait()* to make sure the user buffer can be re-used. Note that in the implementation of the algorithms, processes do not synchronize with each other.

Quadrics supports event notification for both single-rail and multi-rail systems. The destination event (*devent)* is set once in each rail. A target process may call *elan_initEvent()* once for each rail and then wait on each *ELAN_EVENT* to be returned. This guarantees a message has been delivered in order in its entirety.

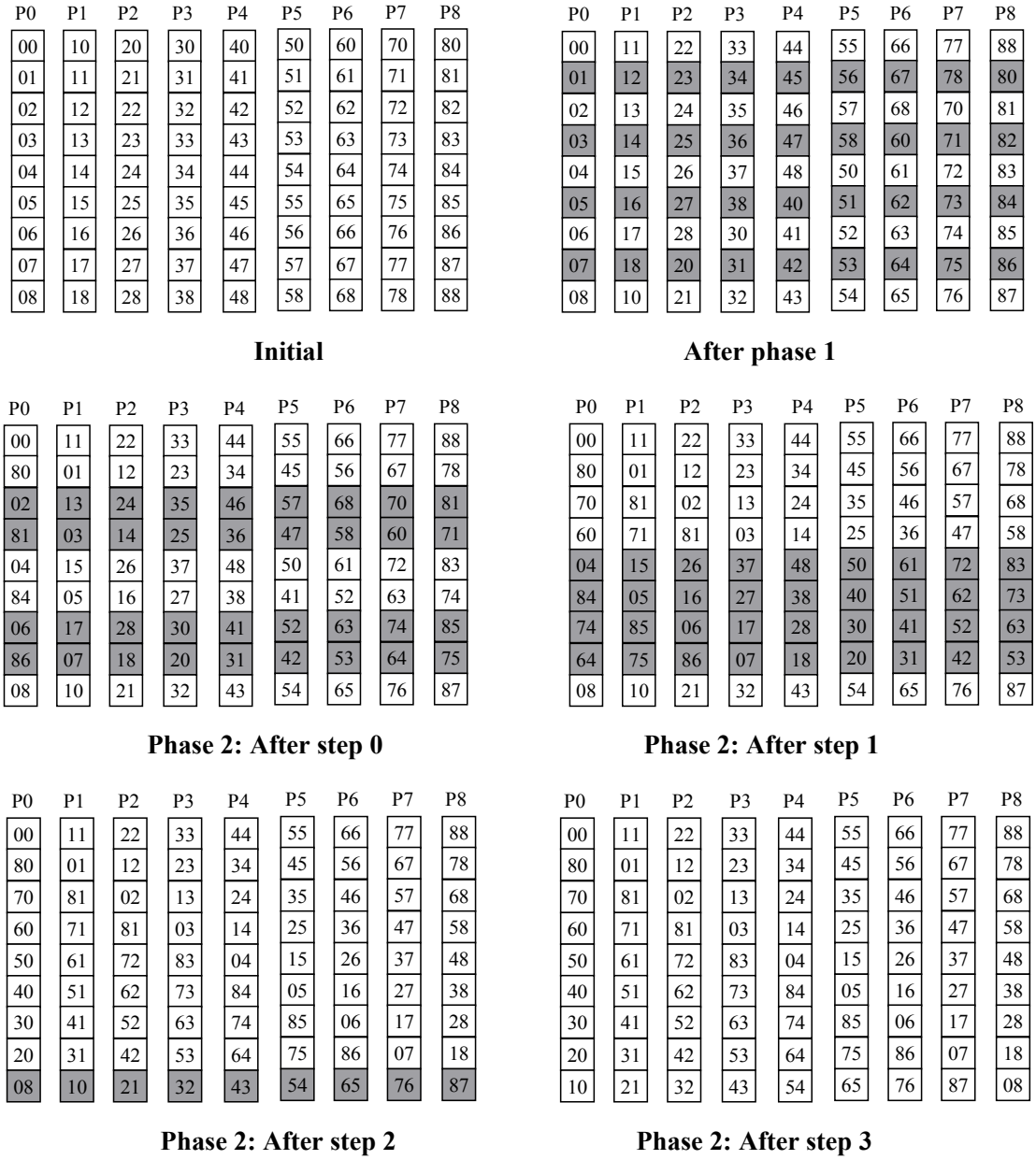

Figure 3.7 Bruck alltoall algorithm for 9 processes under 2-port modeling.

Memory registration/deregistration is a costly operation. Unlike InfiniBand, QsNet^{II} does not need memory registration and address exchange for message transfers. This eases the implementation, and effectively reduces the communication latency. Although I have implemented the proposed algorithms on dual-rail clusters, the algorithms and implementations are robust enough to be used in multi-rail clusters.

3.6 Performance Analysis

In this section, I present the performance of the multi-port collectives introduced in Section 3.4 when they are implemented directly at the Elan layer using RDMA Write on multi-rail QsNet^{II} clusters with striping support.

3.6.1 Evaluation of Scatter

I have implemented the multi-port spanning Binomial tree algorithm and Direct algorithm for scatter operation on multi-rail QsNet^{II} systems at the elan level using RDMA Write. Figure 3.8 compares the performance of the two scatter algorithms on the dual-rail QsNet^{II}. As expected, the Binomial tree algorithm is superior for short messages, while the Direct algorithm has a much better performance for medium and large messages. Figure 3.8 also presents the scalability of the implementation. The scalability figures verify that indeed the Binomial tree algorithm is the better algorithm for short messages with increasing system size.

3.6.2 Evaluation of Gather

The multi-port spanning Binomial tree algorithm for Gather operation has been implemented on multi-rail QsNet^{II} systems using RDMA Write feature. Figure 3.9 compares the performance of the gather algorithm with the *elan_gather()*. The results are very promising as the implementation is much better than the native implementation except slightly for messages less than 512 bytes. The proposed multi-port gather gains an improvement of up to 2.15 for 4KB message. The scalability plots in Figure 3.9 verify the superiority of the gather algorithm for medium and large messages. However, it does show that with increasing number of processes *elan_gather()* is better for very short messages.

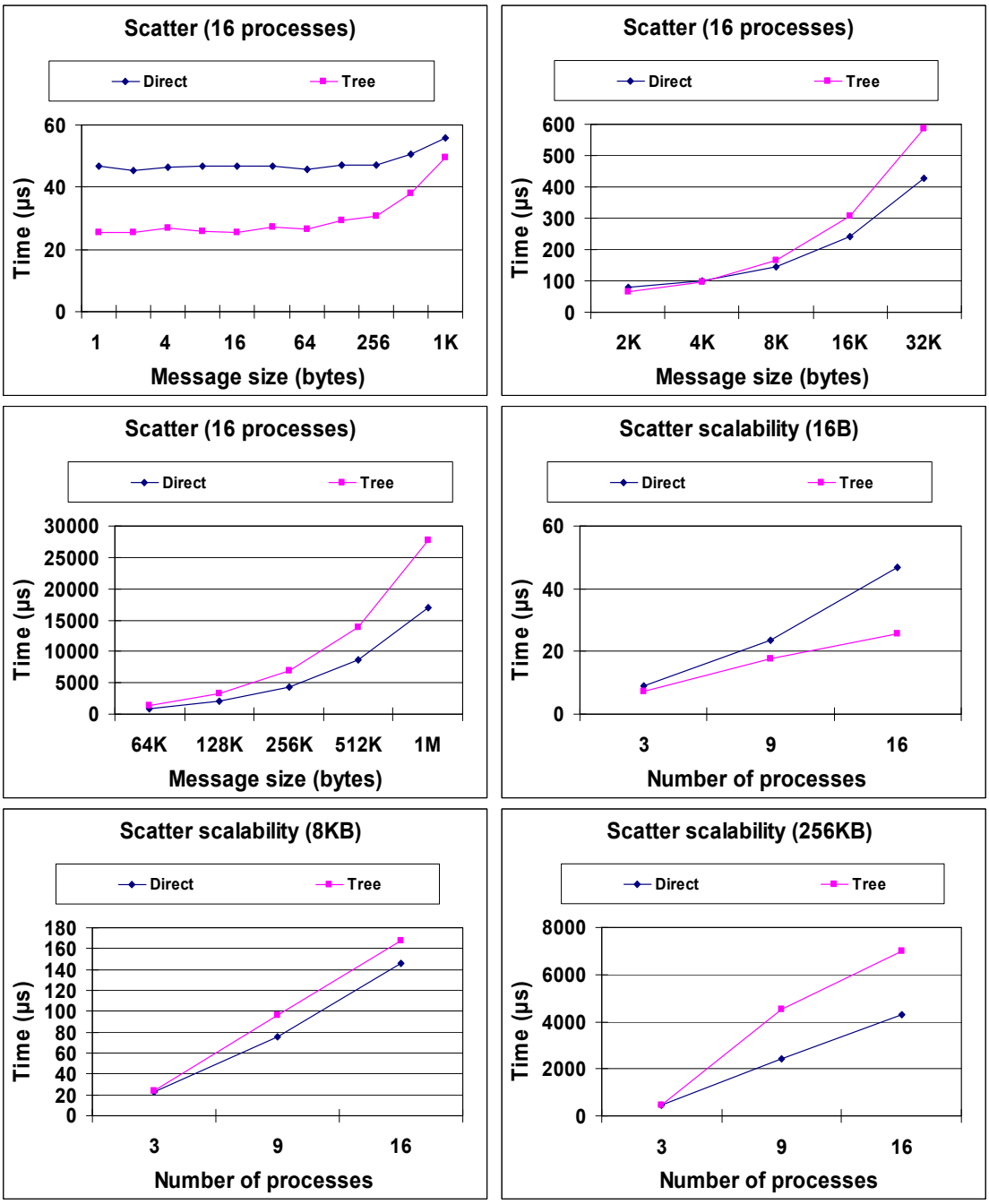

Figure 3.8 Scatter performance and scalability.

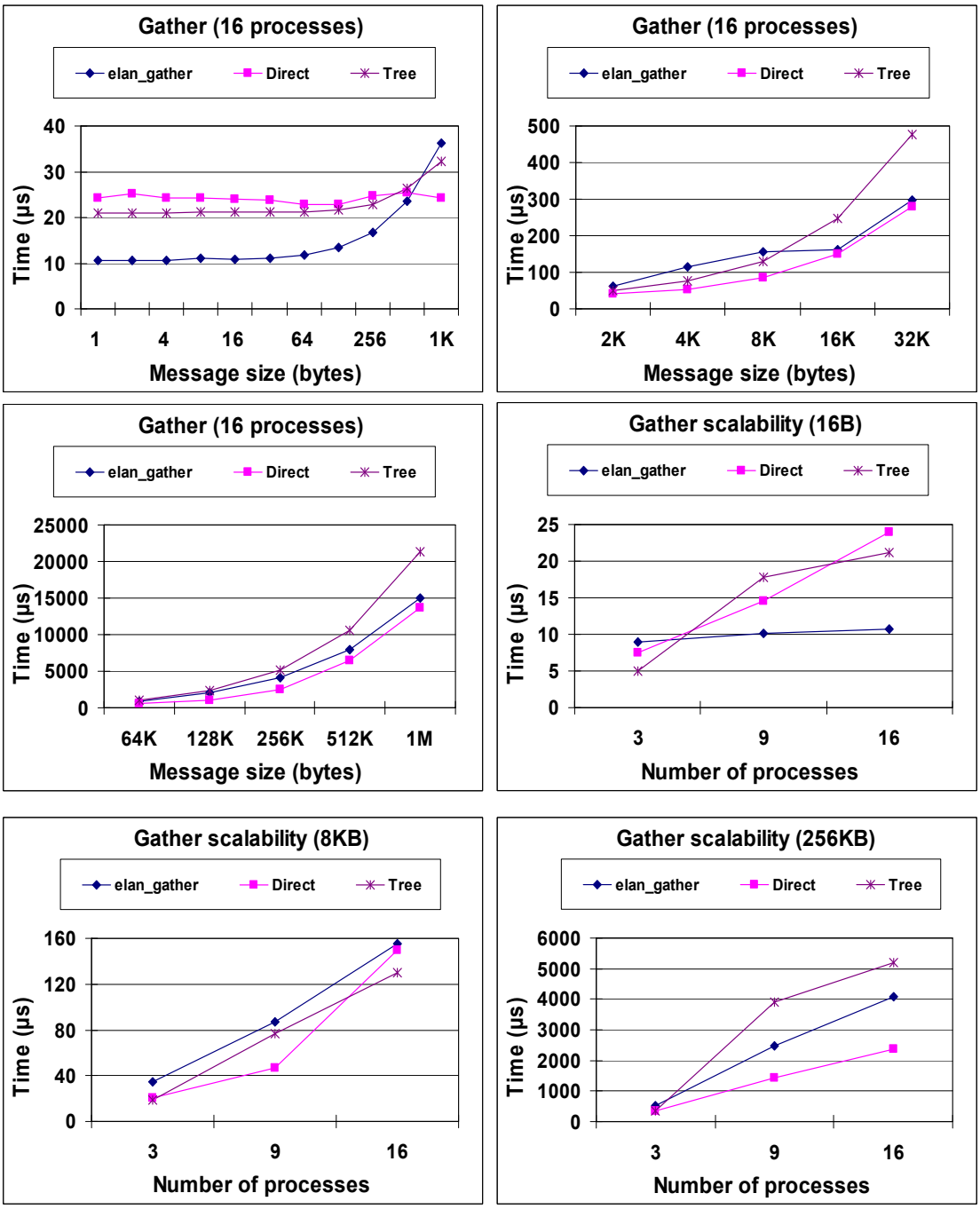

Figure 3.9 Gather performance and scalability.

3.6.3 Evaluation of Allgather

Figure 3.10 compares the performance of the three allgather algorithms, Direct, Standard Exchange, and Bruck on the dual-rail QsNet^{II} cluster. The Bruck algorithm is

superior among the three algorithms for short messages, while the Direct algorithm has a much better performance for medium and large messages. The Standard Exchange algorithm incurs a penalty, due to correction steps, when the number of processes is not a power of k+1 (16 processes in this case). Otherwise, its performance is better than the Bruck algorithm for medium size messages.

The Quadrics implementation of allgather in the *elan_gather()* performs better than my RDMA-based implementations for messages up to and including 2KB. This is most probably because Quadrics uses shared memory point-to-point communication for messages up to 2KB, where its performance is better than the intra-node RDMA. This confirms the hypothesis that traditional algorithms for short messages, such as Bruck and Standard Exchange, are suitable for flat (uniprocessor) clusters, where there is only one process per node. For SMP clusters and the emerging multi-core clusters, shared memory communication is the preferred method for intra-node communications for short messages. I will investigate these techniques to boost the performance of my algorithms for such systems in Chapter 4.

After 2KB, the Direct algorithm performs the best among all algorithms. The multi-port allgather Direct algorithm gains an improvement of up to 1.49 for 32KB messages over the native *elan_gahter()*. The platform used in my study represents a small cluster. However, the scalability plots in Figure 3.10 verify the superiority of the Direct algorithm for medium and large messages. It also shows that with increasing number of processes *elan_gather()* outperforms my algorithms for very short messages.

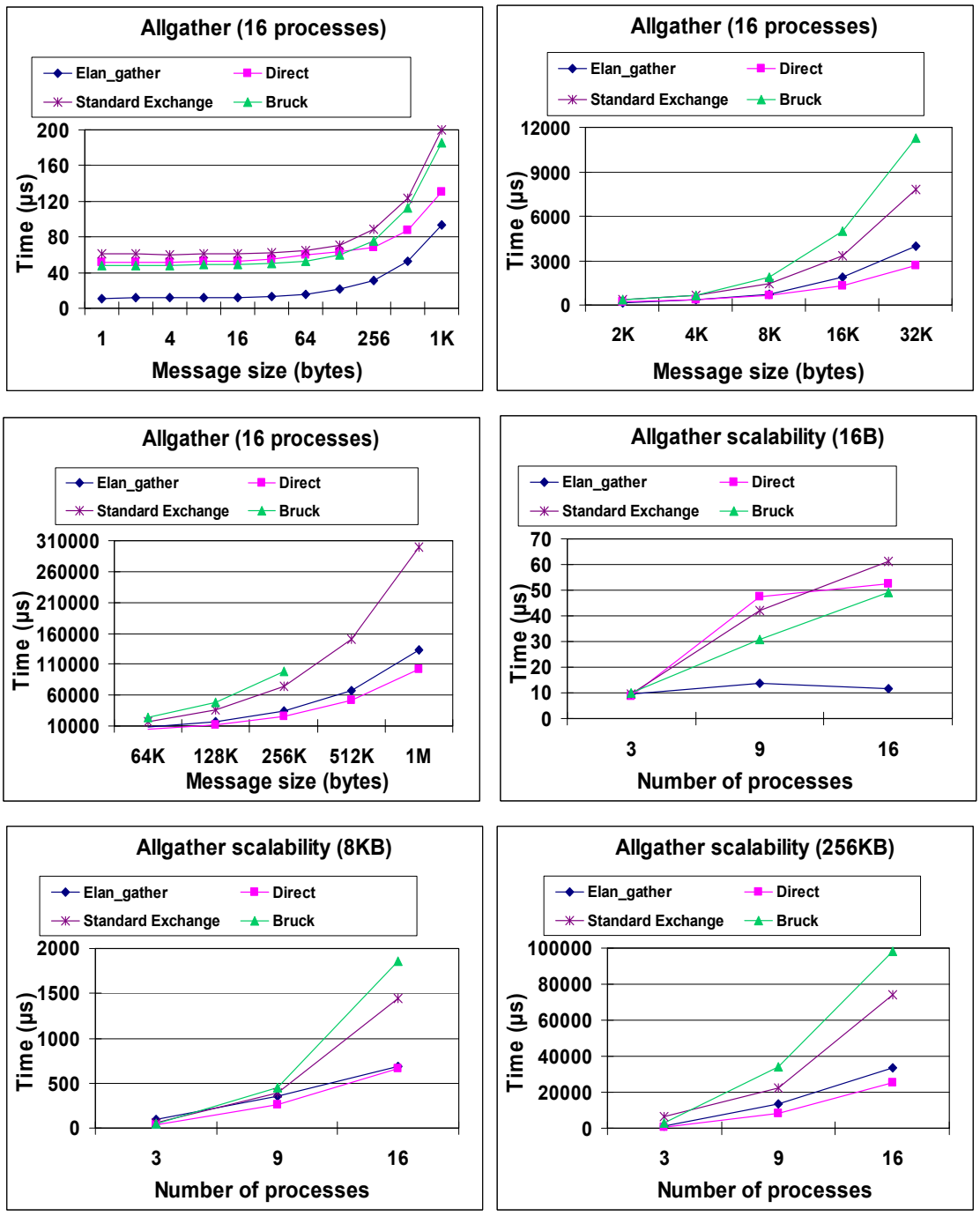

Figure 3.10 Allgather performance and scalability.

3.6.4 Evaluation of Alltoall Personalized Exchange

I have also implemented the multi-port Direct algorithm, Standard Exchange algorithm, and Bruck's index algorithm for alltoall personalized exchange on multi-rail QsNet^{II} systems using RDMA Write.

Figure 3.11 compares the performance of the three alltoall algorithms, with the *elan_alltoall()*. The results are again encouraging. My multi-port Direct alltoall algorithm and its implementation is much better than the native *elan_alltoall()* for medium size messages. In fact, the improvement is up to a factor of 2.26 for 2KB message. However, *elan_alltoall() is* better than the three algorithms for short messages up to 512 bytes. For large message sizes, my two-port algorithm is better. The scalability plots confirm these findings.

3.7 Summary

Scientific applications written in MPI often use collective communications among the parallel processes. In this chapter, I studied the communication performance on a QsNet^{II} dual-rail cluster and found that there is potential to improve the performance of collective using the multi-rail techniques, with respect to the native implementation that uses well-known algorithms. Quadrics MPI directly calls Elan collectives, therefore optimizing Elan collectives is crucial to the performance of MPI applications.

Quadrics supports point-to-point message striping over multi-rail QsNet^{II}. In this work, I have devised and implemented a number of multi-port collectives at the Elan level over multi-rail QsNet^{II} systems. These collectives include scatter, gather, allgather and alltoall personalized exchange. The multi-port Direct implementation outperforms *elan_gather()* as well as the Standard Exchange and Bruck algorithms for messages

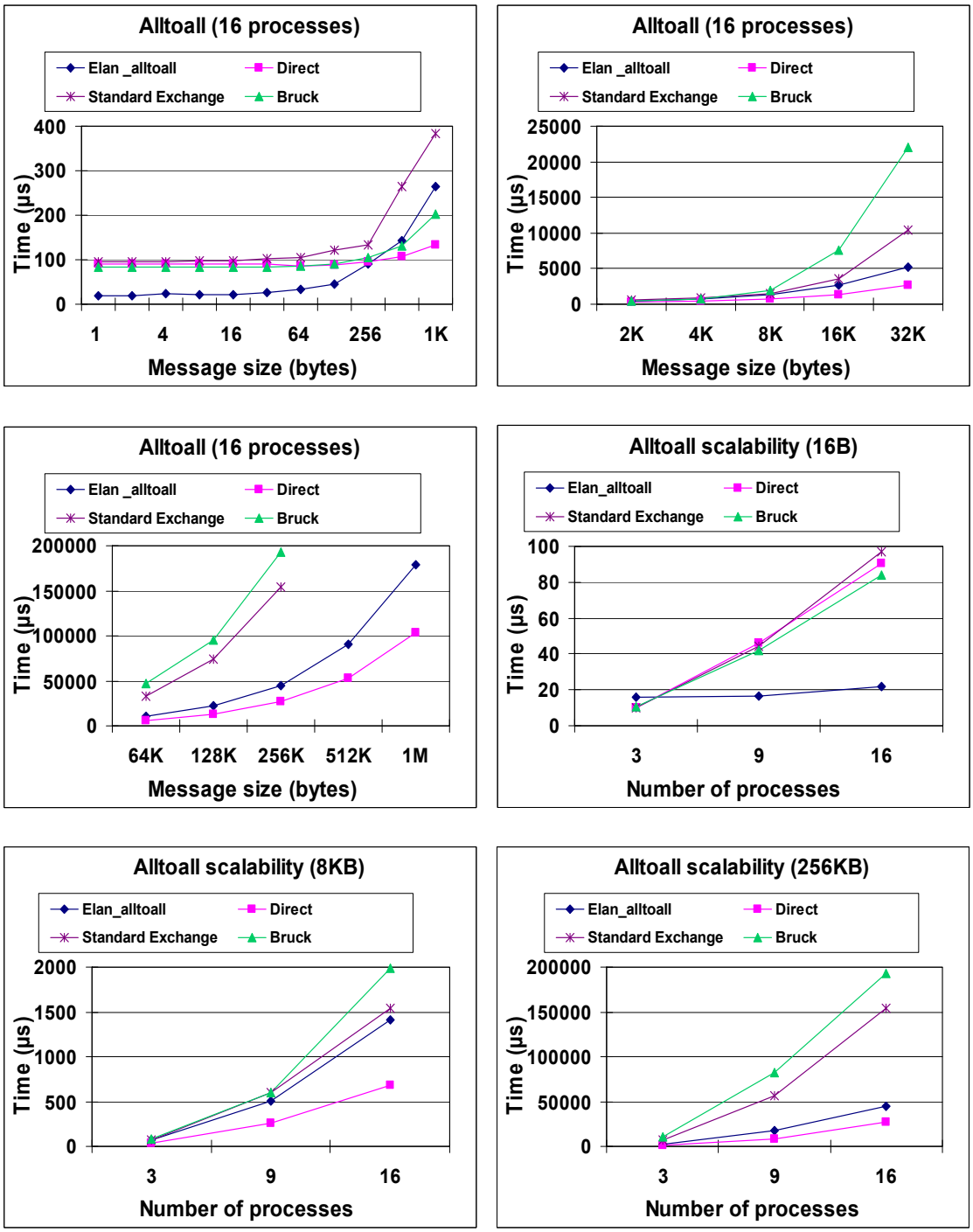

Figure 3.11 Alltoall performance and scalability.

larger than 1KB. The performance results indicate that the multi-port gather gains an improvement of up to 2.15 for 4KB message over the native *elan_gather()*. The multi-

port allgather Direct algorithm gains an improvement of up to 1.49 for 32KB messages over the native *elan_gather()* in the cluster. The proposed multi-port alltoall performs better than the *elan_alltoall()* by a factor of 2.26 for 2KB message.

The results are encouraging. However, the RDMA-based algorithms did not perform well for short messages. The native allgather and alltoall implementation has a better latency for up to 2KB and 512B, respectively. To address this deficiency in RDMA-based algorithms, in the next chapter I will propose shared memory aware algorithms to speedup the collectives for co-located processes on SMP/multi-core nodes.

Chapter 4: RDMA-based and Shared Memory Aware Multi-port Gather and Allgather on Multi-rail QsNet^{II} SMP Clusters

In Chapter 3, I designed and implemented multi-port RDMA-only scatter, gather, allgather and alltoall collectives directly at the Elan level over multi-rail QsNet^{II}. While the performance of the algorithms was excellent for medium to large messages, they lagged behind the native QsNet^{II} implementations for small size messages. This chapter seeks to propose and evaluate efficient gather and allgather for all message sizes, utilizing both shared memory and RDMA features [59, 63].

4.1 Related Work

On SMP clusters, some recent work has been devoted to improve the performance of intra-node communications on SMP nodes [11, 13, 32]. Buntinas et al. [11] have used shared buffers, message queues, Ptrace system call, kernel copy, and NIC loopback mechanisms to improve large data transfers in SMP systems. In [13], Chai and others improved the intra-node communication by using the system cache efficiently and requiring no locking mechanisms. In [32], Jin et al. implemented a potable kernel module interface to support intra-node communications. Chai et al. [14] used both shared memory and OS kernel-assisted direct copy to design efficient MPI intra-node communication. An intra-node shared memory communication for Virtual Machine environments is proposed in [28]. The shared memory communication is a one-copy approach, mapping user buffers between Virtual Machine. This thesis uses shared buffers for shared memory communications.

On collectives for SMP clusters and large SMP nodes, Sistare and his colleagues presented new algorithms taking advantage of high backplane bandwidth of shared

memory systems [74]. In [85], Tipparaju and his colleagues overlapped shared memory intra-node and remote memory access inter-node communications in devising collectives for IBM SP. However, this work is on regular clusters with high-speed interconnects. A leader-base scheme was proposed in [39] to improve the performance of broadcast over InfiniBand. This chapter has looked at a more intensive collective operation, and the proposed algorithms use shared memory. In [91], Wu and others used MPI point-to-point across the network and shared memory within the SMP node to improve the performance of a number of collectives. I use RDMA and multi-rail communications for inter-node communication. In [87], Traff devised an optimized allgather algorithm for SMP clusters. Ritzdorf and Traff [69] used similar techniques in enhancing NEC's MPI collectives. It should be mentioned that the research in [87, 69] is done at MPI level. Mamidala et al. [42] designed allgather over InfiniBand using shared memory for intra-node and single-port Recursive Doubling algorithm for inter-node communication via RDMA. However, in this chapter, I propose a couple of new SMP-aware algorithms. Mamidala et al. [40] systematically evaluated Intel's Clovertown and AMD's Opteron multi-core architectures and used these insights to develop efficient collective operations.

4.2 Native Gather and Allgather implementation on Quadrics QsNet^{II}

QsNet^{II} *elan_gather()* in the Elan library takes care of the gather and allgather collectives. The gather algorithm uses a tree-based algorithm among the processes [70]. Leaf processes send data to their parents. Intermediate processes add their own data and forward to their parents. This process continues until the root process gathers all data. To reduce host processor involvement, the Elan event processor on the NIC is used to chain the RDMA puts [71]. In SMP clusters, data up to 2KB are gathered in the node's shared

memory buffer. Inter-node gather is then performed on a tree formed by the first process of each node. For medium size messages, a tree-based algorithm is used among all processes in the system. For messages larger than 4KB, Tports Send/Recv is used among all processes, which benefits from message striping in multi-rail QsNet^{II}. For allgather, *elan_gather()* uses the gather algorithm followed by broadcast for messages up to 32KB. For larger messages, it switches to the ring algorithm. Note that all the algorithms in *elan-gather()* are single-port algorithms.

4.3 Motivation

In this section, I do a feasibility study of the potential performance that could be gained in the algorithms by using multi-port message striping and shared memory communication.

4.3.1 Shared Memory vs. RDMA

Intra-node communication can be done using shared memory copying via shared buffers/queues, kernel-based copying, and copying through the NIC [11]. In the shared memory copying approach, a memory region is shared between the two processes. The sending process copies its message into the shared buffer and then sets a shared, synchronization flag. The receiving process polls on the flag to realize whether the sending process has finished writing. It then copies the data from the shared buffer to its own buffer. Finally, it resets the flag. The mechanism used guarantees that no race condition occurs.

The NIC-based copying method is basically an intra-node RDMA Write operation. The kernel-based copying method eliminates one of the two copies associated with the

shared memory method. However, it requires an expensive system call. Therefore, I do not consider it in my work.

I have implemented a shared memory point-to-point communication mechanism based on shared buffers. My implementation requires no locking, and uses the *memcpy()* function. Figure 4.1 compares my shared memory implementation (*shm_p2p*) with intra-node RDMA Write, *elan_put(),* and with the concurrent *memcpy()* operations. My experimental platform is the same as the one in Chapter 3. For all the tests, results are averaged over 1000 iterations. By *k-memcpy(),* I mean *k* processes simultaneously writing data onto *k* sections of a shared memory region. I present up to four concurrent *memcpy()* operations as my experimental cluster uses quad-way SMP nodes.

From Figure 4.1, one can conclude that the shared memory implementation is the preferred method for intra-node communication, but only up to 2KB messages; afterwards, RDMA is better. In implementing collectives, this is the main reason why Quadrics uses shared memory intra-node communication among co-located processes only for messages smaller than 2KB.

Prior research [11, 13] has mostly focused on efficient shared memory communication only for point-to-point transactions (such as *shm_p2p*). However, to implement an SMP-aware per-node collective, such as gather, co-located processes just need to concurrently transfer their messages to different sections of a shared memory region using *memcpy()* operations; and then the root process copies the entire shared memory buffer into its own destination buffer using another *memcpy()* operation (synchronization is also needed). Typically, SMP nodes support concurrent *memcpy()* operations efficiently for short to medium size messages. This is clear from the results in

Figure 4.1 as all *k-memcpy()* operations take much less time than an intra-node RDMA operation (in fact, this is true up to 128KB messages). Intuitively, one can argue shared memory regions can be effectively used for per-node collectives for messages larger than 2KB as well, where they should potentially provide better performance than RDMA implementations.

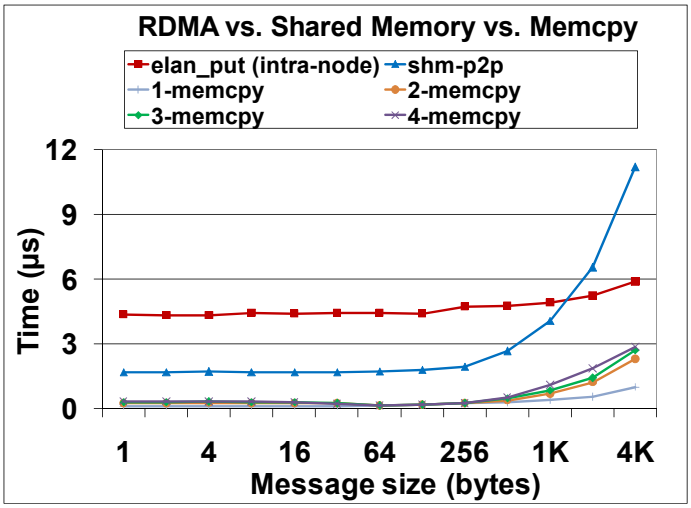

Figure 4.1 Comparison of intra-node communications: RDMA (elan_put), shared memory (shm-p2p) and memory copy.

The proposed SMP-aware allgather algorithms in Section 4.4 use per-node shared memory gather and broadcast. I have implemented these primitives on the 4-way SMP node in order to empirically find the maximum message size that should be transferred via shared memory for an efficient gather and broadcast operation. The per-node shared memory gather described above includes an optimization (as shown in Figure 4.3). For the shared memory broadcast, the Master (root) process copies its data to the shared buffer and then sets a synchronization flag. All other processes poll on this flag and then copy the data to their destination buffers. All processes then synchronize (using *elan_hgsync*) to complete the operation.

Figure 4.2 presents the results for the shared memory gather and broadcast operations on the 4-way SMP node. While the proposed shared memory broadcast (*shm_bcast*) outperforms Elan hardware broadcast (*elan_hbcast*) and Elan software broadcast (*elan_bcast*) for 256B to 32KB messages (with comparable results for very short messages), the proposed shared memory gather (*shm_gather*) is better than, or comparable to, the native *elan_gather()* for up to 8KB messages. Therefore, I use shared memory for messages up to 8KB in my experiments. It is clear that this message size can be found empirically for other single-core/multi-core SMPs, which may have different architectural characteristics than our platform.

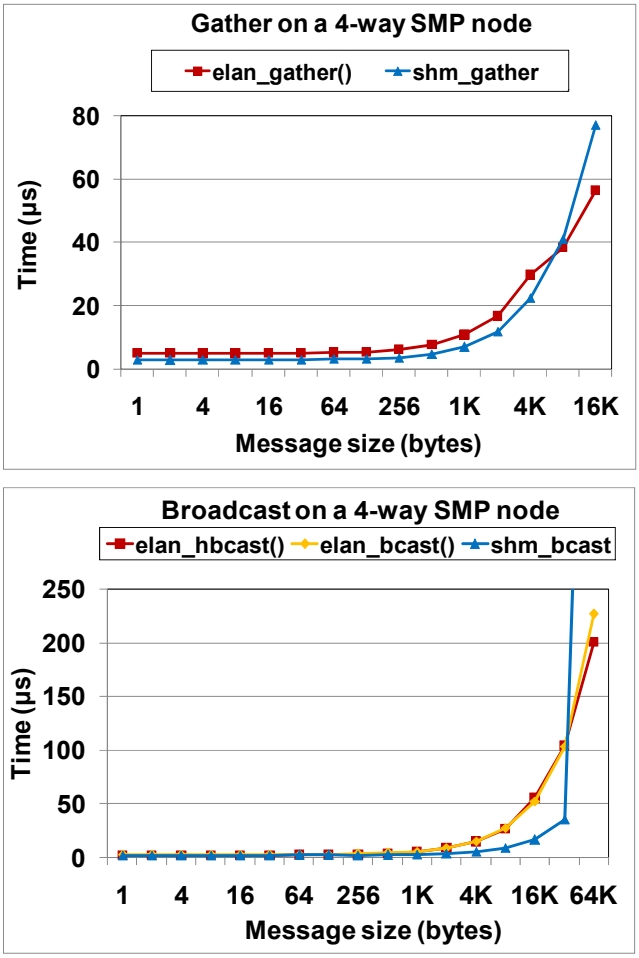

Figure 4.2 Intra-node gather and broadcast.

4.4 SMP-aware Allgather Algorithms

In this section, I propose SMP-aware allgather algorithms. In these algorithms, I distinguish between the intra-node and inter-node communications. However, I do not just simply replace the intra-node communications in the traditional algorithms with shared memory communications. I propose two classes of SMP-aware allgather algorithms. In the first class, I essentially do an SMP-aware gather algorithm across all processes in the system and then broadcast the gathered data to all processes, hence the name *SMP-aware Gather and Broadcast* algorithm.

In the second class, I adapt the traditional multi-port *Direct* and *Bruck* allgather algorithms to SMP clusters by performing them across the SMP nodes rather than processes. I also do shared memory gather and broadcast operations within the nodes. I call these algorithms *SMP-aware Direct and Bruck* algorithms.

4.4.1 SMP-aware Gather and Broadcast Algorithm

This algorithm is essentially done in three phases as follows:

Phase 1: Per-node shared memory gather

Phase 2: Inter-node gather among the Master processes (Tree-based or Direct)

Phase 3: Broadcasting gathered data to all processes

Figure 4.3 shows Phase 1 and Phase 2 of this algorithm for a cluster of four 4-way SMP nodes. Without loss of generality, I assume process 0 is the root process. I choose the first process of each node as the local Master process, in this case processes 0, 4, 8, and 12. In Phase 1, a local shared memory gather is done among the processes of each node. The size of the shared memory buffer is equal to the number of local processes times the message size. Each process has a shared memory flag. Local processes

concurrently copy their data, using *memcpy(),* to the corresponding locations in the shared buffer, and then set their own shared memory flag. The Master process polls on all the local flags and will move on to Phase 2 once all flags are set. Note the optimization for node 0 in Figure 4.3.

In Phase 2, the Master processes involve in a Direct or tree-based inter-node gather operation. For instance, in a Direct inter-node gather algorithm, each Master writes the contents of its local shared memory to the corresponding position in the final destination buffer of the root process. Messages from different Masters are sent on different rails with message striping using RDMA Write. At the end, all processes synchronize using *elan_hgsync()*, and move on to Phase 3 where the root process broadcasts the gathered data to all processes using $QsNet^{II}$ hardware broadcast primitive.

In principle, the proposed *SMP-aware Gather (Direct) and Broadcast* algorithm is similar to the allgather algorithm in *elan_gather()* for short messages. However, the proposed algorithm is host-based, while Quadrics uses a single-port tree-based, NIC-based approach that does not use striping. While NIC-based techniques alleviate cache flushing problems in host-based methods, they incur higher latencies as the NIC processor is slower than the host processors. Moreover, on-board SDRAM is a limited source in NIC-based approaches, which limits the scalability. The proposed algorithms are all multi-port and use striping. For instance, a 256B message with four processes per node will be merged into a 1KB message in the shared buffer. This 1KB message will then be sent in Phase 2 over the two rails using striping. This is not the case in the Quadrics approach.

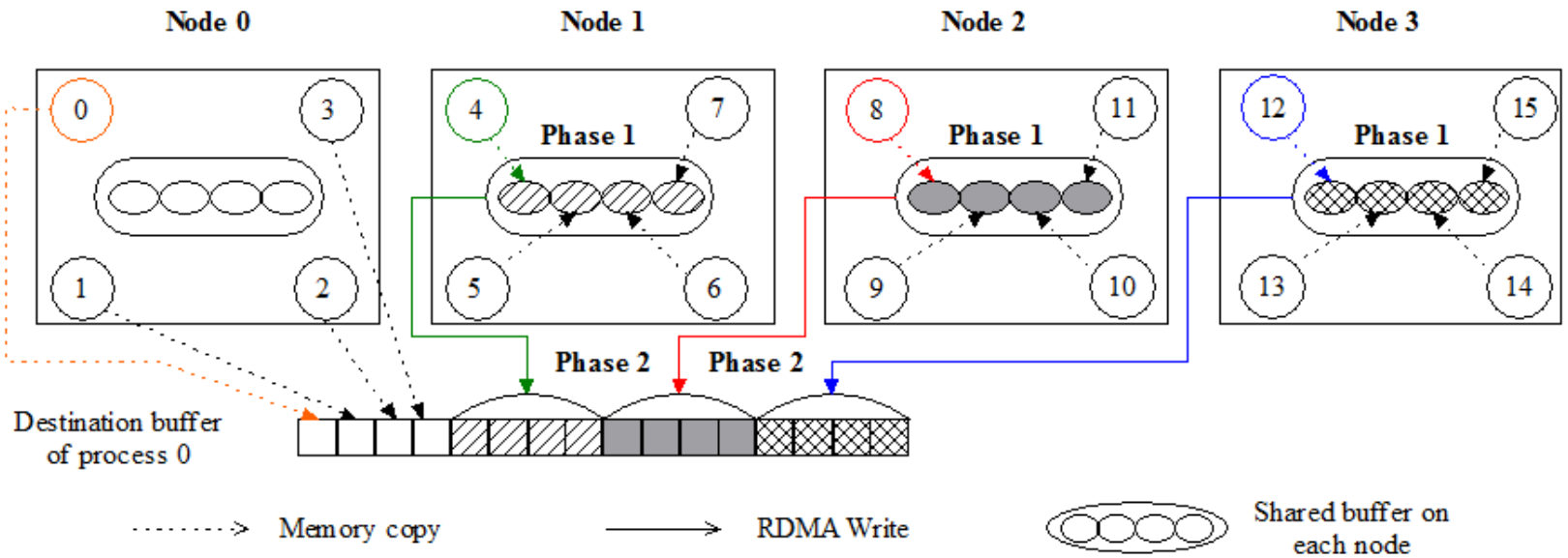

Figure 4.3 Phase 1 and 2 of the SMP-aware Gather and Broadcast on a four 4-way SMP cluster.

4.4.2 SMP-aware Direct/Bruck Algorithms

The *SMP-aware Direct* or *Bruck* allgather algorithms can be done in three steps as follows:

Phase 1: Per-node shared memory gather

Phase 2: Inter-node allgather among the Master processes (Direct or Bruck)

Phase 3: Per-node shared memory broadcast

Figure 4.4 shows the proposed *SMP-aware Direct* allgather algorithm on a quad 4-way SMP cluster. In Phase 1, each SMP node does a shared memory gather operation. However, the size of the shared buffer for this algorithm is four times larger than its counterpart in Section 4.4.1. In Phase 2, Master processes involve in a *Direct* or *Bruck* inter-node allgather operation. Each Master writes the gathered data in Phase 1 to the respective shared memory buffers of the other nodes using the corresponding multi-port allgather algorithm. Each Master then waits for all *devents* to make sure it has received all the data. In Phase 3, Masters use a local shared memory broadcast to copy out the overall contents of the shared buffer to the destination buffers of each process. A final synchronization among all processes completes the collective operation.

In Phase 2, right after posting the RDMA Write operations, I copy the messages in the shared buffer, which have been deposited by local processes, to the destination buffers. This way, I overlap some memory copy operations in Phase 3 with the inter-node communication in Phase 2. Meanwhile, at the end of Phase 2 of the *SMP-aware Bruck* algorithm, all data is available in the shared buffer. However data is not in the right order. Instead of doing a local memory shift, I copy each message from the shared buffer to the right position of the destination buffer for every process.

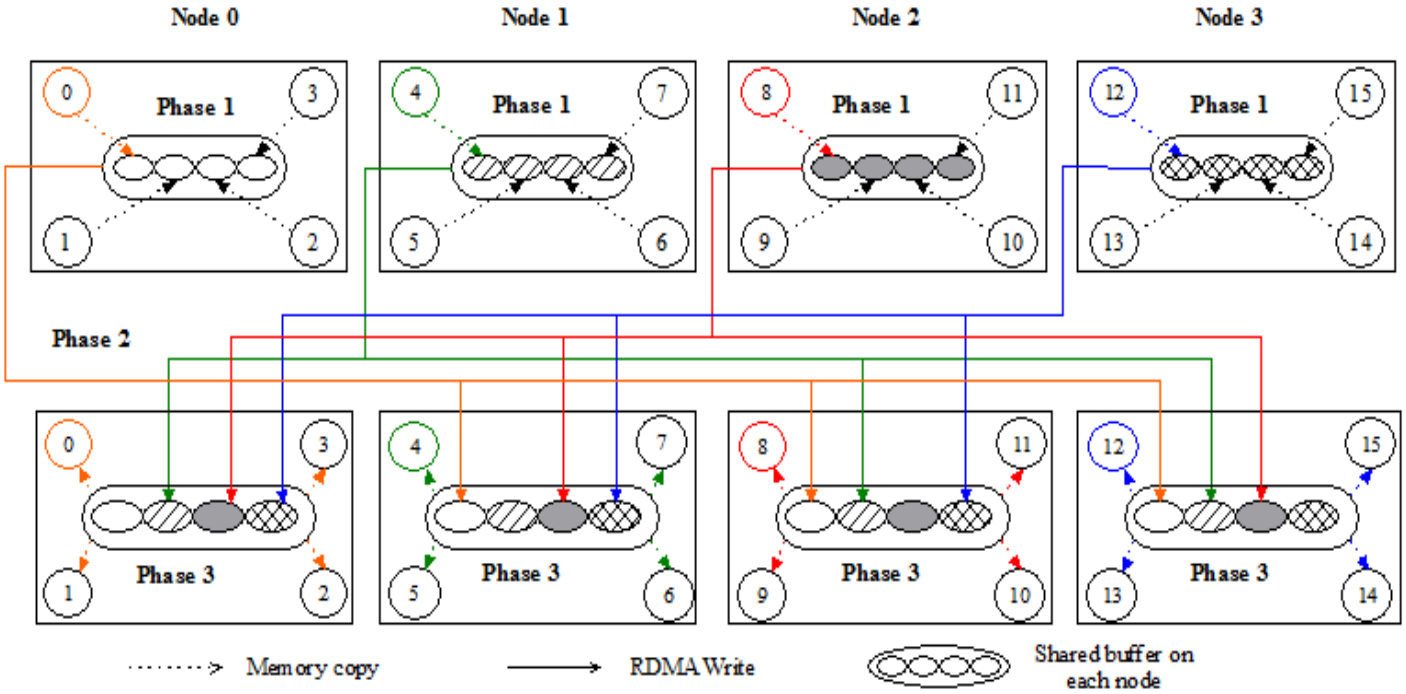

Figure 4.4 SMP-Aware Direct allgather algorithm on a cluster of four 4-way SMP nodes.

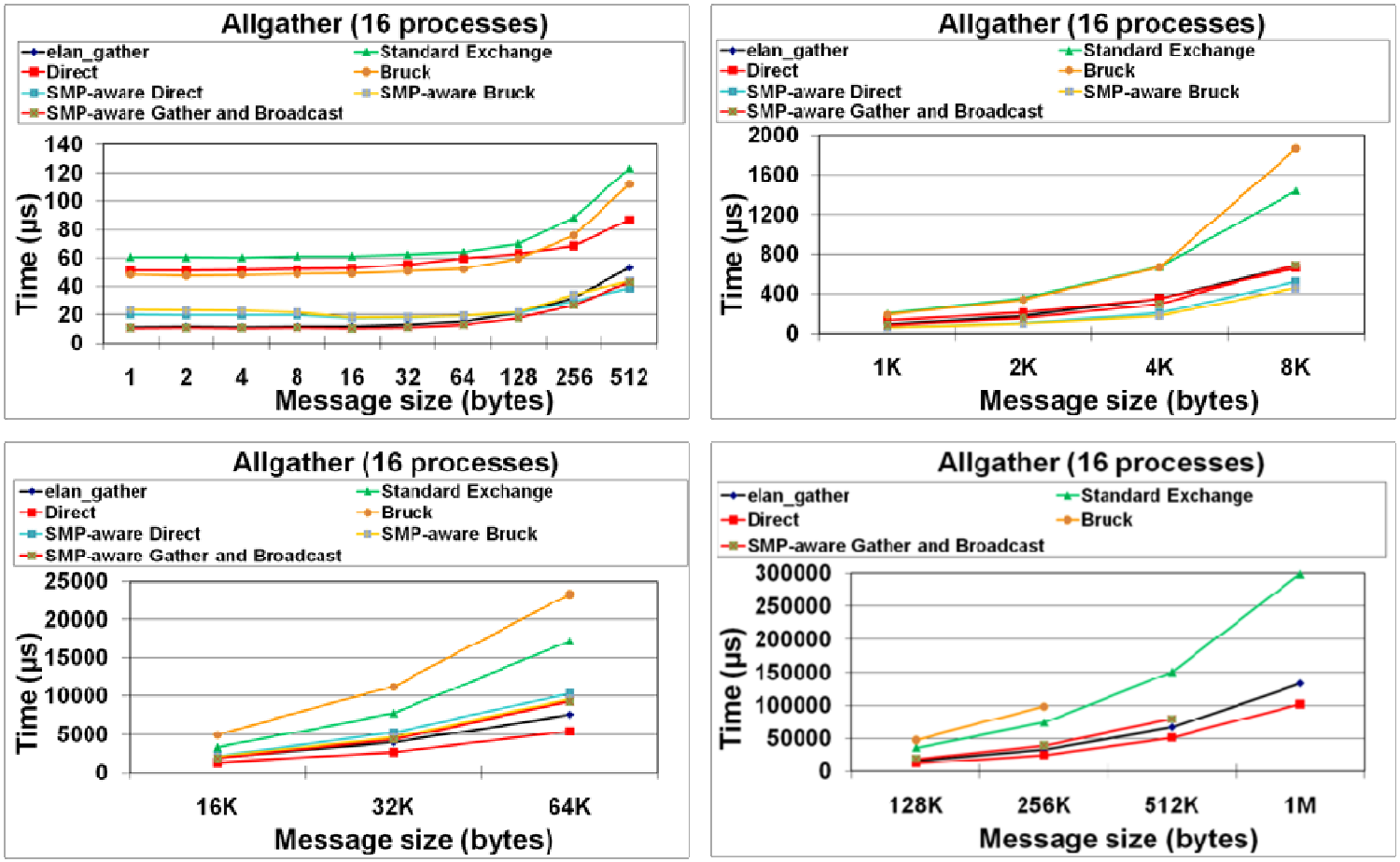

Figure 4.5 Performance of the proposed allgather algorithms on a cluster of four 4-way SMP nodes with dual-rail QsNet^{II}.

The experimental platform represents a small cluster. However, the scalability plots in Figure 4.6 verify the superiority of the proposed algorithms for various message sizes. I have considered 4, 8, and 16 processes in the scalability analysis, where processes are evenly distributed across the nodes. This nicely resembles clusters of four uni-processor nodes, dual-processor nodes, and quad-processor nodes, respectively.

4.4.3 Application Performance

In this section, I will consider two real MPI applications, N-BODY and RADIX [72]. These applications are irregularly structured and use *MPI_Allgather()* collective as well as point-to-point communications. N-BODY simulates the interaction of a system of bodies in three dimensions over a number of time steps, using the Barnes-Hut algorithm. Radix sorts a series of integer keys in ascending order using the radix algorithm.

Table 4.1 shows the application speedup and the communication speedup of N-BODY and RADIX running with 16 processes when using the proposed allgather algorithms. The achieved speedups are within expectation given the size of messages that the *MPI_Allgather()* uses in these applications. *MPI_Allgather()* in RADIX only uses 4KB payload, and the communication speedup of 1.47 is close to the 1.96 speedup that the *SMP-aware Bruck* algorithm achieves in the micro-benchmark test. On the contrary, although N-BODY uses a larger number of *MPI_Allgather()* collectives, 91% of the payloads are less than 64 bytes. The remaining payloads are less than 1KB. Given that the proposed *SMP-aware Gather and Broadcast* algorithm is only slightly better than the

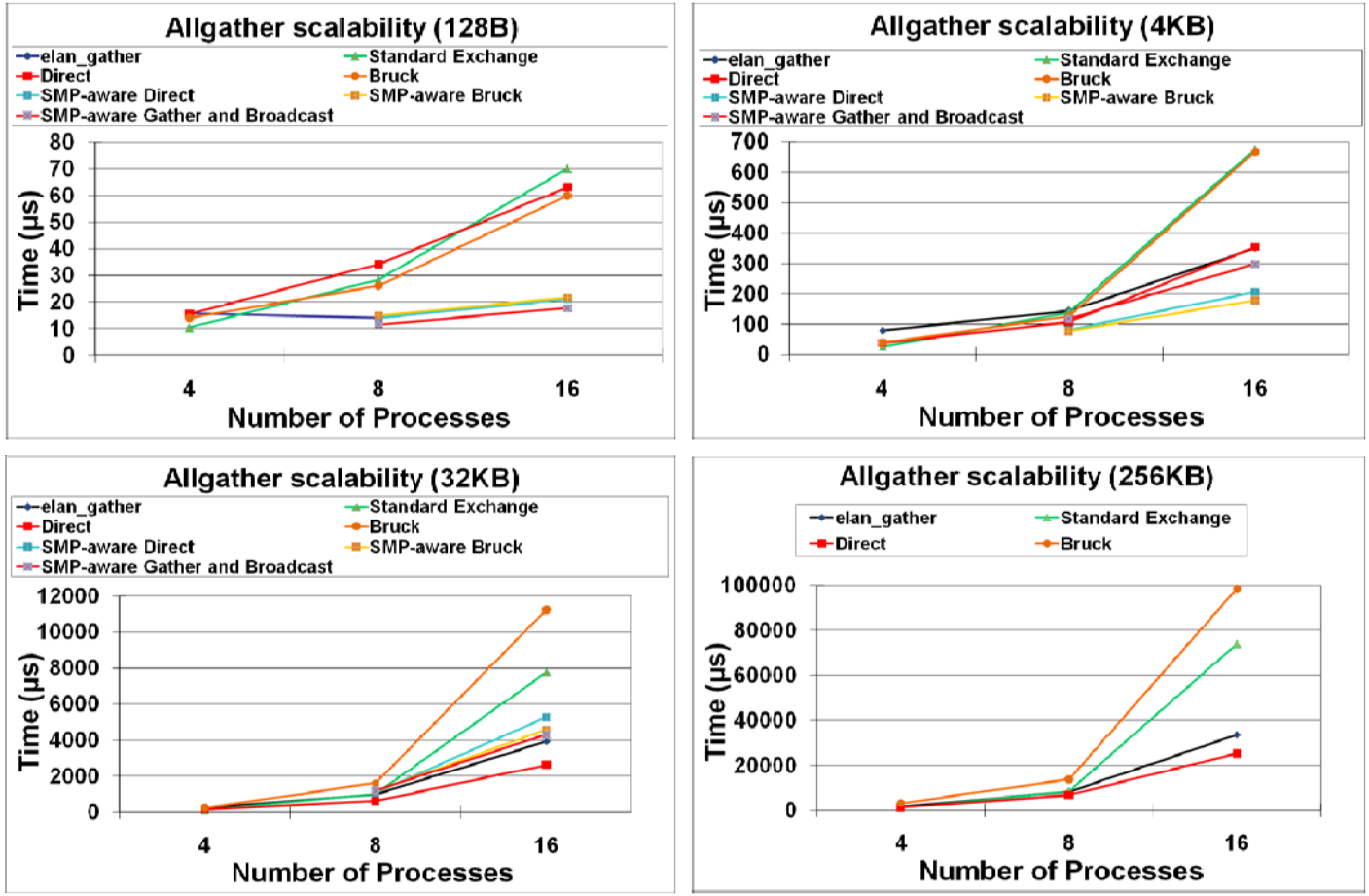

Figure 4.6 Scalability of the proposed allgather algorithms on a cluster of four 4-way SMP nodes with dual-rail QsNet^{II}.

native *elan_gather()* for messages up to 64 bytes, the 9% communication speedup for N-BODY is justified. One has to bear in mind that the micro-benchmark allgather tests at the Elan level were run in a controlled and a synchronized fashion, while real applications may typically suffer from process skew and different process arrival pattern due to imbalanced computation. I will study this and propose process arrival pattern aware allgather and alltoall algorithms in Chapter 6 of this dissertation.

Table 4.1 Application and communication speedup (16 processes) when using the proposed allgather algorithms.

	N-BODY	RADIX
Application speedup	**1.01**	**1.13**
Communication speedup	**1.09**	**1.47**

4.5 Summary

In this chapter, I have proposed and evaluated a number of multi-port allgather algorithms using both RDMA and shared memory communication over multi-rail QsNet^{II} SMP clusters directly at the Elan level. For the allgather operation, and for very short messages up to 256B, the *SMP-aware Gather and Broadcast* algorithm performs slightly better than the native *elan_gather()*. The *SMP-aware Bruck* algorithm outperforms all algorithms including *elan_gather()* for 512B to 8KB messages, with a 1.96 improvement factor for 4KB messages. The multi-port Direct allgather is the best algorithm for 16KB to 1MB, and outperforms *elan_gather()* by a factor of 1.49 for 32KB messages. The proposed allgather algorithms also improve the communication performance of the applications studied in this chapter of dissertation.

It should be mentioned that while this work was focused at gather and allgather collectives, the proposed techniques and algorithms can be adapted to other collective communications such as alltoall.

Chapter 5: Multi-connection and Multi-core Aware Allgather on InfiniBand Clusters

In this chapter, I turn my attention to InfiniBand [31], a leading high-performance networking technology that provides low latency, high bandwidth and good scalability for HPC clusters with thousands of nodes. I provide evidence that the latest generation of InfiniBand HCAs can provide better performance, and to some scalability for simultaneous communication over multiple connections [79] with respect to previous generation of InfiniBand cards [66]. I will then take on the challenge in designing efficient allgather algorithms by utilizing the multi-connection scalability feature of ConnectX InfiniBand networks for inter-node communications using RDMA Write, shared memory operations for intra-node communications in multi-core SMP nodes, as well as multiple cores for better system and network utilization. Specifically, I propose and evaluate three multi-connection and multi-core aware allgather algorithms [64].

5.1 Related Work

In [84], Tipparaju and Nieplocha used the concurrency available in modern networks to optimize *MPI_Allgather()* on InfiniBand and QsNet^{II}. This work, similar to [63], uses multiple outstanding RDMA operations, and perhaps is the closest to my work. However, they do not study the network systematically as I have done in this study, and they do not use shared memory communication and multi-core systems either.

On multi-connection capability of modern interconnects [66], Rashti and Afsahi established a number of connections at the verbs level between two processes running on

two nodes (each node having a NetEffect iWARP or Mellanox InfiniHost III InfiniBand NIC), and then performed point-to-point communications over those connections. It was observed that the normalized multiple-connection latency of small messages is decreased and throughput is increased up to a certain number of connections. In a similar work [79], Sur and others measured the multi-pair RDMA-Write latency and aggregate bandwidth at the InfiniBand verbs level over ConnectX HCAs between multi-core platforms. They established a connection between each pair of processes on different nodes. With increasing number of pairs, the results showed that the network is able to provide almost the same latency for small messages for up to 8 communicating pairs. Both the work in [66] and [79] were focused at point-to-point communication.

5.2 Allgather in MVAPICH

MVAPICH [47] implements the Recursive-Doubling algorithm for *MPI_Allgather()* for power of two number of processes directly using RDMA operations. No shared memory operation is used in this approach. An MPI send-recv approach is used for any other number of processes. Based on the message size, the RDMA-based approach uses two different schemes: (1) a copy-based approach for small messages into a pre-registered buffer to avoid buffer registration cost, and (2) a zero-copy method for large messages, where the cost of data copy is prohibitive [76].

5.3 Experimental Platform

The experiments in Chapter 5 and Chapter 6 were conducted on a 4-node dedicated multi-core SMP cluster, where each node is a Dell PowerEdge 2850 server having two dual-core 2.8GHz Intel Xeon EM64T processors (2MB of L2 cache per core) and 4GB of

DDR-2 SDRAM. Each node has a two-port Mellanox ConnectX InfiniBand HCA installed on an x8 PCI-Express slot. The experiments were done under only one port of the ConnectX HCA. The machines are interconnected through a Mellanox 24-port MT47396 Infiniscale-III switch. In terms of software, I used the OpenFabrics Enterprise Distribution, OFED1.2.5, installed over Linux Fedora Core 5, kernel 2.6.20. For MPI, I used MVAPICH-1.0.0-1625.

ConnectX [17] is the latest generation of InfiniBand HCAs from Mellanox Technologies. It is a two-port HCA that could operate as 4X InfiniBand or 10-Gigabit Ethernet. In this work, I am only concerned with the InfiniBand mode of the ConnectX. In addition, ConnectX supports a number of enhanced InfiniBand features [17] including hardware-based reliable multicast, enhanced atomic operations, fine-grain end-to-end QoS, and extended reliable connection. Such features will enhance the performance and scalability of the communication subsystem. However, to my knowledge, not all these features have been enabled by the ConnectX drivers and firmware.

5.4 Motivation

In multi-core clusters, each core will run at least one process with possible connections to other processes. Therefore, it is very important for the NIC hardware and its communication software to provide scalable performance with the increasing number of connections. In the following, I will show that point-to-point communications as well as collective communications can enjoy scalable performance (up to a certain number of connections) on ConnectX InfiniBand cards. This feature will be useful in devising collectives over multi-core clusters.

I will start with the point-to-point tests. In this experiment, multiple pairs of connections are pre-established between two processes running on different nodes. I perform a ping-pong test using all of the connections in parallel. A message is sent and received over each connection in a round-robin fashion. I vary the number of connections (up to 256 connections in total) and message sizes and report half of the cumulative round trip time divided by the number of connections as the *normalized average latency* in Figures 5.1. This shows how well communications over multiple connections can be performed simultaneously.

For small to medium messages up to 8KB and up to 64 connections, significant time is saved in sending messages over multiple connections. The best performance is achieved at 8 connections. Too many connections will degrade the performance for some message sizes. For messages larger than 8KB, the average latency to send a message over multiple connections is the same as the latency for one pair of connection. This indicates no overlapping is taking place among the connections and that the communication is serialized. One reason behind this is that the ConnectX architecture includes a stateless offload engine for NIC-based protocol processing. Compared to the previous generation of InfiniBand cards, ConnectX improves the processing rate of incoming packets by having hardware schedule the packet processing directly. This technique allows ConnectX to have a better performance for processing simultaneous network transactions.

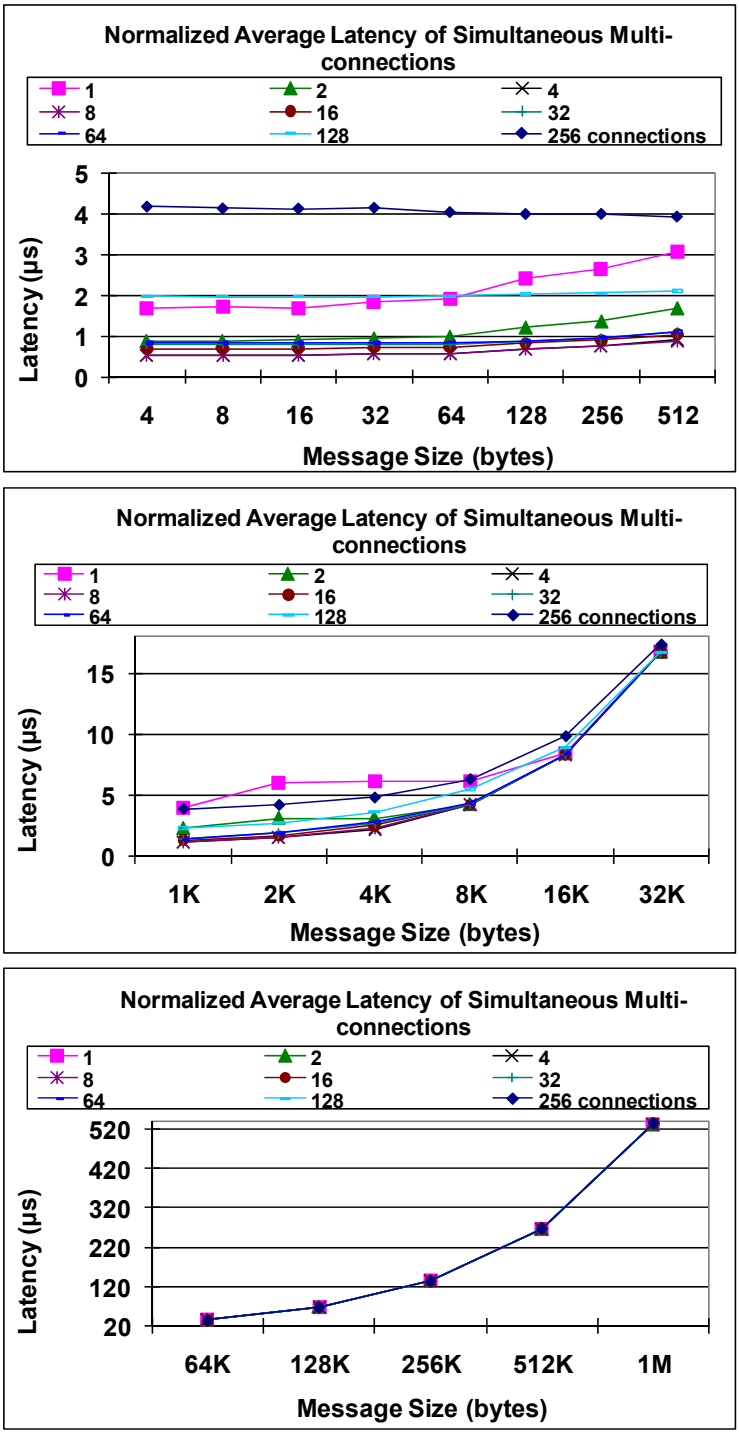

Figure 5.1 Normalized average latency of a 1-byte message sent simultaneously over multiple connections.

To understand whether the multi-connection ability of ConnectX can help allgather (and collectives in general), I have devised a second micro-benchmark in which MPI processes on a set of multi-core nodes are grouped into a number of groups where each group concurrently performs an independent *MPI_Allgather()* operation. Processes of each group are mapped on four different nodes. Therefore, all the communications of an allgather operation within a group are across the network. I have designed the individual allgather algorithm in such a way that each process transfers its own data to the other three members of the group using three back-to-back RDMA operations in a ring-based fashion, leading to three simultaneous active connections per process. With four groups, I will have up to 12 bi-directional active connections per card.

I consider a single-group single-connection ring-based allgather as the baseline operation. I then compare the single- and multiple-group multi-connection allgather with the baseline allgather, as shown in Figure 5.2. The aggregate bandwidth plot in Figure 5.2 shows the total volume of data per second that passes through the ConnectX HCA. To have a better understanding of how multi-connection could enhance the performance, the bandwidth ratio plot in Figure 5.2 shows the bandwidth ratio of the single- and multiple-group multi-connection allgather over the baseline collective operation.

It is evident that the multi-connection allgather operations achieve higher aggregate bandwidth, and can saturate the network card more than a single-connection allgather up to 64KB. The single-group multi-connection case improves the throughput up to 1.8 times the baseline, while the two, three and four groups can achieve up to 2.6, 2.7 and 2.7 times, respectively. The results for the 3-group and 4-group allgather are fairly close to

each other, indicating that the network is almost saturated with nine simultaneous connections. For all multi-connection tests, the ratio drops below one for 64KB messages and above, which shows a performance degradation for very large messages when multiple connections are simultaneously active.

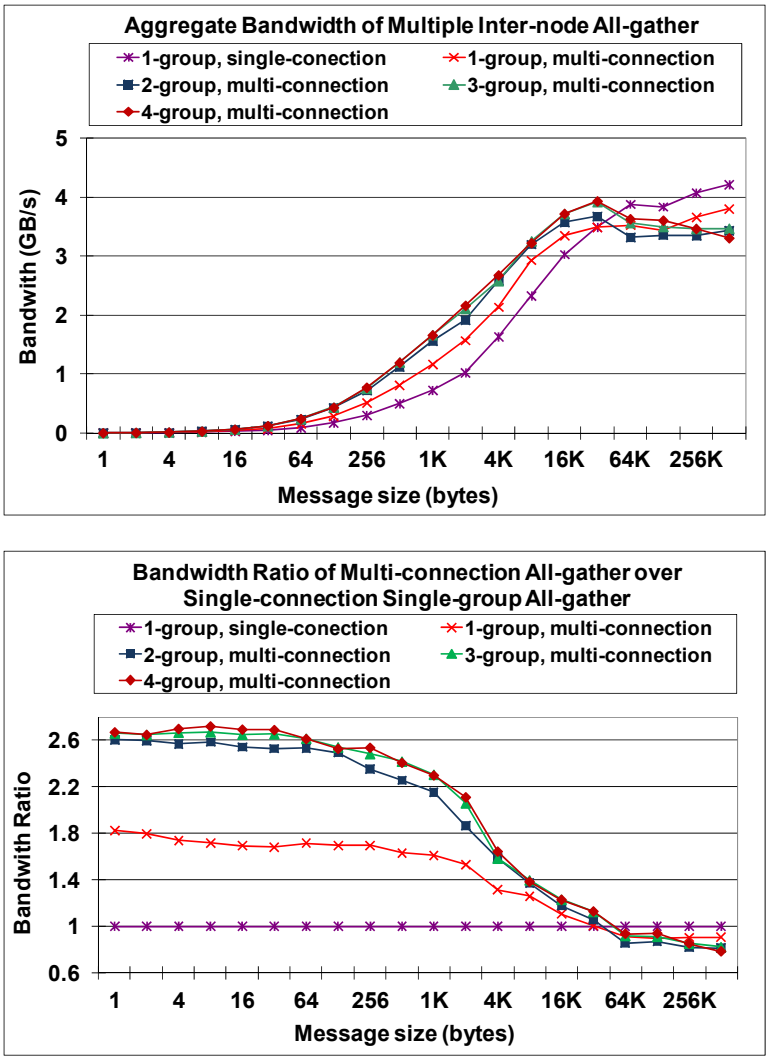

Figure 5.2 Aggregate bandwidth of multiple independent allgather operations.

The results clearly show that the network is capable of providing scalable performance when multiple connections are concurrently active, at least for small to medium size messages and up to a certain number of connections. The message is that there is now a potential to improve the performance of collectives by devising efficient algorithms that use multiple connections concurrently on multi-core systems.

5.5 The Proposed Allgather Algorithms

In this section, I present a number of algorithms for the *MPI_Allgather()* operation. I first propose the *Single-group Multi-connection Aware* allgather algorithm, which is a multi-connection extension of the SMP-aware algorithm proposed in Section 4.4 [59, 63] and is targeted at small to medium messages.

I then propose two different classes of algorithms to enhance the allgather performance for different message sizes. I propose the *Multi-group Gather-based Multi-connection Aware* allgather algorithm to achieve efficient performance for very small messages. This algorithm takes advantage of the availability of multiple cores on the node to distribute the CPU processing load. Finally, to further utilize the multi-connection capability of the InfiniBand network, I propose the *Multi-group Multi-connection Aware* allgather algorithm for medium to large message sizes. This algorithm has less shared memory communication volume, but uses more connections per node.

5.5.1 Single-group Multi-connection Aware Algorithm

Single-connection SMP-aware collective communication algorithms can greatly improve the performance for small to medium message sizes [42, 63]. With the availability of scalable multi-connection performance in modern networks, there is now

an opportunity to improve the performance further. For this, I modify the inter-node communication phase of the SMP-aware allgather algorithm [63] to use multiple connections simultaneously. The algorithm has three phases as follows:

Phase 1: Per-node shared memory gather

Phase 2: Inter-node multi-connection aware allgather among the Master processes

Phase 3: Per-node shared memory broadcast

Each node has a Master process. All node Masters form a group for the inter-node communication in Phase 2 of the algorithm. In Phase 1, the Master process of each node gathers the data from the processes on the same node by a shared memory gather operation. In Phase 2, the Master processes participate in an inter-node allgather operation. Each Master process sends the gathered data in Phase 1, in a multi-connection aware fashion, concurrently over all connections to the other Master processes using RDMA Write operations. In Phase 3, the Master processes perform a shared memory broadcast. They copy out their received data to a shared buffer, from which each process copies the final data to its own destination buffer.

5.5.2 Multi-group Gather-based Multi-connection Aware Algorithm

In Phase 2 of the Single-group Multi-connection Aware algorithm, each Master process is responsible for communication with the Master processes on other nodes. However, the other processes of each node are idle, waiting for the combined data to be shared by their respective Master process. For small messages, the ConnectX has the ability to carry the message data on the work request using programmed input/output (PIO) instead of DMA [79]. This reduces the communication latency, but the CPU/core is

more involved with the communication process, especially when there are multiple simultaneously active connections. Therefore, to take advantage of the multi-core systems and evenly distribute the communication workload on the available cores, I design a multi-group allgather algorithm, in which the outbound connections of each node are now distributed among the cores.

As shown in Figure 5.3, in this algorithm I group the processes across the nodes whose intra-node rank is equal. Each group has a Master process that gathers the data from the processes of the group. The algorithm is performed in three phases:

Phase 1: Per-node shared memory allgather

Phase 2: Per-group inter-node multi-connection aware gather

Phase 3: Per-node shared memory broadcast

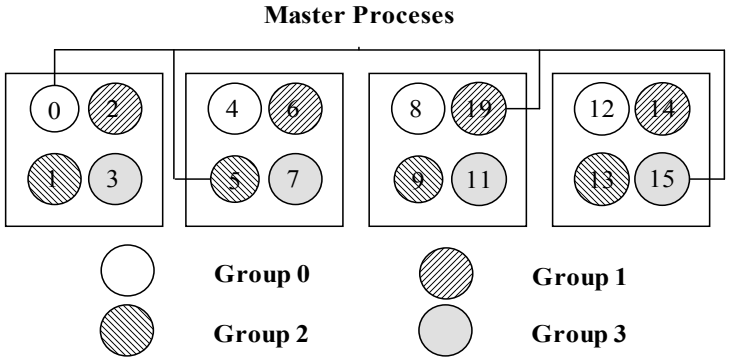

Figure 5.3 Group structure for gather-based allgather algorithm on a 4-node, 16-core cluster.

In Phase 1, an intra-node shared memory allgather is performed among processes on the same node. In Phase 2, each group Master process gathers the data from the processes of its group. This means that at the end of Phase 2 each Master process will have the

entire data from all processes. In Phase 3, each Master process will then broadcast the data to all processes on its own node.

The only limitation of this algorithm is that each node should have at least a group Master process. If the number of nodes is more than the number of groups, some nodes will remain without a Master. To cover this, some groups may need to have two Master processes, with the same duties.

5.5.3 Multi-group Multi-connection Aware Algorithm

Up to this point, I have utilized both multi-connection and multi-core features of a modern InfiniBand cluster in an effort to improve the performance of the allgather collective operation. However, with the 4-node cluster, both the proposed algorithms in Section 5.5.1 and Section 5.5.2 use only a maximum of three simultaneously active connections per card. To examine the multi-connection ability of the NIC more aggressively, I propose the Multi-group Multi-connection aware allgather algorithm, in which an additional number of concurrent active connections is used during the allgather operation. Basically, with an increasing number of cores per node, more groups can be formed in this algorithm, which will even put more pressure on the NIC. I first propose an algorithm with only two independent groups.

Two-group allgather: As discussed earlier in Section 5.4, two independent multi-connection allgather operations achieve a much higher bandwidth than a single allgather case, at least for small to medium size messages. Therefore, I expect this algorithm to perform well for a range of message sizes. Figure 5.4 shows the group structure for a two-group multi-connection allgather on the cluster. Each group has a Master process on

each node, and includes half of each node's processes. The algorithm is performed in three phases as follows:

Phase 1: Per-node/per-group shared memory gather by each group Master process

Phase 2: Per-group inter-node multi-connection aware allgather among group Master processes

Phase 3: Per-node shared memory broadcast from each group Master process

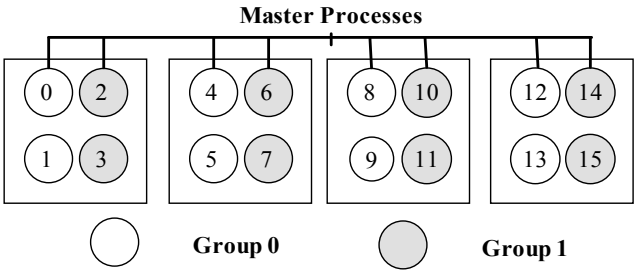

Figure 5.4 Group structure for 2-group allgather algorithm on a 4-node, 16-core cluster.

In Phase 1, each Master process gathers the data of all intra-node processes that belong to its group. For example in Figure 5.4, process 0 and 2 gather the data from process 1 and 3, respectively. In Phase 2, an inter-node allgather is done in each group among Master processes to transfer the data gathered in Phase 1. Each Master process in Phase 2 concurrently performs RDMA Write operations on all its connections. For example in Figure 5.4, Master process 2 concurrently sends data to Master processes 6, 10, and 14. Therefore, there will be six concurrently active connections for this case. In Phase 3, all Master processes broadcast their received data to all the other processes on the same node.

Four-group allgather: A more connection-intensive algorithm that can be implemented on the platform is a 4-group multi-connection aware algorithm. Figure 5.5 shows the group structure for such an algorithm on the cluster. The algorithm is done in two phases as follows:

Phase 1: Per-group inter-node multi-connection aware allgather

Phase 2: Per-node shared memory allgather

In Phase 1, each process performs an inter-node allgather within its group. This operation is done in a way similar to Phase 2 of the 2-group algorithm. However, this time 12 concurrent active connections exist per NIC. In Phase 2, each process shares the combined data received from Phase 1 with all other processes on its own node. Based on the preliminary results in Figure 5.2, I do not expect the cards to scale well with this number of connections, at least on the platform I have experimented with.

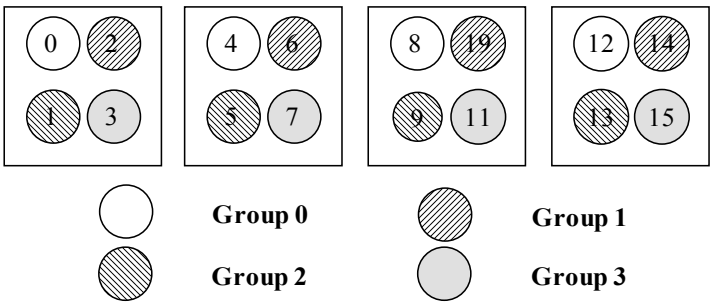

Figure 5.5 Group structure for 4-group allgather algorithm on a 4-node, 16-core cluster.

5.5.4 Complexity Analysis of the Algorithms

In order to estimate the performance, in this section I compare the per-node complexity of the proposed algorithms. The analysis is done based on a 4-node cluster, each node having 4 cores. I compare the number of active connections, the amount of

shared memory Read/Write operations, and the volume of messages communicated through the network in the algorithms. I assume the message size for the allgather operation is M bytes. The following notation is used to present the complexity of the algorithms in Table 5.1:

αM_βS: $\alpha \times M$ bytes of data are communicated in each of β consecutive steps.

αM_βC: There are β active connections, and $\alpha \times M$ bytes of data are concurrently communicated over each connection.

αM_βW / αM_βR: There are β concurrent shared memory Writes/Reads of $\alpha \times M$ bytes of data each.

As shown in Table 5.1, the inter-node communication volume is fixed and equal to 12M for all cases. Moving from the 1-group single-connection algorithm to the 1-group multi-connection algorithm, I just change the way the inter-node communication is done. In essence, instead of sending 4M data in three consecutive steps, I send it over three concurrent active connections.

The other notable difference among the algorithms is in their shared memory transactions. Consider the 1-group algorithms on a 4-node (16-core) cluster. In Phase 1 of the 1-group algorithms, all processes first write their data to the shared memory (1M_4W), and the Master process then reads the data into its send buffer (4M_1R). In Phase 3, the Master process will share all the data that it has received, from all other Master processes in Phase 2 of the algorithm, with its local processes (16M_1W). The local processes will then read the data into their own destination buffers (16M_4R).

Table 5.1 Per-node complexity of the proposed allgather algorithms on a 4-node, 16-core cluster.

		1-group single-connection	1-group multi-connection	4-group gather-based multi-connection	2-group multi-connection	4-group multi-connection
Concurrent active connections		1 In	3 In	3 In	6 In	12 In
		1 Out	3 Out	3 Out	6 Out	12 Out
Network – outbound or inbound data		4M_3S	4M_3C	4M_3C	2M_6C	1M_12C
Shared memory Read/Write operations	**Phase 1**	1M_4W + 4M_1R	1M_4W + 4M_1R	1M_4W + 4M_4R	1M_4W + 2M_2R	-
	Phase 2	-	-	-	-	4M_4W + 16M_4R
	Phase 3	16M_1W + 16M_4R	16M_1W + 16M_4R	16M_1W+ 16M_4R	8M_2W + 16M_4R	-

The gather-based algorithm has a slightly higher shared memory volume than the 1-group and 2-group algorithms. This means that for larger size messages, I do not expect the gather-based algorithm to perform better. With almost the same level of shared memory volume in the 1-group and 2-group algorithms, the 2-group algorithm has an edge over the 1-group and gather-based algorithms, mostly due to distribution of its inter-node communication among more cores, effectively utilizing both multi-connection and multi-core ability of the InfiniBand cluster. The 4-group algorithm has the lowest shared memory volume. However, I expect to see some overhead due to its aggressive use of simultaneous network connections (over 12 connections).

5.6 Performance Results

In this section, I present the performance results of the proposed algorithms and the native MVAPICH on the cluster, along with a brief description of the implementation. I use a cycle-accurate timer to record the time spent in an allgather operation (1000 iterations) for each process, and then calculate the average allgather latency among all processes.

For the implementation, I am directly using RDMA Write operations. I have implemented two different schemes: *zero-copy* and *copy-based* schemes in RDMA-based communications. The zero-copy approach is designed for large messages to avoid the extra data copy. To be able to have a direct data movement, the application buffers are required be registered. Also, each source process needs to know the address of the remote destination buffers before the RDMA communications can take place. For this, each process will advertise its registered destination buffer addresses to all other processes by writing into their pre-registered and pre-advertised control buffers.

To avoid the high cost of application buffer registration and address advertisement for small messages, the copy-based technique involves a data copy to pre-registered and pre-advertised intermediate data buffers at both send and receive sides. The sending process can copy its messages to the pre-registered intermediate destination buffers using RDMA Write.

To find the switching point of two schemes, I have implemented the proposed algorithms using both copy-based and zero-copy techniques, and evaluated them for 1B

to 512KB messages. The results shown in Figure 5.6, for 16 processes on the test cluster, are the best results of the two schemes for each algorithm.

In general, the proposed multi-connection aware algorithms perform much better than the native MVAPICH implementation except for 128KB messages and above, mostly due to shared memory bottleneck and poor multi-connection performance for very large messages. As expected, the gather-based algorithm has the best performance for very small messages up to 32 bytes, mostly because this algorithm is using multiple cores on the node and lightly utilizes the multi-connection ability of the cards for network communication. The 1-group multi-connection aware algorithm outperforms all other algorithms from 64 bytes up to 2KB, since it has a lighter shared memory volume. From 4KB to 64KB, the 2-group multi-connection-aware algorithm performs the best, due to a lighter shared memory volume compared to 1-group algorithms, and use of multiple concurrent connections and multi-cores.

5.7 Summary

Collective operations are the most data intensive communication primitives in MPI. In this chapter, I proposed three multi-core and/or multi-connection aware allgather algorithms over ConnectX InfiniBand networks. The implementation results confirm that utilizing the advanced features of modern network cards to process the communication over multiple simultaneously active connections can greatly improve the performance of collective operations. The proposed 1-group multi-connection aware algorithm performs better than the 1-group single-connection method for most of the message sizes.

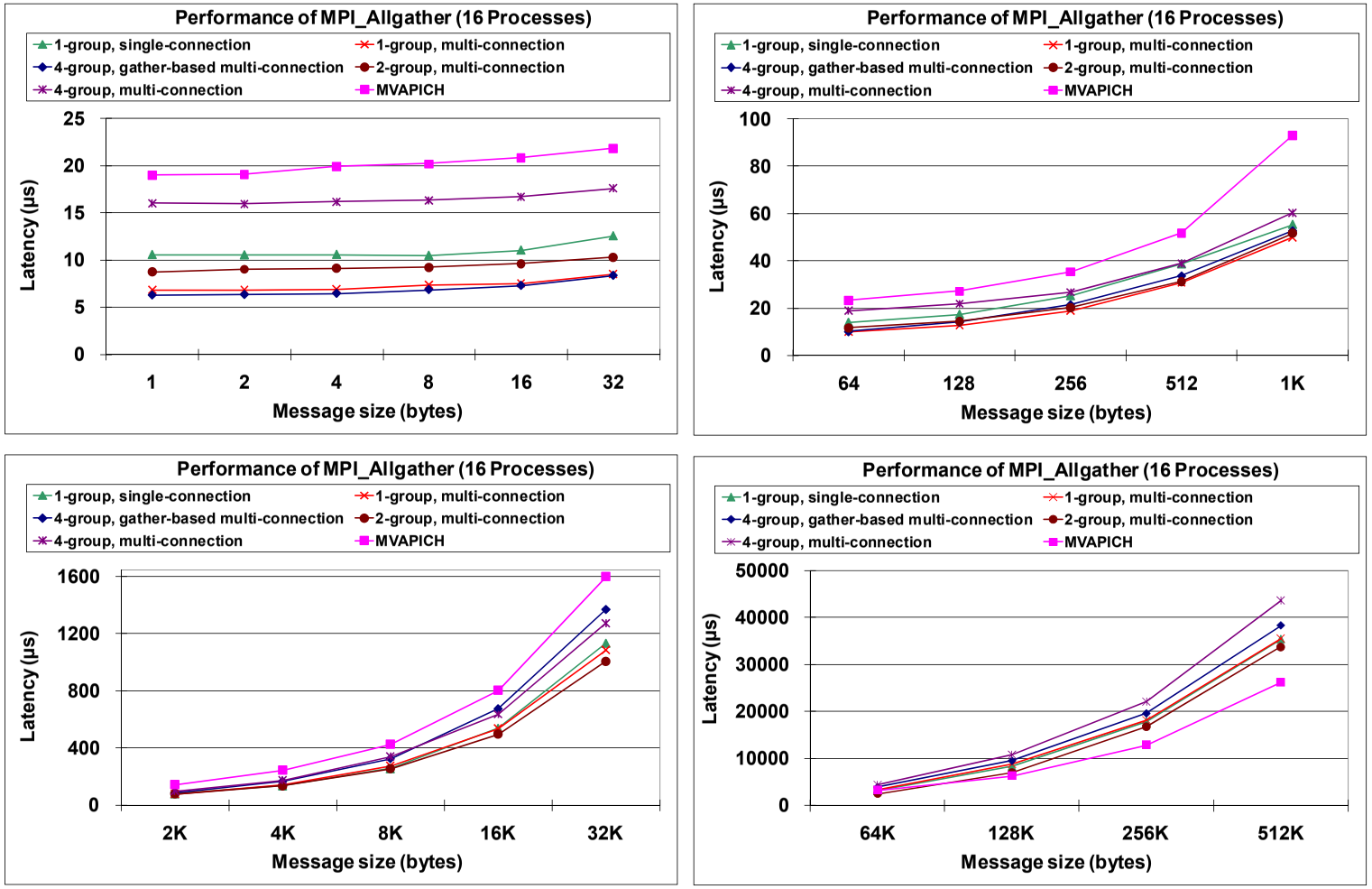

Figure 5.6 Allgather performance.

Using the gather-based algorithm, I could also improve the performance by distributing the communication load over multiple cores for small message sizes that involve CPU in communication processing. Another factor that has affected the achieved performance compared to the pure RDMA-based implementation is the use of shared memory. In general, the multi-group multi-connection aware algorithms perform better than the RDMA-only allgather, except for very large message sizes.

Chapter 6: Process Arrival Pattern Aware Collectives on InfiniBand

In preceding chapters, I have proposed a number of collective operations on top of modern interconnects. The micro-benchmark studies, and not the application studies, were done when processes arrived at the collective call almost simultaneously. However, this is not the case in real-life running applications. Recent research has shown that *process arrival pattern* (PAP) for collective operations have significant influence on the performance of collectives and consequently on the applications [21]. In this chapter, I will take another important factor into account, that is PAP awareness, and propose PAP-aware alltoall [62] and algather collectives on top of InfiniBand [61].

6.1 Related work

Most research on developing and implementing efficient collective communication algorithms assume all MPI processes involved in the operation arrive at the same time at the collective call. However, recent studies have shown that processes in real applications can arrive at the collective calls at different times. This imbalanced process arrival pattern can significantly affect the performance of the collective operations [21]. In addition, it has been found that different collective communication algorithms react differently to PAP [21]. In this regard, the authors in [52] have recently proposed PAP aware *MPI-Bcast()* algorithms and implemented them using MPI point-to-point primitives.

My research is similar to the work in [52]. However it has a number of significant differences. First, the authors in [52] have incorporated control messages in their algorithms at the MPI layer to make the processes aware of and adapt to the PAP. These control messages incur high overhead, especially for short messages. My proposed PAP

aware *MPI_Alltoall()* and *MPI_Allgather()* instead are RDMA-based, and I use its inherent mechanism for notification purposes. Therefore, there are no control messages involved and thus there is no overhead. Secondly, while [52] is targeted at large messages, I propose and evaluate two RDMA-based schemes for small and large messages. Thirdly, I propose an intra-node PAP and shared memory aware scatter operation to boost the performance for small messages.

6.2 MPI_Alltoall() and MPI_Allgather() in MVAPICH

The study in this chapter is done on an InfiniBand ConnectX cluster, and the platform is described in Chapter 5. MVAPICH is the native MPI implementation. In MVAPICH, point-to-point and some MPI collective communications have been implemented directly using RDMA operations. However, *MPI_Alltoall()* uses the two-sided MPI send and receive primitives, which transparently uses RDMA. Different alltoall algorithms are employed for different message sizes: the *Bruck* algorithm for small messages, the *Recursive-Doubling* algorithm for large messages and power of two number of processes, and the *Direct* algorithm for large messages and non-power of two number of processes. The MVAPICH *MPI_Allgather()* implementation was discussed in Chapter 5.

dsfasdfas

6.3 Motivation

In this section, I first show that indeed MPI processes arrive at collective calls at different times at runtime. Then, I present the impact of this skew time on the performance of *MPI_Alltoall()* and *MPI_Alllgather()*.

6.3.1 Process arrival pattern

I first describe the parameters that are used to characterize PAP, similar to [21]. An example for four processes is given in Figure 6.1. Let n processes, p_0 , …, p_{n-1} be involved in the collective operations. a_i represents the arrival time of p_i , and e_i represents the time when p_i exits the collective. Let d_i be the time difference between each process's arrival time a_i and the *average arrival time* $\bar{a}$. Average arrival time $\bar{a}$, *average-case imbalance time* $\bar{d}$ and *worst-case imbalance time* θ are defined in Equation (6.1) (6.2) and (6.3), respectively.

$$\bar{a} = \frac{a_0 + \ldots + a_{n-1}}{n} \tag{6.1}$$

$$d_i = \left| a_i - \bar{a} \right|$$

$$\bar{d} = \frac{d_0 + \ldots + d_{n-1}}{n} \tag{6.2}$$

$$\theta = \max_i \{a_i\} - \min_i \{a_i\} \tag{6.3}$$

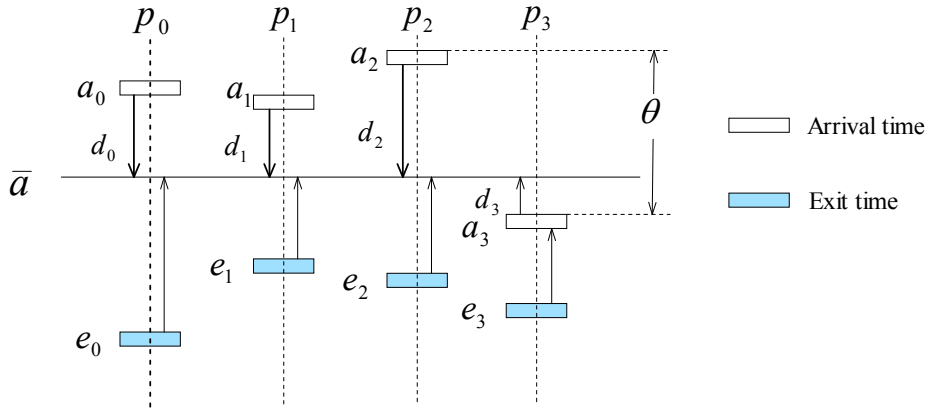

Figure 6.1 Process arrival pattern for 4 processes.

The worst-case imbalance time and the average-case imbalance time are normalized over the time it takes to communicate a message (the size of this message is equal to the collective payload). These new metrics are called the *average-case imbalance factor* $\frac{\bar{d}}{T}$ and the *worst-case imbalance factor* $\frac{\theta}{T}$ [21].

To investigate the PAP behavior of MPI collectives in real applications, I have developed a profiling tool using the MPI wrapper facility. Using *MPI_Wtime(),* the wrapper records the arrival and exit times for each collective. An *MPI_Barrier()* operation is used in the MPI initialization phase in *MPI_Init()* to synchronize all the processes.

I have chosen several benchmarks from NAS Parallel Benchmarks (NPB), version 2.4 [49]. The average-case imbalance factor and worst-case imbalance factor results running for 16 processes are shown in Table 6.1. The results show that the averages-case and worst-case imbalance factors for all applications are quite large. For larger class C, the imbalance factors are even larger. The results confirm previous studies [21] that processes arrive at collective sites at different times.

6.3.2 Impact of Process Arrival Pattern on Collectives

To show how the native MVAPICH *MPI_Alltoall()* and *MPI_Allgather()* perform on the platform under random PAP, I use a micro-benchmark similar to [52]. Processes first synchronize using an *MPI_Barrier()*. Then, they execute a random computation before entering the *MPI_Alltoall()* and *MPI_Allgather()* operations. The random computation

time is bounded by a maximum value MIF (*maximum imbalanced factor* [52]) times the time it takes to send a message.

Table 6.1 The average of worst-case and the average-case imbalance factors for FT LU and MG benchmarks.

	Major routine	Number of calls	Average message size (bytes)	Average-case imbalance factor $\frac{\overline{d}}{T}$	Worst-case imbalance factor $\frac{\theta}{T}$
FT (class B)	alltoall	22	2097152	597	1,812
	reduce	20	16	438,701	1,460,811
FT (class C)	alltoall	22	8388608	614	1,871
	reduce	20	16	1,888,964	5,732,913
LU (class B)	allreduce	9	22	56,033	163,523
	bcast	10	10	2,799	9,553
LU (class C)	allreduce	9	22	118,695	388,537
	bcast	10	10	11,519	32,938
MG (class B)	allreduce	12	89	9,101	2917
MG (class C)	allreduce	12	89	162,146	560,769

To get the performance of *MPI_Alltoall()* and *MPI_Allgather()*, a high-resolution timer is inserted before and after the *MPI_Alltoall()* and *MPI_Allgather()* operations. The completion time is reported as the average execution time across all the processes. Figure 6.2 and Figure 6.3 present the performance of MVAPICH *MPI_Alltoall()* and *MPI_Allgather()* when MIF is 1, 32, 128 and 512, respectively. Clearly, the completion time is greatly affected by the increasing amount of random computation. The results confirm that the PAP can have significant impact on the performance of collectives. It is therefore crucial to design and implement PAP aware collectives to improve their

performance and consequently the performance of the applications that use them frequently.

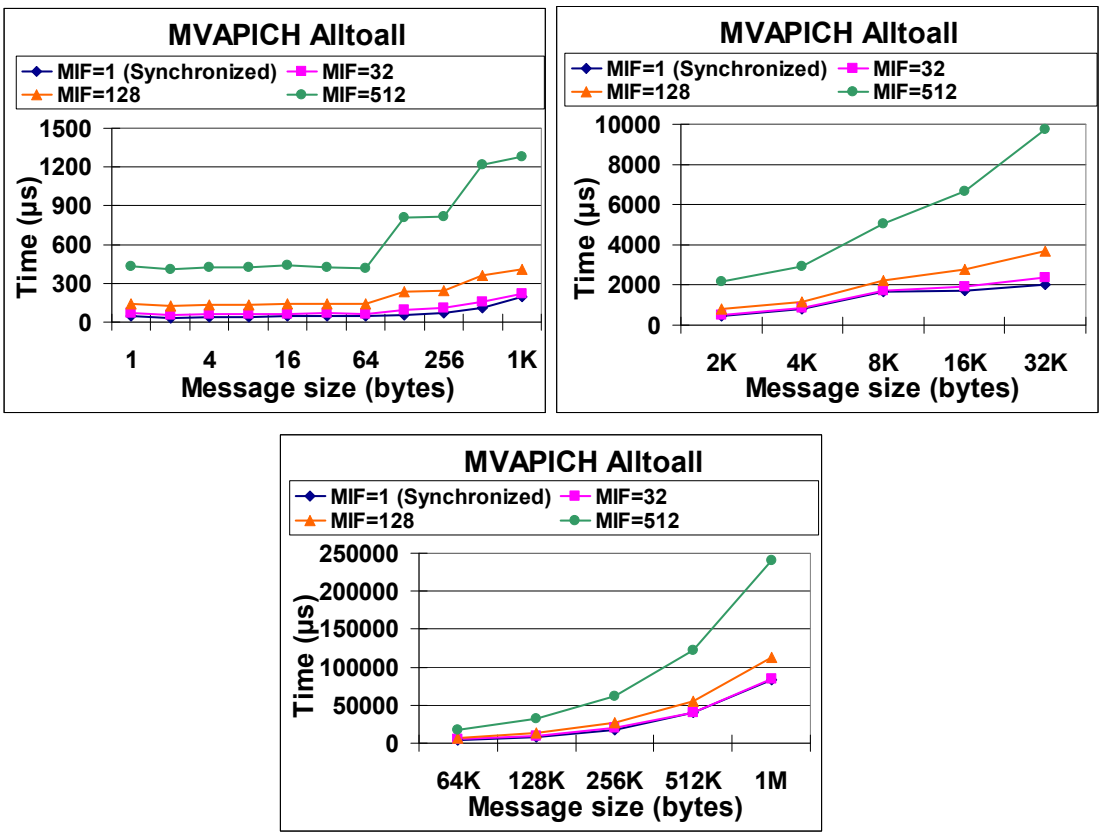

Figure 6.2 Completion time of MVAPICH Alltoall under different process arrival patterns.

6.4 The Proposed Process Arrival Pattern Aware MPI_Alltoall() and MPI_Allgather()

In this section, I propose PAP aware algorithms for *MPI_Alltoall()* and *MPI_Allgather()*. My algorithms are based on the *Direct* and the *SMP-aware Direct* allgather (and alltoall) algorithms proposed in Chapter 5. The basic idea in the algorithms is to let the early-arrived processes do as much work as possible. One critical issue in my PAP aware algorithms is how to let every other process know who has already arrived at

the collective call. Previous work on PAP aware *MPI_Bcast()* [52] has introduced control messages that would add extra overhead, especially for small messages. However, in my work I do not send distinct control messages and instead I utilize the inherent features of RDMA-based communication to notify the arrival of a process. In the following, I first propose my RDMA-based PAP aware *MPI_Alltoall()* and *MPI_Allgather()*, and then extend them to be shared memory aware for better performance for small messages.

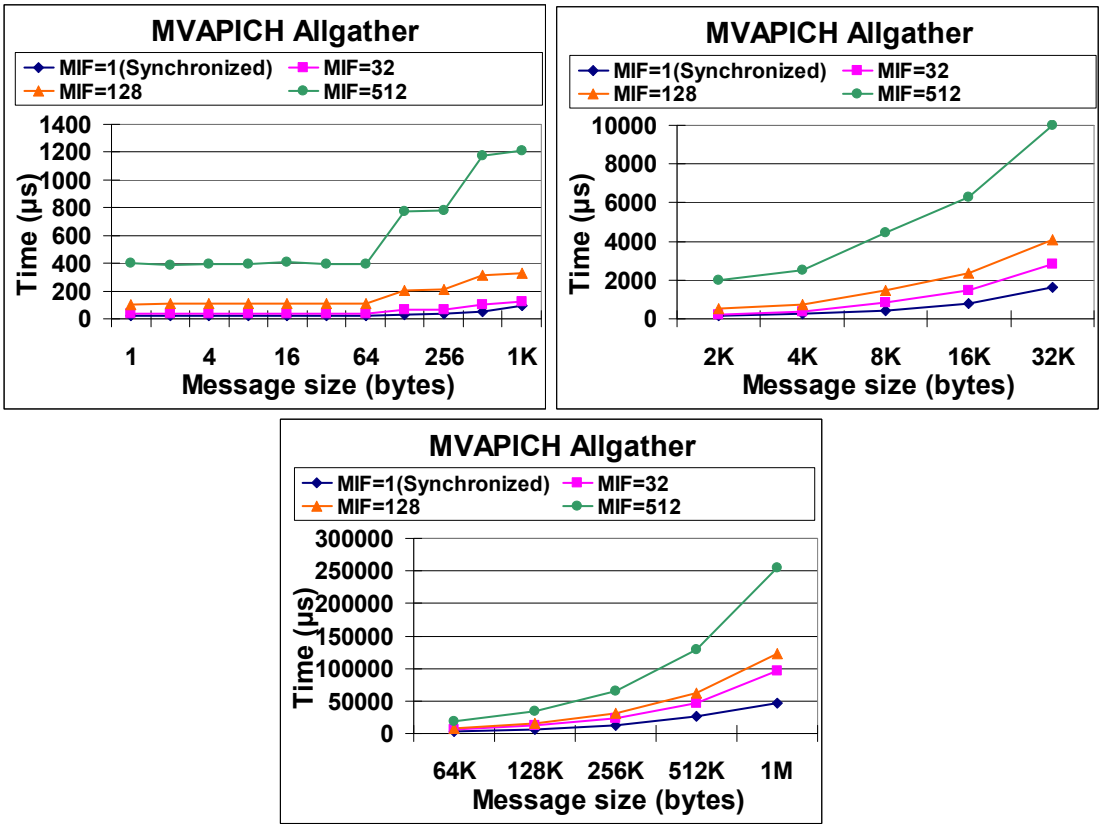

Figure 6.3 Completion time of MVAPICH Allgather under different process arrival patterns.

6.4.1 Notification Mechanisms for Early-arrival Processes

The basic idea in my PAP aware algorithms is for each process to send its distinct data to the already-arrived processes as soon as possible. It is therefore very important to have an efficient mechanism in place to inform others of the early-arrival processes. For this, I have devised two different notification mechanisms for *zero-copy* and *copy-based* schemes used in RDMA-based communications. These notification mechanisms do not incur any communication overhead.

In the zero-copy approach, where the cost of data copy is prohibitive for large messages, the application buffers are registered to be directly used for data transfer. However, for an RDMA Write message transfer to take place each source process needs to know the address of the remote destination buffers. For this, each process will advertise its registered destination buffer addresses to all other processes by writing into their pre-registered and pre-advertised control buffers. This inherent destination address advertisement mechanism can be interpreted as a control message to indicate a process has arrived at the collective call. Therefore, processes can poll their control buffers to understand which other process has already arrived at the collective call.

The copy-based technique involves a data copy to pre-registered and pre-advertised intermediate data buffers at both send and receive sides. Therefore, the received data in the pre-registered intermediate destination buffer can be used as a signal that the sending process has already arrived at the site. This can be checked out easily by polling the intermediate destination buffer.

6.4.2 RDMA-based Process Arrival Pattern Aware Alltoall

My base algorithm is the Direct alltoall algorithm. Let N be the total number of processes involved in the operation. In this algorithm, at step i, process p sends its message to process $(p + i)$ *mod* N, and receives from $(p - i)$ *mod* N. To implement this algorithm, each processes p first posts its RDMA Writes to all other processes in sequence (after it receives the destination buffer addresses). It then polls the completion queues to make sure its messages have been sent to all other processes. Finally it waits to receive the incoming messages from all processes.

To make this algorithm PAP aware using the zero-copy scheme, each process p polls its control buffers for the advertised remote destination buffer addresses starting from process $(p + i)$ *mod* N. It then sends its data to final destination buffers of the early-arrived processes using RDMA Write. Subsequently, it waits for the remaining processes to arrive in order to send its messages to them. Finally, each process waits for all incoming messages by polling its own destination buffers. The beauty of this PAP aware algorithm over the non-PAP aware algorithm is that a sending process will never get stuck for a particular process to arrive in order to proceed with the next message transfer.

Under the copy-based scheme, each process p polls its intermediate destination buffers, starting from process $(p - i)$ *mod* N. Any received data indicates that the corresponding process has already arrived. The process p then copies its messages using RDMA Write to all early-arrived processes. It then sends its data to the rest of processes who have not yet arrived. All processes also need to wait to receive messages from all

other processes into their intermediate buffers, and then copy them to their final destination buffers.

6.4.3 RDMA-based Process Arrival Pattern Aware Allgather

RDMA-based Process Arrival Pattern Aware allgather algorithm is very similar to the alltoall algorithm explained in Section 6.4.2. It is based on the Direct allgather algorithm. I have also used the same notification mechanisms for *zero-copy* and *copy-based* schemes used in RDMA-based communications. The only difference is that each process sends the same data/address to the other processes in the allgather operation.

6.4.4 RDMA-based Process Arrival Pattern and Shared Memory Aware Alltoall

Up to this point, I have utilized the RDMA features of a modern InfiniBand cluster along with PAP awareness in an effort to improve the performance of *MPI_Alltoall()*. Previous research has shown that shared memory intra-node communication can improve the performance of collectives for small to medium messages. However, it is interesting to see how shared memory intra-node communication might affect the performance under different process PAP. For this, I propose an SMP-aware and RDMA-based PAP aware *MPI_Alltoall()* algorithm that has the following three phases:

Phase 1: Intra-node shared memory gather performed by a Master process

Phase 2: Inter-node PAP aware Direct alltoall among the Masters

Phase 3: Intra-node PAP and shared memory aware scatter by a Master process

A Master process is selected for each node (without loss of generality, the first process in each node). Phase 1 cannot be PAP aware because the Master process has to wait for all intra-node messages to arrive into a shared buffer before moving on to the

PAP aware Phase 2. Phase 2 is the same as the algorithm proposed in Section 6.4.2, and is performed among the Master processes. In Phase 3, an intra-node shared memory and PAP aware scatter is devised. Because the Master processes may arrive in Phase 2 at different times, this awareness can be passed on to Phase 3 by allowing the intra-node processes to copy their destined data available in the shared buffer to their final destinations without having to wait for data from all other Masters.

In a shared memory but non-PAP aware Phase 3, a Master process waits to receive the data from all other Master processes. It then copies them all to a shared buffer and sets a shared *done* flag. All other intra-node processes poll on this flag, and once set they start copying their own data from the shared buffer to their final destinations.

In a shared memory and PAP aware Phase 3, we consider multiple shared done flags, one for data from each Master process (four flags in our 4-node cluster). As soon as a Master process receives data from any other Master process, it copies it to the shared buffer and then sets the associated done flag. All other intra-node processes poll on all done flags, and as soon as any partial data is found in the shared buffer they copy them out to their final destination buffers.

6.4.5 RDMA-based Process Arrival Pattern and Shared Memory Aware Allgather

Similar to the *MPI_Alltoall()* algorithm in Section 6.4.4, I propose an SMP-aware and RDMA-based PAP aware *MPI_Allgather()* algorithm that has the following three phases:

Phase 1: Intra-node shared memory gather performed by a Master process

Phase 2: Inter-node PAP aware Direct allgather among the Masters

Phase 3: Intra-node PAP and shared memory aware broadcast by a Master process

Phase 1 is similar to the Phase 1 of the *MPI_Alltoall()* algorithm, but with less data movements. Phase 2 employs the same *MPI_Allgather()* algorithm explained in Section 6.4.3, among the Master processes. In Phase 3, an intra-node shared memory and PAP aware broadcast is done instead of scatter in *MPI_Alltoall()*.

6.5 Experimental Results

In this section, I present the performance results of the proposed algorithms, the RDMA-based PAP aware Direct (*PAP_Direct*) alltoall and allgather, and RDMA-based PAP and Shared memory aware Direct (*PAP_Shm_Direct*) alltoall and allgather, and compare them with the non-PAP aware RDMA-based Direct (*Direct*) and RDMA-based and Shared memory aware Direct (*Shm_Direct*) algorithms as well as with the native MVAPICH on the cluster.

I have evaluated the proposed algorithms using both copy-based and zero-copy techniques for 1B to 1MB messages. The results shown in the section are the best results of the two schemes for each algorithm. Again, I use a cycle-accurate timer to record the time spent in the collectives (1000 iterations) for each process, and then calculate the average alltoall latency across all processes.

6.5.1 MPI_Alltoall() Micro-benchmark Results

Figure 6.4 compares the PAP aware *MPI_Alltoall()* algorithms with the native MVAPICH implementation and non-process arrival pattern aware versions, with MIF equal to 32 and 512. Clearly, the PAP aware algorithms, PAP_Direct and PAP_Shm_Direct, are better than their non-PAP aware counterparts for all message sizes.

This shows that indeed such algorithms can adapt themselves well with different PAP. My algorithms are also superior to the native MVAPICH, with an improvement factor of 3.1 at 8KB for PAP_Direct and 3.5 at 4B for PAP_Shm_Direct, with MIF equal to 32. With a larger MIF of 512, the improvements are 1.5 and 1.2, respectively.

Comparing the PAP_Shm_Direct with PAP_Direct, one can see that the PAP_Shm_Direct is the algorithm of choice up to 256 bytes for MIF equal to 32. However, this is not the case for MIF of 512 where processes may arrive at the call with more delay with respect to each other. This shows that the SMP version of my process arrival pattern algorithm introduces some sort of implicit synchronization in Phase 1 that may degrade its performance under large maximum imbalanced factors.

To evaluate the scalability, I compare the performance of the PAP_Direct *MPI_Alltoall()* with those of MVAPIVH and Direct algorithm for 4, 8, and 16 processes, as shown in Figure 6.5 (shared memory algorithms are not shown due to limited data points). One can see that the proposed PAP aware algorithm has scalable performance and is always superior to the non-PAP aware algorithms. I have found similar results for other MIFs and messages sizes.

In the previous micro-benchmark, the arrival time of each process is random. In another micro-benchmark, I control the number of late processes. In Figure 6.6, I present the *MPI_Alltoall()* results for MIF equal to 128 when 25% or 75% of processes arrive late. My proposed algorithms are always better than their counterparts for the 25% case, and mostly better in the 75% case. The PAP_Shm_Direct alltoall is always better than MVAPICH, although with a less margin in the 75% case.

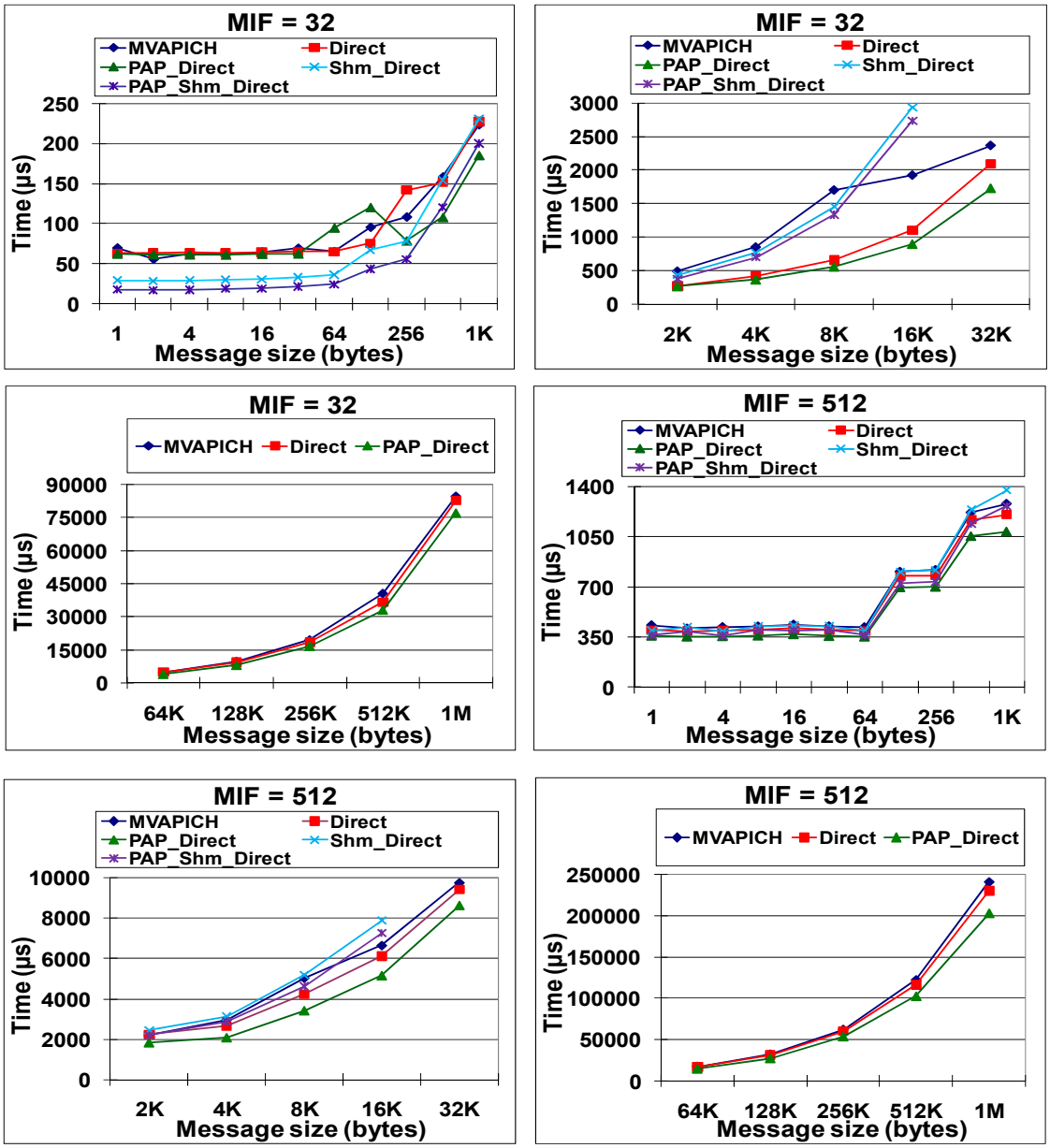

Figure 6.4 Performance of the proposed MPI_Alltoall(), 16 processes on a 4-node, 16-core cluster.

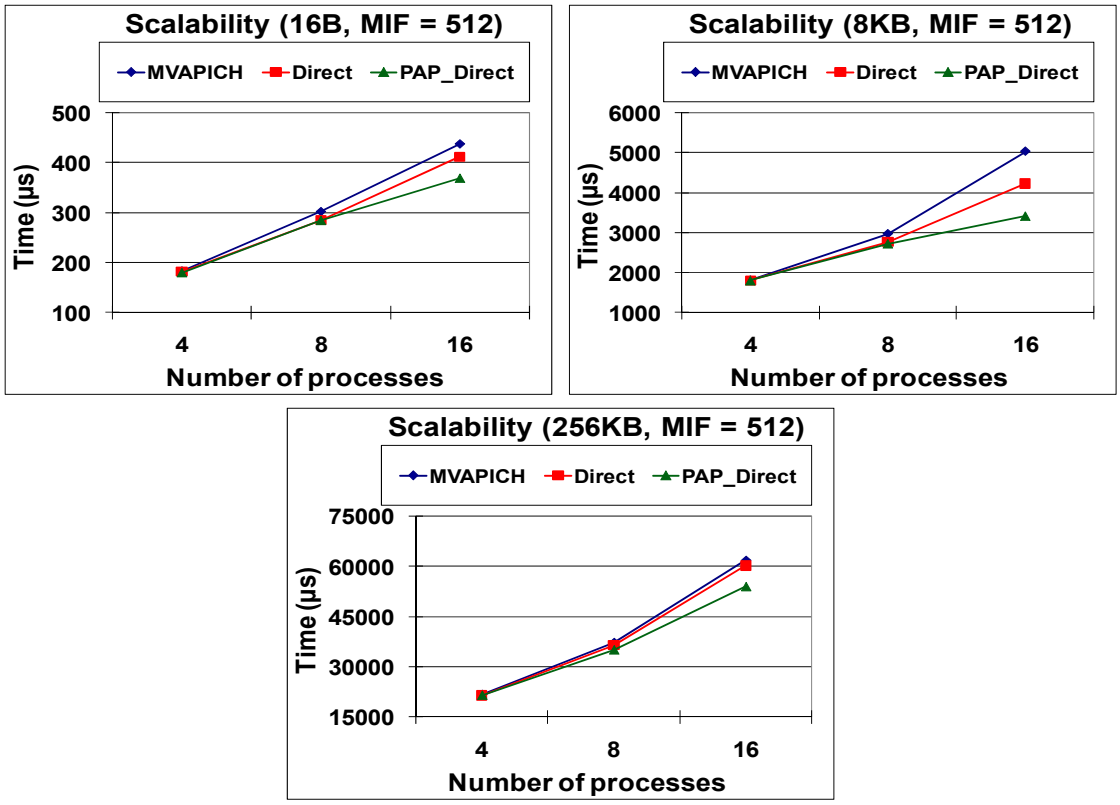

Figure 6.5 MPI_Alltoall() scalability.

6.5.2 MPI_Allgather() Micro-benchmark Results

The *MPI_Allgather()* results are shown in Figure 6.7, 6.8 and 6.9. They are also the best results of the two schemes for each algorithm. Figure 6.7 compares the PAP aware *MPI_Allgather()* algorithms with the native MVAPICH implementation and non-PAP aware version, with MIF equal to 32 and 512. The PAP aware *MPI_Allgather()* algorithms, PAP_Direct and PAP_Shm_Direct, are better than their non-process arrival pattern aware counterparts for all message sizes. This shows that such algorithms can improve performance for different collective operations. My algorithm is also superior to the native MVAPICH, with an improvement factor of 3.1 at 8KB for PAP_Direct and 2.5

at 1B for PAP_Shm_Direct, with MIF equal to 32. With a larger MIF of 512, the improvement is 1.3 and 1.2, respectively.

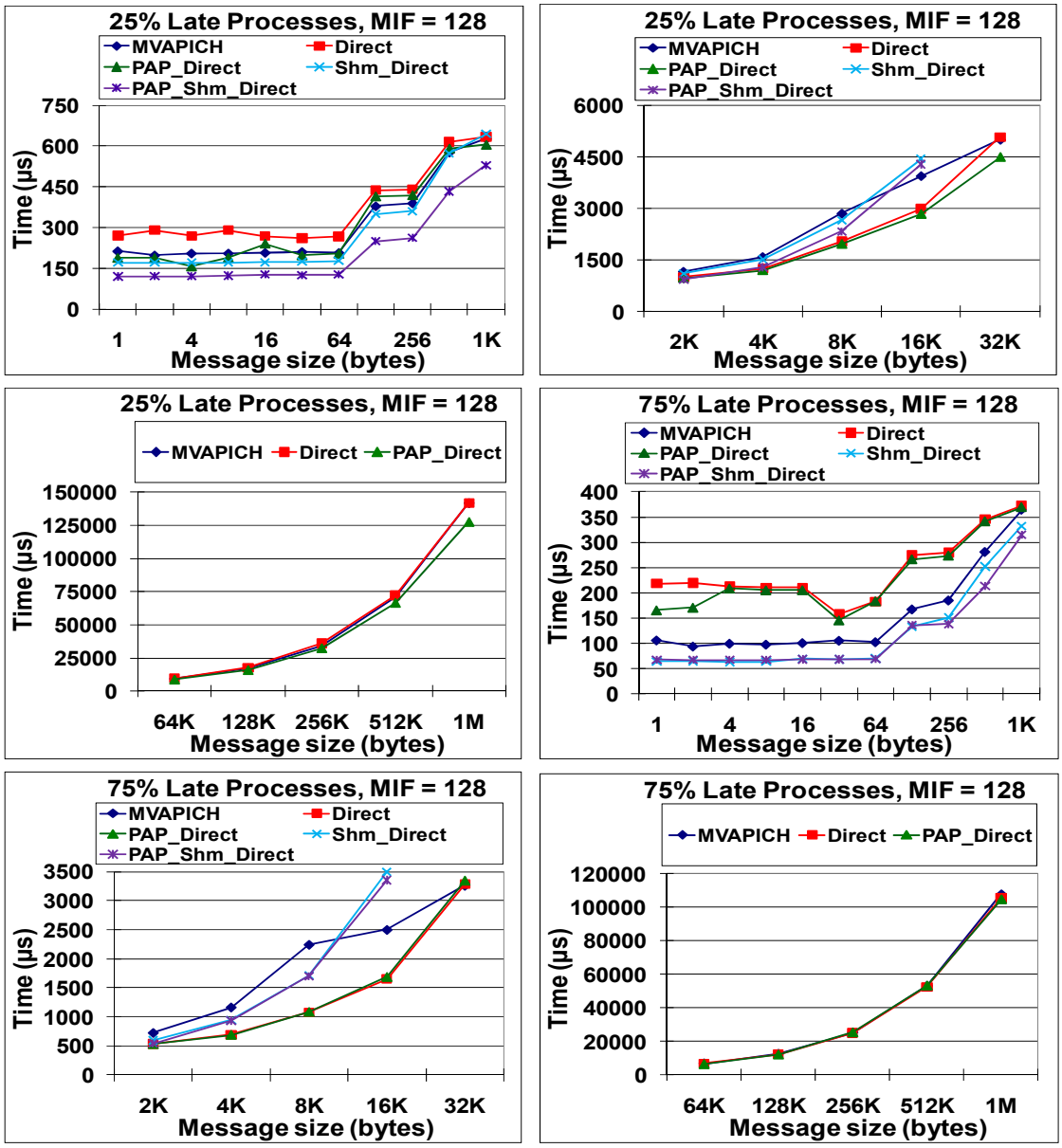

Figure 6.6 Performance of the proposed MPI_Alltoall() with 25% and 75% late processes.

Comparing the PAP_Shm_Direct with PAP_Direct, the PAP_Shm_Direct is the algorithm of choice up to 1KB for MIF equal to 32. This is better than the one achieved by *MPI_Alltoall()*. The reason is because *MPI_Alltoall()* has more shared memory data movements than *MPI_Allgather()*. For MIF of 512, PAP_Shm_Direct and Shm_Direct perform much better than other algorithms up to 32KB. This indicates that shared memory implementation can speed up *MPI_Allgather()* greatly. The performance of PAP_Shm_Direct and Shm_Direct are very close for MIF=512 results. This indicates that the SMP version of the algorithm introduces some sort of implicit synchronization, which may degrade its performance under large maximum imbalanced factors.

I compare the scalability performance of the PAP_Direct *MPI_Allgather()* with those of MVAPIVH and Direct algorithm for 4, 8, and 16 processes, as shown in Figure 6.8. The PAP algorithm performs better for large message sizes. The proposed PAP aware algorithm has scalable performance and is always superior to the non-PAP aware algorithms.

In Figure 6.9, I present the *MPI_Allgather()* results for MIF equal to 128 when 25% or 75% of processes arrive late. My PAP_Direct allgather algorithm is always better than its counterpart for the 25% case, and for messages larger than 4KB in the 75% case. PAP_Shm_Direct allgather algorithm is very close to its counterpart for both cases, and it is the best algorithm up to 8Kbyes for 25% case and 2KB for 75% case.

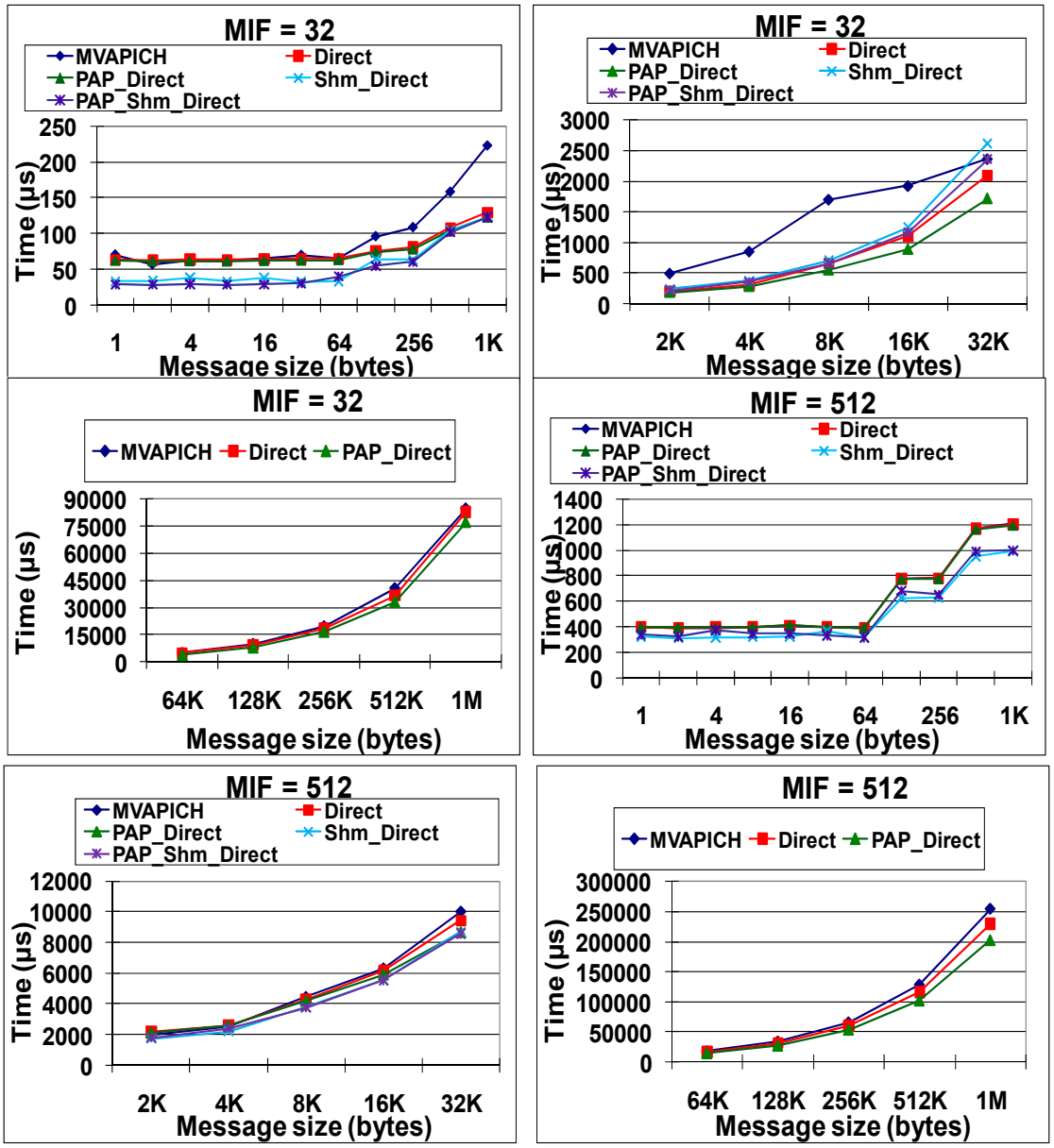

Figure 6.7 Performance of the proposed MPI_Allgather(), 16 processes on a 4-node, 16-core cluster.

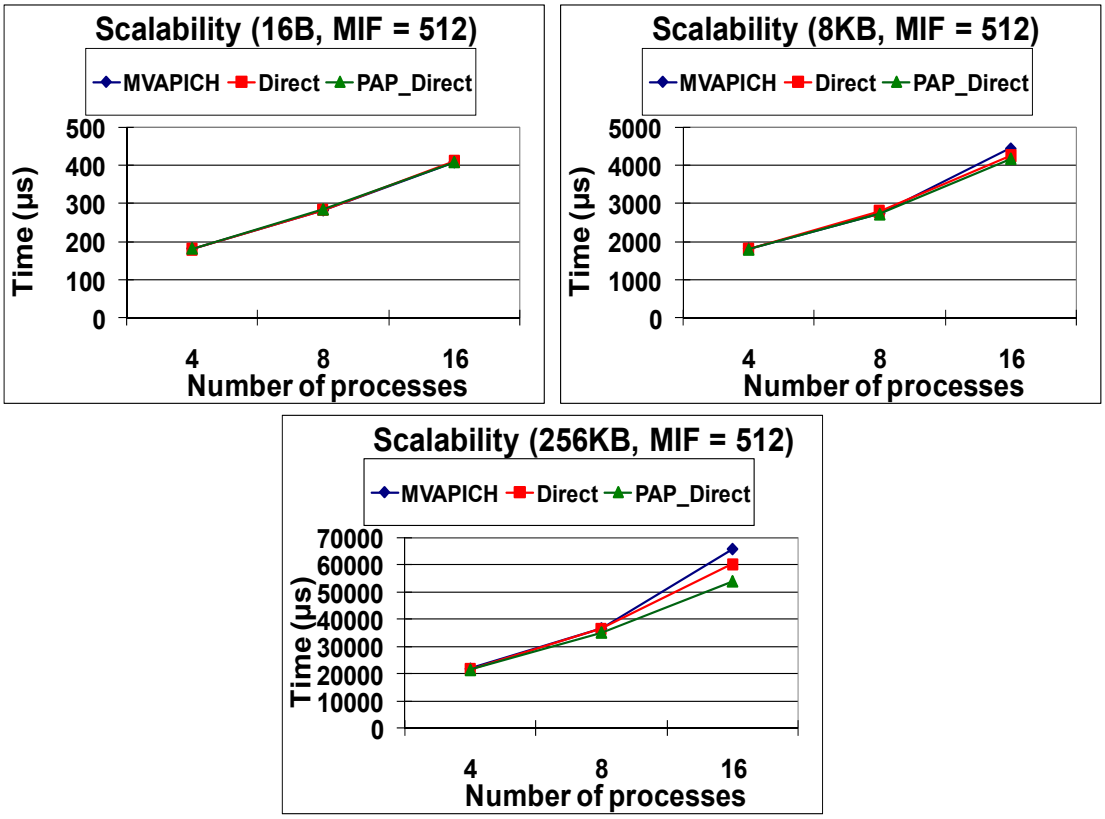

Figure 6.8 MPI_Allgather() scalability.

6.5.3 Application Results

In this section, I consider the FT application benchmark from NPB, version 2.4 [49], and N-BODY and RADIX [72] applications to evaluate the performance and scalability of the proposed PAP aware *MPI_Alltoall()* and *MPI_Allgather()*. FT uses *MPI_Allltoall()* as well as a few other collectives communications. I have experimented with class B and C of FT, running with different number of processes, which use payloads larger than 2MB. Table 6.2 shows the PAP aware *MPI_Alltoall()* speedup over the native MVAPICH and the Direct algorithms for FT running with 4, 8, and 16 processes. Clearly, the proposed algorithm outperforms the conventional algorithms. The results also show

that the PAP aware *MPI_Alltoall()* has modest scalability as speedup improves with increasing number of processes.

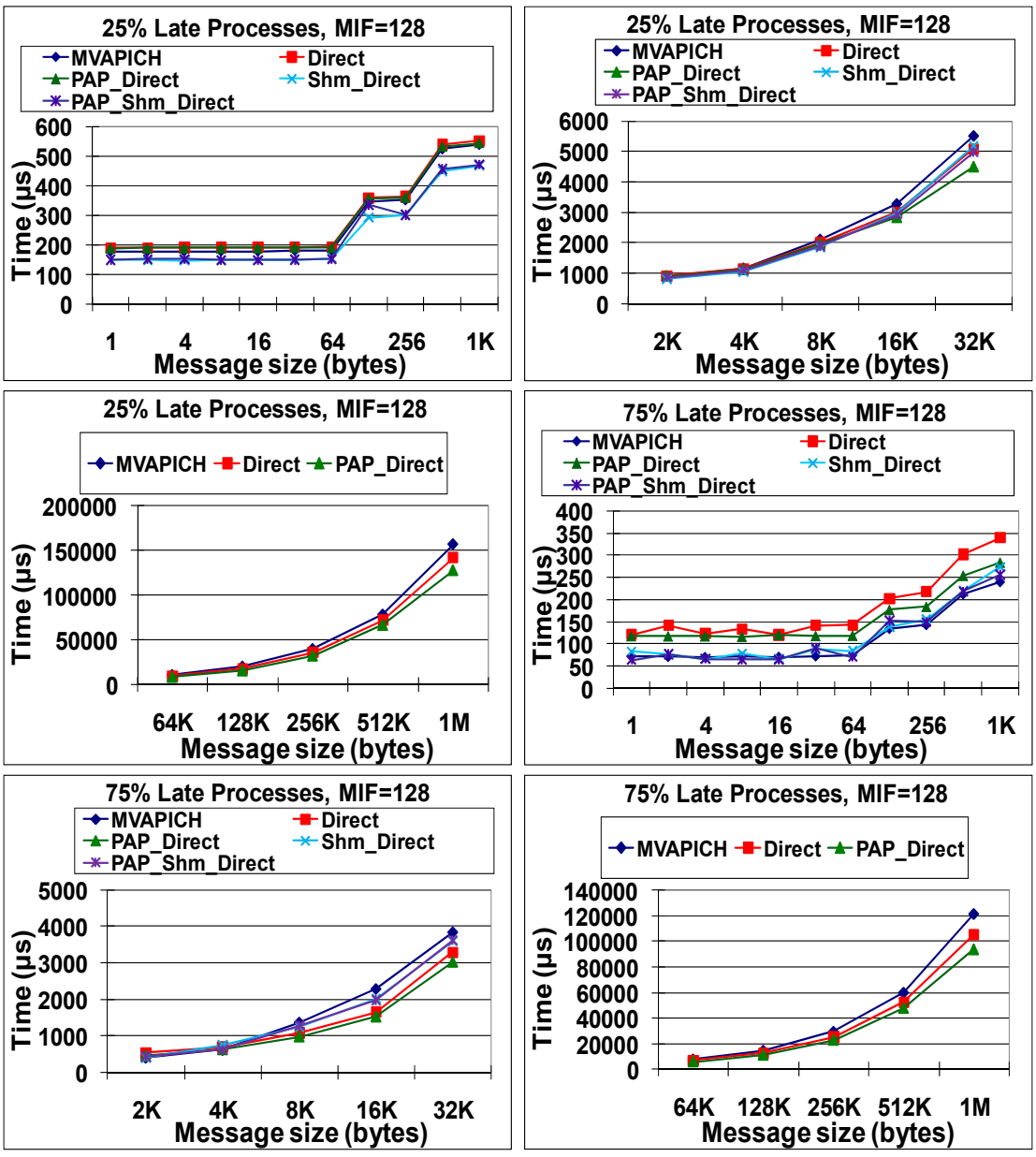

Figure 6.9 Performance of the proposed MPI_Allgather() with 25% and 75% late processes.

Table 6.2 PAP_Direct MPI_Alltoall() speedup over native MVAPICH and the Direct algorithms for NAS FT running with different number of processes and classes.

	Speedup over native MVAPICH algorithm		Speedup over Direct algorithm	
	FT (class B)	FT (class C)	FT (class B)	FT (class C)
4 processes	1.08	1.01	1.16	1.04
8 processes	1.10	1.04	1.04	1.14
16 processes	1.14	1.17	1.42	1.63

N-BODY and RADIX mainly use *MPI_Allgather()* with relatively small message sizes, 4KB for RADIX and mostly 64B for NBODY (some with less than 1KB payload). Therefore, the best choice will be PAP_Shm_Direct algorithm. Table 6.2 shows the SMP version of PAP aware *MPI_Allgather()* speedup over the native MVAPICH and the SMP version of Direct algorithms for N-BODY and RADIX running with 4, 8, and 16 processes. The proposed PAP algorithm outperforms the conventional algorithms. The improvement increases with more number of processes.

Table 6.3 PAP_Shm_Direct MPI_Allgather() speedup over native MVAPICH and the shared memory aware Direct algorithms for N-BODY and RADIX running with different number of processes.

	Speedup over native MVAPICH algorithm		Speedup over Shm_Direct algorithm	
	N-BODY	RADIX	N-BODY	RADIX
4 processes	1.01	0.92	0.97	1.11
8 processes	1.03	1.53	0.92	1.19
16 processes	1.52	1.62	1.31	1.22

6.6 Summary

MPI_Alltoall() and *MPI_Allgather()* are two of the most communication-intensive primitives in MPI. Imbalanced PAP has an adverse impact on their performance. In this chapter, I have proposed RDMA-based PAP aware *MPI_Alltoall()* and *MPI_Allgather()* algorithms and extended them to be shared memory aware without introducing any extra control messages.

The performance results indicate that the proposed PAP aware *MPI_Alltoall()* and *MPI_Allgather()* algorithms perform better than the native MVAPICH and the traditional Direct and SMP-aware algorithms when processes arrive at different times. They also improve the communication performance in the applications studied.

While this study was focused at *MPI_Alltoall()* and *MPI_Allgather()*, it can be directly extended to other collectives. The proposed techniques can be applied to other alltoall and allgather algorithms such as Bruck or Recursive Doubling. However, one has to bear in mind that due to synchronization between different steps of these algorithms they may not achieve the highest performance as in the Direct algorithm.

Chapter 7: Conclusion and Future Work

In this dissertation, I have proposed and evaluated a number of algorithms for MPI collective operations over high-performance interconnects. I have proposed how to take advantage of advanced features provided by modern interconnects such as RDMA, multi-rail communication, and multi-connection capability in order to design efficient collective operations. My work has also taken into account the multi-core and SMP clusters architectures, as well as the runtime process arrival pattern issue.

In Chapter 3, I have presented new designs that exploit multi-rail communication techniques over multiple independent networks/rails, or multi-port NICs, to overcome bandwidth limitations. I have adapted well-kown multi-port algorithms for a number of collective operations, including scatter, gather, allgather and alltoall personalized exchange to work over multi-rail networks using RDMA techniques. I have evaluated in detail the performance improvement offered by the new approaches over QsNet^{II} dual-rail systems. The proposed techniques can achieve superior bandwidth improvement. The RDMA-based multi-port scatter and gather algorithms include a tree-based and a Direct algorithm. The allgather and alltoall algorithms include the Direct, Standard Exchange, and Bruck algorithms. The performance results show that the algorithms are superior over the native implementations. In fact, the multi-port RDMA-based Direct algorithms for gather and allgather collectives gain an improvement of up to 2.15 for 4KB messages and 1.49 for 32KB messages over *elan_gather()*, respectively. In addition, the RDMA-based Direct alltoall outperforms *elan_alltoall()* up to 2.26 for 2KB messages

The RDMA-based algorithms, however, did not perform well for short messages. The native gather, allgather and alltoall implementation had a better latency for up to 512B, 2KB, and 512B, respectively. The reason is because shared memory operations have absolute advantage over RDMA Reads or Writes for small messages on SMP/multi-core nodes. To address this deficiency, in Chapter 4, I proposed RDMA-based and shared memory aware multi-port algorithms to speedup the collectives for co-located processes on SMP/multi-core nodes. I showed that concurrent shared memory transfer can greatly improve the collectives performance for small to medium size messages. I proposed two classes of SMP-aware allgather algorithms: *SMP-aware Gather and Broadcast* algorithm and *SMP-aware Direct and Bruck* algorithms.

The *SMP-aware Gather and* Broadcast algorithm performed best for very short messages up to 256B. The *SMP-aware Bruck* algorithm outperformed all algorithms including *elan_gather()* for 512B to 8KB messages, with a 1.96 improvement factor for 4KB messages. The multi-port Direct allgather proposed in Chapter 3 was still the best algorithm for 16KB to 1MB. The scalability results verifed the superiority of the algorithms for various message sizes. In addition, the performance of NBODY and RADIX applications as well as their communication performance was improved using the proposed algorithms.

InfiniBand has been proposed as a high-performance interconnect. I showed that the latest InfiniBand cards can provide much better performance and salability for simultaneous communication over multiple connections. By taking advantage of this feature, in Chapter 5, I proposed three multi-core and/or multi-connection aware

MPI_Allgather() algorithms over ConnectX InfiniBand networks: (1) the *Multi-group Gather-based Multi-connection Aware MPI_Allgather()* algorithm targeted at very small messages. (2) the *Single-group Multi-connection Aware MPI_Allgather()* algorithm targeted at small to medium messages, and (3) the *Multi-group Multi-connection Aware MPI_Allgather()* algorithm for medium to large message sizes. I also compared the per-node complexity of each of the proposed algorithm to estimate their performance.

The multi-group multi-connection aware algorithms performed better than the native allgather implementation from 4KB to 64KB, mostly due to the use of multiple concurrent connections and multiple cores. The gather-based algorithm showed the best performance for very small messages, up to 32 bytes, mostly because this algorithm efficiently used the available cores and lightly utilized the network communications. Finally, the single-group multi-connection aware algorithm outperformed all other algorithms from 64B to 2KB, since it had a lighter shared memory volume.

Lastly, in Chapter 6, I took into account the process arrival pattern impact on the collective communications. I studied the process arrival pattern behavior of NAS parallel benchmarks and showed that indeed processes arrive at different times at collectives. I then evaluated the impact of the process arrival pattern on the performance of collectives. The results confirmed that it is essential to have process arrival pattern aware collecives. For this, I proposed RDMA-based process arrival pattern aware alltoall and allgather algorithms and then extended them to be shared memory aware. For performance reasons, I utilized the inherent features of RDMA data transfer mechanisms for notification purposes to avoid introducing any extra control messages.

The performance results indicated that the proposed PAP aware *MPI_Alltoall()* and *MPI_Allgather()* algorithms perform better than the native MVAPICH and the traditional non PAP aware Direct and shared memory algorithms when processes arrive at different times. The shared memory and RDMA-based PAP aware algorithms were designed to target smaller message size. The proposed *MPI_Alltoall()* algorithms outperformed the native implementation by up to 3.1 times at 8KB for PAP_Direct and 3.5 times at 4B for PAP_Shm_Direct. The proposed *MPI_Allgather()* algorithms gained an improvement factor of 3.1 at 8KB for PAP_Direct and 2.5 times at 1B for PAP_Shm_Direct. In addition, the communication performance of NAS FT, NBODY and RADIX applications were improved by up to 63% over the native and non PAP aware implementations.

It should be mentioned while this dissertation was focused at Quadrics and InfiniBand networks, the proposed collective algorithms can be used on top of any RDMA-based interconnects. For instance, the work proposed on InfiniBand can be directly applied to iWARP Ethernet without any modifications, as the algorithms were designed on top of OFED, and thus they are portable.

7.1 Future Work

The high-performance and rich features offered by modern interconnects such as $QsNet^{II}$ and InfiniBand make them very attractive for large scale system design. In this dissertation, I have exploited such modern features along with multi-core/SMP architectures to design efficient and scalable MPI collective communications. I would like to extend this work and also explore several other interesting research topics in the future.

- **Extension of Multi-connection and Multi-core Aware Algorithms**

I would like to extend my work in this area to other collective communications of interest such as *MPI_Reduce(), MPI_Allreduce(),* etc. It is also interesting to discover how such algorithms behave on emerging large multi-core clusters with new architectures (such as NUMA) and more number of cores per nodes. I would also like to investigate the performance of the proposed collectives over other InfiniBand transport protocols.

- **Extension of PAP Aware Algorithms to Other Collectives**

My PAP algorithms are based on the Direct algorithm. I would like to extend it to PAP Bruck and Recursive Doubling algorithms for allgather and alltoall operations. I am interested in devising other process arrival pattern aware collectives over emerging multi-core clusters. I would also like to use a larger multi-core SMP cluster in the future to evaluate the algorithms as the process arrival pattern will become even more crucial for larger systems. It is very important to optimize the algorithms for such large systems.

- **Collectives for Next-Generation Programming Models**

One of the challenges to petascale computing is the programmer productivity. The *Partitioned Global Address Space* (PGAS) programming model [23] has been gaining rising attention due to its prospects as the basis for productive parallel programming. The PGAS model provides for ease-of-use through its global shared address space view. The DARPA HPCS [20] program has also introduced new promising PGAS languages, such as X10 [83] and Chapel [18]. Devising efficient collective communications for such languages is crucial for their performance. I would like to exploit the techniques used in

this dissertation as well as new novel techniques designed according to the specific features offered by these languages.

- **Onloading vs. Offloading vs. Hybrid Communication Stacks**

Some of the networking companies such as Mellanox have consistently tried to offload most of the communication tasks to programmable processors on the network interface cards. On the contrary, some other companies like QLogic have designed their host channel adapters with limited offloading capability. Their argument is that computing nodes are becoming more powerful due to the availability of multi- and soon many-core processors. Therefore, it is better to onload most of the protocol processing tasks. There is right now a debate among the research community as towhether we should move toward onloading or offloading. There is also a middle-ground, where some communication tasks should be offloaded while others are handled by the host processors. I would like to investigate the impact of these design alternatives on the performance of collective communications.

References

[1] A. Alexandrov, M. Ionescu, K.E. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation," *Proc. of 7th Annual Symposium on Parallel Algorithms and Architecture (SPAA)*, July, 1995.

[2] Q. Ali, S.P. Midkiff, and V.S. Pai, "Efficient High Performance Collective Communication for the Cell Blade," *Proc. of 23rd ACM International Conference on Supercomputing (ICS),* 2009.

[3] O. Aumage, E. Brunet, G. Mercier, and R. Namyst, "High-Performance Multi-Rail Support with the NewMadeleine Communication Library," *16th International Heterogeneity in Computing Workshop (HCW), Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, CA, 2007.

[4] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Traeff, "MPI on a Million Processors," *16th EuroPVM/MPI*, *Lecture Notes in Computer Science (LNCS)* 5759, Espoo, Finland, September, 2009.

[5] A. Bar-Noy and S. Kipnis, "Designing Broadcasting Algorithms in the Postal Model for Message Passing Systems," *Proc. of 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, San Diego, CA, pages 13-22, 1992.

[6] J. Beecroft, D. Addison, D. Hewson, M. McLaren, D. Roweth, F. Petrini, and J. Nieplocha, "QsNet^{II}: Defining High-performance Network Design," *IEEE Micro*, 25(4): 34-47, 2005.

[7] R. A.F. Bhoedjang, T. Rühl, and H. E. Bal, "User-Level Network Interface Protocols," *IEEE Computer*, 31(11): 53-60, November, 1998.

[8] S.H. Bokhari, "Multiphase Complete Exchange on Paragon, SP2, and CS-2," *IEEE Parallel and Distributed Technology*, 4(3): 45-59, 1996.

[9] R. Brightwell, D. Doerfler, and K.D. Underwood, "A Comparison of 4X InfiniBand and Quadrics elan-4 Technologies," *Proc. of 6th IEEE International Conference on Cluster Computing (Cluster)*, pages 193-204, 2004.

[10] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient Algorithms for All-to-all Communications in Multiport Message-passing Systems," *IEEE Trans. Parallel and Distributed Systems*, 8(11): 1143-1156, 1997.

[11] D. Buntinas, G. Mercier, and W. Gropp, "Data Transfers between Processes in an SMP System: Performance Study and Application to MPI," *Proc. of 35th International Conference on Parallel Processing (ICPP)*, 2006.

[12] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini, "User-level Communication in Cluster-based Servers," *Proc. of the 8th Symposium on High-Performance Architecture (HPCA)*, February 2002.

[13] L. Chai, A. Hartono, and D.K. Panda, "Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters," *Proc. of 8th IEEE International Conf. on Cluster Computing (Cluster)*, Barcelona, Spain, 2006.

[14] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda, "Designing An Efficient Kernel-level and User-level Hybrid Approach for MPI Intra-node Communication on Multi-core Systems," *Proc. of 37th International Conference on Parallel Processing (ICPP)*, Portland, Oregon, September, 2008.

[15] E. Chan, R. Van de Geijn, W. Gropp, and R. Thakur, "Collective Communication on Architectures that Support Simultaneous Communication over Multiple Links," *Proc. of 11th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, New York City, NY, pages 2-11, 2006.

[16] S. Coll, E. Frachtenberg, F. Petrini, A. Hoisie, and L. Gurvits, "Using Multirail Networks in High Performance Clusters," *Concurrency and Computation: Practice and Experience*, 15(7-8): 625-651, 2003.

[17] ConnectX InfiniBand Adapters, product brief, Mellanox Technologies, Inc. http://www.mellanox.com/pdf/products/hca/ConnectX_IB_Card.pdf

[18] Cray, Inc. Chapel Programming Language, http://chapel.cs.washington.edu/.

[19] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eiken, "LogP: Towards a Realistic Model of Parallel Computation," *Proc. of 4th ACM SIGPLAN Symposium on Principles and Practice of*

Parallel Programming (PPoPP), 1993.

[20] DARPA. High Productivity Computer Systems, http://www.highproductivity.org/.

[21] A. Faraj, P. Patarasuk, X. Yuan, "A Study of Process Arrival Patterns for MPI Collective Operations," *International Journal of Parallel Programming*, 36(6): 543-570, 2008.

[22] A. Faraj, P. Patarasuk, and X. Yuan, "Bandwidth Efficient All-to-All Broadcast on Switched Clusters," *International Journal of Parallel Programming* 36(4): 426-453, 2008.

[23] M. Farreras, G. Almasi, C. Cascaval, and T. Cortes, "Scalable RDMA performance in PGAS languages," *Proc.of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, May, 2009.

[24] E. Garbriel, G. Fagg, G. Bosilica, T. Angskun, J. J. D. J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T.Woodall, "OpenMPI: Goals, Concept, and Design of a Next Generation MPI Implementation," *Proc. of 11th European PVM/MPI*, 2004.

[25] R. Hockney, "The Communication Challenge for MPP, Intel Paragon and Meiko CS-2," Parallel Computing, 20(3): 389–398, 1994.

[26] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately Measuring Collective Operations at Massive Scale," *Proc. of 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Miami, FL, pages 1-8, April, 2008.

[27] T. Hoefler and J.L. Traff, "Sparse Collective Operations for MPI," International Workshop on High-level Parallel Programming Models and Supportive Environments, Proc. of 23rd *IEEE* International *Parallel and Distributed Processing Symposium* (IPDPS), Rome, Italy, May, 2009.

[28] W. Huang, M. Koop, and D.K. Panda, "Efficient One-Copy MPI Shared Memory Communication in Virtual Machines," *Proc. of 10th IEEE International Conf. on Cluster Computing (Cluster)*, Tsukuba, Japan, September, 2008.

[29] W. Huang, G. Santhanaraman, H. -W. Jin, and D. K. Panda, "Scheduling of MPI-2 One Sided Operations over InfiniBand," *Workshop on Communication Architecture*

on Clusters (CAC),), Proc. of 19th International Parallel and Distributed Processing Symposium (IPDPS), pages 215, 2005.

[30] IBM BlueGene/P, IBM, Http://www.research.ibm.com/journal/rd/521/team.html.

[31] InfiniBand Architecture, http://www.infinibandta.org.

[32] H.-W. Jin, S. Sur, L. Chai, and D.K. Panda, "LiMIC: Support for High-performance MPI Intra-node Communication on Linux Clusters," *Proc. of 34th International Conference. on Parallel Processing (ICPP),* Denver, Colorado, 2005.

[33] S. Kamil, J. Shalf, L. Oliker, and D. Skinner, "Understanding Ultra-Scale Application Communication Requirements," *Proc. of IEEE International Workload Characterization Symposium (IISWC)*, pages 178 – 187, October, 2005.

[34] T. Kielmann, H.E. Bal, and K. Verstoep, "Fast Measurement of LogP Parameters for Message Passing Platforms," 4th Workshop on Runtime Systems for Parallel Programming (RTSPP), Proc. of 14th International Parallel and Distributed Processing Symposium (IPDPS), 2000.

[35] M. Koop, T. Jones, and D. K. Panda, "MVAPICH-Aptus: Scalable High-Performance Multi-Transport MPI over InfiniBand," *International Parallel and Distributed Processing Symposium (IPDPS 2008)*, Miami, Florida, April, 2008.

[36] M. Koop, J. Sridhar, and D.K. Panda, "Scalable MPI Design over InfiniBand using eXtended Reliable Connection," *Proc. of 10th IEEE International Conf. on Cluster Computing (Cluster)*, Tsukuba, Japan, September, 2008.

[37] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The deep Computing Messaging Framework: generalized Scalable Message Passing on the Blue Gene/P Supercomputer," *Proc. of 22nd International Conference on Supercomputing (ICS)*, 2008.

[38] J. Liu, A. Vishnu, and D.K. Panda, "Building Multirail InfiniBand Clusters: MPI-level Design and Performance Evaluation," *Proc. of 2004 ACM/IEEE Conf. on Supercomputing (SC 2004)*, 2004.

[39] A.R. Mamidala, L. Chai, H.-W. Jin, and D.K. Panda, "Efficient SMP-aware MPI-

level Broadcast over InfiniBand's Hardware Multicast," *Workshop on Communication Architecture on Clusters (CAC), Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS),* Pittsburgh, PA, November, 2006.

[40] A. Mamidala, R. Kumar, D. De, and D. K. Panda, "MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics," *Proc. of 8th International Symposium on Cluster Computing and the Grid (CCGrid)*, Lyon, France, May, 2008.

[41] A. Mamidala, J. Liu, and D. K. Panda, "Efficient Barrier and Allreduce on IBA clusters using hardware multicast and adaptive algorithms," *Proc. of 6th IEEE International Conf. on Cluster Computing (Cluster)*, San Diego, California, September, 2004.

[42] A.R. Mamidala, A. Vishnu, and D.K. Panda, "Efficient Shared Memory and RDMA based Design for MPI-allgather over InfiniBand," *Proc. of EuroPVM/MPI*, pages 66-75, Bonn, Germany, September, 2006.

[43] P.K. Mckinley, Y.-J. Tsai, and D.F. Robinson, "Collective communication in wormhole-routed massively parallel computers." *IEEE Computer*, 28(12):39-50, 1995.

[44] Mellanox technologies, http://www.mellanox.com/.

[45] MPI: Message Passing Interface Forum, http://www.mpi-forum.org/.

[46] MPICH - A Portable MPI Implementation, http://www.mcs.anl.gov/mpi/mpich.

[47] MVAPICH, http://mvapich.cse.ohio-state.edu/.

[48] Myricom, http://www.myri.com/.

[49] NPB, http://www.nas.nasa.gov/Resources/Software/npb.html

[50] R. Rabenseifner, "Automatic MPI counter profiling of all users First results on a CRAY T3E 900-512," *the Message Passing Interface Developer's and User's Conference*, pages 77–85, 1999.

[51] S. Pakin**,** "Receiver-initiated Message Passing over RDMA Networks," *Proc. of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Miami, Florida, April, 2008.

[52] P. Patarasuk and X. Yuan, "Efficient MPI_Bcast across Different Process Arrival Patterns," *Proc. of 22nd International Parallel and Distributed Processing Symposium (IPDPS)*, Miami, Florida, April, 2008

[53] PDSH, http://www.llnl.gov/linux/pdsh/.

[54] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie, "Hardware- and Software-Based Collective Communication on the Quadrics Network," *Proc. of 2001 IEEE International Symposium on Network Computing and Applications (NCA),* Cambridge, MA, pages 24, October, 2001.

[55] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie, "Performance Evaluation of the Quadrics Interconnection Network," *Journal of Cluster Computing*, 6(12): 125-142, 2003.

[56] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Identifying and Eliminating the Performance Variability on the ASCI Q Machine," *Proc. of the 2003 Conference on High Performance Networking and Computing (SC)*, November 2003.

[57] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, and J.J. Dongarra, "Performance Analysis of MPI Collective Operations," *Proc. of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS),* Denver, Colorado, April, 2005.

[58] Y. Qian and A. Afsahi, "Efficient RDMA-based Multi-port Collectives on Multi-rail QsNet^{II} Clusters," *6th Workshop on Communication Architecture for Clusters (CAC)*, *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April, 2006.

[59] Y. Qian and A. Afsahi, "Efficient Shared Memory and RDMA based Collectives on Multi-rail QsNet^{II} SMP Clusters," *Cluster Computing, Journal of Networks, Software Tools and Applications*, 11(4): 341-354, 2008.

[60] Y. Qian and A. Afsahi, "High Performance RDMA-based Multi-port All-gather on Multi-rail QsNet^{II}," *Proc. of the 21st International Symposium on High Performance Computing Systems and Applications (HPCS)*, Saskatoon, SK, Canada, May, 2007.

[61] Y. Qian and A. Afsahi, "Process Arrival Pattern Aware Alltoall and Allgather on InfiniBand Clusters", *International Journal of Parallel Programming*, under review.

[62] Y. Qian and A. Afsahi, "Process Arrival Pattern and Shared Memory Aware Alltoall on InfiniBand", *16th EuroPVM/MPI*, *Lecture Notes in Computer Science (LNCS)* 5759, 250-260, Espoo, Finland, September, 2009.

[63] Y. Qian and A. Afsahi, "RDMA-based and SMP-aware Multi-port Allgather on Multi-rail QsNetII SMP Clusters", *Proc. of 36th International Conference on Parallel Processing (ICPP)*, XiAn, China, September, 2007.

[64] Y. Qian, M.J. Rashti, and A. Afsahi, "Multi-connection and Multi-core Aware All-Gather on InfiniBand Clusters", *Proc. of 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Orlando, Florida, USA, November, 2008.

[65] Quadrics, http://www.quadrics.com/.

[66] M.J. Rashti and A. Afsahi, "10-Gigabit iWARP Ethernet: Comparative Performance Analysis with InfiniBand and Myrinet-10G", *7th Workshop on Communication Architecture for Clusters (CAC)*, *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS),* Long Beach, California, USA, March, 2007.

[67] M.J. Rashti and A. Afsahi, "A Speculative and Adaptive MPI Rendezvous Protocol over RDMA-enabled Interconnects", *International Journal of Parallel Programming*, 37(2): 223-246, April, 2009.

[68] RDMA Consortium. iWARP protocol specification, http://www.rdmaconsortium.org/

[69] H. Ritzdorf and J.L. Traff, "Collective Operations in NEC's High-performance MPI Libraries," *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS)*, April, 2006.

[70] D. Roweth and D. Addison, "Optimized Gather Collectives on QsNetII," *Proc. of EuroPVM/MPI*, pages 407-414, 2005.

[71] D. Roweth, A. Pittman, and J. Beecroft, "Performance of All-to-all on QsNetII," Quadrics White Paper, 2005, http://www.quadrics.com/.

[72] H. Shan, J.P. Singh, L. Oliker, and R. Biswas, "Message Passing and Shared Address Space Parallelism on an SMP Cluster," *Parallel Computing*, 29(2): 167–186, 2003.

[73] SHMEM: Shared Memory Access, Cray Man Page Collection: S-2383-23, Available: http://docs.cray.com/.

[74] S. Sistare, R. vandeVaart, and E. Loh, "Optimization of MPI Collectives on Clusters of Large-scale SMPs," *Proc. of 1999 ACM/IEEE Conf. on Supercomputing (SC),* 1999.

[75] M. Small and X. Yuan, "Maximizing MPI Point-to-Point Communication Performance on RDMA-enabled Clusters with Customized Protocols," *Proc. of 23rd International Conference on Supercomputing (ICS)*, 2009

[76] S. Sur, U.K.R. Bondhugula, A. Mamidala, H.-W. Jin, and D.K. Panda, "High Performance RDMA based All-to-all Broadcast for InfiniBand Clusters," *Proc. of 12th International Conference on High Performance Computing (HiPC)*, 2005.

[77] S. Sur, H. Jin, L. Chai, and D.K. Panda, "RDMA Read based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits," *Proc. of 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 32-39, 2006.

[78] S. Sur, H.-W. Jin, and D.K. Panda, "Efficient and Scalable All-to-all Personalized Exchange for InfiniBand Clusters," *Proc. of 33rd International Conf. on Parallel Processing (ICPP)*, pages 275-282, 2004.

[79] S. Sur, M. Koop, L. Chai, and D.K. Panda, "Performance Analysis and Evaluation of Mellanox ConnectX InfiniBand Architecture with Multi-core Platforms," *Proc. of 15th IEEE International Symposium on Hot Interconnects (HotI)*, Palo Alto, CA, 2007.

[80] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, 19(1): 49-66, 2005.

[81] The GM and MX API. Available: http://www.myri.com.

[82] The OpenFabrics Alliance, http://www.openfabrics.org.

[83] The X10 Programming Language, http://www.research.ibm.com/x10/.

[84] V. Tipparaju and J. Nieplocha," Optimizing All-to-all Collective Communication by Exploiting Concurrency in Modern Networks," *Proc. of 2005 ACM/IEEE Conf. on Supercomputing (SC)*, 2005.

[85] V. Tipparaju, J. Nieplocha and D.K. Panda, "Fast Collective Operations using Shared and Remote Memory Access Protocols on Clusters," *Proc. of 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003.

[86] Top 500 Supercomputing Sites, http://www.top500.org/

[87] J.L. Traff, "Efficient Allgather for Regular SMP-clusters", *Proc. of EuroPVM/MPI*, pages 58-65, 2006.

[88] S.S. Vadhiyar, G.E.Fagg and J. Dongarra, "Automatically Tuned Collective Communications," *Proc. of 2000 ACM/IEEE Conf. on Supercomputing (SC)*, 2000.

[89] J.S. Vetter and F. Mueller, "Communication Characteristics of Large-scale Scientific Applications for Contemporary Cluster Architectures," *Journal of Parallel and Distributed Computing, 63*(9): 853-865, 2003.

[90] Virtual Protocol Interconnect (VPI), Whitepaper, Mellanox Technologies, Inc. http://www.mellanox.com/pdf/prod_architecture/Virtual_Protocol_Interconnect_VPI.pdf

[91] M. Wu, R.A. Kendall and K. Wright, "Optimizing Collective Communications on SMP Clusters," Proc. of 34th International Conference on Parallel Processing (ICPP), pages 399- 407, 2005.

[92] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li, "Expericences with VI communication for Database Storage." *Proc. of International Symposium on Computer Architecture (ISCA)*, 2002.