High-Performance Interconnects and Computing Systems: Quantitative Studies

By

Ying Qian

A thesis submitted to the Department of
Electrical and Computer Engineering
in conformity with the requirements for
the degree of Master of Science (Engineering)

Queen's University
Kingston, Ontario, Canada
May, 2004

ABSTRACT

As symmetric multiprocessors become commonplace, there are a number of factors affecting the performance of parallel applications, including the application characteristics, parallel programming paradigms used by the applications, and the machine system's architecture. In recent years, with the introduction of high speed networks, the high performance computing community has seen a trend to use network-based computing systems such as cluster of symmetric multiprocessors. In such systems, the communication subsystems become another crucial factor which affects the application performance.

In this thesis, the communication characteristics of one widely used parallel benchmarks, NAS parallel benchmarks written in MPI, are studied for different class sizes B, and C, and the newly released class D. Moreover, the performance of three different implementations of the NAS benchmarks in MPI, OpenMP, and Java, is compared on a small 4-way SMP (Dell PowerEdge 6650) and on a large 72-way SMP (Sun fire 15K server). The memory bandwidth, MPI communication latency and bandwidth are provided on these two SMPs, as well. Our results indicate that the performance of applications is affected by their characteristics. The new class D has much more communication than class B and class C, larger message size and larger number of messages. On both SMPs, the MPI version has better performance than the OpenMP and the Java.

Two interconnects, the Sun Fire Link Interconnect and the Myrinet, are studied in this thesis. The Sun Fire Link is a memory-based interconnect, where Sun MPI uses the Remote Shared Memory (RSM) model for its user-level inter-node messaging protocol. I give an overview of the Sun Fire Link, RSM, and the Sun MPI implantation on top of RSM. I provide an in-depth performance evaluation of the Sun Fire Link interconnect cluster of four Sun Fire 6800s at the RSM layer, and at the micro benchmark level. Our results include the performance of the Remote Shared Memory API primitives, MPI overhead on top of the RSM, latency and bandwidth under different communication modes, parameters of the LogP

model, collective communications, and different permutation communications. I also provide the performance of a Myrinet cluster with eight 2-way SMP nodes (Dell PowerEdge 2650), at the micro-benchmark level and application level. The Sun Fire Link and Myrinet achieve 5 μs and 6 μs latency, along with 695 MB/s and 444 MB/s bandwidth, respectively. In general, they both perform relatively well in most cases.

Acknowledgements

I would like to thank the guidance and support of my supervisor Professor Ahmad Afsahi. Without him, this project would have never been possible. I would like to acknowledge the financial support from the Ontario Graduate Scholarship for Science and Technology (OGSST). I am indebted to the Queen's University for awarding me as a Teaching Assistant and Dr. Afsahi for supporting me as a Research Assistant.

I would like to thank my friends at the Parallel Processing Research Laboratory, Nathan R. Fredrickson and Reza Zamani for their great help. I also would like to thank Dr. Ken Edgecombe, and Dr. Hartmut Schmider at High Performance Computing Virtual Laboratory at the Queen's University, and Mr. Gary Braida at the Sun Microsystems for their kind help in accessing the Sun Fire cluster with its Sun Fire Link interconnect.

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

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Figure **Page**

Glossary of Symbols and Abbreviations

ADI	Abstract Device Interface
BT	Block Tridiagonal
CC-NUMA	Cache-Coherent Non-Uniform Memory Access
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient
COW	Cluster of Workstations
DSM	Distributed Shared Memory
EP	Embarrassingly Parallel
FFT	Fast-Fourier Transform
FT	3-D Fast-Fourier Transform
HCA	Host Channel Adapters
HPF	High Performance Fortran
IS	Integer Sort
LU	Lower-upper Diagonal
MIMD	Multiple-Instruction Streams Multiple-Data Streams
MISD	Multiple-Instruction Streams Single-Data Stream
MG	Multigrid
MPI	Message Passing Interface
MPP	Massively Parallel Processors
NIC	Network Interface Cards
NPB	NAS Parallel Benchmarks
PDE	Partial Differential Equation
RDMA	Remote Direct Memory Access
RSM	Remote Shared Memory
RSMAPI	Remote Shared Memory Application Programming Interface

SAN	System Area Networks
SIMD	Single-Instruction Stream Multiple-Data Streams
SISD	Single-Instruction Stream Single-Data Stream
SMP	Symmetric Multiprocessor
SP	Scalar Pentadiagonal
SSOR	symmetric successive over-relaxation
TCA	Target Channel Adapters
VCSEL	Vertical Cavity Surface Emitting Laser
VI	Virtual Interface
VIA	Virtual Interface Architecture

Chapter 1

Introduction

1.1 Motivation

In late 80s and early 90s, several parallel machines with different architectures appeared. They include *Symmetric Multiprocessors* (SMP) such as Cray T-90, *massively parallel processor* (MPP) systems such as Cray T3D, Intel Paragon, and Thinking Machines CM-5, and *Distributed Shared Memory* (DSM) multiprocessors such as Stanford DASH and SGI Origin 2000. Considerable work has gone into the design of SMP systems, and several vendors such as IBM, Sun, Compaq, SGI, and HP offer small to large scale shared-memory systems [16]. Recently, *network of workstations* (NOW) and *cluster of multiprocessors* (CLUMPs) have been proposed as viable platforms for high performance computing. SMPs are the backbone of such high-performance cluster computing systems.

Parallel machines are being built to satisfy the increasing demand of higher performance for parallel applications. The parallel applications can be written using a variety of parallel programming paradigms, including message passing, shared memory, data parallel, bulk synchronous, and mixed-mode. The message passing and shared memory paradigms are the two most important programming paradigms. In the message passing paradigm, data transfer is done using explicit communications through send and receive calls. Collective communications and synchronization are also supported. The shared memory paradigm is originally designed for shared memory systems, although researchers are investigating to extend it to cluster of SMPs too [33]. In the shared memory paradigm, the threads running on separate processors, can communicate with each other by writing data to the shared memory

and then reading from it. *Message Passing interface* (MPI) [39] and OpenMP [43] are the de facto standards for these two paradigms. However, it is open to debate which parallel programming paradigm is the programming of choice for high performance [9][24]. It is really interesting to compare their performance on different systems.

To measure and to predict the performance of parallel computer systems, parallel benchmarks are designed. A benchmark is a performance testing program that captures processing and data movement characteristics of a class of applications [21]. A benchmark suite is a set of benchmark programs together with a set of specific rules. NAS parallel benchmark suite [42] is one of the most popular parallel benchmarks, which consists of eight benchmarks, each having different communication characteristic. It has several implementations, written in MPI [5], OpenMP [23], Java [18] and High Performance Fortran [17], respectively.

There are a number of factors affecting the performance of parallel applications on SMP systems. These include the applications' characteristics, the choice of parallel programming paradigms used by the applications, and the machine system's architecture. Understanding the applications' characteristics will give us insights to design better high performance computing systems in the future. It will also provide us with reasons why some applications perform better or worse on a specific system. In this thesis, I am interested in the communication characteristics of the NAS parallel benchmark suite, along with its performance on different SMPs.

In recent years, with the introduction of high speed networks, the high performance computing community has seen a trend to use network-based computing systems such as *network of workstations* (NOW) and *cluster of multiprocessors* (CLUMPs), to achieve high performance. The parallel applications developed for these computing systems require intensive co-operations between the processors. Therefore, the communication subsystem becomes a crucial factor which may affect the application performance. The network interconnects, the communication protocols and the messaging middleware form some of the important components of the communication subsystem.

Currently there are several high performance interconnects that provide low latency and high bandwidth. Three of the most famous products are *Myrinet* [7], *InfiniBand* [38] and *Quadrics* [30], using the user-level messaging layers *GM* [50], *VAPI* [12][28] and *Elan3lib* [30], respectively. Recently, Sun Microsystems has introduced the *Sun Fire link interconnect* [31][34][46] to provide ultra-high bandwidth needed to fuse a collection of large SMP servers into a cluster. *Remote Shared Memory* (RSM) [45] provides the inter-node user-level communications over the Sun Fire Link interconnect.

Usually, there are two messaging layers between these network interconnects and the applications: user-level messaging layer and the message passing layer. The messaging layers mentioned above provide protected user-level access to the network interface. Kernel-based protocols like TCP/IP are not capable of effectively utilizing the performance provided by the network interconnects, because every data transfer involves operating system intervention. On the contrary, 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. It is also crucial to provide an efficient implementation of message passing interface on top of the user-level protocol. In this thesis, the performance of the user-level messaging layer, message passing layer, and the application layer are provided. With these results, one can discover how each layer performs and how well each layer is implemented on top of the lower layer.

In this thesis, I am interested in evaluating the performance of single SMPs, as well as the cluster of SMPs. To design better architectures in the future, it is important to discover how the communication subsystems, parallel programming paradigms, and the application characteristics may affect the performance. I am interested in measuring the performance of several computing platforms. Our platforms consist of a 72-way SMP node from Sun Microsystems (Sun Fire 15K), a 4-way Intel Xeon MP from Dell (Dell PowerEdge 6650), a cluster of four 24-way SMP (Sun Fire 6800) interconnected by the Sun Fire Link interconnect, and a cluster of eight 2-way Intel Xeon MP from Dell (Dell PowerEdge 2650) interconnected

by Myrinet. The software environment, including the operating system, compiler and the MPI version, are shown in Table 1.1.

Table 1.1 Software environments of computing systems.

	Dell PowerEdge 6650 4-way SMP	Sun Fire 15K 72-way SMP	Sun cluster 4x24 Sun Fire 6800	Myrinet cluster 8x2 Dell PowerEdge 2650
OS	Linux, Redhat 9	Solaris 9	Solaris 9	Linux, Redhat 9
Compiler	Intel compiler 7.1	Sun One Studio 7	Sun One Studio 7	gcc
MPI	MPICH-1.2.5 ch_shmem	Sun MPI 6	Sun MPI 6	MPICH-1.2.5..10 (MPICH-GM)

1.2 Contributions

This thesis discusses a number of issues that are the contributing factors affecting the performance of the parallel computing systems. This thesis makes four major contributions.

- I obtain the communication characteristics of five NAS benchmarks written in MPI, including a newly released class D, which has not been characterized before. This thesis compares the communication patterns of the applications running under different number of processes and different problem sizes: class B, class C, and class D. The communication characteristics include the message size and the number of messages, and the distribution of the message destinations.
- For the two SMP platforms, the memory bandwidth, and the performance of a set of micro-benchmark suite implemented on top of MPI are presented. With these results and the communication characteristics of the NAS parallel benchmarks, I explain the performance of these parallel applications. This thesis also compares the performance of the NAS benchmarks under different parallel programming paradigm, including MPI, OpenMP and Java.
- This thesis studies the newly released user-level protocol “RSM” for the Sun Fire Link interconnect. The performance of RSM *Application Programming Interface* (API) calls is provided. This will help in determining the mechanisms that should be

used at a high-level (the MPI level) to achieve performance. I also look at how the Sun MPI is implemented on top of the RSM protocol.

- This thesis presents a framework to evaluate the performance of communication subsystems of two clusters. I provide a set of micro-benchmarks implemented on top of MPI to evaluate the performance of communications seen by the applications. The micro-benchmarks include traditional point-to-point latency, bandwidth, bandwidth under load, LogP parameters, permutation traffic patterns, and collective communications. These, along with the performance of the low level protocol, can be used to determine what percentage of the performance at the lower layer is delivered to the MPI level. These micro-benchmarks are also used to assess the quality of the given MPI implementation as well. Finally, I measure the performance of the NAS parallel benchmarks on the Myrinet cluster.

1.3 Outline of Thesis

In this thesis, I characterize one popular parallel benchmark suite, NAS parallel benchmarks, and evaluate its performance on a small SMP, a large SMP and an SMP cluster. I also evaluate the performance of two recently introduced high performance interconnects, the Sun Fire Link and the Myrinet, at the user level (just for the Sun Fire Link), and at the micro-benchmark level.

In chapter 2, I provide the background of this thesis. I take a look at the popular high performance architectures, high performance interconnects, user-level protocols, and different parallel programming paradigms. In chapter 3, the NAS parallel benchmarks are introduced, along with their communication characteristics. The performance of MPI micro-benchmarks and the NAS benchmarks on a small SMP and a large SMP is compared in chapter 4. In chapter 5, I introduce the Remote Shared Memory (RSM) model in detail, along with the performance of some RSM API calls. The implementation of SunMPI over RSM is also discussed in this chapter. In chapter 6, the performance of the Sun Fire Link and Myrinet is evaluated by several micro-benchmarks, including latency, bandwidth, aggregate

bandwidth, different traffic patterns and collective communications. I also provide the performance of application benchmarks on the Myrinet cluster. Finally, I conclude the thesis and provide some directions for future work in chapter 7.

Chapter 2

Background

In the past decade, high performance computers have been implemented using a variety of architectures: *Massively Parallel Processors* (MPP), *Symmetric Multiprocessors* (SMP), *Distributed Shared Memory* (DSM) multiprocessors, and Clusters. The current trend in high performance computing is for hybrid architectures, such as *networks of workstations* (NOW) and *clusters of multiprocessors* (CLUMPs). In section 2.1, I briefly describe the most common parallel computer architectures. Not to mention, our focus in this thesis is on SMPs, and CLUMPs, as they are and will remain the trends for years to come. In section 2.2 through section 2.5, I will discuss the different components of a high performance computing system including the nodes, the interconnects, the massaging layers, and the parallel programming paradigms.

2.1 Parallel Computer Architectures

Based on the Flynn's classification [21], there are four kinds of machine architectures, *single-instruction stream single-data stream* (SISD), *single-instruction stream multiple-data streams* (SIMD), *multiple-instruction streams single-data stream* (MISD) and *multiple-instruction streams multiple-data streams* (MIMD). SISD models conventional sequential computers. MISD was seldom used. In an SIMD machine, all processors execute the same instruction at the same time. So it is a synchronous machine, and mostly used for

special purpose applications. An MIMD machine is a general-purpose machine, where processors operate in parallel but asynchronously.

MIMD machines are generally classified into four practical machine models: *Symmetric Multiprocessors* (SMP), *Massively Parallel Processors* (MPP), *Distributed Shared Memory* (DSM) multiprocessors, *Cluster of Workstations* (COW), and *Cluster of Multiprocessors* (CLUMP), as shown in Figure 2.1.

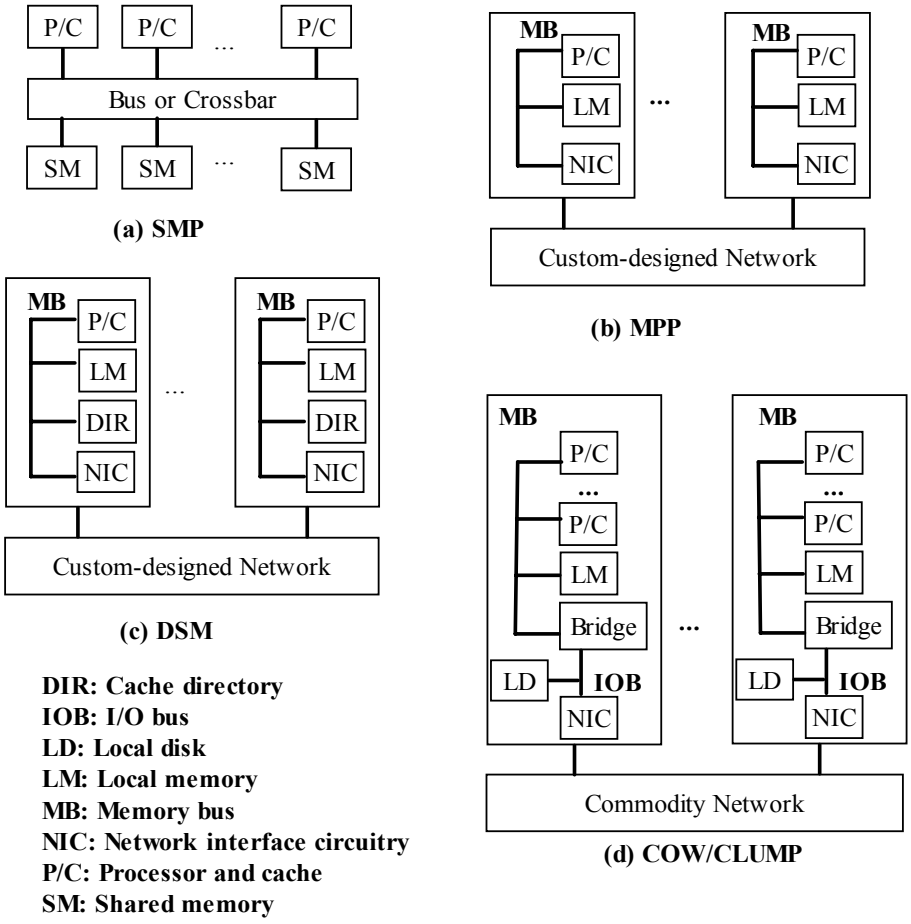


Figure 2.1. Parallel computer models: (a) SMP (b) MPP (c) DSM (d) COW/CLUMP.

SMP is a *Uniform Memory Access* (UMA) system, where all memory locations are the same distance away from the processors, so it takes roughly the same amount of time to access any memory location. SMP systems have gained prominence in the market place. Considerable work has gone into the design of SMP systems, and several vendors such as IBM, Sun, Compaq, SGI, and HP offer small to large-scale shared memory systems [16].

MPP, DSM multiprocessors, COW, and CLUMP and are distributed-memory systems, where there are multiple nodes each having one or more processors and its own local memory. For MPP, COW and CLUMP systems, one node's local memory is considered remote memory for other nodes. DSM machines use cache directory protocols to implement coherent caches. MPP, CLUMP and COW machines do not have cache directory, and processes communicate by exchanging messages. MPP machines consist of a number of nodes interconnected by a high-speed custom-designed network. SMPs are called *tightly coupled* [21]. COW and CLUMP machines are low-cost variation of MPP machines, which use low-cost commodity networks. 41.9% of the top 500 supercomputers in the world are clusters [49].

2.2 Computing Nodes

MPP, COW, CLUMP and DSM multiprocessors each contain multiple nodes, which are connected by custom-designed or commodity networks. Each node can be a uni-processor, an SMP, or a *Simultaneous Multi-Threading* (SMT) system. SMP was introduced in section 2.1. The SMPs that I will explore in this thesis include Dell PowerEdge 2560, Dell PowerEdge 6650, Sun Fire 6800 and Sun Fire 15K server. Two-way Dell PowerEdge 2560 and four-way Dell PowerEdge 6650 use Intel Xeon MP Processors. Sun Fire 6800 and Sun Fire 15K servers have 24 and 72 Sun Ultrasparc III cu processors, respectively. SMT is the technology that allows a single physical processor to execute multiple threads concurrently in hardware. Intel implemented SMT with Hyper-Threading, which is used on the Intel Xeon MP Processor. Table 2.1 shows some popular processors used in high performance computers.

Table 2.1. Processor and the system using them.

Processor	System
IBM PowerPC	IBM SP cluster
Intel Xeon MP	Dell PowerEdge
Sun Ultrasparc IV cu	Sun Fire server
Intel Itanium2	HP Integrity Server
Compaq Alpha	Compaq AlphaServer

2.3 High-Performance Interconnects

To have high-performance computer systems, the interconnect that connects the nodes of the system plays a crucial role. Currently, there are several high performance interconnects that provide low latency (less than 10 us) and high bandwidth, such as Myrinet [7], InfiniBand [38], and Quadrics [30]. Recently, Sun Microsystems has introduced Sun Fire link [46] to provide ultra-high bandwidth needed to fuse a collection of large SMP servers into a cluster.

2.3.1 Myrinet

Myrinet was developed by Myricom [7] based on packet-switching technology, which was originally designed for Massively Parallel Processor systems [7]. The packets are wormhole-routed through a network consisting of switching elements and host interfaces. The core of the switch is a pipelined crossbar. The programmable Myrinet network interface cards provide much flexibility in designing communication software.

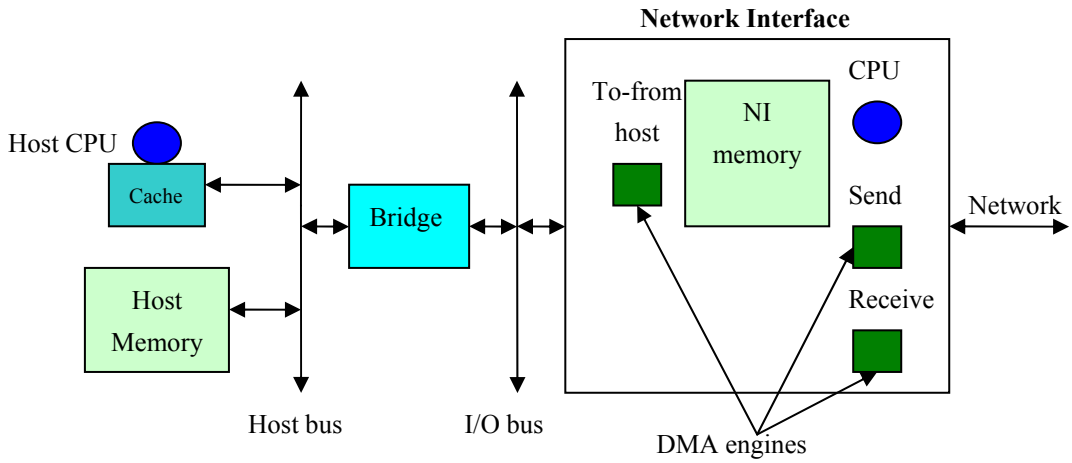


Figure 2.2. The host and network interface architecture of Myrinet

Figure 2.2 illustrates the architecture of a node in a Myrinet network system. Each host has a *Network Interface Card* (NIC) that contains a processor and some memory, which is used to store the control program and data. The NIC connects to the host's I/O bus. The M3F2-PCIXE-2 "**E card**" Myrinet/PCI-X interface has been released recently. The "E card" has a 64-bit, 133MHz PCI-X interface, and has a programmable Lanai-2Xp RISC processor

operating at 333MHz with 2MB local memory. Each port is 2.0+2.0 Gbps data rate. The standard firmware has two ports working at same time, which acts as a 4.0+4.0 Gbps data-rate port.

2.3.2 Quadrics

Quadrics networks (QsNet) [30] is based on two building blocks, a programmable network interface called *Elan* and a low-latency high-bandwidth communication switch called *Elite* [30]. The newly released QM500 PCI-X network adapter for Quadrics QsNet II [1], uses Quadrics Elan 4 network processor, and is connected to the hosts via 64bit, 133MHz PCI-X Bus. It provides full duplex 900Mbytes/s peak bandwidth at each direction. It has 64Mbytes onboard DDR-SDRAM memory. Quadrics switch uses a full crossbar connection and supports wormhole routing. The performance of Quadrics is provided in [30].

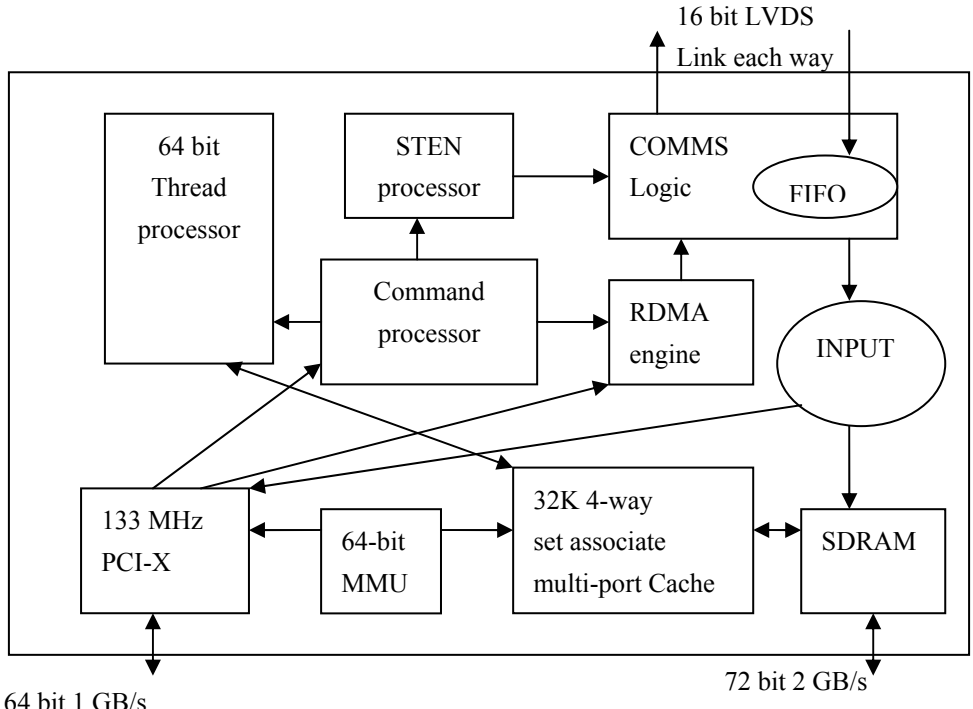


Figure 2.3. Structure of Elan4.

QsNet II provides efficient and protected access to a global virtual memory using *remote direct memory access* (RDMA) operations. The Elan4 chip contains the following major logic blocks, as outlined in Figure 2.3. The 64-bit multi-threaded control processor with independent hardware state machines controls pipelined output DMA issue, input transaction

processing, synchronization processing, the scheduling of the thread processor, and the command queue processing. It also generates output packets issued directly over the PCI-X interface by the main processor. MMU is used to translate 64-bit virtual addresses into either local SDRAM physical addresses or 64-bit physical addresses for PCI-X master addresses. A 64-bit Thread Processor helps to implement high-level messaging libraries without explicit intervention from the main CPU. The short message processing unit is called STEN (Small Transaction Engine).

2.3.3 Sun Fire Link

Sun Fire Link is a high-bandwidth, low-latency interconnect recently introduced to cluster Sun Fire 6800 and 15K/12K systems [13][46]. The system's interface to the Sun Fire Link network is provided by a Sun Fire Link specific I/O subsystem which is called the *Sun Fire Link assembly*. Each Sun Fire Link assembly contains two optical transceiver modules called Sun Fire Link optical modules. Each optical module supports a full-duplex optical link. The transmitter uses a Vertical Cavity Surface Emitting Laser (VCSEL), and has a 1.65 GB/s raw bandwidth, and a theoretical 1 GB/s sustained bandwidth after protocol handling [34]. The Sun Fire Link assembly is installed in pair. Each pair is called a computer domain of the system, which means that each compute domain contains four optical link connections to the Sun Fire Link network. A Sun Fire 6800 server can have one compute domain, while a Sun Fire 15K/12K server can have up to four compute domains, with a maximum count of 16 optical links connected to the network.

A Sun Fire cluster can have different network structure depending on the type of topology used: *direct connect* or *switched*. The switches are not needed when the Sun Fire cluster has two or three domains. The optical cables connect directly to the servers. Figure 2.4 shows how two domains connect to each other, and Figure 2.5 shows how three domains connect to each other.

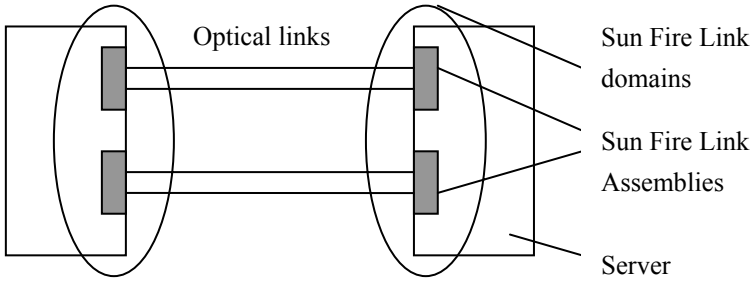


Figure 2.4. Two domains direct connect configuration.

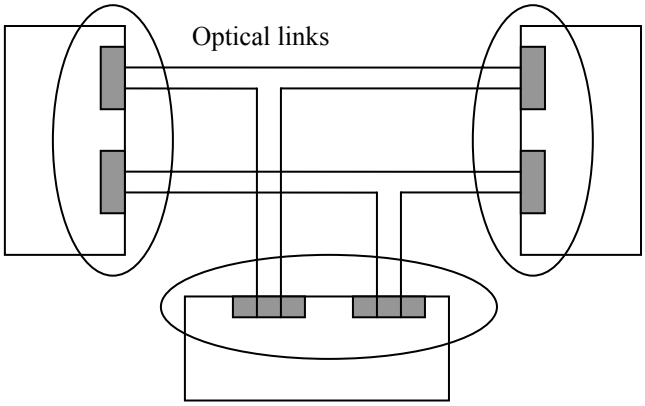


Figure 2.5. Three domains direct connect configuration.

For more than four domains, switches are needed. The Sun Fire Link switch is an eight-port optical switch, each of which handles one optical network link. So the current Sun Fire switch supports only up to eight hosts, while theoretically, the Sun Fire Link can support up to 254 hosts. There are two standard *switched* configurations. One can have up to four domains and two Sun Fire Link switches. Figure 2.6 shows this configuration. The other can have up to eight domains and four switches, which is shown in Figure 2.7.

The network interface does not have a DMA engine. It can initiate interrupts as well as do polling for data transfer operations. It provides uncached read and write accesses to remote memory regions on the other nodes. Layered system software components implements a mechanism for user-level messaging based on direct access to remote memory regions of other nodes [2]. This is referred to as *Remote Shared Memory* (RSM) [45]. Nodes can communicate through a TCP/IP network for cluster administration issues, and exchanging control and

status/error information. Sun MPI is a complete library of message-passing routines, based on RSMAPI. Details will be given in chapter 5.

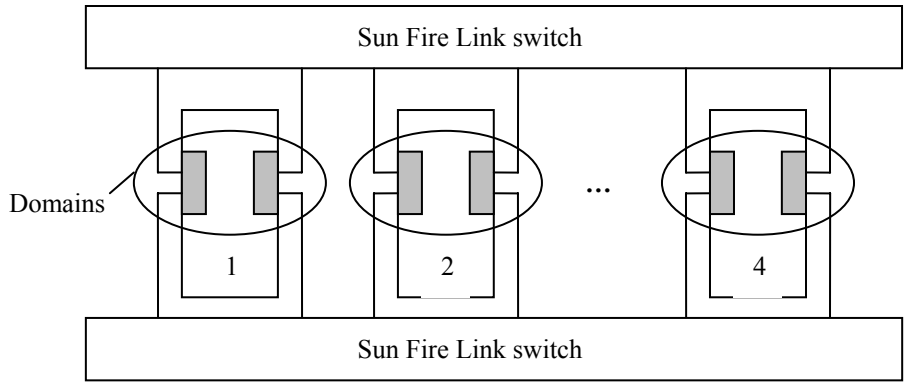


Figure 2.6. Four domains and 1 switch configuration.

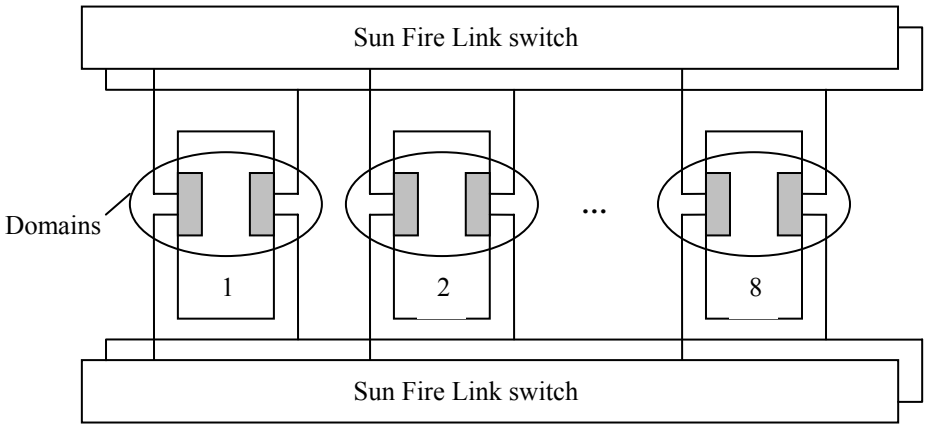


Figure 2.7. Eight domains and 4 switches configuration.

2.3.4 InfiniBand

The InfiniBand Architecture [38] is a packet switched network, initially proposed as a generic interconnect for inter-process communication and I/O. In the InfiniBand network, nodes are connected to the fabric by *Host Channel Adapters* (HCAs) and *Target Channel Adapters* (TCAs). 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 bus. They also contain an interface to the memory and hardware engines, which provides virtual to physical address translations and memory protection [38].

The fundamental concept of the channel interface is the *queue pair* (QP) [38] which serves as a virtual communication port. Each QP has two queues: a send queue and a receive queue. The completion of communication requests is reported through c*ompletion queues* (CQ).

2.4 User-Level Protocols

TCP/IP, a very popular kernel-based communication protocol, incurs in performance penalties, which is unbearable in *System Area Networks* (SAN) due to its layered structure [6]. SAN is a local network designed for high-speed interconnection in cluster environments (server to server), multiprocessing systems and storage area networks. TCP/IP stack is generally built into the operating system kernel, so every data transfer involves operating system intervention. Data copying in TCP/IP layers (from kernel space to user space or vice versa) causes performance degradation.

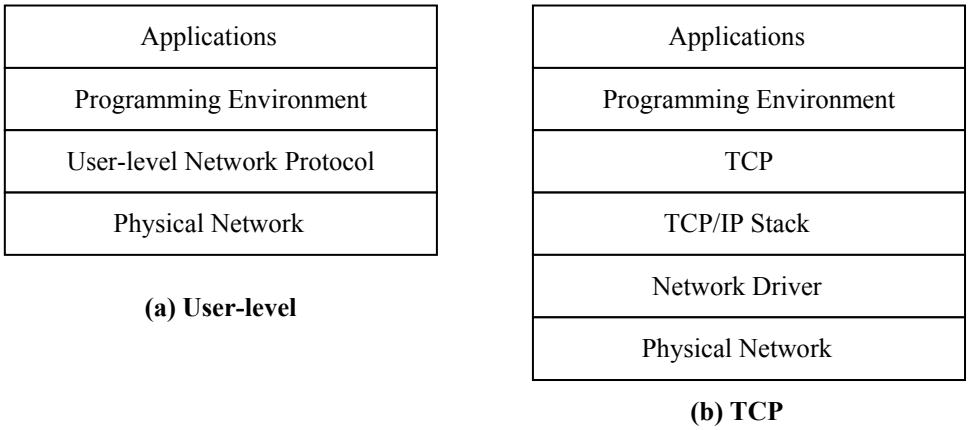


Figure 2.8. Layers of abstraction from Network to Applications.

To provide low latency, the user-level network protocols move some of the services normally provided by kernel into the user level. Bypassing the operating system, the user-level protocols avoid the costs associated with switching to the privileged mode. The layers of abstraction of TCP and user-level protocols are shown in Figure 2.8. In the following, we briefly discuss some important user-level protocols.

2.4.1 GM

GM [50] is a commercial open source user-level networking protocol from Myricom Corporation, which runs on top of the Myrinet network. GM provides a protected interface to the network interface cards so that multiple user applications can share the NIC simultaneously. GM supports both send/receive and RDMA operations, and its performance is provided in [20][28]. User buffers need to be registered and pinned down in the physical memory to enable DMA transfer in and out of these memory regions.

The GM communication system provides reliable and ordered delivery between the communication endpoints, called *ports*. For the send/receive model, ports need to be opened before any communications, by calling the *gm_open* function. All the buffers used by send and receive must be registered using *gm_dma_calloc*. As shown in Figure 2.9, the send side may send a message by calling a GM API send function, *gm_send_with_callback*. When the send completes, GM calls the *callback* function, and waits for the receiving event to indicate if the send has been completed successfully.

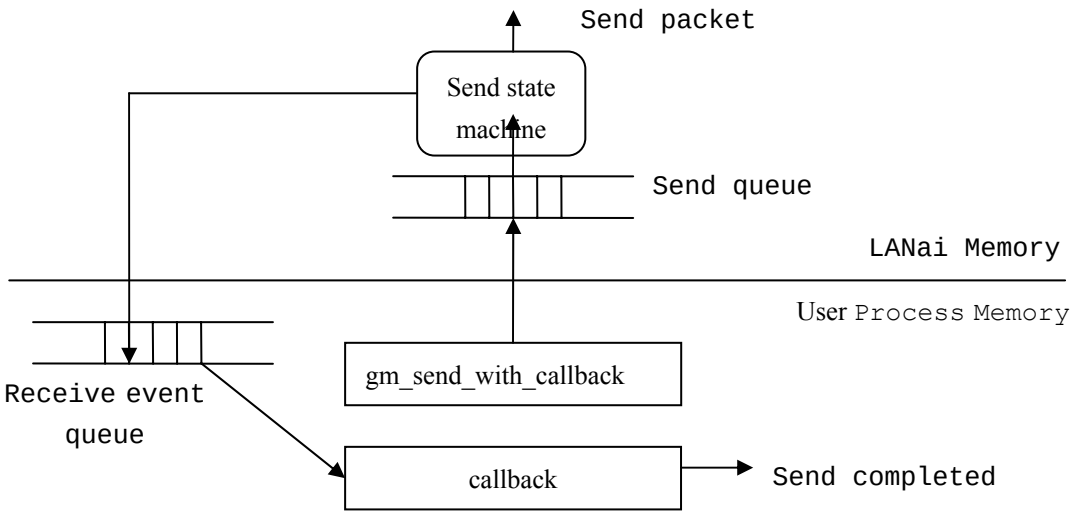


Figure 2.9. Send side flow chart.

In the receive side, opening the port and registering the buffers are also needed to be done at the beginning. API call *gm_allow_remote_memory_access* allows these local buffers

to be modified by any other GM process. Then the receive side waits for events that indicate the incoming messages. Figure 2.10 shows the major steps at the receive side.

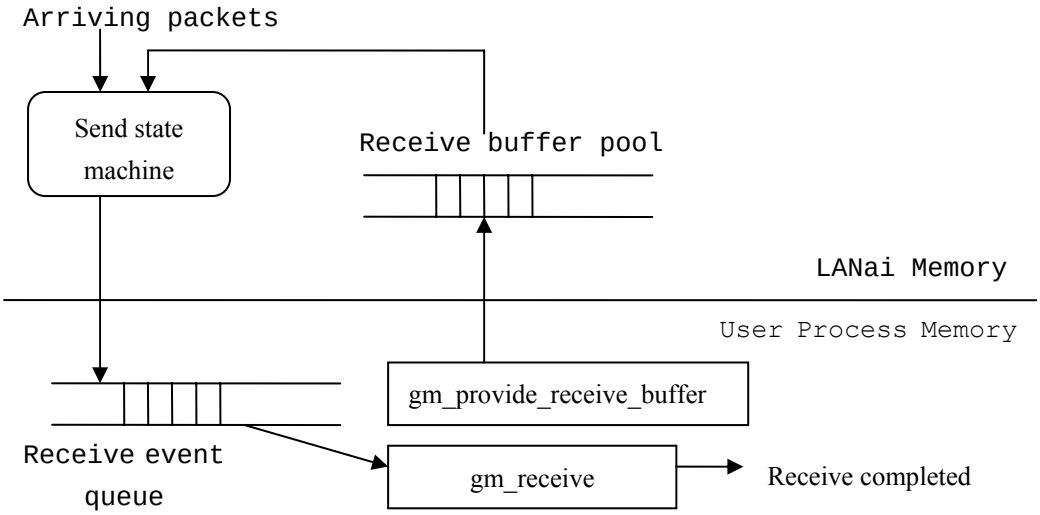


Figure 2.10. Receive side flow chart.

2.4.2 Elan3lib and Elanlib

Figure 2.11 illustrates the Elan programming libraries [30]. Elan3lib [30] provides the lowest-level, user space programming interface to the Elan3 [30] network. At this level communications between processes can be done though an abstraction of a distributed, virtual shared memory. Elanlib is a higher-level machine independent communication library to provide low level accesses [30]. It provides a global virtual address space by integrating the address spaces of individual nodes. One node can use RDMA to access a remote node's memory [28]. A general-purpose synchronization mechanism based on events stored in memory is provided so that the completion of RDMA operations can be reported. It also provides basic mechanism for point-to-point message passing, called *tagged message ports* (Tports). Unlike GM and VAPI, the QsNet does not require the communication buffers to be registered.

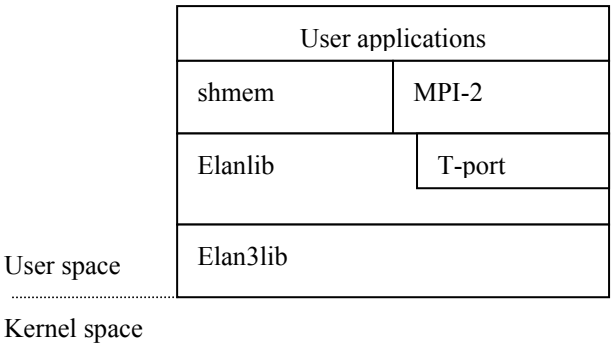


Figure 2.11. Elan programming libraries.

2.4.3 Remote Shared Memory

Remote Shared Memory (RSM) [45] is designed for the Sun Fire link interconnect, to provide low latency and high bandwidth communications. It is a high-performance memory-based mechanism, which implements user-level inter-node messaging with direct access to memory that is resident on remote nodes. To establish the communications, an application process creates an RSM export segment from the process' local address space. One or more remote application processes create an RSM import segment with a virtual connection between export and import segments across the interconnect. All processes make memory references for the shared segment with addresses local to their specific address spaces. After the RSM segment is published through one or more interconnect controllers, the segment is remotely accessible. RSM also provides a notification mechanism to synchronize local and remote accesses. An export process can call a function to block while an import process finishes a data write operation. When the import process finishes writing, the process unblocks the export process by calling a signal function. Once unblocked, the export process processes the data. The detailed API is studied in Chapter 5.

2.4.4 VAPI

Verb-Based API (VAPI) is the software interface for the InfiniBand. The interface is based on the InfiniBand verbs layer, which is an abstract description of functionalities of a HCA. Three communication operations are provided: send/receive, RDMA operations and Atomic.

The performance of send/receive and RDMA operations is shown in [12][28]. Both reliable connection and unreliable datagram services have been implemented on HCAs. Similar to the GM, memory buffers must be registered with HCA before being used. Existing designs of MPI over InfiniBand use send/receive operations for small data messages and control message, and RDMA operations for large data messages [12].

2.4.5 VIA

The Virtual Interface Architecture (VIA) [6][15] is designed to provide high bandwidth and low latency over a System Area Network, by providing a protected and directly accessible network interface called the Virtual Interface (VI). Two VI endpoints on different nodes can be connected by a bidirectional point-to-point communication channel. The virtual memory used by user communication buffers needs to be registered, so that these buffers can be accessed by network interface. The VIA specifies two types of data transfer models: the traditional send/receive messaging model and the Remote Direct Memory Access (RDMA) model.

2. 5 Parallel Programming Paradigms

Parallel computers provide support for a wide range of parallel programming paradigms. The HPC programmer has several choices for the parallel programming paradigm, including the message passing, shared memory, data parallel, bulk synchronous, and mix-mode.

Message Passing interface (MPI) [39], and OpenMP [43] are the de facto standards for message passing, and shared memory paradigms. The shared address space within each SMP node is suitable for OpenMP parallelization and POSIXThread [44]. Message passing can be employed within and across the nodes of a cluster. Programming with shared memory paradigm is generally easier but it is not highly scalable, while message passing is harder to program but it is more scalable. Data parallel paradigm is for the SIMD architecture, where a single control unit issues each instruction to the processing elements [32]. The *Bulk Synchronous Parallel* (BSP) model is a universal abstraction of parallel computation that can be used to design portable parallel programs [29]. The mix-mode programming is a

combination of message passing and shared memory paradigms. Which programming paradigm is better depends on the nature of the given problem and the architecture being used. In this section, MPI, OpenMP, Java, and other paradigms are introduced.

2.5.1 Message Passing

MPI [39] is a well known message passing environment. MPI has good portability, because programs written using MPI can run on distributed-memory multicomputers, shared-memory multiprocessors, and networks of workstations. On top of shared memory systems, message passing is implemented as writing to and reading from the shared memory. So MPI can be implemented very efficiently on top of the shared memory systems. Another 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. The disadvantage is that existing sequential applications require a fair amount of restructuring for parallelization based on MPI.

MPI provides the user with a programming model where processes communicate with each others by calling library routines. There are two kinds of communications in MPI, point-to-point and collective.

Point-to-point communication is the basic communication mechanism used to transmit data between a pair of processes in MPI, as shown in Figure 2.12. Point-to-point communications can be divided into blocking and non-blocking. A blocking procedure will not return until the user is allowed to reuse resources specified in the call. A non-blocking procedure may return before the operation completes. Blocking calls support four different modes: *standard* send and receive, *buffered*, *synchronize* and *ready*.

Standard – The completion of a send implies that the message either is buffered internally or has been received. So after the call returns, the user is free to overwrite the message.

Buffered – The user guarantees a certain amount of buffering space.

Synchronous – *Rendezvous* semantics is used between the sender and receiver. The sending process blocks until the corresponding receive has been posted.

Ready – A send operation can be started only after the matching receive is already posted. The *ready* mode is a way for the programmer to notify the system that the receive has been posted so that the underlying system can use faster protocol if it is available.

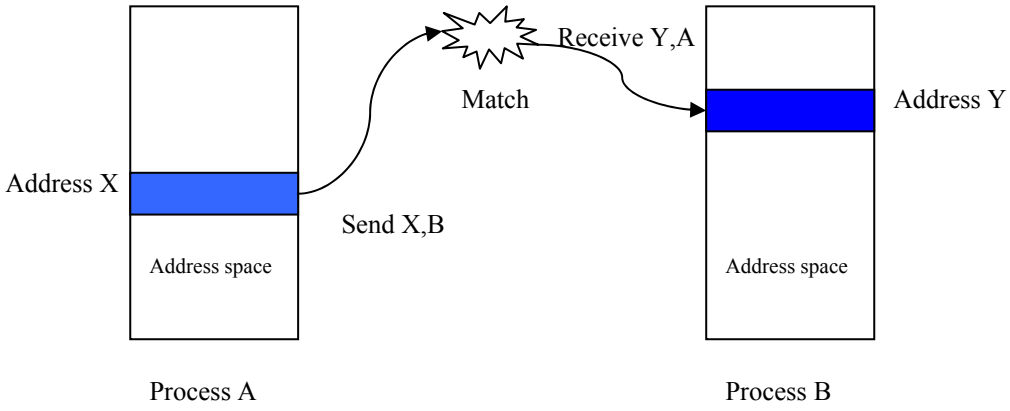


Figure 2.12. Standard send/receive model.

Collective communications transmit data among all processes in a group. *Barrier* operation synchronizes across all processes in the group. In a *Broadcast* operation, a process sends a unique message to all other processes of the group. In a *Gather* operation, each process sends a message to a specific process. *Scatter* operation is the inverse operation to the *gather* operation, where a process sends a different message to all processes in the group. *Alltoall* operation sends messages from all processes to all processes. *Reduce* operation gets the combined value from the messages received from all other process in the group, using the operation *op*.

2.5.1.1 Sun MPI

Sun MPI [48] is a complete library of message-passing routines, developed by Sun Microsystems. Figure 2.13 describes the architecture of Sun MPI. Sun MPI treats on-node and off-node communications differently. For on-node communications, shared memory protocol is used, while Remote Shared Memory [45] protocol is used for off-node

communications. A Remote Shared Memory application programming interface (RSMAPI) offers a set of user level function calls for remote memory operations [45].

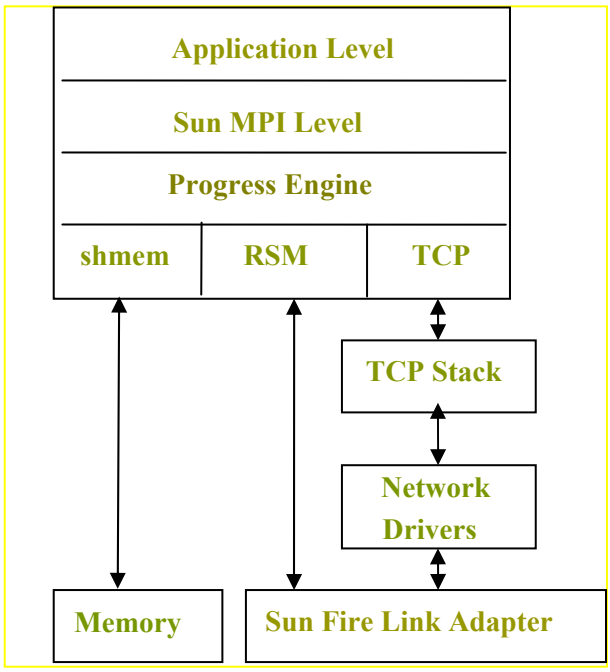


Figure 2.13. Sun MPI structure.

For on-node point-to-point message passing, the sender writes to shared-memory buffers, depositing pointers to these buffers into shared-memory postboxes [47]. After the sender finishes writing, the receiver can read the postboxes and the buffers. Figure 2.14 shows the different ways a message is sent based on its size. For small messages, instead of putting pointers to the buffers into postboxes, data itself is placed into the postboxes. For medium-size messages, one postbox is used pointing to the buffers with data. For large messages, which may be separated into several buffers, the reading and writing can be pipelined. For very large messages, to keep the message from overrunning the shared-memory area, the sender is allowed to advance only one postbox ahead of the receiver. Thus, the footprint of the message in shared memory is limited to at most two postboxes at any one time, along with associated buffers. Sun MPI uses *eager* protocol for small messages, where it writes the messages without explicitly coordinating with the receiver. For large

messages, it employs *rendezvous* protocol, where the receiver must explicitly notify the sender that it is ready to receive the message, before the message can be sent.

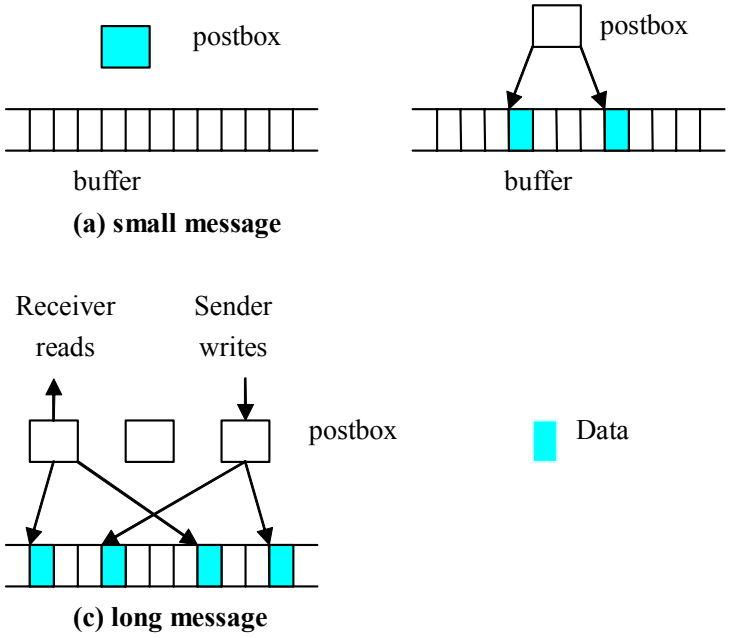


Figure 2.14. On-node messages. (a) small (b) medium-size (c) long.

For off-node communications, Sun MPI supports high performance message passing by means of the RSM protocol, which is running on top of the Sun Fire Link. Sun MPI over RSM achieves low latency bypassing the operating system, and high bandwidth from striping messages through multiple channels. Messages sent over RSM are in one of two fashions depending on the size of message [31][47]. Short messages are fit into multiple postboxes and no extra buffers are used. Pipelined messages are sent in 1024-byte buffers under the control of multiple postboxes.

Standard MPI communications are two-sided. To complete the transfer of information, both the sending and receiving side processes must call proper functions. This form of communication requires synchronization between the sending and receiving processes. Sun MPI supports one-sided communication designed to reduce the amount of synchronization required. In one-sided communication, a process opens a window in memory, and exposes it to some processes that it wants to communicate with, using a particular communicator. These processes must reside on the same node. As long as the window is open, any process in that

particular communicator and node can *put* (write) data into it and *get* (read) data out of it. *MPI_Put* and *MPI_Get* functions are used for the *put* and *get* operations.

Efficient implementation of collective communication algorithms is one of the keys to the performance of cluster computer systems. Sun MPI takes advantage of the symmetric multiprocessors' characteristics for efficient implementation of collective communication algorithms for on-node, and off-node communications in clusters of SMPs. For on-node collective communications, the optimized algorithms use the local exchange method instead of point-to-point approach. As stated earlier, on a single SMP node, any process may communicate with any other node via shared memory. Thus, the time to complete the operation is limited by the memory bandwidth. For off-node collective communications, one representative process for each SMP node is chosen [35]. This process is responsible for delivering the message to all other processes on the same node, which are involved in the collective operation.

2.5.1.2 MPICH

MPICH [40] is a portable implementation of the full MPI specification for a variety of parallel computing environments. MPICH is designed through implementation of an *abstract device interface* (ADI). Each implementation of ADI is called a *device*. Several devices are available, such as *ch_p4*, *ch_shmem* and *ch_gm*. The *ch_p4* device supports SMP nodes and heterogeneous collections of systems. The *ch_shmem* device is appropriate for an SMP. It uses shared memory to pass messages between processes. The *ch_gm* is used for the GM user-level protocol. There are four protocols in MPCH, namely *eager*, *rendezvous*, *short*, and *get*. In the *eager* protocol, the sender sends data to the receiver without request, while in the *rendezvous* protocol the sender sends the data only after the receiver notifies the sender that it can accept the message. In the *short* protocol the data was sent in to the control message envelop. In the *get* protocol the receiver just gets the data from the sender

2.5.2 OpenMP

Message-passing codes written in MPI are obviously portable and should transfer easily to SMP cluster systems. However, it is not immediately clear that message passing is the most efficient parallelization technique within an SMP box, where in theory a shared memory model such as OpenMP [43] should be preferable.

OpenMP is a loop level programming style. It is popular because it is easy to use and enables incremental development. Parallelizing a code includes two steps, (1) discover the parallel loop nests contributing significantly to the computations time; (2) add directives for starting/closing parallel regions, managing the parallel threads (workload distribution, synchronization), and managing the data.

OpenMP provides a fork-and-join execution model, as shown in Figure 2.15. A program begins execution as a single process or thread, until a parallelization directive for a parallel region is found. At this time, the thread creates a number of threads, and becomes the master thread of these threads. All threads execute the statements together until the end of the parallel region.

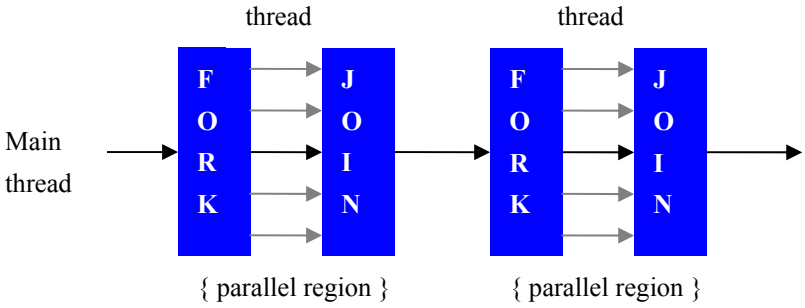


Figure 2.15. Fork-join model.

The advantage of OpenMP is that an existing code can be easily parallelized by placing OpenMP directives around the loops which do not contain data dependences. The disadvantage is that it may not scale very well with the number of processors. In [33], OpenMP is extended for the cluster of SMPs by “compiler-directed” distributed shared memory system.

2.5.3 JAVA

Java is a relatively new language for High Performance Computing. Although Java programs suffer from poor performance, running much slower than C and Fortran, it offers a number of benefits as a language for HPC, such as portability, software engineering, security and GUI development [36]. Java offers a higher level of platform independence to generate portable code which compiles and runs on a diverse range of platforms.

There are several parallel programming models of Java: Java threads, MPI-like API of Java [10] and an OpenMP-like API for Java (JOMP) [8]. It is possible to write shared memory parallel programs using Java's native threads model by running a single multi-threaded Java application. The Java thread class is part of the standard Java libraries. Most current virtual machines implement this class on top of the native OS threads allowing threads distributed across the processors. A thread is spawned by creating an instance of the *java.lang.thread* class, and has the methods to control the threads [36]. Currently, there are several Java MPI-like bindings available, which are generally implemented in two ways - as a wrapper around the existing native MPI library, or as a pure Java implementation. JOMP [8] is a prototype Java version of OpenMP, which provides a familiar parallelism model without the complexity of Java threads.

2.5.4 Other Parallel Programming Paradigms

In the mixed MPI-OpenMP programming style, each SMP node executes one MPI process that has multiple OpenMP threads. This kind of hybrid parallelization might be beneficial when it utilizes the high optimization of the shared memory model on each node. As small to large SMP clusters become more prominent, it is open to debate whether pure message-passing or mixed MPI-OpenMP is the programming of choice for higher performance. Previous works on small SMP clusters have shown contradictory results [9][19].

In Data Parallel paradigm, a single program controls the distribution of and operations on data distributed across all processors. The compiler is responsible for generating the codes to

distribute the array elements on the available processors. Data parallel applications may be run on MIMD and SIMD architectures [32]. *High Performance Fortran* (HPF) [32] is a programming language designed to support the data parallel programming style.

In the BSP model [29], the computation is divided into a sequence of supersteps. In each superstep, the processors perform some local computation, initiate communications to other processors, and synchronize at the end of each superstep.

2.6 Summary

General idea about parallel computer architectures is given in this chapter. I discussed the architecture of some popular systems, especially SMPs and CLUMPs, which are the trends now and will remain the trends for future. I discussed different components of such systems, including the processor, interconnect, user-level messaging layer and high-level parallel programming paradigms. In the following chapters, I will discuss the performance of such parallel computer systems. In chapter 3, I will introduce the application benchmarks.

Chapter 3

Application Benchmarks and Their Characteristics

To evaluate the performance of high-performance parallel computer systems, application benchmarks are developed. In this chapter, I describe the NAS parallel Benchmark (NPB) suite [42], used in this thesis along with the communication characteristics of its MPI versions. The architectural requirements and scalability of NAS parallel benchmarks (class A) was presented in [37]. The communication characteristics of some of the NAS benchmarks were studied in [5]. In [9], the NAS benchmarks were used to evaluate two communication libraries over the IBM SP machine. In this section, I include the characteristics of the newly released class D, as well as the results for classes B and C. I have gathered the communication traces under three system sizes of 16, 32, and 64 processes.

3.1 NAS Parallel Benchmarks

The NAS Parallel Benchmark suite has been developed at the NASA Ames Research Center [42] to help evaluate the performance of parallel supercomputers. The benchmark suite, which mimics the computation and data movement characteristics of large scale computational fluid dynamics (CFD) applications consists of eight programs, including five kernels and three pseudo-applications. Namely, the three simulated CFD application benchmarks are *block tridiagonal* (BT), *lower-upper diagonal* (LU), and *scalar pentadiagonal* (SP), and the kernels are *conjugate gradient* (CG), *embarrassingly parallel* (EP), *3-D fast-Fourier transform* (FT), *integer sort* (IS), and *multigrid (*MG). The five kernels mimic the computational core of five numerical methods used by CFD applications.

The simulated CFD applications reproduce much of the data movement and computation found in full CFD codes.

Implementation of the NAS benchmarks are based on either Fortran 90 (including Fortran-77), or C language (except for NPB-JAVA 3.0 and NPB-HPF 3.0) because of the observation that Fortran and C are the most commonly used programming languages by the scientific parallel computing community.

I used three different versions of the NAS benchmarks, version 2.3, version 2.4, and version 3.0. NPB 2.3 is implemented with MPI-based source-code. They are intended to be run with little or no tuning. NPB 2.3 comes with five problem sizes for each benchmark: small class S, workstation class W, large class A and larger classes B and C. I study the characteristics of class B and class C in this chapter, and then the performance of class B in chapter 4.

High-performance computer systems have grown significantly in size and capabilities, including increases in cache and memory size, improved compiler technology and increased network bandwidths. The latest release, NPB 2.4 is also implemented with MPI, but contains a new and even larger class D, whose characteristics is studied in Section 4.2. Each class D benchmark involves approximately 20 times as much work, and a data set that is approximately 16 times as large, comparing with class C benchmark. Note that the class D implementation of the IS benchmark is not yet available.

NPB 3.0 is implemented in three different ways, OpenMP, High Performance Fortran (HPF) [17], and Java, which are called NPB–OMP 3.0 [23], NPB-JAVA 3.0 [18] and NPB-HPF 3.0, respectively. They were derived from the NPB-serial implementations released with NPB 2.3, with some additional optimization. I study the class B performance of NPB–OMP 3.0 and NPB-JAVA 3.0 in chapter 4.

3.1.1 EP

An embarrassingly parallel kernel, EP [5] provides an estimate of the upper achievable limits for floating point performance, by generating pairs of Gaussian random deviates

according to a specific scheme. That is the performance without significant interprocessor communication. The MPI version of the kernel benchmark requires a power of two number of processors.

3.1.2 MG

Simplified multigrid kernel, MG [5] uses a V-cycle multgrid method to compute an approximate solution to the discrete Poisson problem, the 3-D scalar Poisson equation. The partitioning of the grid onto processors occurs such that the grid is successively halved, starting with the *z* dimension, then the *y* dimension and then the *x* dimension, and repeating until all power-of-two processors are assigned. The algorithm requires highly structured long distance communication between coarse and fine, so that it tests both short and long distance data communication. The MPI version of the kernel benchmark requires a power of two number of processors.

3.1.3 CG

The *conjugate gradient* kernel, CG [5] computes an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix, using conjugate gradient method. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, which employs unstructured matrix vector multiplication. The MPI version of the kernel benchmark requires a power of two number of processors.

3.1.4 FT

FT [5] is a 3-D fast-Fourier transform (FFT) partial differential equation (PDE) benchmark. It numerically solves a certain partial differential equation using forward and inverse FFTs. The implementation of FT follows a fairly standard scheme. The 3-D array of data is distributed according to *z*-planes of the array – one or more planes are stored in each processor. The forward 3-D FFT is then performed as multiple 1-D FFTs in each dimension, first in the *x*- and then *y*- dimensions. An array transposition is then performed which amounts to an exchange from all nodes to the others. The final set of 1-D FFTs is then

performed. The MPI version of the kernel benchmark requires with a power of two number of processors.

3.1.5 LU

The *lower-upper diagonal* benchmark, LU [5], uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system which results from finite-difference discretization of Naview-Stokes equations in 3-D by splitting it into block lower and upper triangular systems. 2-D partitioning of the grid onto processors occurs by halving the grid repeatedly in the first two dimensions, alternately *x* and then *y*, until all power-of-two processors are assigned, resulting in vertical pencil-like grid partitions. The ordering of point based operations constituting the SSOR procedure proceeds on diagonals which progressively seep from one corner on and given *z* plane to the opposite corner of the same *z* plane, there upon proceeding to the next z plane. Communication of partition boundary data occurs after completion of computation on all diagonals that contact an adjacent partition. The MPI version of this benchmark requires a power of two number of processors.

3.1.6 IS

Parallel sort over small integers, IS [5], kernel benchmark, sorts *N* keys in parallel, which are generated by the sequential key generation algorithm given initially must be uniformly distributed in memory. IS is the only NAS benchmark that written in C language (except for the JAVA version). The MPI version of the kernel benchmark requires a power of two number of processors.

3.1.7 BT and SP

Block Tridiagonal, BT [5], and Scalar Pentadiagonal, SP [5] have a similar structure; each solves three sets of uncoupled systems of equations. BT uses an implicit algorithm to

solve 3-D compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the *x*, *y* and *z* dimensions. The resulting systems are Block-Tridiagonal of 5x5 blocks and are solved sequentially along each dimension. Differently, SP is based on a Beam-Warming approximate factorization that decouples the *x*, *y* and *z* dimensions. Both BT and SP require a square number of processors. Those codes have been written such that if the number of processors is different than a square number, then the unneeded processors are set inactive and are not used during computations.

3.2 Characteristics of the NAS Benchmark Suite

A proper understanding of the communication patterns of parallel applications is important for determining how to maximize their performance within a given environment. This section presents the communication patterns of the MPI version of the NAS benchmark suite. This includes the message sizes, the number of messages, and the destination distribution. I executed the applications on a 4-node Sun Fire 6800s, and a Sun Fire 15K server at the *High Performance Computing Virtual Laboratory* (HPCVL) for gathering their communication traces. I wrote my own profiling codes using the wrapper facility of the MPI. I did this by inserting recording operations in the profiling MPI library and saving the communication related activities into log files.

I chose five benchmarks from the NAS benchmark suite, BT, SP, LU, CG and MG. I characterized three problem sizes: class B and class C (from NPB 2.3 MPI), class D (from NPB 2.4 MPI). I have not included EP and FT because all of the communications in EP and FT are collective communications. In IS, each process is sending messages to only one fixed destination process. Note that I was not able to gather the results for the class D of BT with 16 processes and 32 processes due to their large problem sizes.

Traditionally, the communication properties of parallel application have been characterized by three attributes: the spatial, temporal, and volume components [11][26]. The spatial behavior is presented by distribution of message destinations. The temporal behavior

is defined by message generation rate. (The temporal characteristics are not provided in this thesis). The volume of data transfer is characterized by the distribution of message sizes and the average number of messages.

Figure 3.1 shows the number of send events per process for the benchmarks. Class D has a much larger number of send events than class B and class C, more than two times for MG, CG, and LU (64 processes). Class B and class C behave very similar. With the increasing number of processes, from 16 to 64 processes, the number of sends in MG and LU does not change much, while the number of sends in BT and SP is increasing. For CG, BT and SP, the send events are evenly assigned to each process, because the minimum, average and maximum number stay very close. LU has the largest number of send events among all the benchmarks; each process is sending more than 200000 messages for class D. MG has the smallest number of send events, smaller than 10000 events for class D, with 64 processes.

Figure 3.2 shows the average, minimum and maximum message size (Kbyte) transferred among processes. The average message sizes of the whole applications are shown in Figure 3.3. Class D has a much larger average message size than class B, which is slightly larger than class C. All benchmarks have decreasing average message sizes, when running with more processes. LU has the smallest average message size, around 0.7 Kbytes for class D and 64 processes, while BT has the biggest average message size, around 60 Kbytes for class D and 64 processes.

Figure 3.4 shows the number of destinations. In BT and SP, there are two different communicators used. I define <destination #, communicator> as one unique destination. CG, BT and SP have the same number of destinations for each process. It is clear that the processes in the benchmarks do not have many partners. LU and CG have up to four different destinations; MG has up to nine destinations; BT and SP have up to 12 destinations. So, in conclusion, in the NAS benchmarks, each process has relatively constant number of partners to communicate with.

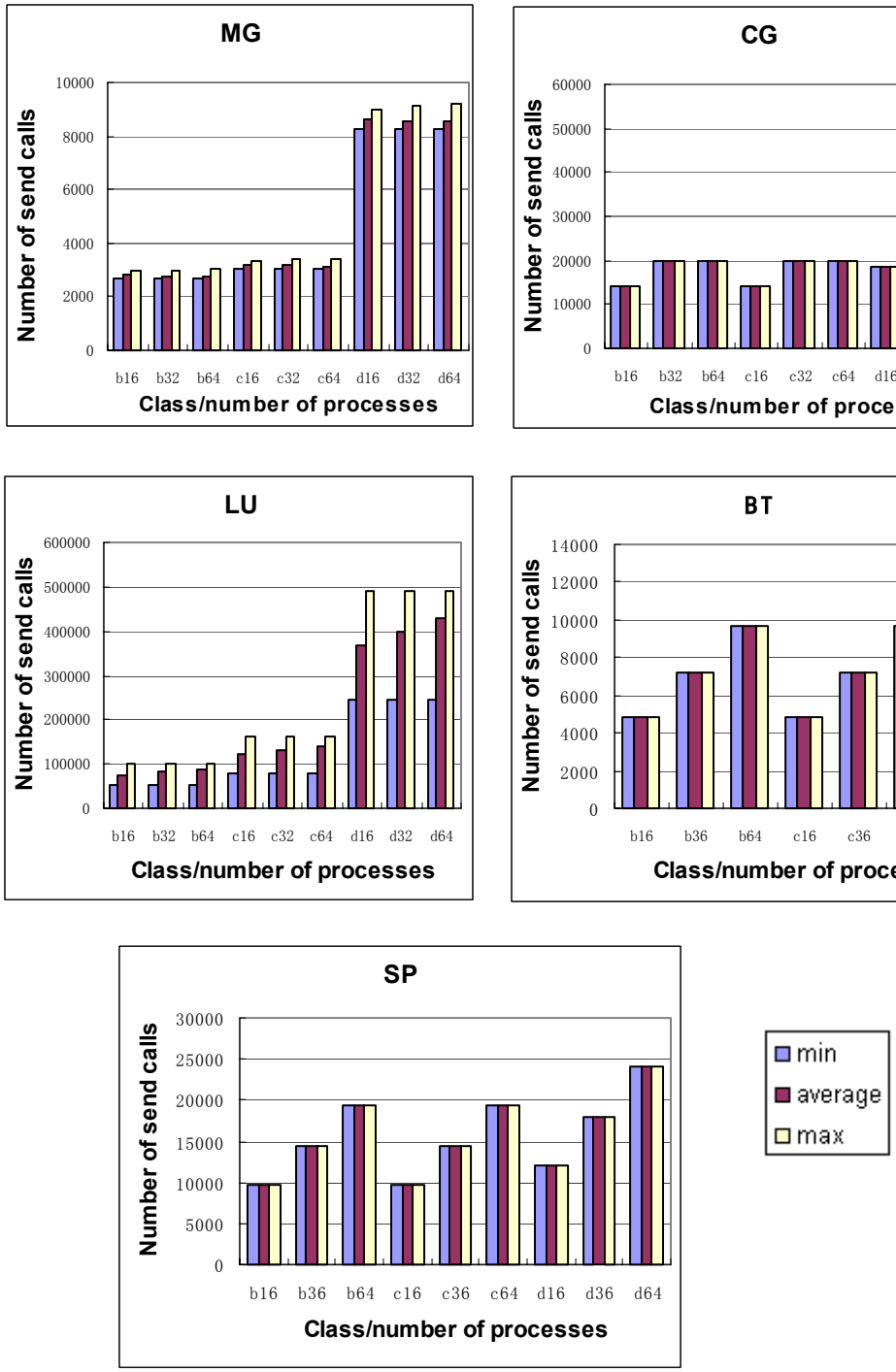


Figure 3.1. Number of send events per process.

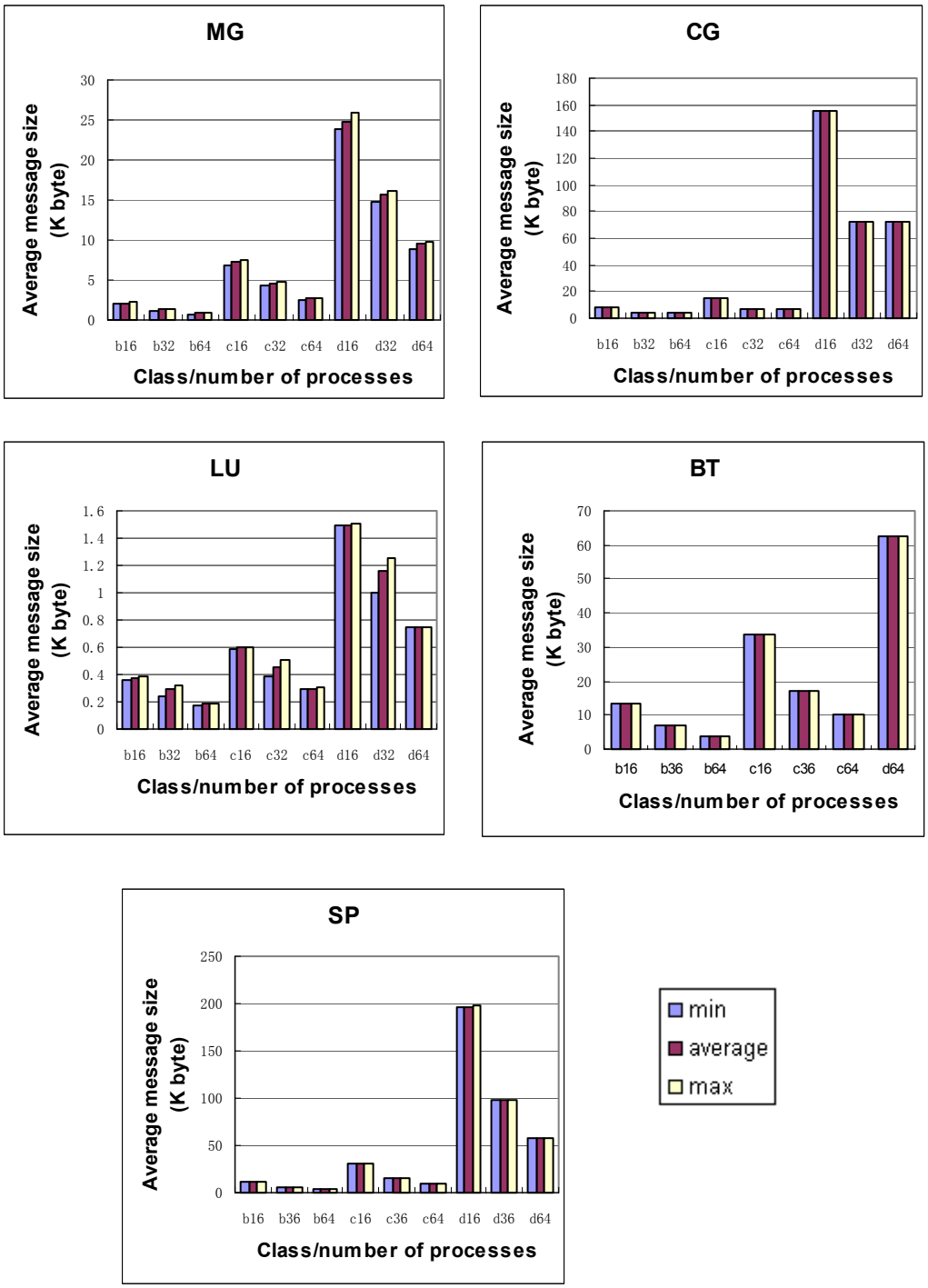


Figure 3.2. Average message size (Kbytes) per process.

Figure 3.5, Figure 3.6, and Figure 3.7 show the cumulative distribution of message sizes. The value in the Y-axis presents the percentage of the messages with message sizes smaller than the X-axis value. MG has many different sizes of messages, from 10 bytes to more than 100 Kbytes. Almost half of the messages sent in CG are around 0.6 Kbytes, while another half is more than 10Kbytes. LU sends many small messages around 500 bytes, which makes

the average message size around 700 bytes. BT and SP have quite large message sizes, where almost all the messages are more than 10 Kbytes.

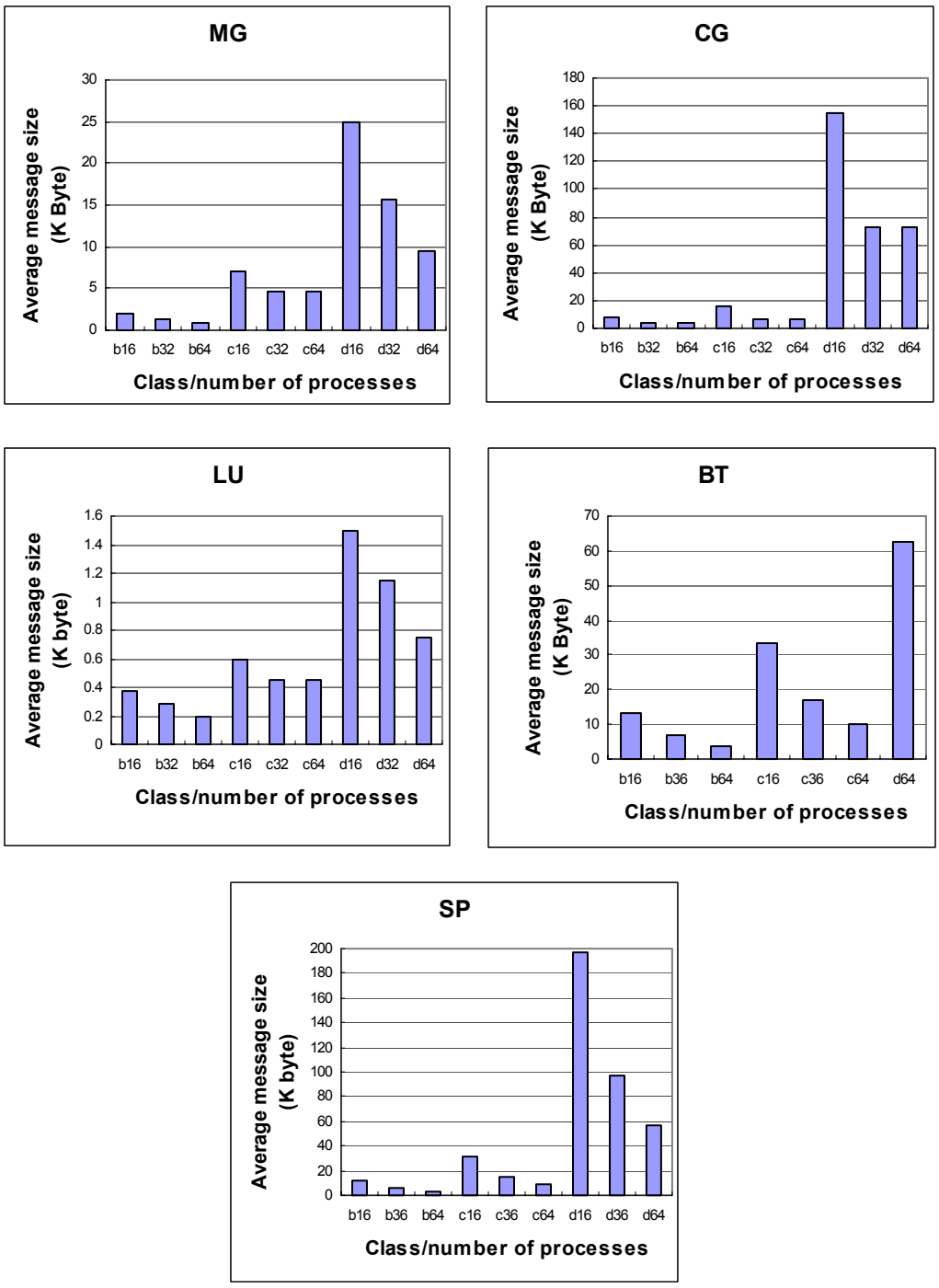


Figure 3.3. Average message size (Kbytes).

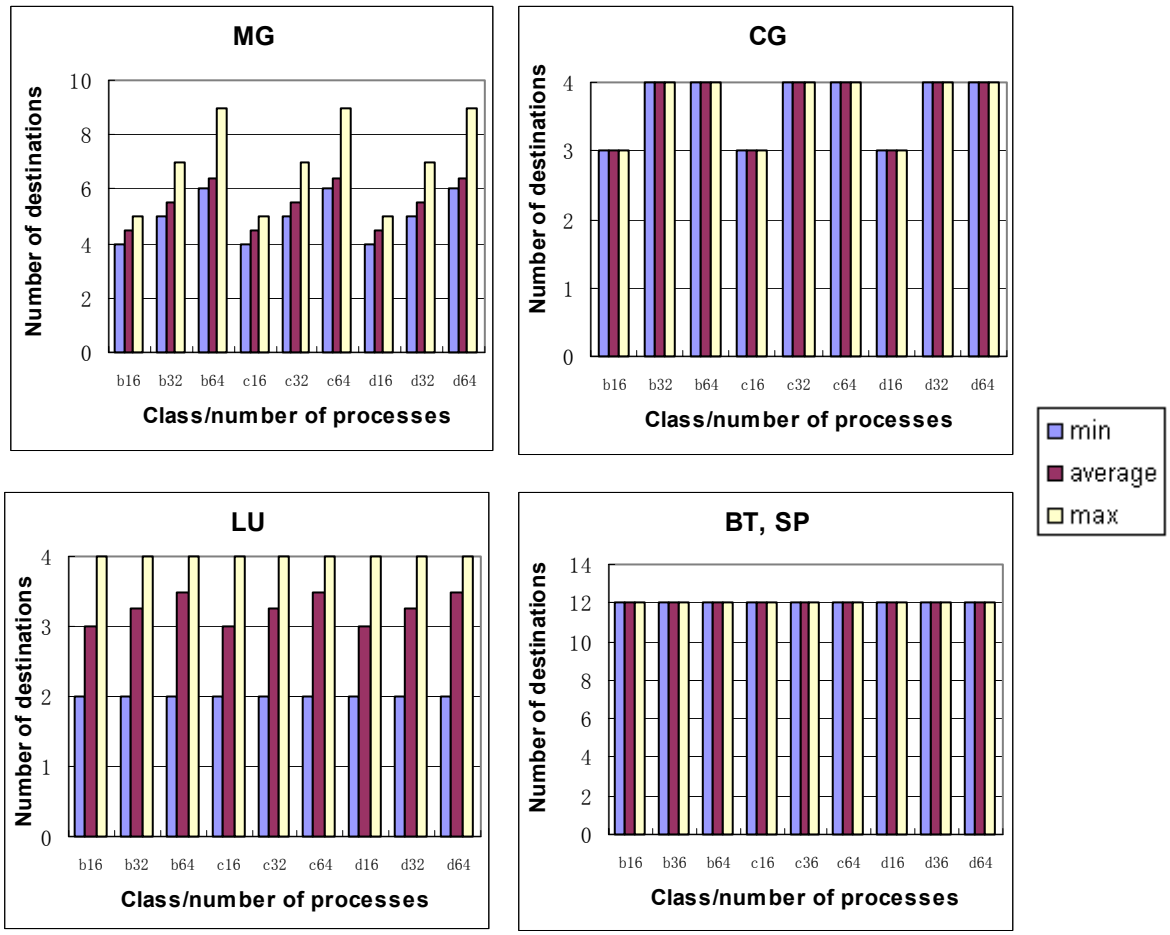


Figure 3.4. Number of destinations.

Figure 3.8 and Figure 3.9 show the destination distribution of process 0, running with 64 processes. For class B and class C, CG, BT and SP have exactly the same destinations, and the same number of messages is sent to each destination. For class D, all benchmarks have similar destinations as in class B and class C, but larger number of messages is sent. It is clear that the benchmarks do not show many communication partners.

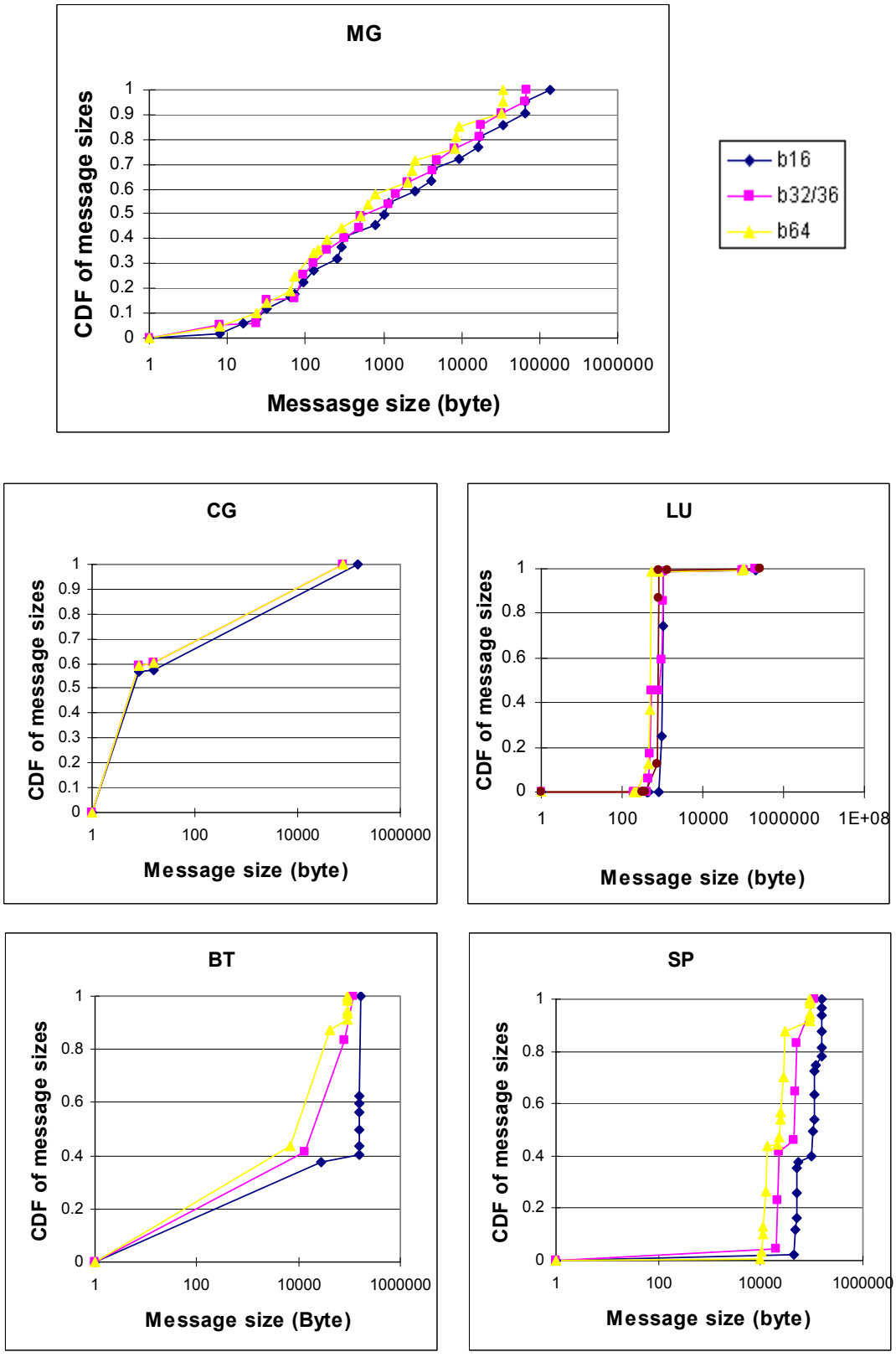


Figure 3.5. Cumulative distribution of message sizes, class B.

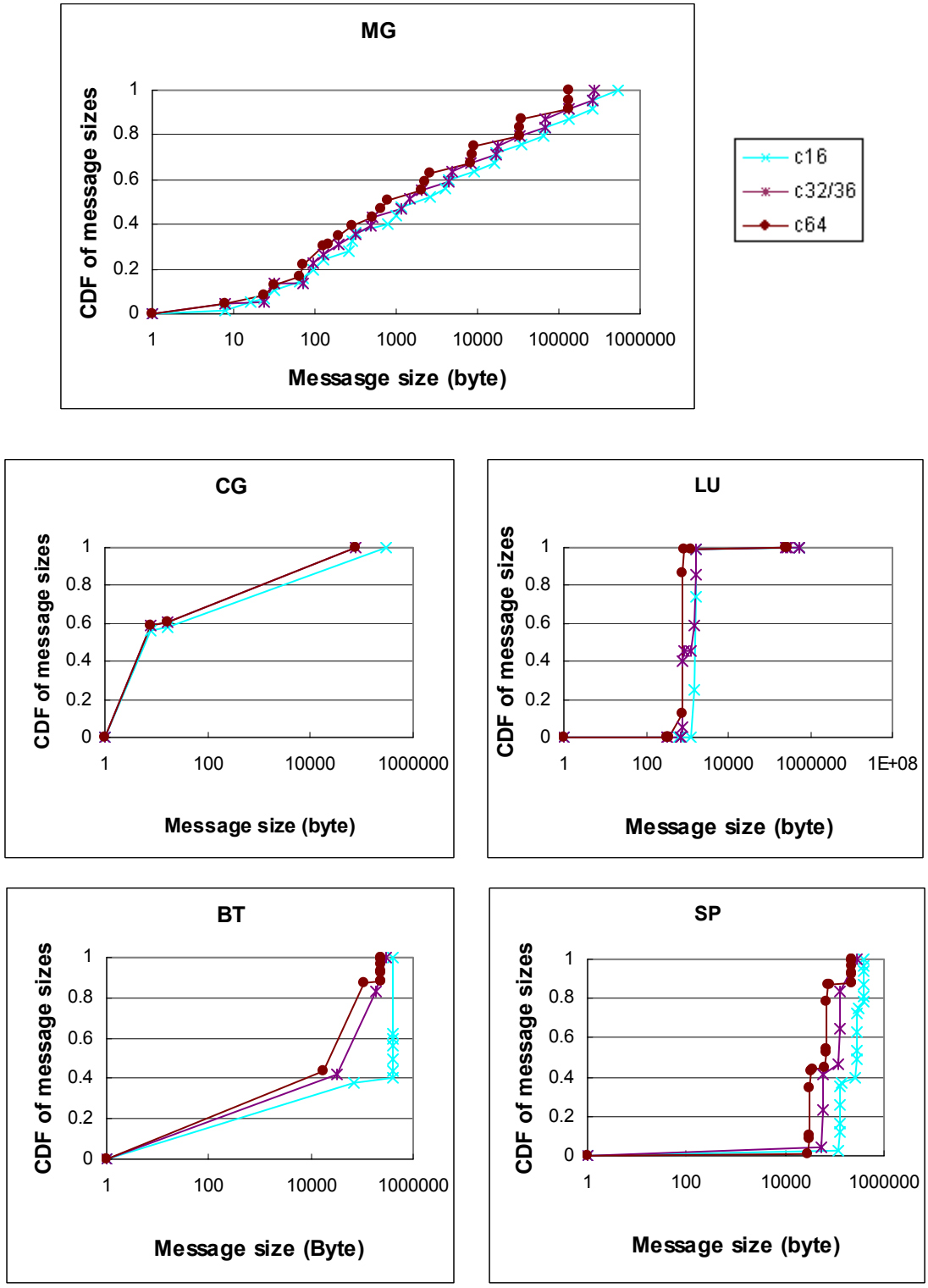


Figure 3.6. Cumulative distribution of message sizes, class C.

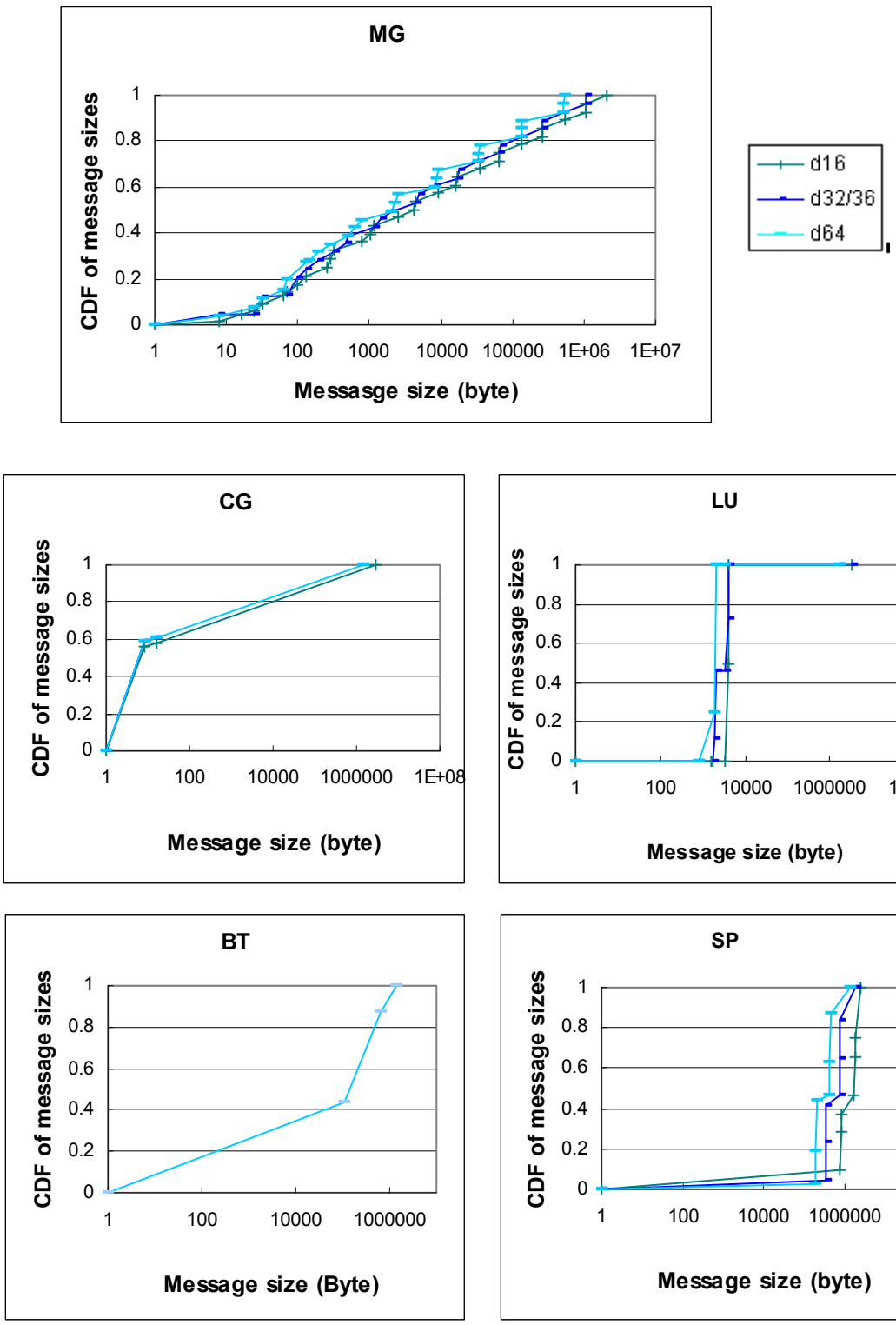


Figure 3.7. Cumulative distribution of message sizes, class D.

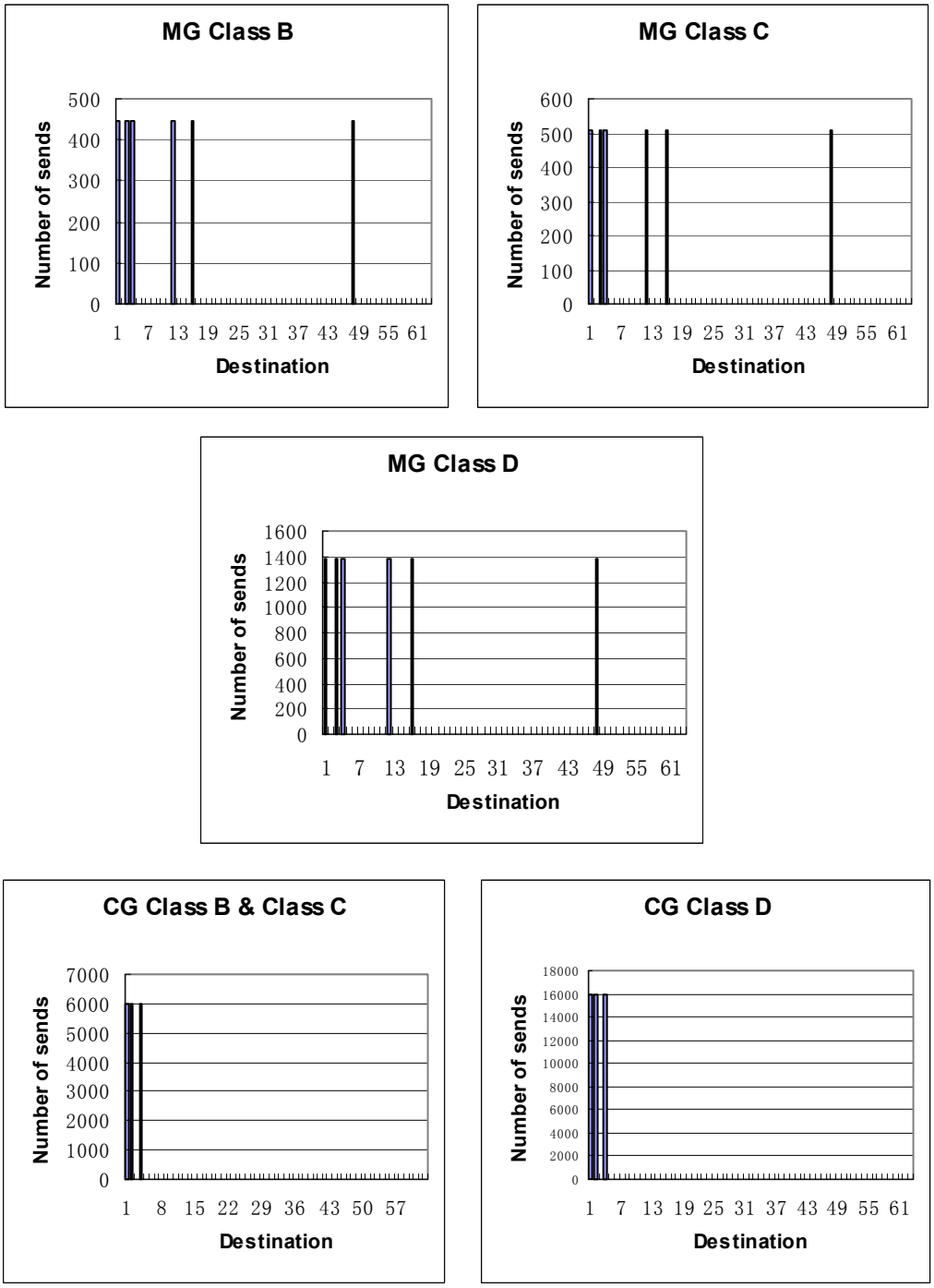


Figure 3.8. Destination distribution of process 0 (64 processes)

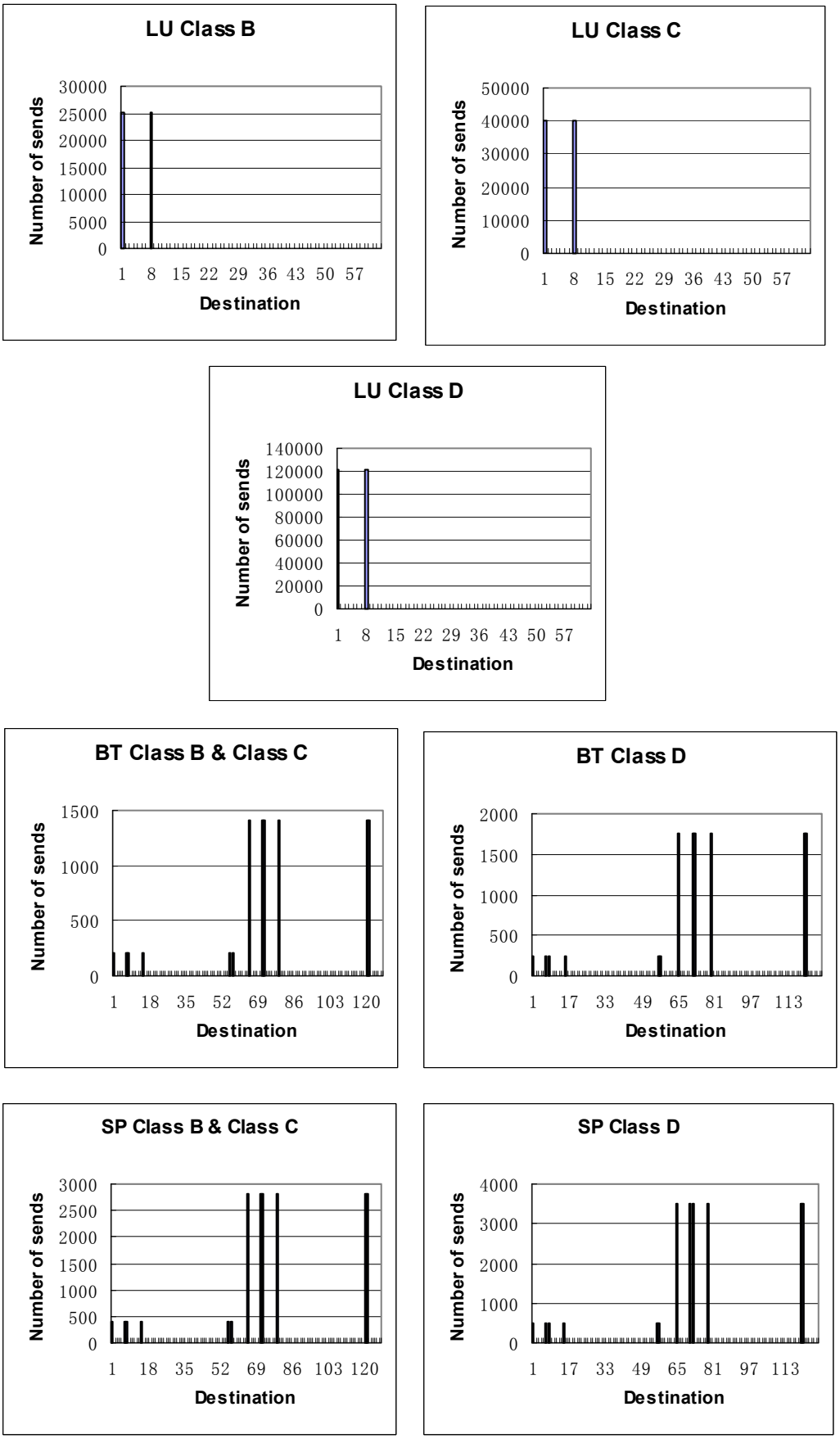


Figure 3.9. Destination distribution of process 0 (64 processes).

3.3 Summary

In this chapter, the NAS Parallel benchmark suite has been introduced. I analyzed the communication characteristics of the MPI version, where it could provide insights as to the performance of applications on high performance computers. I presented the characteristics of applications under different problem sizes (classes B, C and D), and different system sizes (16, 32 and 64 processes).

I have found that the number of send calls and average message sizes are increasing with larger workloads. Class D has much more communication than class B and class C. For the same workload, running with larger number of processes, the benchmarks have larger number of send calls, but with smaller average message sizes. LU has the largest number of send calls, and smallest average message size. LU is the benchmark that sends a lot of small size messages. MG has many different message sizes. BT and SP are very similar, both sending relatively large size messages.

Knowing more about the application benchmarks, I will evaluate the micro-benchmark and application benchmark performance on two different SMPs in the next chapter. I will try to correlate the performance of these applications on different platforms (SMPs and CLUMPs) with their communication characteristics.

Chapter 4

Performance on Small and Large SMPs

Having analyzed the communication characteristics of the NAS benchmark suite, I would like to know their performance on two different SMP systems. An SMP machine is the building block of cluster of SMP machines. For this, I would like to see their performance under a suite of parallel application benchmarks. In this chapter, I evaluate the performance on two SMPs, a 4-way Dell PowerEdge 6650, and a 72-way Sun Fire 15K server. The results include memory bandwidth, latencies and bandwidths of point-to-point communications, latencies of collective communications and the performance of three versions of the NAS parallel Benchmarks, implemented in MPI, OpenMP, and JAVA, respectively. I compare the results for different classes, as well as for different programming paradigms. In [24], the performance of OpenMP, MPI, and hybrid programming paradigms are provided on Sun Fire 15K server. However, they only used the BT benchmark for comparative study. The performance between the MPI and OpenMP version of a large-scale application benchmark suite, *SPECseis* was compared in [4].

4.1 SMP Platforms

Dell PowerEdge 6650 has 4 Intel Xeon MP processors (1.4-GHz), and 2 GB of RAM. In our tests in this chapter and following chapters, Hyper-Threading is turned off on the Dell PowerEdge machines. The system uses Linux, RedHat 9, as the operating system, the Intel compiler 7.1 and MPICH 1.2.5 for the MPI messaging layer. I use the *ch_shmem* device which is suitable for SMP, because of its highly optimized use of shared memory model on SMP. The Sun Fire 15K server has 72 UltraSPARC III Cu processors (1.2-GHz), and 144 GB

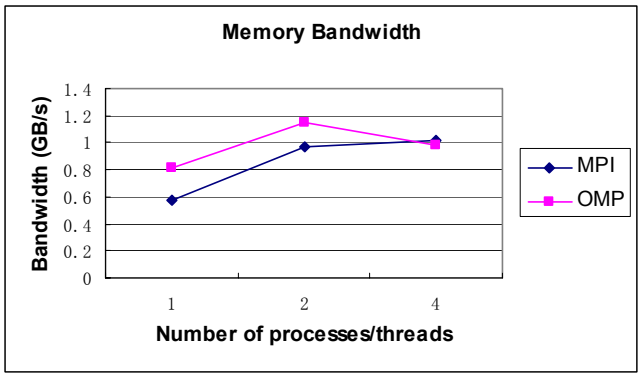
of RAM. The environment includes the Solaris™ 9 Operating Environment, and Sun MPI 6.0 for message passing.

4.2 Memory Bandwidth

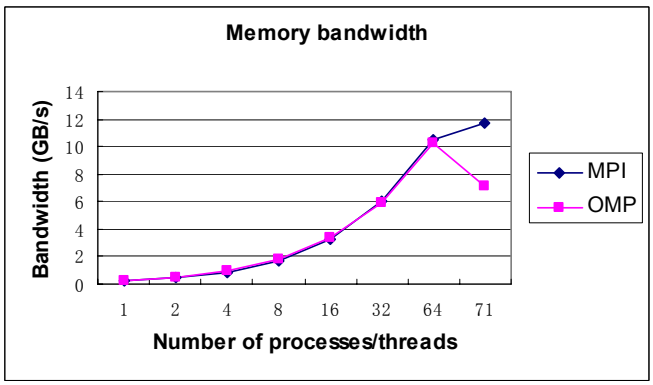
Computer CPUs are getting faster much more quickly than computer memory systems. As this continues, more and more programs will be limited in performance by the memory bandwidth of the system, rather than by the computational performance of the CPU. In this Section, I study the memory bandwidth of the Dell PowerEdge 6650, and the Sun Fire 15K server.

We wrote the benchmark codes for memory bandwidth in MPI and OpenMP based on the *STREAM* benchmark [51]. The STREAM benchmark is specifically designed to work with datasets much larger than the available cache on any given system, so that the results are (presumably) more indicative of the performance of very large, vector style applications. What we do is to have all processes in the MPI version, or threads in the OpenMP version, do a number of memory operations at the same time. These operations are copy, scalar, addition, and triad, which is the combination of the scalar and addition operations.

Figure 4.1 shows the memory bandwidth of the Dell PowerEdge 6650, and the Sun Fire 15K server. For the MPI version with one process, PowerEdge 6650 achieves 0.6 GB/s bandwidth, comparing to 0.8 GB/s for the OpenMP version with one thread. The OpenMP version also has a better performance than the MPI version when running with two processes/threads. They both have degrading performance for 4 processes or threads. The Sun Fire 15K server shows very good scalability. The performance is dropping for 71 processes/threads. The MPI version has similar performance as the OpenMP version, which indicates that Sun MPI is highly optimized using shared memory within the SMP. The Dell PowerEdge 6650 has a larger memory bandwidth when running with 2 processes/threads than the Sun Fire server.



(a)



(b)

Figure 4.1. Memory bandwidth on (a) Dell PowerEdge 6650 (b) Sun Fire 15K.

4.3 Communication Latency and Bandwidth

I am also interested in the performance at the MPI level. In this section, I provide the performance of point-to-point latencies and bandwidths. Latency is defined as the time it takes for a message to travel from the sender process address space to the receiver process address space. I provide here our framework for measuring the latency of a message transfer. I do the standard *ping-pong* test under different MPI send modes; measure the latency with different send buffers; and find the unidirectional latency. In the ping-pong test, the sender sends a message and the receiver upon receiving the message, immediately replies with the same message size [31]. In the unidirectional test, the sender sends a number of messages and the receiver replies after receiving all these messages.

The *bi-directional latency* test is the ping-pong test that is repeated sufficient number of times to eliminate the transient conditions of the network. Then, the average round-trip time

divided by two is reported as the one-way latency. This test is repeated for messages of increasing sizes. I tested using matching pairs of blocking sends and receives under different MPI send modes; that is, the *standard* mode, the *synchronous* mode, the *buffered* mode, and the *ready* mode.

In the standard latency test with buffer management, I modify the standard ping-pong test such that each send operation uses a different message buffer. This test exposes the buffer management cost at the MPI level. The results are obtained with 16 different buffers. I experimented with different message sizes.

The unidirectional bandwidth test shows the capacity of the network. In this measurement, the sending process constantly pumps messages into the network without waiting for an acknowledgement. The receiving node sends back an acknowledgment upon receiving all the messages. Bandwidth is reported as the total number of bytes per unit time delivered during the time measured.

All the results are averaged over running the tests for 1000 times. I do not have the results for the synchronous mode for the Sun Fire 15K due to limited exclusive access to the system. Figure 4.2 shows the latencies on Dell PowerEdge 6650, and Figure 4.3 shows the results on Sun Fire 15K server. For the Dell PowerEdge 6650, the latency stays at 10 μs for up to 512-byte message, for the *standard*, *ready*, *buffered* and *diffbuf* modes. The *Unidirectional* mode has a little bit larger latency, 12 μs for small size messages, and 32 μs for the *synchronous* mode. The Sun Fire 15K server has a better performance, 3 μs up to 64-byte messages for *unidirectional* ping, 5μs for *standard ready* and *diffbuf*, and 6 μs for *buffered* mode. For large message size, the Sun Fire 15K server also has smaller latency.

Figure 4.4 and Figure 4.5 show the bandwidth on these two systems. The Dell PowerEdge 6650 achieves up to 600 MB/s, while the Sun Fire 15K server gets up to around 550MB/s. The performance of Dell PowerEdge 6650 is degrading after 64 Kbytes messages. This might be due to several reasons: protocol switch from eager to rendezvous protocol, poor MPI implementation, or lack of sufficient memory.

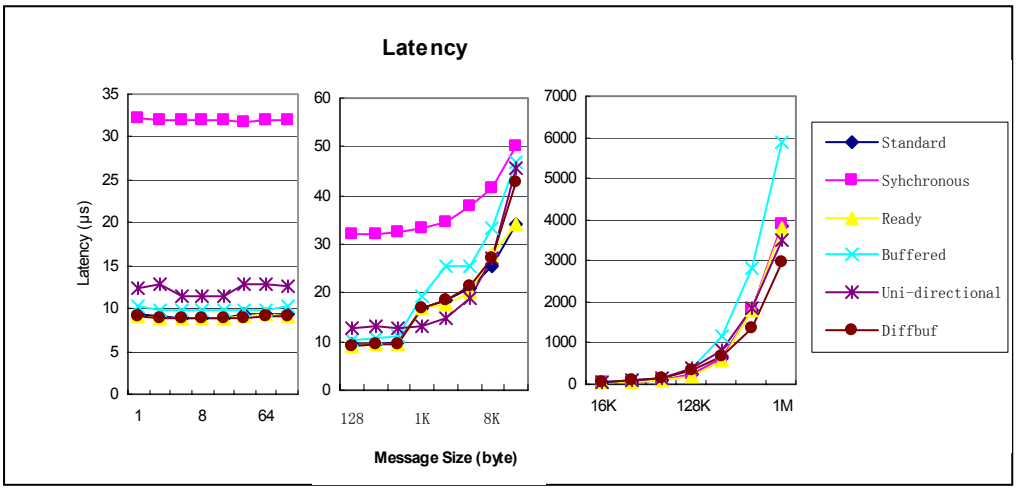


Figure 4.2. Point-to-point latency on Dell PowerEdge 6650.

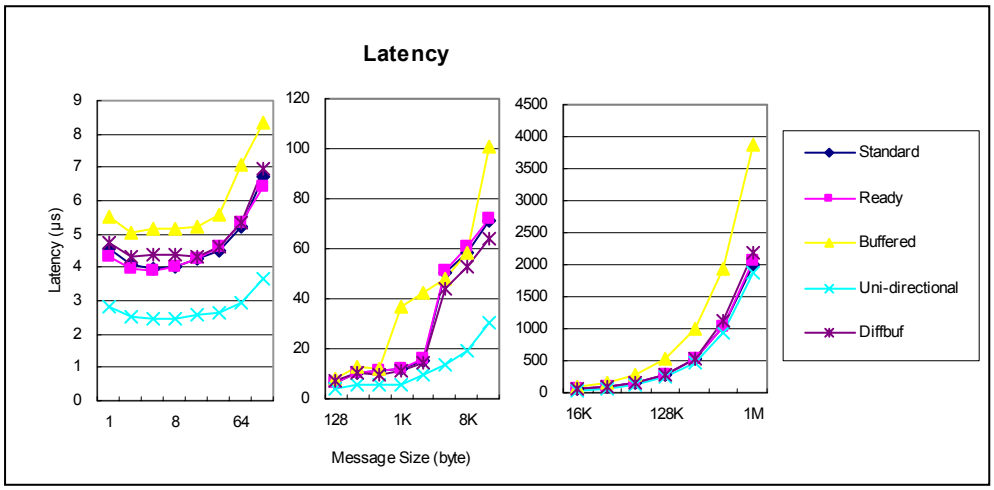


Figure 4.3. Point-to-point latency on Sun Fire 15K.

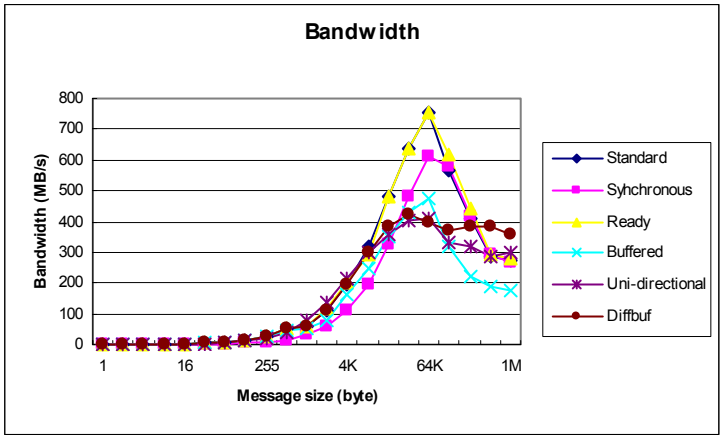


Figure 4.4. Bandwidth on Dell PowerEdge 6650.

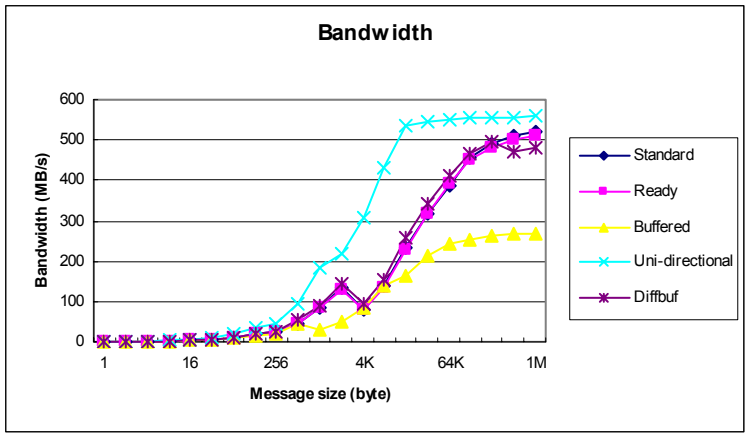


Figure 4.5. Bandwidth on Sun Fire 15K.

I have presented the MPI latencies and bandwidths on Dell PowerEdge 6650 and Sun Fire 15K. From the results, it can be concluded that the Sun Fire 15K server has a better memory hierarchy system and a better MPI implementation than the MPICH on the Dell PowerEdge 6650 server. The Dell PowerEdge 6650 server has a degrading performance for message sizes larger than 64 Kbytes. In the next section, I will look at the performance of collective communications, an important part in message passing applications.

4.4 Collective Communications

Efficient implementation of collective communication operations is one of the keys to the performance of parallel applications. I have chosen the *broadcast*, *scatter*, *gather*, *alltoall*, *alltoallv*, *reduce* and *allreduce* operations as representatives of the mostly used collective communication primitives in parallel applications. I have measured the performance in terms of their average completion time over 500 times running. An overall look at their running time shows that the *reduce* and *allreduce* operations take the longest, and *broadcast* operations the shortest.

On the Dell PowerEdge 6650, I present the results running with 4 processes. On the Sun Fire 15K, I provide the results running with 4 and 64 processes. Figure 5.7 shows the completion time of collective communications on Dell PowerEdge 6650, and Figure 5.8 shows the results on Sun Fire server, both with 4 processes. For the Sun Fire 15K, *alltoall* and *alltoallv* have a special long time for 1 byte and 512 bytes message size. I have not found any reason for that. In general, Sun Fire 15K server has a better performance.

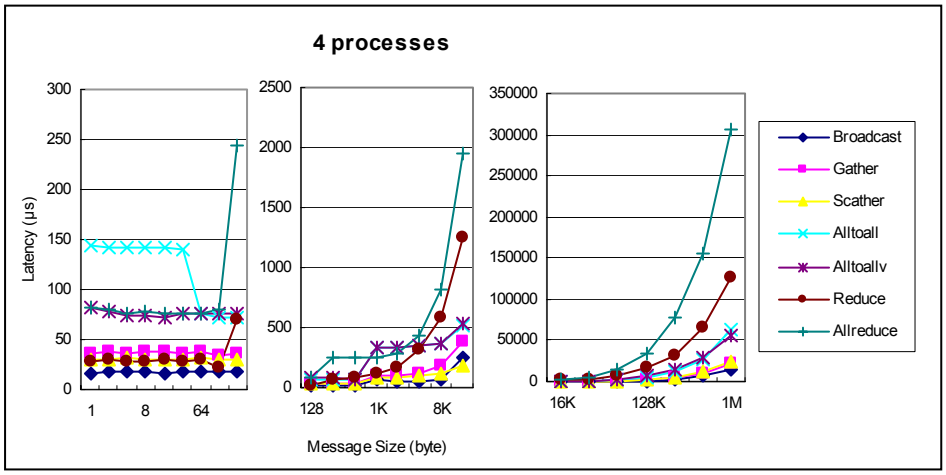


Figure 4.6. Latency of collective communications on Dell PowerEdge 6650.

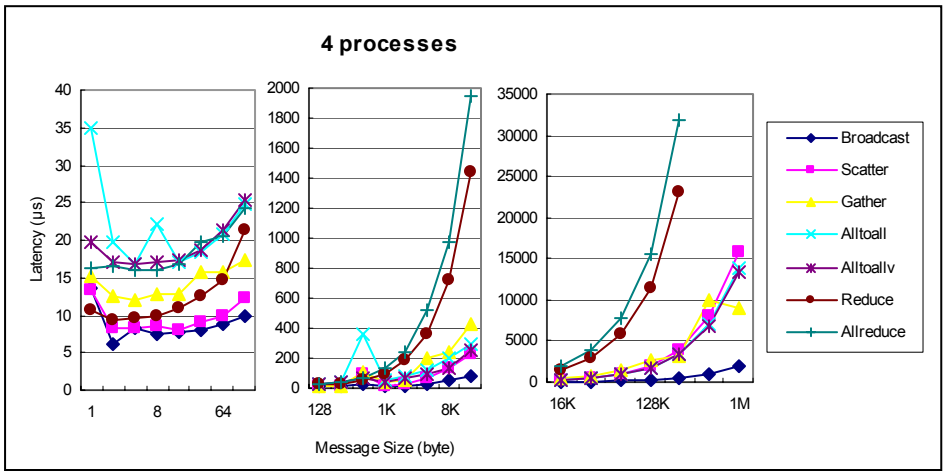


Figure 4.7. Latency of collective communications on Sun Fire 15K, 4 processes.

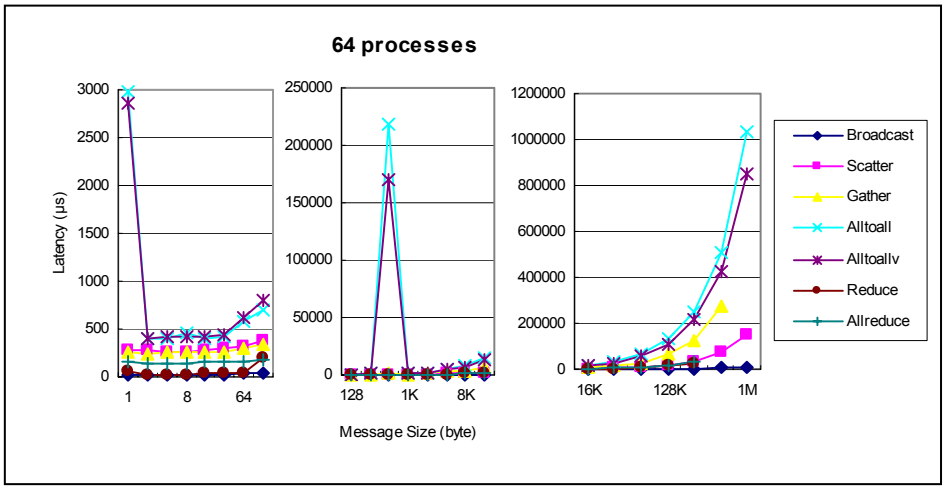
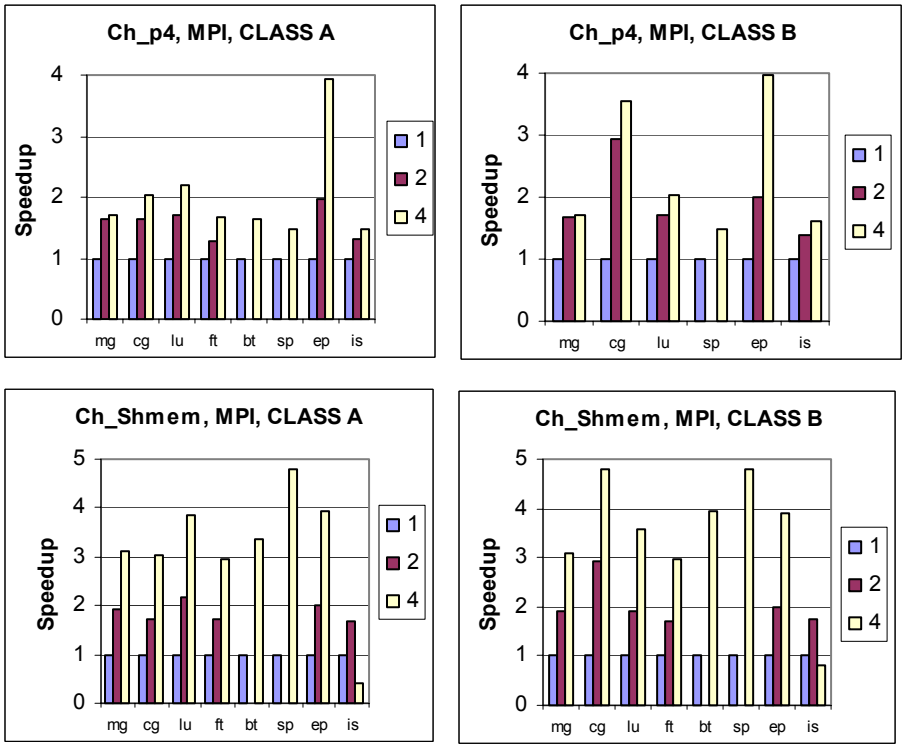


Figure 4.8. Latency of collective communications on Sun Fire 15K, 64 processes.

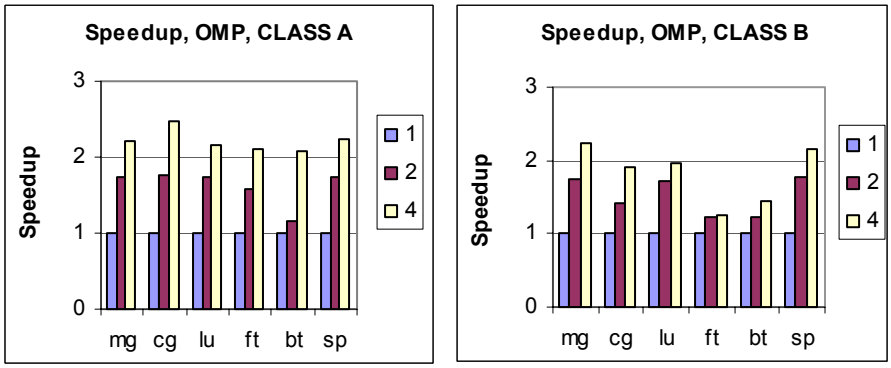
4.5 Performance of Application Benchmarks

Having known the performance at the MPI level, it is the time to run some application benchmarks to evaluate their performance. I evaluate the performance of NPB 2.3 (MPI version), NPB 3.0 (JAVA version) and NPB 3.0 (OpenMP version) on our 4-way, and 72-way SMP nodes. Because of the size of the SMPs, I run the NPB suite with number of processes/threads 1, 2 and 4 for the 4-way SMP, and from 1 to 64 for the 72-way SMP, respectively. I have chosen the class A and class B problem sizes, due to the memory size of the Dell PowerEdge 6650. I ran all the benchmarks three times, and the results shown in this section are the average completion time. Note that the NPB 3.0 JAVA version does not have an implementation for EP. I do not have the results of EP and IS for all OpenMP cases due to memory limitation.

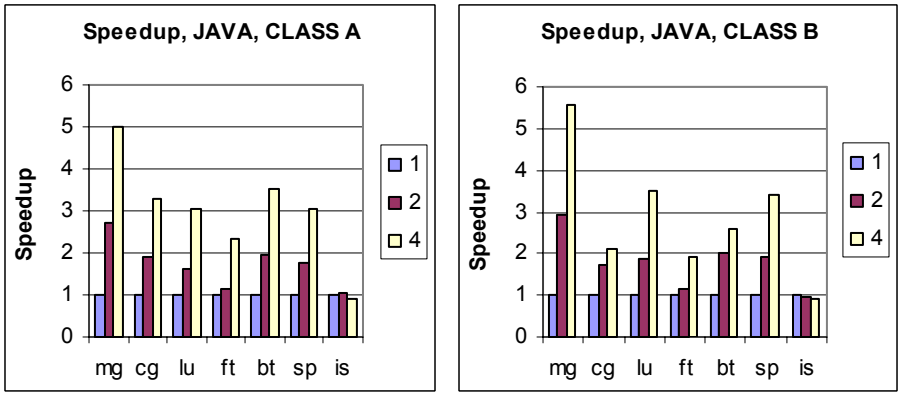
Figure 4.9 shows the speedup for the MPI, OpenMP and JAVA versions of the NAS parallel benchmarks on Dell PowerEdge 6650. I include the results for MPICH on two devices, the *ch_p4* and *ch_shmem*. The *ch_shmem* has a better scalability than the *ch_p4*, which indicates that the *ch_shmem* has a better implementation on SMP. For the MPI results, the performance of class A and class B are similar except for CG. I believe it is because the serial version of the CG program of MPI version takes quite long time for class B. Although LU has many small size messages, discussed in chapter 3, the speedup is better than the other benchmarks except for CG and EP. It is because the Dell PowerEdge 6650 does not have a good performance for large messages. EP does not have many communications. It has a linear speedup. FT has a poor performance because it requires larger memory, exceeding our memory system size.



(a)



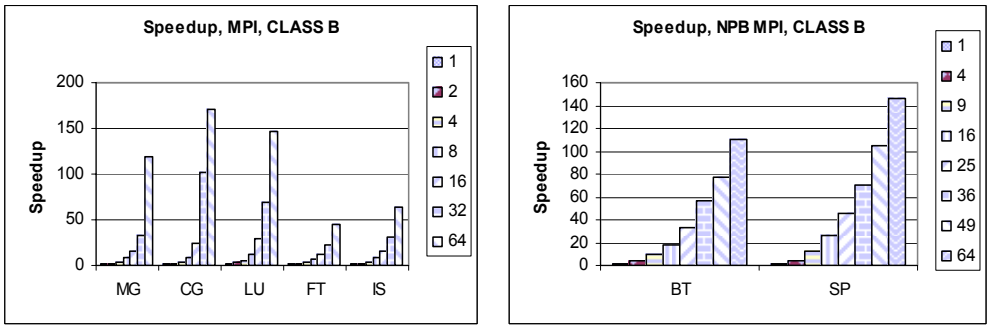
(b)



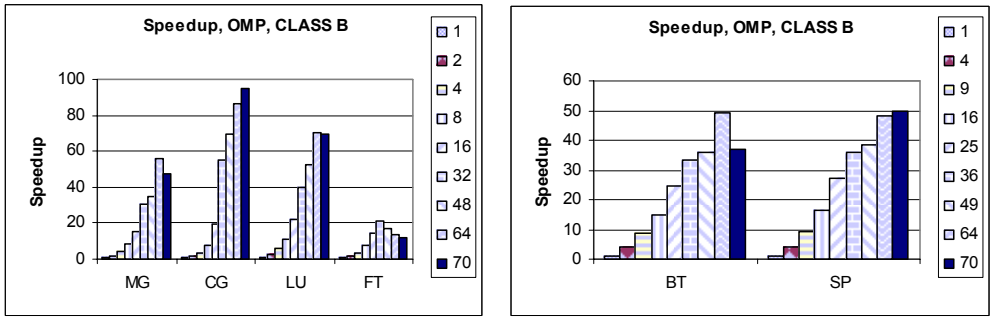
(c)

Figure 4.9. Speedup on Dell PowerEdge 6650, Class A and Class B, of (a) NPB2.3-MPI (b) NPB3.0-OMP (c)NPB3.0-JAVA.

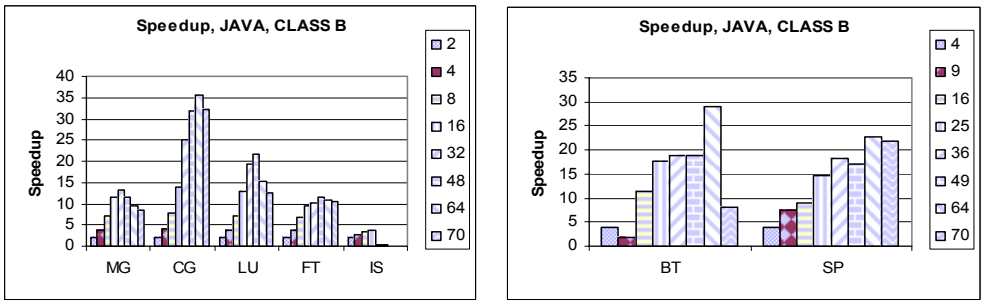
For NPB3.0 OMP version, the performance of class A is better than class B. MG and SP have almost the same performance in class A and class B. BT and FT have lower speedup because they require more memory. Figure 4.9 (c) shows the speedup of NPB3.0 Java version. The performance of class A and class B are similar. MG has the best performance, almost super-linear. In general, NPB3.0 MPI version on device *ch_shmem* has the best performance in the three versions of NAS benchmarks. NPB3.0 Java has slightly better performance than NPB2.3 OMP. The speedups of NPB3.0 OMP with 4 threads are all around or smaller than 2.5.



(a)



(b)



(c)

Figure 4.10. Speedup on Sun Fire 15K, Class A and Class B, of (a) NPB2.3-MPI (b) NPB3.0-OMP (c)NPB3.0-JAVA.

For the Sun Fire 15K server, I have chosen the class B. (Unfortunately I did not have enough exclusive access time to get the results for class A.) Figure 4.10 (a) shows the speedup of NPB 2.3 MPI. All benchmarks have very good scalability. CG has the best performance, while FT has the worst. NPB 2.3 MPI has the best performance in three versions, while NPB 3.0 JAVA does not have good scalability.

Table 4.1 compares the execution times for the class B on two SMPs, for applications running with 4 threads/processes. In most of the MPI and OpenMP version of the benchmarks, the Sun Fire 15K shows a better performance. For the Java version, except for LU and FT, the Dell PowerEdge 6650 shows a better performance than the Sun Fire 15K.

Table 4.1. Execution time of NPB 3.0 OMP, NPB 3.0 Java, NPB2.3 MPI, Class B, on Dell PowerEdge 6650 and Sun Fire 15K, with 4 threads or processes

MPI	Dell p4	Dell shmem	SunFire	OMP	Dell	SunFire	JAVA	Dell	SunFire
CG	815.9	668.9	283.0	CG	934.0	753.9	CG	796.9	1010.1
LU	796.8	2296.0	1051.8	LU	1022.5	773.4	LU	16261.5	6090.5
FT	295.5	411.2	250.8	FT	326.9	201.2	FT	1023.0	977.8
MG	59.8	82.6	58.0	MG	38.7	35.1	MG	68.5	180.7
BT	3290.2	3302.0	1962.2	BT	1150.1	662.8	BT	3079.4	3637.6
SP	1728.0	2877.9	1489.3	SP	609.5	739.7	SP	1170.6	3628.0

4.6 Summary

In this chapter, I compared the performance of memory bandwidths, point-to-point latencies and bandwidths at the MPI level. I also compared three versions of the NAS applications benchmarks on two SMPs, implemented in MPI, OpenMP and Java.

I discovered that the large-scale SMP (Sun Fire 15K server) has a better MPI performance, while the small-scale SMP (Dell PowerEdge 6650) has a degrading performance for large size message transfers. The MPICH implementation of Dell PowerEdge 6650 is not as good as the Sun MPI implementation. For both systems, the MPI version of the application benchmarks has a better scalability than the OpenMP and Java versions.

Because of the excellent performance of the Sun MPI on the Sun Fire 15K server, I am interested to see how Sun MPI has been implemented, and how it performs among different nodes over the Sun Fire Link interconnect. In chapter 5, I will look into the RSM user-level communication layer over the network hardware. Then the performance at higher level will be provided on both the Sun Fire Link interconnect and the Myrinet in chapter 6.

Chapter 5

Remote Shared Memory over Sun Fire Link Interconnect

In chapter 4, I have studied the performance of two SMPs which can be used as a node in a CLUMP. The communication overhead is one of the most important factors affecting the performance of high-performance cluster computer systems. The interconnection network hardware and the communication system software and libraries are the keys to the performance. In this chapter, I introduce a new memory-based interconnect, Sun Fire Link, recently released by the Sun Microsystems. I study the user-level messaging layer, *Remote Shared Memory* (RSM) [45] and the Sun MPI [48] implementation on top of RSM in section 5.1, along with the performance of RSM API in section 5.2.

5.1 Remote Shared Memory

The Sun Fire Link cluster interconnect is used to cluster the Sun Fire 6800s and the Sun Fire 15K/12K systems. Information is transferred over fiber-optic communication links. Each link supports full-duplex, point-to-point communication [34]. The raw data transfer rate of each link is 1.64 GB/s. After coding, framing and protocol overhead, one link can theoretically sustain 1 GB/s of unidirectional user data bandwidth [34]. RSM is a high-performance memory-based mechanism, which implements user-level inter-node messaging with direct access to memory that is resident on remote nodes. Its performance directly affects the performance of upper software layers such as MPI, and application levels.

Table 5.1 shows some of the RSM API calls with their definitions. The complete API calls can be found in [45]. The API calls can be classified into five categories: interconnect controller operations, cluster topology operations, memory segment operations, barrier operations, and event operations.

Table 5.1. RSM API calls and their definitions (partial).

Interconnect controller operations	
rsm_get_controller	get a controller handle
rsm_release_controller	release a controller handle
Cluster topology operations	
rsm_free_interconnect_topology	get or free interconnect topology
rsm_get_interconnect_topology	get interconnect topology
Memory segment operations including segment management and data access	
rsm_memseg_export_create	resource allocation functions for export memory segments
rsm_memseg_export_destroy	resource release functions for export memory segments
rsm_memseg_export_publish	allow a memory segment to be imported by other nodes
rsm_memseg_export_republish	re-allow a memory segment to be imported by other nodes
rsm_memseg_export_unpublish	disallow a memory segment to be imported by other nodes
rsm_memseg_import_connect	create logical connection between import and export segments
rsm_memseg_import_disconnect	break logical connection between import and export segments
rsm_memseg_import_get	read from a segment
rsm_memseg_import_put	write to a segment
rsm_memseg_import_map	map imported segment
rsm_memseg_import_unmap	unmap imported segment
Barrier operations	
rsm_memseg_import_close_barrier	remote memory access error detection functions
rsm_memseg_import_destroy_barrier	destroy barrier for imported segment
rsm_memseg_import_init_barrier	create barrier for imported segment
rsm_memseg_import_open_barrier	remote memory access error detection functions
rsm_memseg_import_order_barrier	impose the order of write in one barrier
rsm_memseg_import_set_mode	set mode for barrier scoping
Event operations	
rsm_intr_signal_post	signal for an event
rsm_intr_signal_wait	wait for an event

5.1.1 Remote Shared Memory Structure

Communication under the RSMAPI involves two basic steps: 1. segment *setup* and *teardown*; 2. the actual data transfer operations using the direct read and write models [2][31]. In essence, an application process running on one node, for example process 0 in Figure 5.1, should first create an RSM export segment from its local address space, and publish it to make it available for other processes. Then one or more remote processes running on remote nodes, for example process 1 in Figure 5.1, create an RSM import segment with a virtual connection between the import and export segments across the memory-based interconnect in order to connect to them. This is called the *setup* step. After the establishment of connection between the export and import segments, the processes can communicate with each other by writing into and reading from the memory. This is called the data transfer phase. After the data are successfully transferred, the last step is to tear down the connection between the export process and the import process. The import side process disconnects the connection and the export side process unpublishes the segments and destroys the memory handle. The RSMAPI provides barrier and signal notification mechanisms to support remote access error detection and synchronization [45].

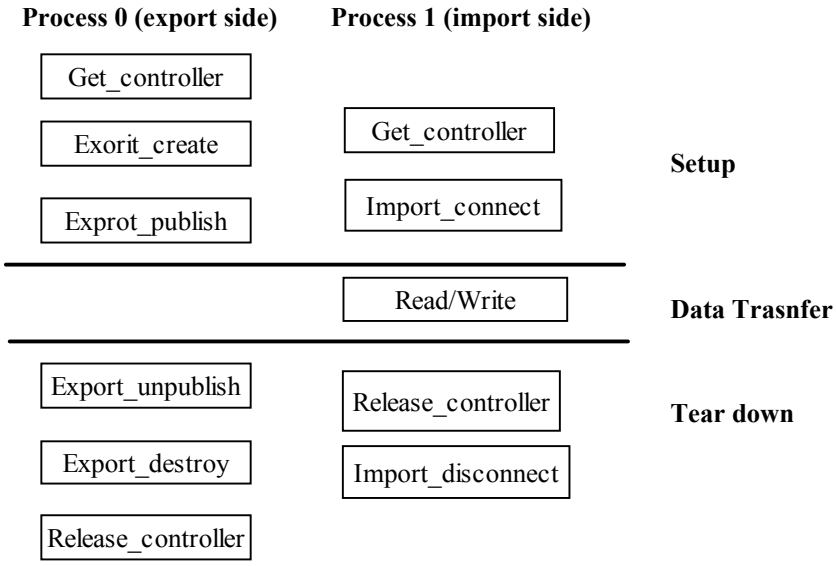


Figure 5.1. Setup, Data transfer, and Tear down in Remote Shared Memory communication.

The “Import” side can use three different methods, namely “*map*”, “*put*”, and “*get*”, to read or write data. The *“map”* method uses CPU block store operation to write to the address, which is mapped to the exported memory segment from the other process. “*put*” and “*get*” writes to and reads from the exported memory segment through the connection, respectively. Figure 5.2 illustrates the main steps for each of the three approaches. The *barrier* operations ensure the data transfers are successfully completed. The *signal* operation is used to inform the “Export” side that the “Import” side has written something onto the exported segment. *Barrier* and *signal* operations may or may not be used for small messages, because of the high overhead of those operations.

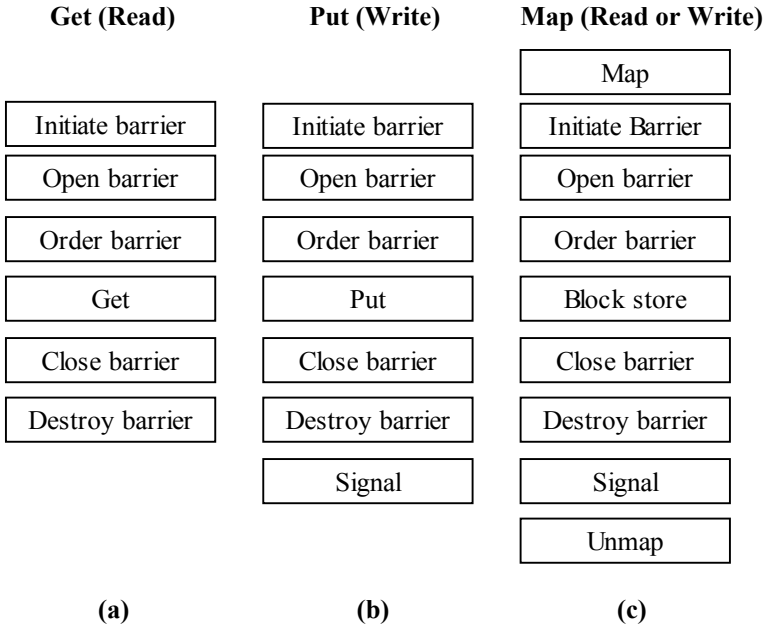


Figure 5.2. Different steps in the data transfer phase. (a) get (b) put (c) map.

5.1.2 Performance at the Remote Shared Memory level

The RSMAPI is the closest layer to the Sun Fire Link, so I would like to test the performance of RSMAPI calls, (described in Table 5.1), with varying parameters over the Sun Fire Link interconnect. Communication of data using RSMAPI involves two processes running on two different nodes, one as the export side and the other as the import side.

Table 5.2 shows the execution time of several RSMAPI calls. For those that are affected by the size of memory segment, 16 KB memory size is used. The *get_controller* takes the

longest, 841 μs. Before the export side can be accessed by the import side, it needs to execute *get_interconnect_topology*, *export_create* and *export_publish* primitives. This takes 1.7 ms long. This is quite long compared to a *put* operation. Similarly, the import side needs to run *get_controller* and *import_connect* primitives, which takes 1.0 ms long. Figure 5.3 shows that the "connect" and "disconnect" calls use more than 80% of the total communication time at the import side. To get a better performance, these preparations cannot be done for every communication. In the Sun MPI implementation, they are done in the initialization phase. The time for barrier operations, such as open, close and signal are also not small compared to the time to "put" a small message size. That is why explicit barrier is not used for small message size transaction.

Figure 5.4 shows the execution time of *export_create*, *export_publish*, *export_unpublish*, *export_destroy* and *release_controller* with different memory segment size. The *export_publish* is most sensitive to the segment size. Figure 5.5 shows the performance of *put* and *get*. To write a message smaller than 64 bytes, the *put* takes 35 μs long, but it takes only 0.6 μs long for a 64-byte message. That is why Sun MPI writes a complete 64 bytes to the postbox even if the message size is smaller than 64 bytes. The *put* has a much better performance than the *get*, except for message sizes smaller than 64 bytes. To *get* 16 Kbytes, it takes 373 μs long, while *put* takes only 28 μs long for the same message size. That is why Sun MPI implementation uses *put* rather than *get*.

Table 5.2. Execution time for RSMAPI calls. 16KB memory size is used for "export_create", "export_publish", "export_unpublish", "export_destroy", "release_controller", and "import_put".

Export side	Time(μs)	Import side	Time(μs)
get_interconnect_topology	12.65	import_connect	173.45
get_controller	841.00	import_map	13.56
free_interconnect_topology	0.61	import_init_barrier	0.33
export_create	103.61	import_set_mode	0.38
export_publish	119.36	import_open_barrier	9.93
export_unpublish	73.48	import_order_barrier	16.80
export_destroy	16.73	import_put	27.73
release_controller	3.63	import_close_barrier	7.13
		import_destroy_barrier	0.14
		import_signal	23.78
		import_unmap	21.40
		import_disconnect	486.31

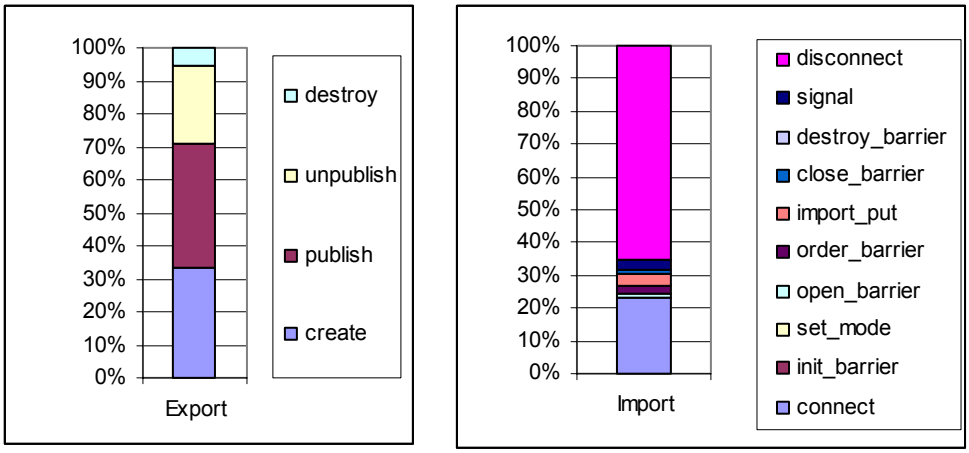


Figure 5.3. Percentage comparison for the export and import side. (16 KB)

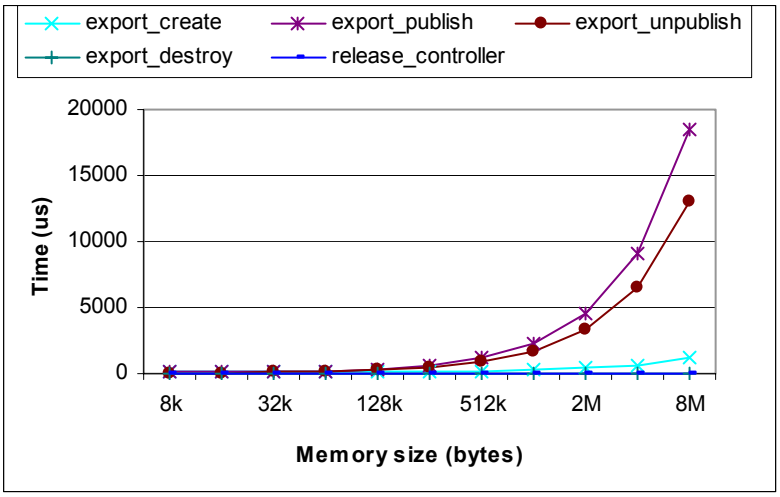


Figure 5.4. Execution times of several RSMAPI calls.

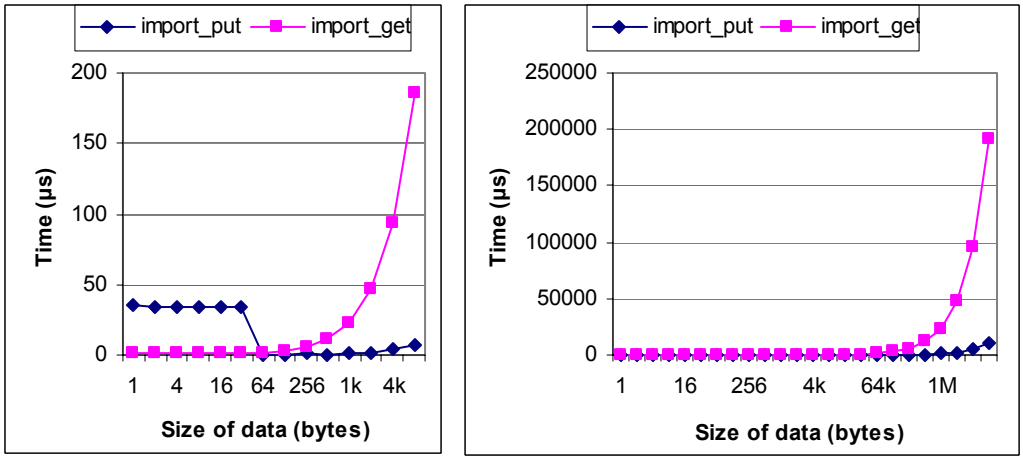


Figure 5.5. Comparison of the RSM put and get with different message sizes.

5.1.3 MPI Implementation over Remote Shared Memory

In this section, I look at how MPI_Send and MPI_Recv primitives have been implemented in Sun MPI over RSM. Because the segment setup and teardown have quite large overhead as seen in Section 5.2, Sun MPI establishes several logical connections to each node when the program starts. Each connection is also called a *stripe*.

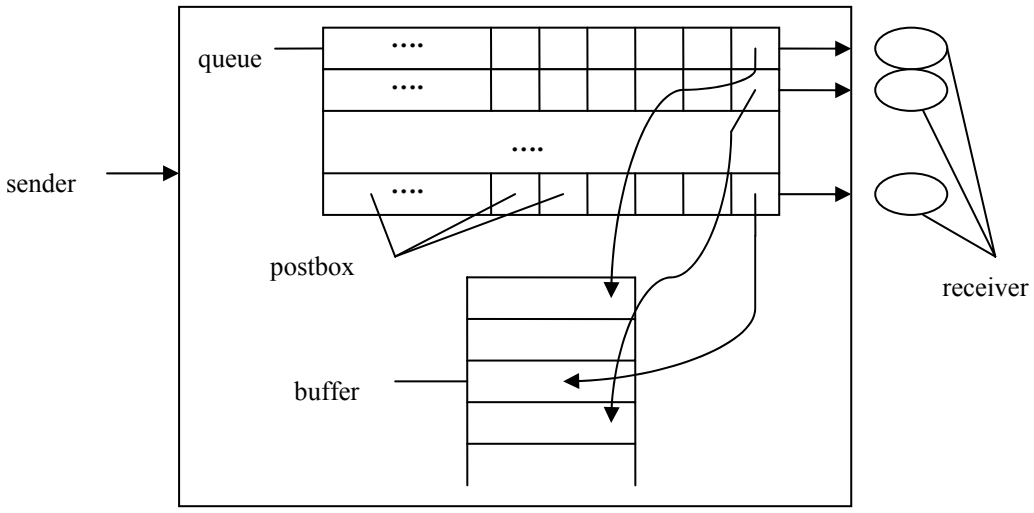


Figure 5.6. Structure of messages.

Messages are sent in one of two fashions short messages (smaller than 3912 bytes) and long messages [31]. Short messages are fit into multiple postboxes, 64 bytes each, because remote memory accesses are provided only to full 64-byte cache lines. Buffers, barriers, and signal operations are not used due to the high overhead. Note that even for messages smaller than 64 bytes, a full 64-byte postbox is used. Long messages are sent in 1024-byte buffers under the control of multiple postboxes, shown in Figure 5.6. Each postbox in the queue has the pointer to point to the buffer. These postboxes are sent to receiver who can recover the messages from buffers. Barriers are opened for each stripe to make sure the writes are successfully done.

The environment variable MPI_POLLALL can be set to “1” or “0”. In the *general polling* (MPI_POLLALL=1), Sun MPI polls for all incoming messages even though their corresponding receive calls may or may have not been posted earlier. In the *directed polling*

(MPI_POLLALL=0), the implementation searches only for the specified connection. Directed polling has a better performance when the user's code is perfectly organized.

To write the data, the "rsm_memseg_import_put" API call or the CPU block store operation can be used at the sender side, where "put" has a better performance than the "get". Figure 5.7 shows the flowchart for MPI_Send and MPI_Recv primitives, which are using the CPU block store operation to write data.

When "put/get support" is not defined, block store operations are used in MPI_Send. There are three block store operations which can be used, *atomic_copy*, *atomic_copy64*, and *cheetah_copy*. *Atomic_copy* can write any length of data. The *atomic_copy64* is used to write only 64-byte cache lines. The *cheetah_copy* perform the *prefetch* technique so it has to write data more than two cache lines. *Prefetch* is a technique that attempts to minimize the time a processor spends waiting for instructions to be fetched from memory. It will try to fetch next memory copy instruction before the previous copy completes.

For small messages, atomic_copy64 is used to write postboxes (64 bytes) directly. Long messages are divided and copied into buffers. For each buffer, if there are more than 8 cache lines data to write, *cheetah_copy* is used, where prefetching can provide better performance. Otherwise, data will be written using *atomic_copy64* with 64 bytes at a time. Figure 5.8 shows when each block store operation is used.

```
if send to itself
        copy the message into the buffer
else if general poll
                exploit the progress engine
        endif
        establish the forward connection (if not done yet)
        if message < short message size (3912 bytes)
                set envelop as data in the postbox
                write data to postboxes
        else if message < rendezvous size (256 KB)
                        set envelop as eager data
                else
                        set envelop as rendezvous request
                        wait for rendezvous Ack
                        set envelop as rendezvous data
                endif
                reclaim the buffer if message Ack received
                prepare the message in cache-line size
                open barrier for each connection
                        write data to buffers
                close barrier
                write pointers to buffers in the postboxes
        endif
endif
```

(a) MPI_Send pseudo-code

```
if receive from itself
        copy data into the user buffer
else if general poll
                exploit the progress engine
        endif
        establish the backward connection (if not done yet)
        wait for incoming data, and check out the envelope
        switch (envelope)
                case: rendezvou request
                        send rendezvous Ack
                case: eager,    rendezvou data, or postbox data
                        copy data from buffers to user buffer
                        write message Ack back to the sender
        endswitch
    endif
```

(b) MPI_Recv pseudo-code

Figure 5.7. Pseudo-nodes for (a) MPI_Send, (b) MPI_Recv.

```
If for each buffer there are more than 8*64 data to write
        cheetah_copy
        atomic_copy64 the leftovers
else
        atomic_copy64
```

Figure 5.8. Block store opertaioins.

5.2 Summary

In this chapter, I studied the RSM model, along with the performance of its API calls. However, it should be pointed out that the performance seen at the user-level may not be delivered at the higher levels. In the next chapter, I will evaluate the performance of Sun Fire link interconnect and Myrinet at the micro-benchmark levels. Then I can find exactly how much overhead is added by the MPI implementation on top of the user-level protocol.

Chapter 6

SMP Clusters' Performance at the Micro-benchmark and Application Levels

In chapter 5, I introduced the Remote Shared Memory model. However, applications may run over another layer on top of the user-level. For our NAS benchmarks, this layer is the MPI layer. I have studied how Sun MPI point-to-point communications are implemented on top of RSM. Now, it is the time to find the actual overhead added by the Sun MPI messaging layer. In this chapter, I evaluate the performance of two popular interconnects at the micro-benchmark level: one is the Sun Fire Link interconnect and the other is the Myrinet. I also provide the performance of our Myrinet cluster at the application level. The results presented in this chapter include point-to-point latencies and bandwidths, parameterized LogP performance, different traffic patterns bandwidths, collective communications, and application benchmarks.

6.1 Cluster Platforms

I would like to test the performance on two clusters of SMPs, one is four Sun Fire 6800 nodes interconnected by the Sun Fire link interconnect, and the other is eight Dell PowerEdge 2650 nodes interconnected by the Myrinet. The Sun Fire 6800 has 24 UltraSPARC-III Cu processors and 24 GB of shared memory. The Sun Fire 6800 nodes offer a flat memory system, such that all memory locations approximately have the same distance to each processor. The Sun MPI library uses the Remote Shared Memory (RSM) model to implement high performance messaging protocol between the nodes, and uses shared memeory model

within the nodes. Each Dell PowerEdge 2650 node has two Intel Xeon MP Processors running at 2 GHz with 64-bit PCI-X slots for Myrinet interconnect. I used the MPICH-GM 1.2.5..12 as the message passing library on top of GM 2.1, which has been very recently released as the user-level message-passing system for the Myrinet networks.

6.2 Latency

I present some measurements similar to the ones introduced in chapter 4. The measurements have been done for on-node communication, as well as for off-node communication. I also measure the MPI latency under load when the sending and receiving nodes are on different nodes. I do the average standard ping-pong latency test when simultaneous messages are in transit between pairs of send and receive processes. Note that, the send and receive nodes are on different nodes, and pair-wise processes are spread evenly across different nodes in the cluster.

In our experiments on the Sun Fire Link, I have used the default environment variable MP_POLLALL equal to 1. Figure 6.1 shows the latency for on-node pair-wise communication on Sun Fire Link. The latency for 1-byte message is 2 μs for unidirectional ping, 3 μs for *Standard, Ready, Buffered, Synchronous,* and the *Diff buf* modes. For the *unidirectional*, it remains at 2 μs up to 64 bytes, and for *bidirectional ping*, is almost constant at 3 μs. It is clear that the *buffered mode* has a higher latency for larger messages. It is interesting to see that for messages up to 1KB the latency is 5 μs for the *unidirectional ping*, and between 7 to 9 μs for *bidirectional.* The latency for on-node communication on Myrinet is shown in Figure 6.2. The on-node latency for 1-byte message remains at 1.3 μs for *unidirectional ping*, and 1.6 μs for *Standard, Ready, Synchronous, Diff buf* modes, and 2.5 μs for the *buffered* mode.

Figure 6.3 shows the latency for off-node communication on Sun Fire Link, where the endpoints are on different SMP machines. Quite interestingly, the latency for 1-byte message remains at 2 μs for *unidirectional ping*, but 5 μs for *Standard, Ready, Synchronous, Diff buf* modes, and 6 μs for the *buffered* mode. This is almost fixed up to 64 bytes. Figure 6.3 shows

that the MPI uses the short message method for up to 3912 bytes messages and then switches to the long message method for larger messages. For off-node communications, the *diff buf* has slightly worse performance compared to others (except for the buffered mode). For large size messages, the off-node communications have similar performance with on-node communications.

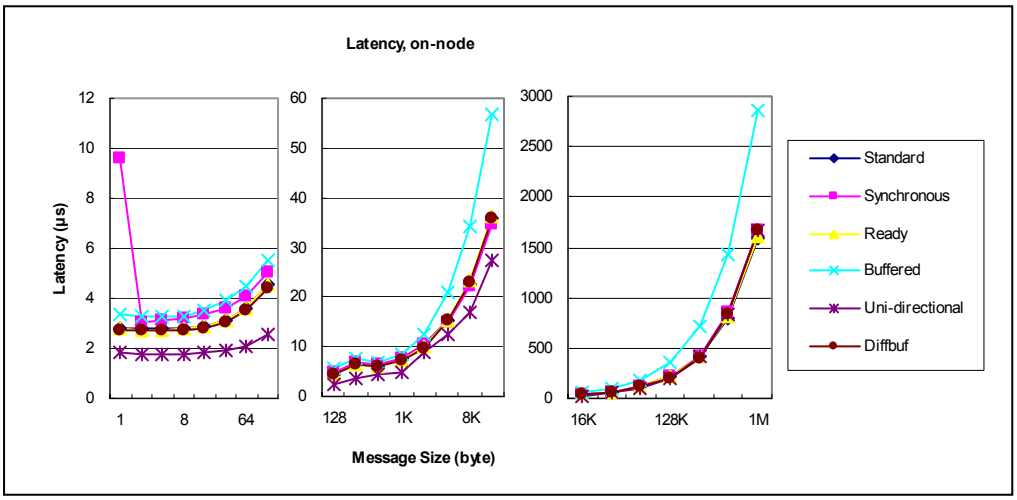


Figure 6.1. On-node MPI latencies on Sun Fire Link cluster.

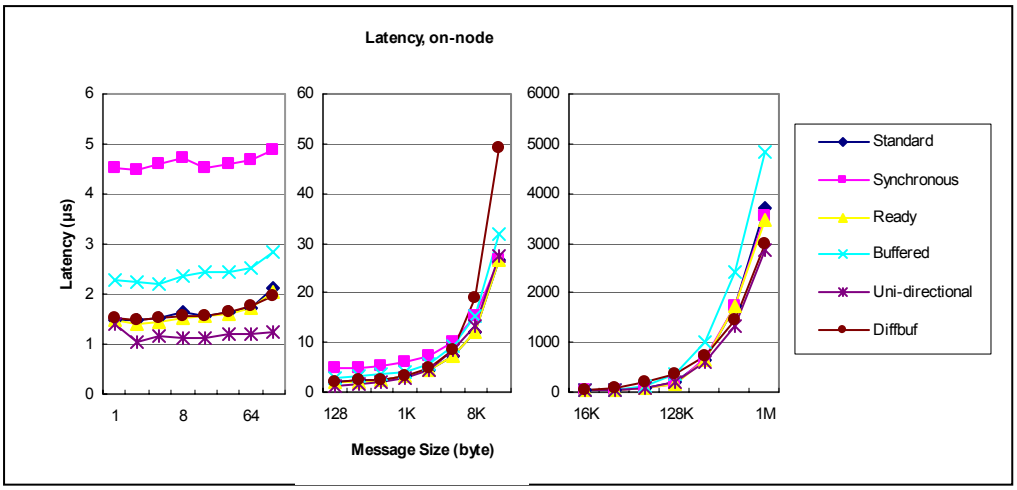


Figure 6.2. On-node MPI latencies on Myrinet cluster.

The latency for off-node communication on the Myrinet is shown in Figure 6.4. In the *synchronous* mode, the Myrinet shows quite large latency, equal to 23 µs for up to 64 byte message. Latency for 1-byte message remains at 5 µs for *unidirectional ping*, 6 µs for *Standard, Ready, Diff buf* modes, and 7 µs for the *buffered* mode. The off-node latencies for small size message on the Myrinet are a little bit larger than on the Sun Fire Link. For large size message,

off-node communications have better performance than on-node communications. Table 6.1 shows a summary of small message size latencies for on-node and off-node communications for both clusters. Table 6.2 compares the best reported short message latencies for MPI over Quadrics QsNet [27], and QsNet II [1], Myrinet D-card [27], Myrinet E-card [41], InfiniBand [27], Sun Fire Link [34], and our Sun Fire Link *. The performances of the Sun Fire Link interconnect for short messages are better than the results for MPI over QsNet and InfiniBand, but almost comparable to the results for QsNet II and Myrinet E-card. The performance of our Myrinet for short messages is not as good as reported in [41].

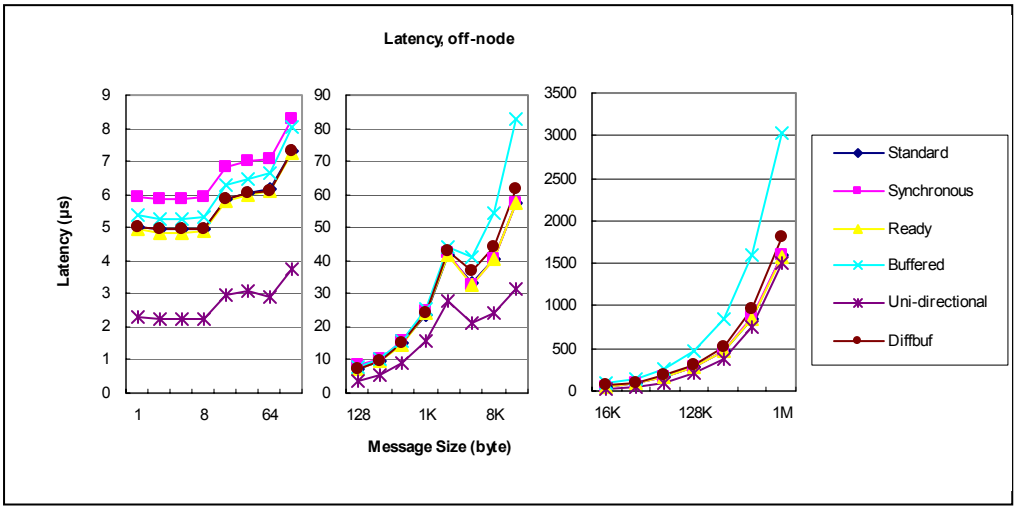


Figure 6.3. Off-node MPI latencies over Sun Fire Link.

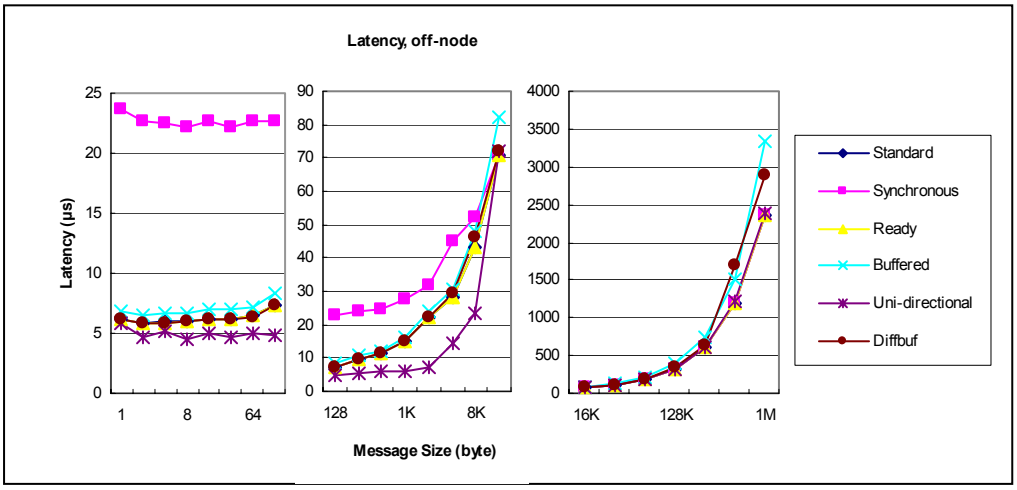


Figure 6.4. Off-node MPI latencies over Myrinet.

I also measured the average standard ping-pong latency test when simultaneous messages are in transit between pairs of send and receive processes, as shown in Figure 6.5. For each curve, the message size is held constant, while the number of off-node pairs is increased. The results are interesting as the latency in each case for both interconnects does not change much as the number of pairs is increased. The flatness of the curves verifies that Sun Fire Link and Myrinet interconnects deliver a robust low latency for short messages that is not sensitive to the load. For Myrinet, the latencies stay at 6 μs from 2-byte to 64-byte messages, running with up to 8 processes. Sun Fire Link stays at less than 6 μs up to 8-byte messages, 7μs for 16-byte messages, and more than 10 μs for 32-byte messages.

Table 6.1. Half-way ping-pong latency for small message sizes

Message Size (bytes)	Sun Fire Link		Myrinet	
	On-node (μs)	Off-node (μs)	On-node (μs)	Off-node (μs)
1	2.8	5	1.5	6.2
2	2.8	4.9	1.5	5.8
4	2.8	5	1.5	5.9
8	2.8	5	1.6	6.0
16	2.9	5.9	1.5	6.2
32	3.1	6	1.6	6.2

Table 6.2. Comparison of Sun Fire Link and Myrient short message latency (in microseconds) with other high-performance interconnects.

QsNet	QsNet II	Myrinet (D card)	Myrinet (E card)	Myrinet*	InfiniBand	Sun Fire Link (Sun)	Sun Fire Link*
4.6	~3	6.7	3.5	6.2	6.8	3.7	5

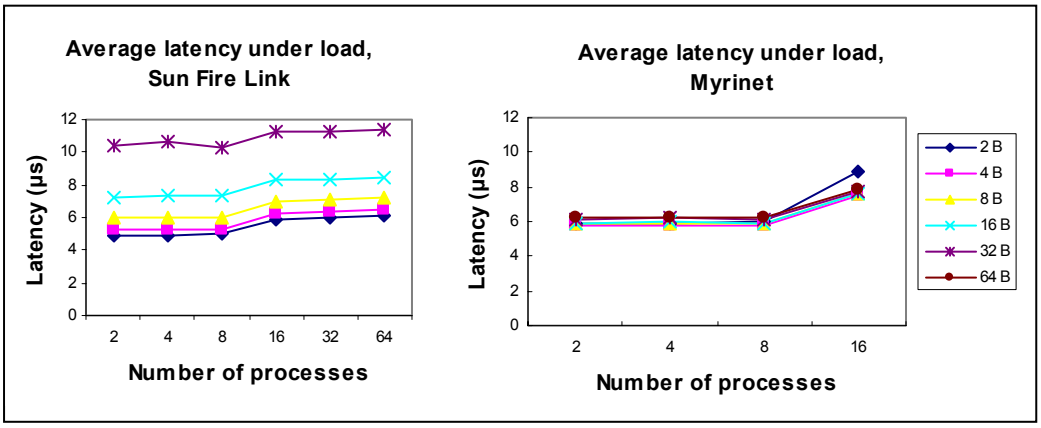


Figure 6.5. Off-node latency under load.

Figure 6.6 shows the overhead of the standard MPI ping-pong latency over the RSM *put* primitive. Note that I have assumed the same execution time for *put* with 1 to 64 bytes.

I measured the communication latencies under different modes on Sun Fire Link and Myrinet in this section. Myrinet has smaller on-node latencies for small size message, but the Sun Fire link has a better performance for off-node communications and for large size message.

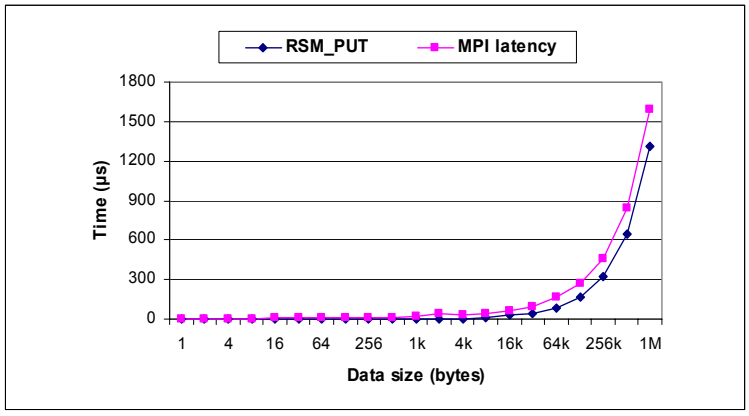


Figure 6.6. RSM put and MPI latency comparison.

6.3 Bandwidth

The same bandwidth tests, as in the chapter 4, are done here. Moreover, I measure the aggregate bandwidth when simultaneous messages are in transit between pairs of send and

receive processes. In this test, send and receiving nodes are on different nodes, and pair-wise processes are spread evenly across different nodes in the cluster.

Figure 6.7 presents the bandwidth for on-node communication on the Sun Fire Link and the Myrinet. The *unidirectional* bandwidth can be considered as the peak performance of the network, while sending packets in both directions may expose the network bottlenecks. For Sun Fire Link, the *unidirectional ping* achieves the highest bandwidth of 695 MB/s for internode communication. The network shows roughly similar performance for both *unidirectional* and *bidirectional* cases (except for the *buffered* mode). The *bidirectional ping* achieves a bandwidth of approximately 660 MB/s, except for the *buffered* mode, where it has the lowest bandwidth of 346 MB/s. This is clear as it has the overhead of buffer management. However, the *diff buff* has a better performance of 582 MB/s. It is interesting to see that for on-node communication, except for the *buffered* mode, all others achieve roughly the same maximum bandwidth. For Myrinet, one can see clearly that the on-node bandwidths arrive at maximum 700 Mbyte/s around 4 Kbytes. Except for *diffbuf* mode, the bandwidths start to drop from 64 Kbytes. This shows that either MPICH-GM does not have good implementation for on-node large size messages, or the memory system cannot handle such large size messages.

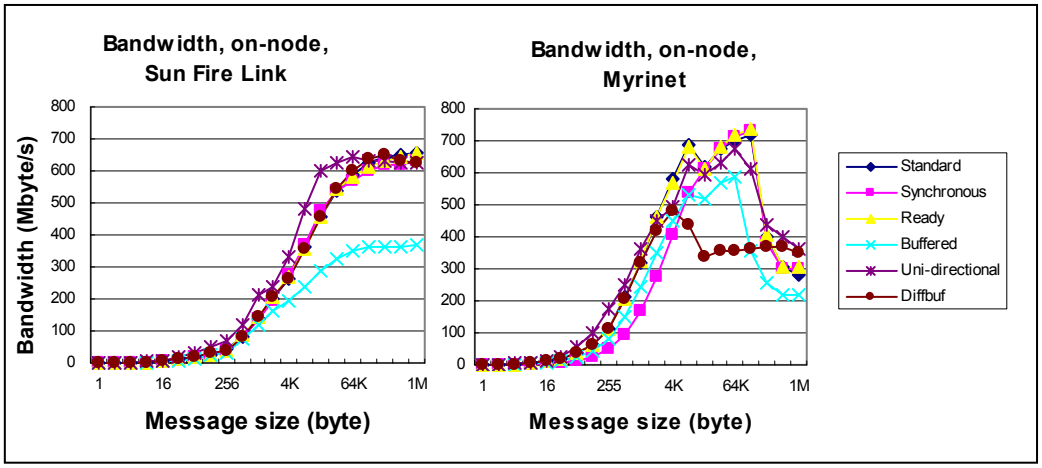


Figure 6.7. On-node bandwidths.

The bandwidths for off-node communications on the Sun Fire Link and Myrinet are shown in Figure 6.8. For Sun Fire Link, the transition point between the short and long

message protocols is at the 3912 bytes message size. Except for *buffered* mode, both *unidirectional ping* and *bidirectional ping* achieve around 600 MB/s. For Myrinet, the *bidirectional ping* except for the *buffered* mode, achieves a bandwidth of approximately 440 MB/s, which is not as good as the Sun Fire Link. The *buffered* mode, where it has the lowest bandwidth of 360 MB/s is similar to the Sun Fire Link. Table 6.3 shows a summary of maximum *bidirectional* bandwidths for on-node and off-node communications. Table 6.4 compares the reported bandwidths for MPI over Quadrics QsNet [27], and QsNet II [1], Myrinet D-card [27], Myrinet E-card [41], InfiniBand [27], Sun Fire Link [34], and our Sun Fire Link *.

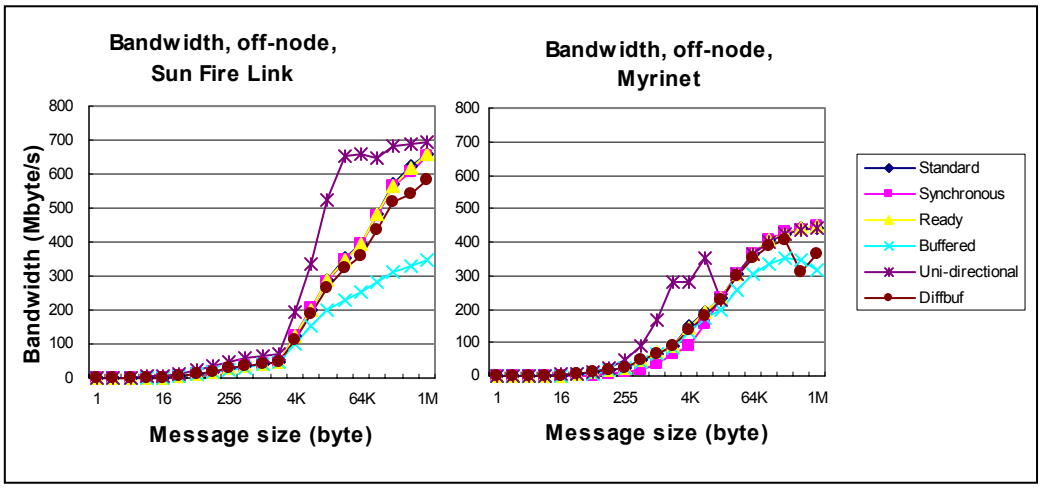


Figure 6.8. Off-node bandwidths.

Table 6.3. Bidirectional bandwidth

	Sun Fire Link		Myrinet	
	On-node	Off-node	On-node	Off-node
Bandwidth (MB/s)	658.6	659.5	720.4	443.9

Table 6.4. Comparison of Sun Fire Link and Myrinet MPI bandwidths (Mbytes/s) with other high-performance interconnects.

QsNet	QsNet II	Myrinet (D card)	Myrinet (E card)	Myrinet*	InfiniBand	Sun Fire Link (Sun)	Sun Fire Link*
308	900	235	495	444	841	792	695

Figure 6.9 shows the aggregate bandwidth with a standard bidirectional ping-pong test for varying number of communicating pairs, on the two interconnects. The aggregate bandwidth is the sum of individual bandwidths. Again, the inflection point is shown at the 3912 bytes message size for the Sun Fire Link in Figure 6.8. From 256kB message size, rendezvous protocol is used, where the sender needs to wait for the acknowledge message from receiver. Figure 6.8 also shows that up to 256kB message size, the network is capable of providing higher bandwidth with increasing number of communication pairs. However, with 256kB message size and above, aggregate bandwidth is higher for 16 pairs of communication than for 32 pairs. For 16 processes, Myrinet achieves the maximum bandwidth 2500 Mbyte/s.

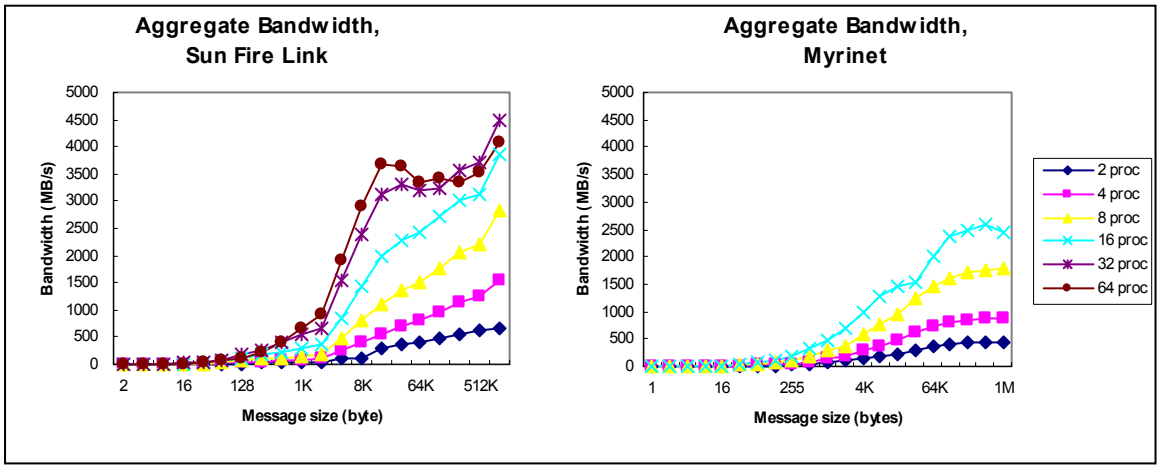


Figure 6.9. Aggregate off-node bandwidth.

6.4 LogP Parameters

Traditionally, message latency and bandwidth have been used for measuring the network performance. However, LogP model provides greater detail about different component of a communication step [14]. It shows how much time is spent by the processor at the sending side, o_s, and by the processor at the receiving side, o_r. It also provides the network hardware latency, L, and the gap in between consecutive sends, $g(m)$.

In [25], the authors proposed the *LogP* model for parallel computation and parallel algorithm development. LogP parameters have been proposed to gain insights into different

components of a communication step. Essentially, it captures the relevant aspects of message passing systems. It considers the communication cost as well as the cost for integrating communication into computation. LogP models sequences of point-to-point communications of short messages. *L* is the network hardware latency for one-word message transfer. *O* is the combined overhead in processing the message at the sender (o_s) and receiver (o_r). *P* is the number of processors. The gap, *g*, is the minimum time interval between two consecutive message transmission from a processor. *LogGP* model [3] extends LogP to also cover long messages. The Gap per byte for long messages, *G*, is defined as the time per byte for a long message.

An efficient method for measurement of LogP parameters has been recently proposed in [25]. The method is called parameterized LogP, shown in Figure 6.10, where it subsumes both LogP, and LogGP models. This model defines five parameters (L, o_s, o_r, g, P). In this model, the latency, *L*, is the end-to-end latency from a process to another process, combining all contributing factors. The most significant advantage of this method over the method introduced in [22] is that it only requires saturation of the network to measure *g(0),* the gap between sending messages of size zero. For a message size *m*, the latency, *L*, and the gaps for larger messages, $g(m)$, can be calculated from *g(0),* and round trip times, *RTT(m),* as shown in Equation 1, and Equation 2. Table 6.5 defines LogP/LogGP parameters in terms of parameterized LogP.

$$L = \frac{RTT(0)}{2} - g(0) \quad (1)$$

$$g(m) = RTT(m) - RTT(0) + g(0) \quad (2)$$

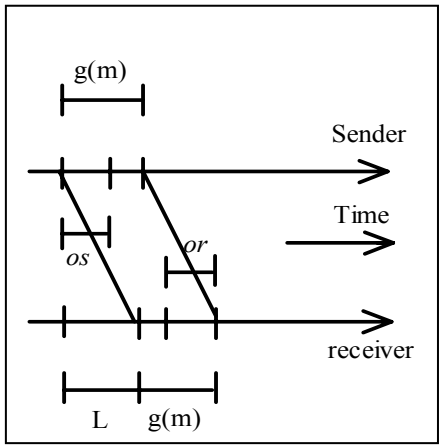
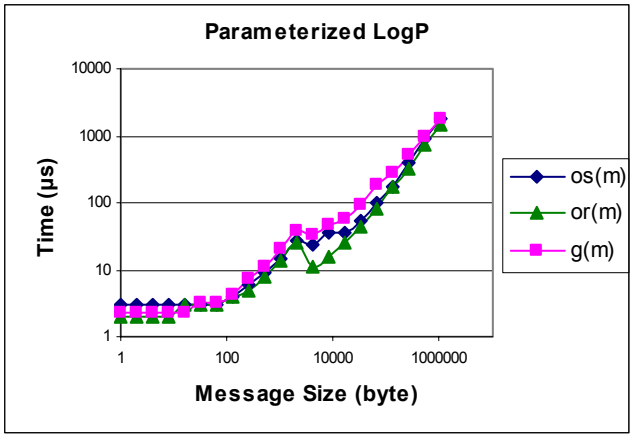


Figure 6.10. Message transmission modeled by parameterized LogP.

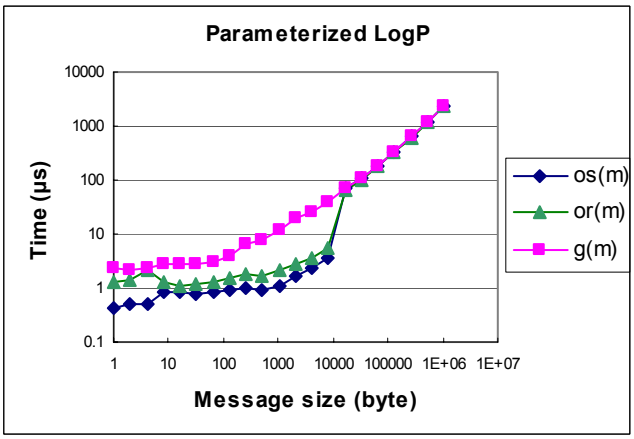
Table 6.5. LogP/LogGP parameters in terms of parameterized LogP.

LogP	Parameterized LogP
L	$L + g(1) - o_s(1) - o_r(1)$
O	$\frac{o_s(1) + o_r(1)}{2}$
G	$g(1)$

We applied the parameterized LogP method to our Sun Fire cluster and Myrinet cluster. We used two processes on different SMP nodes. The o_s, o_r, and g for Sun Fire Link are drawn in Figure 6.11(a). It is interesting to see that all three parameters, o_s (3 µs), o_r (2 µs), and g (2.29 µs), remain fixed for the message sizes of one to 64 bytes. However, they increase with larger messages sizes (except with a decrease at 3912 bytes). It seems to us that probably the network interface is not quite efficient as the CPU has to do more work with larger message sizes, both at the send and at the receiving sides. The decrease at 3912 bytes message size is related to switching from the short message protocol to long message protocol. Figure 6.11(b) shows the o_s, o_r, and g for the Myrinet. All three parameters, o_s (0.9 µs), o_r (1.1 µs), and g (2.9 µs), also remain fixed for the message sizes of one to 64 bytes. Both o_s and o_r have a sudden increase at 16 Kbytes. The two interconnects show similar *gaps*, but Myrinet cluster has smaller o_s and o_r for small messages. Table 6.6 shows the LogP parameters.



(a)



(b)

Figure 6.11. LogP parameters, g(m), os(m), and or(m). (a) Sun Fire Link (b) Myrinet.

Table 6.6. LogP parameters.

LogP	Sun Fire Link (us)	Myrinet (us)
L	0.51	4.2
o	2.5	0.8
g	2.29	2.3

6.5 Traffic Patterns

In the previous section, I analyzed the performance under specific conditions to get the "peak" performance of the networks. Even in the "under load" experiments, the destination of each process to communicate with and the size of message are fixed. In this section, I expand the experiments with some different conditions; that is, different message sizes and different destinations. In these experiments, we analyze the network performance under several different traffic patterns, where each node selects a random or fixed destination for its

transactions. These communication patterns are mostly representative of parallel numerical algorithm behavior found in scientific applications [31]. We generate random message size and inter-arrival time between two transactions with two different distributions: uniform and exponential. Note that these patterns may generate both on-node and off-node traffic.

6.5.1 Uniform Traffic

The uniform traffic is one of the most frequently used traffic patterns for evaluating the network performance [31]. Each node selects its destination randomly with uniform distribution.

6.5.2 Permutation Patterns

In these traffic patterns, each node communicates with a fixed destination process. We experiment with the following permutation patterns:

- *Bit-reversal*. The process with binary coordinates $a_{n-1}, a_{n-2}, ..., a_1, a_0$ always communicates with process $a_0, a_1, ..., a_{n-2}, a_{n-1}$.
- *Butterfly*. The *i*th butterfly permutation is defined by

  $\beta_i(a_{n-1}, ..., a_{i+1}, a_i, a_{i-1}, a_1, a_0) = a_{n-1}, ..., a_{i+1}, a_0, a_{i-1}, ..., a_1, a_i$ $(0 \le i \le \text{n-1})$.
- *Complement*. The process with binary coordinates $a_{n-1}, a_{n-2}, ..., a_1, a_0$ always communicates with process $\overline{a_{n-1}, a_{n-2}, ..., a_1, a_0}$.
- *Matrix transpose*. The process with binary coordinates $a_{n-1}, a_{n-2}, ..., a_1, a_0$ always communicates with process $a_{\frac{n}{2}-1}, ..., a_0, a_{n-1}, ..., a_{\frac{n}{2}}$.
- *Perfect-shuffle:* The process with binary coordinates $a_{n-1}, a_{n-2}, ..., a_1, a_0$ always communicates with process $a_{n-2}, a_{n-3}, ..., a_0, a_{n-1}$.
- *Neighbor:* Processes are divided in pairs. Each pair consists of two adjacent processes. For example, process 0 with process 1, process 2 with process 3, …, and process n with

process n+1. In our system, two adjacent processes are in same node, so all traffic in the pattern are on-node traffic.

- *Cube:* The *i*th butterfly permutation is defined by

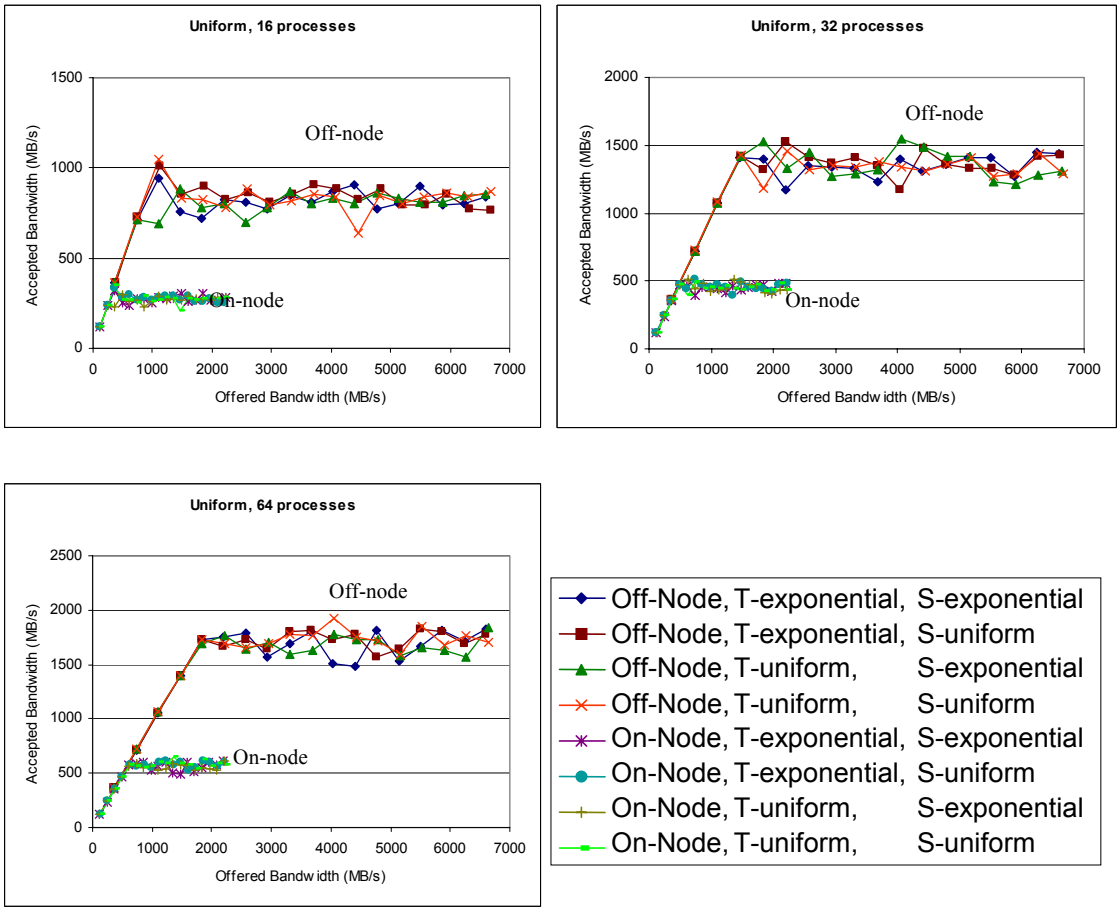
$$\beta_i(a_{n-1},...,a_{i+1},a_i,a_{i-1},...,a_0) = a_{n-1},...,a_{i+1},\overline{a_i},a_{i-1},...,a_0 \quad (0 \le i \le \text{n-1}).$$

- *Baseline:* The *i*th butterfly permutation is defined by

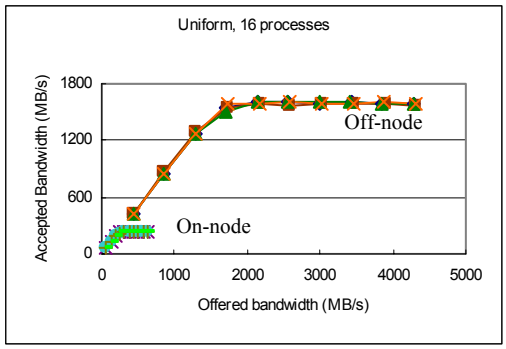
$$\beta_i(a_{n-1},...,a_{i+1},a_i,a_{i-1},...,a_1,a_0) = a_{n-1},...,a_{i+1},a_0,a_i,a_{i-1},...,a_1 \quad (0 \le i \le \text{n-1}).$$

6.5.3 Results

In the experiments, we consider uniform and exponential distributions for both message size (identified by "S" in the figures) and inter arrival time ("T"). We chose 10 Kbytes as the mean message size. From the results, the performance is not much sensitive to these distributions especially for the Myrinet. Figure 6.12 shows the accepted bandwidth under the traffic with uniform distribution. For the Sun Fire Link, the off-node accepted bandwidth can be up to around 2000 MB/s with 64 processes, 1500 MB/s with 32 processes, and around 900 MB/s with 16 processes. Compared with the aggregate bandwidth, Sun Fire Link achieves 2900 MB/s with 64 processes for 8 Kbytes message, 2400 MB/s with 32 processes and 1500 MB/s with 16 processes. It is clear that the Sun Fire link interconnect performance scales with the number of processes. For Myrinet, the off-node accepted bandwidths, shown in Figure 6.12(b), are staying almost the same for all message sizes and inter-arrival time with different distributions. The bandwidth is around 1600 MB/s with 16 processes, comparing to 1300 MB/s as the aggregate bandwidth for 8 Kbytes messages.



(a)



(b)

Figure 6.12. Uniform Traffic accepted bandwidth (a) Sun Fire Link, (b) Myrinet.

Butterfly, *Cube*, and *Baseline* have *single-stage* and *multi-stage* patterns. *Single-stage* is the highest stage permutation, and *multi-stage* is the full stage permutation. Figure 6.13 shows the accepted bandwidth of permutation patterns with 32 processes for Sun Fire Link. There is only off-node traffic for *Complement*, *multi-stage Cube* and *single-stage Cube* patterns. They achieve 3100 MB/s maximum accepted off-node bandwidth, which is similar

to 3128 MB/s aggregate bandwidth with 16 KB message size. *Neighbor* has only on-node traffic which stays at 4000 MB/s accepted bandwidth. There is half on-node traffic and half off-node traffic for *Butterfly single-stage*. The other traffic patterns have one fourth on-node and 3 fourths off-node traffic, where off-node accepted bandwidth stays at 2000 MB/s, and on-node bandwidth arrives at 4000 MB/s. *Matrix transpose* and *Butterfly multi-stage* permutations have similar performance, 2400 MB/s off-node bandwidth and up to 1200 MB/s on-node bandwidth. *Bitreversal* and *Baseline single-stage* (*Inverse perfect shuffle*) get 2500 MB/s off-node bandwidth, and 2000 MB/s on-node bandwidth. All those four patterns have similar on-node off-node ratio traffic, but behave differently. The reason is that for *Matrix transpose* and *Butterfly multi-stage* permutations, the on-node traffic are evenly assigned to two nodes, while in *Bitreversal* and *Baseline single-stage* (*Inverse perfect shuffle*), the on-node traffic are evenly assigned to four nodes. The results indicate that the on-node and off-node communications can be influenced by each other.

Figure 6.14 shows the accepted bandwidth with 16 processes for the Myrinet. *Complement* (Cube *multi-stage*) and *Neighbor* permutations have only off-node traffic with bandwidths close to 2000 MB/s. The aggregate bandwidth is 1260 MB/s for 8 Kbytes messages, 1460 MB/s for 16 Kbytes messages, and 1530 MB/s for 32 Kbytes messages, which indicates that Myrinet does not perform very well around 16 Kbytes. *Cube single-stage* permutation has only on-node traffic with bandwidth up to 3600 MB/s. *Bitreversal*, *Matrix transpose* and *Baseline single-stage* (*Inverse perfect shuffle*) permutations have one fourth on-node traffic, and three fourths off-node traffic. They have similar performance of 2000 MB/s off-node bandwidth and 1500 MB/s on-node bandwidth. *Butterfly single-stage* has half on-node traffic, and half off-node traffic. *Butterfly multi-stage* and *Perfect shuffle* have less than 1500 MB/s off-node bandwidth, and the other patterns have 2000 MB/s. The difference between them is that for *Butterfly multi-stage* and *Perfect shuffle* permutations, each process does not send message to and receive message from same process. In this experiment for Myrinet, the only on-node traffic is from the process 0 and process 15 both sending to and receiving from themselves. So the on-node traffic did not affect the performance of off-node

traffic. In general, the Myrinet cluster has comparable on-node performance to the Sun Fire cluster, while the Sun Fire cluster has better off-node performance.

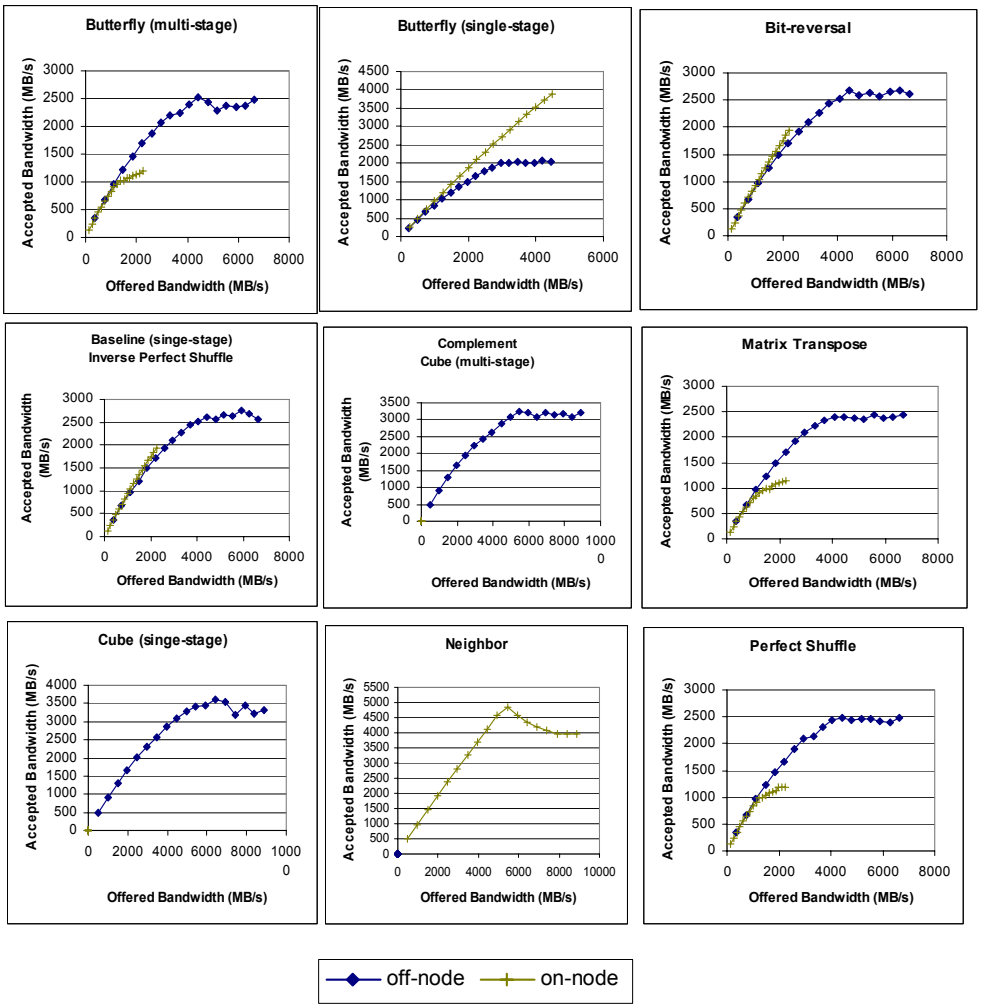


Figure 6.13. Permutation patterns accepted bandwidth (Sun Fire Link).

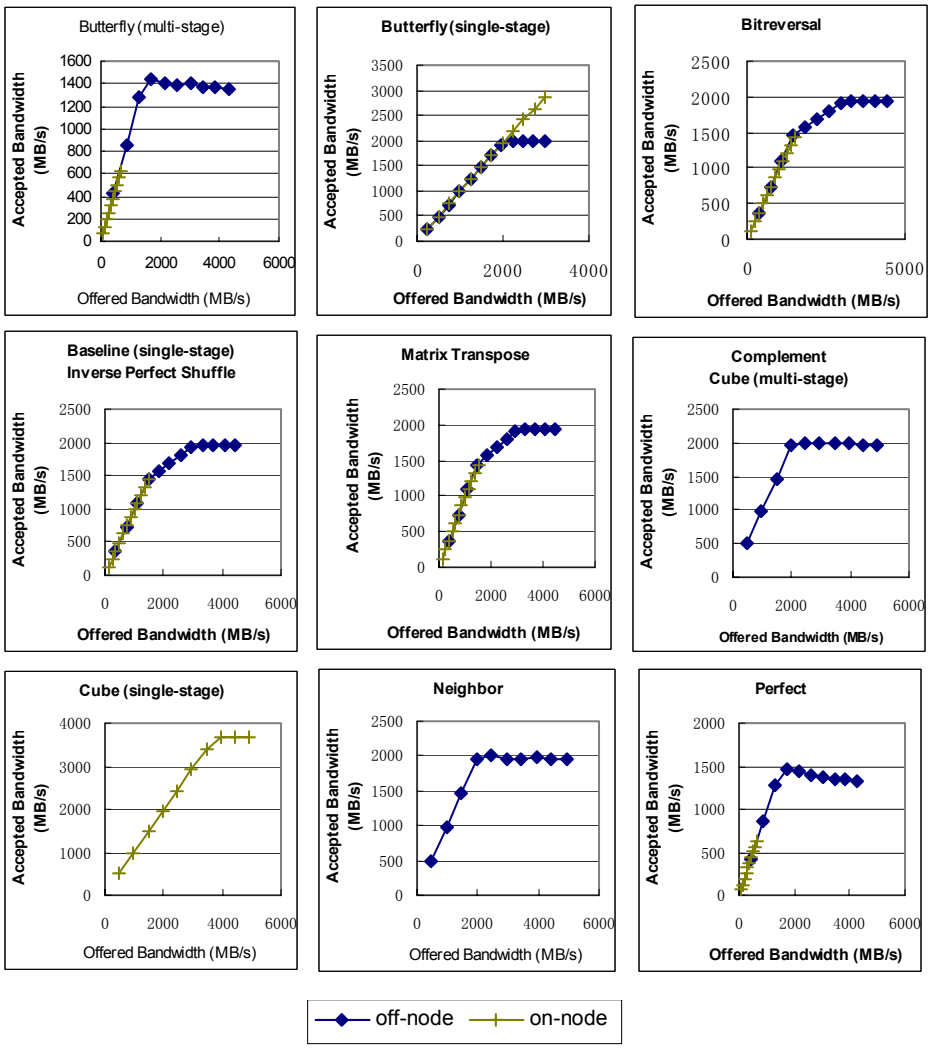
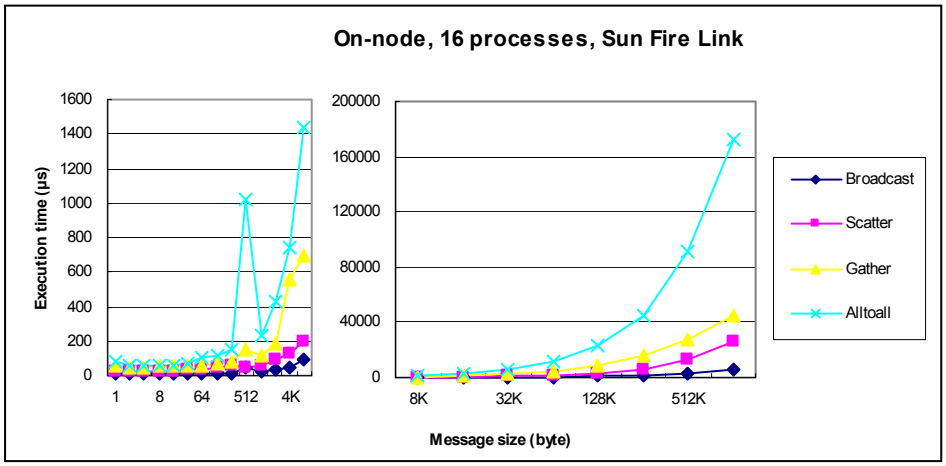
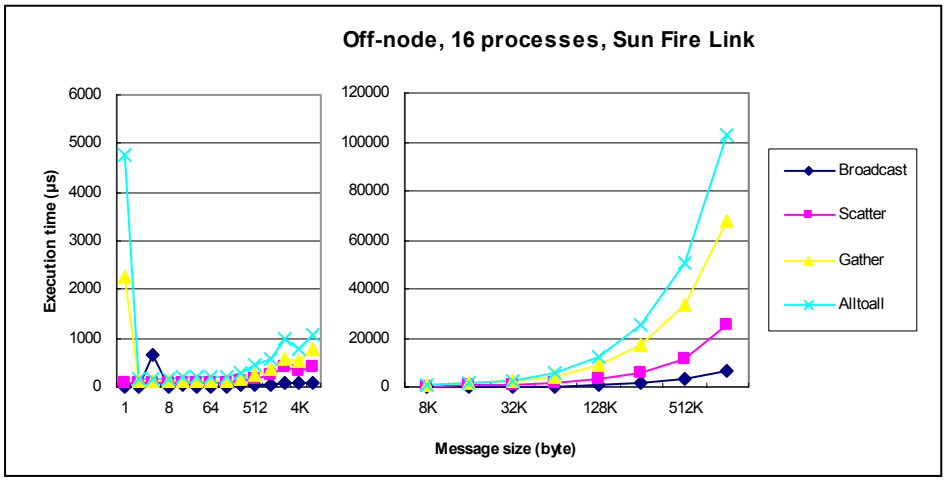


Figure 6.14. Permutation patterns accepted bandwidth (Myrient).

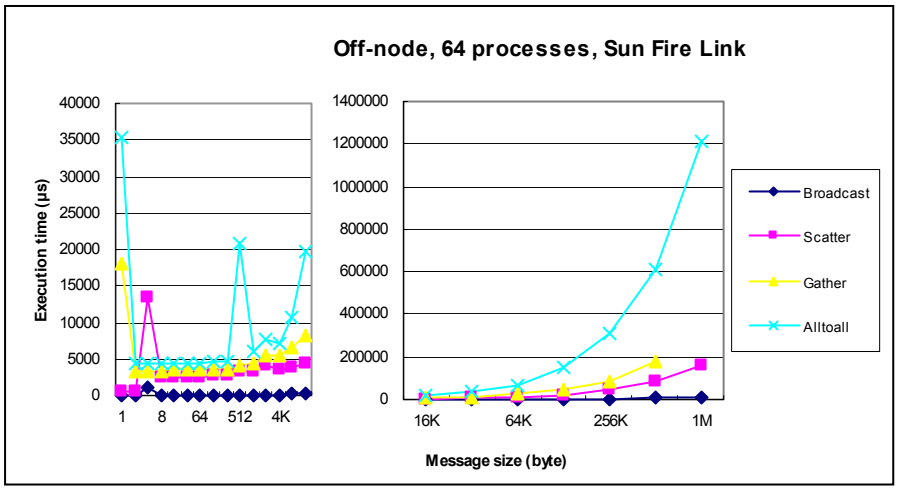
6.6 Collective Communications

In this experiments, I choose *broadcast*, *scatter*, *gather*, *alltoall*, and *barrier* operations as representatives of the mostly used collective communication primitives in parallel applications. Our experiments are done with processes located on the same node and/or on different nodes. In the off-node case, we evenly divided the processes among the 4 Sun Fire 6800 nodes.





(a)



(b)

Figure 6.15. Collective communication performance, Sun Fire Link. (a) 16 processes (b) 64 processes

I measure the performance in terms of their completion time on Sun Fire cluster and Myrinet cluster. I also obtain the performance of *alltoallv*, *reduce*, and *allreduce* operation on

Myrient cluster. An overall look at their running time shows that the *alltoall* operation takes the longest, followed by the *gather*, *scatter,* and *broadcast* operations, for Sun Fire cluster. This is true for on-node, as well as for off-node communications. For the Sun Fire 15K, *alltoall* and *gather* have a special long time for 1 byte, 4 bytes and 512 bytes message size. I have not found the reason of it.

In *broadcast*, for Sun Fire cluster, I see that for all cases, the on-node case performs better then the off-node case (except for 64K byte messages with 2 and 16 processes). The broadcasting has been highly optimized for on-node communications [35]. In all other cases, the on-node performance is better than the off-node performance. However, with very large message sizes of 64K and 1M bytes, the difference in performance gradually decreases. This is related to the degraded performance of shared-memory system on the Sun Fire due to multiple large messages in transit. Figure 6.16 shows the collective communication performance of Myrinet cluster. *Broadcast* again has the best performance, but *allreduce* has the largest completion time, followed by *alltoallv* and *reduce*. I do not know the reasons behind the spikes for Sun Fire Link in Figure 6.15. I run tests 1000 times and got the average. The spikes were present in all cases.

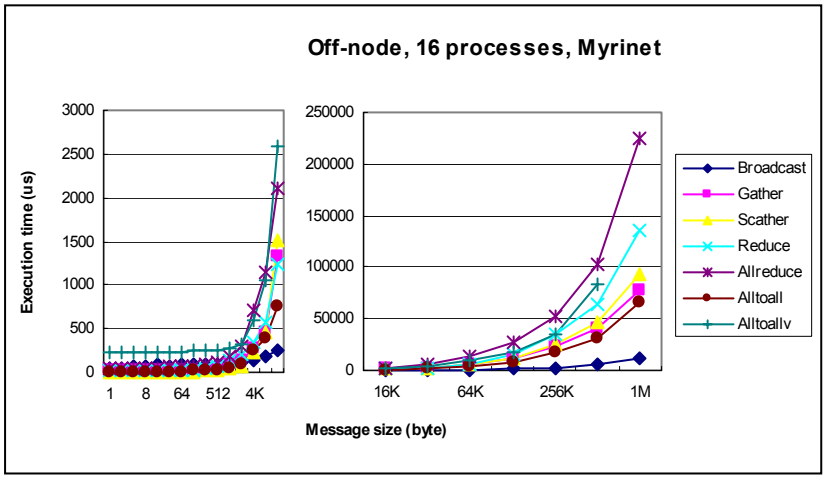


Figure 6.16. Collective communication performance, Myrinet.

Table 6.7 Barrier performance (microseconds)

# of processes	Sun Fire Link		Myrinet
	On-node	Off-node	Off-node
2	1.21	-	9.84
4	1.80	1156.05	19.12
8	1.80	1355.17	29.50
16	3.10	966.50	52.44
32	-	971.45	-
64	-	1785.60	-

Table 6.7 shows the time for barrier operation. For Sun Fire Link, there is a large performance gap between the on-node and the off-node performance, mostly because the barrier operation is highly optimized for an SMP node.

6.6 Application Benchmarks

I analyzed the performance of the 4-node Sun Fire cluster and the 8-node Myrinet cluster at the micro-benchmark level in the previous sections. Now I want to evaluate the performance of actual application benchmarks. I chose the NPB 2.3 benchmark suite with the two problem sizes, class A and class B, fro this study.

Legend "*n*×*m*" in the Figure 6.17 means running with *n* nodes and *m* processes in each node. For BT and SP, where they require square number of processes, legend "*n*×*m*+1" means running with *n* nodes and *m* processes in each node plus an extra process in one of the nodes.

Speedups for class A and class B are shown in Figure 6.17. Running with the same number of processes, the applications show better speedup when there are fewer processes in each node. For example "*2*×*1*" has a larger speedup than "*1*×2", and "*4*×*1*" has a larger speedup than "*2*×*2*". This is related to the poor performance of on-node communications for large message sizes. In the NAS benchmark suite, all benchmarks, except for EP and FT, have large number of long message sizes communications. That is why EP has similar speedup for any combinations with same total number processes. LU has more small size messages, so it is less influenced. Because FT requires more memory, it has less speedup having 2 processes in each node than the case with only one process per node. An extreme case is "*1*×2" for class B. Some serial programs take too long time, which makes them have super linear speedup, such as SP of class

A, and BT, CG, and FT of class B. In general all benchmarks have very good speedup for class A, close to linear.

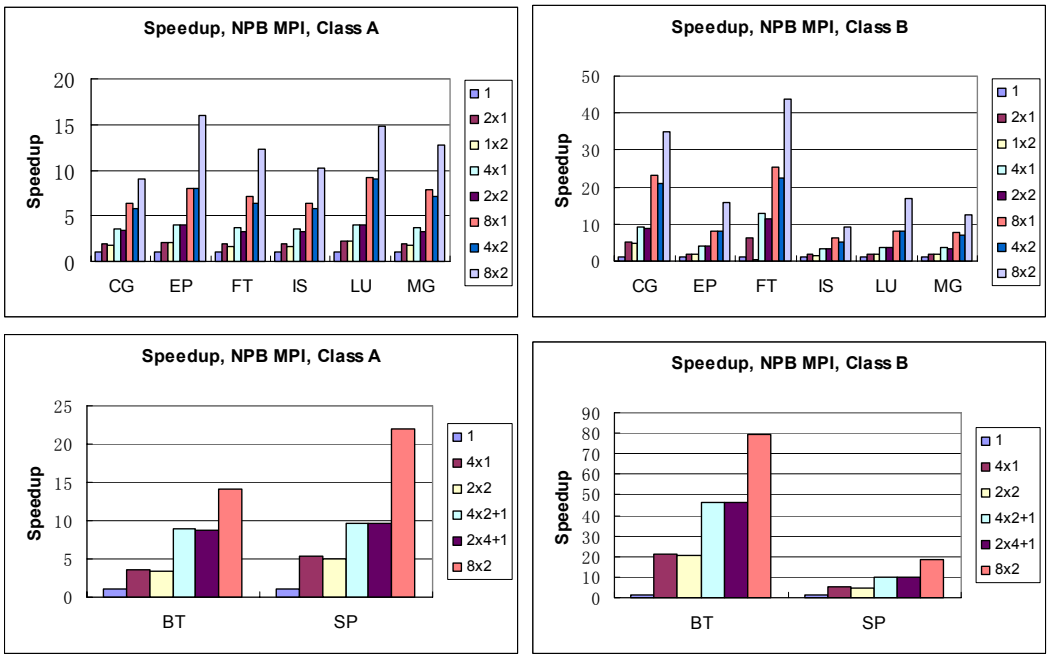


Figure 6.17. NAS benchmark performance, class A and class B.

6.7 Summary

In this chapter, I provided the performance at the MPI micro-benchmark level for two clusters, a 4-node Sun Fire 6800s interconnected by Sun Fire Link, and an 8-node Dell PowerEdge 2560s interconnected by Myrinet. The performance results include the traditional point-to-point latencies and bandwidths, latencies and bandwidths under load, and collective communications. Both network interconnects had relatively good performance. Sun Fire Link achieved 5 μs latency for small size messages, while Myrinet made 6 μs latency. The maximum bandwidth for the Sun Fire Link was 700 MB/s, while it was 444 MB/s for the Myrinet. The on-node performance of Myrinet dropped for messages larger than 64 Kbytes.

I also measured the bandwidth under different traffic patterns. Both interconnects are not sensitive to different distributions for message size and inter-arrival time of messages. For *Complement* permutation, the Sun Fire Link delivered around 3300 MB/s off-node bandwidth.

For *Neighbor* permutation, Myrinet delivered around 2000 MB/s off-node bandwidth. For collective communications on Sun Fire Link, the on-node performance was better than the off-node. For Myrinet, *allreduce* took the longest time, *broadcast* the shortest.

Finally, I provided the performance of NAS application benchmark for the Myrinet cluster. We found that Sun MPI has relatively good implementation for both on-node and off-node communications. MPICH-GM has degrading on-node performance for long messages, where it affects the performance of application benchmarks.

Chapter 7

Conclusion

In this thesis, I provided a complete measurement of the performance on the SMPs and cluster of SMPs. I looked into all factors which may affect the performance of parallel applications on those systems. I found all the results are reasonable.

Firstly, I discussed and characterized a popular parallel benchmark suite, NAS benchmarks suite. I looked at several communication parameters, including the message size, the number of messages, and destination distributions. I compared these characteristics among five different benchmarks from the NAS benchmark suite. I ran them under different number of processes and different problem sizes. I found that with larger problem size, all benchmarks have larger average message sizes, and larger number of messages. The newly released class D has much more communications and larger message size than the other problem sizes. All benchmarks have small number of message destinations, at most 10 destinations when running with 64 processes.

Along with the performance of the memory bandwidth, point-to-point latency and bandwidth at the MPI level, I presented the performance of the NAS parallel benchmarks on a small SMP and a large SMP. The Dell PwerEdge 6650 achieved around 10 μs for small size messages, while the Sun Fire 15K stayed at 6 μs. The performance of Dell PowerEdge 6650 degraded for messages larger than 64 Kbytes. Because the Sun Fire 15K has a better memory hierarchy system and better MPI implementation, it showed a very good scalability for the performance of the application benchmarks. I believe the performance of the Dell PowerEdge 6650 is affected by the lack of sufficient memory. I also compared the performance of NAS parallel benchmarks implemented with MPI, OpenMP and Java, on both SMP machines. For both machines, the MPI version had the best performance among the three versions.

I discussed the programming model of the Remote Shared Memory (RSM) user-level protocol and the Sun MPI implementation on top of it. The performance of RSM API calls was assessed, and compared with the performance of the Sun MPI. The *setup* and *tear down* steps took very long compared to the execution time of the actual data transfer. *put* had a better performance than *get* for messages larger than 64 bytes. These results explained how the Sun MPI is implemented on top of RSM.

I introduced a set of micro-benchmark suite at the MPI level to evaluate the performance of two interconnects: the Sun Fire Link interconnect and Myrinet. The suite includes not only the traditional point-to-point latency and bandwidth, but also the bandwidth under load, LogP parameters, different traffic patterns, and collective communications. They were useful in finding out the bottlenecks, giving insights about the MPI implementation, and analyzing the performance at the application level. Our performance results include the on-node and the off-node latency, bandwidth measurements under different communication modes. For Sun Fire Link interconnect, to write 64 bytes to a remote node took only around 0.6 μs at RSM level. The Sun MPI implementation achieved 2 to 5 μs for off-node latency. The difference between them is generated by the MPI implementation. The performance of on-node communications was better than off-node communications. But I found the on-node performance degraded for large message size on the Myrinet, which also affected the performance of the benchmark applications on the Myrinet cluster. All benchmarks had good speedups, close to linear. In general, the performance results indicated that the Sun Fire Link and Myrinet performed very well in most cases.

I observed that the low level implementations delivered better results, and the MPI implementations were adding large percentage overheads. I saw that extra copies were done in the data critical path. The implementations of some of the collective communications are not as good as the *broadcast* operation, which is highly optimized.

Generally, the MPI implementations on two SMPs were excellent. They showed better scalability than OpenMP and Java. Both interconnects, the Sun Fire Link and Myrinet, also presented relatively good performance (low latency and high bandwidth). However, the Sun

Fire Link interconnects had a better performance at the micro-benchmark level. The Myrinet cluster also performed well under the NAS parallel benchmark suite, except for some cases due to lack of enough memory. Knowing the performance at the RSM level, I discovered the MPI overhead. I believe there is room to improve the MPI implementation.

7.1 Future Work

The performance measurements in this thesis are mainly in MPI, OpenMP, and Java are also included for the experiments on SMPs. The mixed-mode (MPI+OpenMP) programming, may be suitable for the cluster of multiprocessor systems. It will be very interesting to compare its performance with pure MPI on such systems.

I am interested in measuring the performance of the GM messaging layer on top of the Myrinet both for the send/receive and RDMA models.

In this thesis, I looked at the performance of the Sun Fire Link and Myrinet. It is desirable to evaluate the performance of other high performance interconnects such as QsNet II and InfiniBand.

I looked into the implementations of regular point-to-point communications of Sun MPI. The implementations of collective communications in Sun MPI over RSM are also interesting.

References

[1] D. Addison, J. Beecroft, D. Hewson, M. McLaren and F. Petrini, "Quadrics QsNet II: A Network for Supercomputing Applications". In *Hot Chips 15*, Stanford University, CA, Aug. 2003.

[2] A. Afsahi and Y. Qian, "Remote Shared Memory over Sun Fire Link interconnect", *15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, 2003, pp. 381-386.

[3] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman, "Incorporating Long Messages Into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation", *7th Annual ACM Symposium on Parallel Algorithms and Architecture*, Jul. 1995.

[4] B. Armstrong, S. W. Kim, and R. Eigenmann, "Quantifying Differences between OpenMP and MPI Using a Large-Scale Application Suite", *International Workshop on OpenMP: Experiences and Implementations*, Tokyo, Japan, Oct. 2000.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), "The Nas Parallel Benchmarks", Nas Technical Report RNR-91-002, NASA AAmes Research Center, Moffett Field, CA, 1991.

[6] M. Bertozzi, M. Panella, and M. Reggiani, "Design of a VIA based communication protocol for LAM/MPI suite", *9th Euromicro Workshop on Parallel and Distributed Processing*, Sept. 2001.

[7] N. J. Boden, D. Cohen, R. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W-K Su, "Myrinet: A Gigabit-per-Second Local Area Network", *IEEE Micro*, 1995.

[8] J. M. Bull, M. E. Kambites, "JOMP an OpenMP-like interface for Java", *Java Grande 2000*, 44-53.

[9] F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks", *Supercomputing Conference (SC'00): High Performance Networking and Computing Conference*, 2000.

[10] B. Carpenter, V. Getov, G. Judd, T. Skjellum, G. Fox, "MPI for Java, Position document and Draft API Specification". Available: http://www.javagrande.org/jgpapers.html.

[11] S. Chodnekar, et al., "Towards a Communication Characterization Methodology for Parallel Applications," *High-Performance Computer Architecture*, 1997.

[12] A. Cohen, "A Performance Analysis of 4X InfiniBand Data Transfer Operations", *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, *2003*.

[13] A. Charlesworth, "The Sun Firplane Interconnect", *IEEE Micro*, Vol. 22, No. 1, 2002, pp. 36-45.

[14] 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", *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[15] V. Dunning, G. Regnier, G. McAlpine, D. Cameron, B, Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd, " The Virtual Interface Architecture", *IEEE Micro*, March/April, 1998, pp. 66-76.

[16] N. R. Fredrickson, Ahmad Afsahi, and Ying Qian, "Performance Characteristics of OpenMP Constructs, and Application Benchmarks on a Large Symmetric Multiprocessor", *17th Annual ACM International Conference on Supercomputing, ICS'03*, San Francisco, CA, USA, June, 2003, pp. 140-149.

[17] M. Frumkin, H. Jin and J. Yan, "Implementation of NAS Parallel Benchmarks in High Performance Fortran", Nas Technical Report, NAS-98-009, NASA Ames Research Center, Moffett Field, CA,1998

[18] M. Frumkin, M. Schultz, H. Jin, and J. Yan, "Implementation of the NAS Parallel Benchmarks in Java", Nas Technical Report NAS-02-009, NASA AAmes Research Center, Moffett Field, CA, 2002.

[19] D. S. Henty, "Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling", *Supercomputing Conference (SC'00),* 2000.

[20] J. Hsieh, T. Leng, V. Mashayekhi, R. Rooholamini. "Architectural and Performance Evaluation of GigaNet and Myrinet Interconnects on Clusters of Small-Scale SMP Servers", *Super Computing Conference (SC'00)*, *2000*.

[21] K. Huang, Z. Xu, "Scalable Parallel Computing: Technology, Architecture, Programming".

[22] G. Iannello, M. Laurio, and S. Mercolino, "LogP performance characterization of fast messages atop Myrinet", *6th EUROMICRO Workshop on Parallel and Distributed Processing* (*PDP98),* 1998.

[23] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementations of NAS Parallel Benchmarks and its Performance", NAS Technical Report NAS-99-011, 1999.

[24] G. Jost, H. Jin, Dieter an Mey and F. F. Hatay, "Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster", NAS-03-019.

[25] 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), held in conjunction with IPDPS 2000*, 2000.

[26] J. Kim and David J. Lilja, "Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs", *the Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, International Symposium on High Performance Computer Architecture*, Feb. 1998, pp. 202-216.

[27] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda, "Performance comparison of MPI implementations over InfiniBand, Myrinet, and Quadrics", *Supercomputing Conference (SC'03)*, 2003.

[28] J. Liu, B. Chandrasekaran, Weikuan Yu, Jiesheng Wu, Darius Buntinas, Sushmitha Kini, Dhabaleswar K. Panda, Pete Wyckoff, "Microbenchmark Performance Comparison of High-Speed Cluster Interconnects", *IEEE Micro* 24(1): 42-51 (2004).

[29] M. V. Nibhanupudi and B. K. Szymanski, "Adaptive Parallelism in the Bulk-Synchronous Parallel model", *Proceedings of the Second International Euro-Par*

Conference, Lyon, France, Aug 1996.

[30] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie, "Performance Evaluation of the Quadrics Interconnection Network", *Journal of Cluster Computing*, to appear.

[31] Y. Qian, Ahmad Afsahi, Nathan R. Fredrickson, and Reza Zamani, "Performance Evaluation of the Sun Fire Link SMP Clusters", *18th International Symposium on High Performance Computing Systems and Applications, HPCS 2004*, Winnipeg, Canada, May 16-19, 2004,145-156.

[32] H. Richardson, "High Performance Fortran. History, Overview and Current Status", Technology Watch Report. Available: http://www.epcc.ed.ac.uk/epcc-tec/documents/techwatch.html.

[33] M. Sato, et al, "Design of OpenMP Compiler for an SMP Cluster", *1st European Workshop on OpenMP (EWOMP'99)*, 1999.

[34] S. J. Sistare, and C. J. Jackson, "Ultra-High Performance Communication with MPI and the Sun Fire Link Interconnect", *Supercomputing Conference (SC'02)*, 2002.

[35] S. J. Sistare, F. Vande Vaart, and E. Loh, "Optimization of MPI Collectives on Clusters of Large-Scale SMPs", *Supercomputing Conference (SC'99)*, 1999.

[36] L. A. Smith and J. M. Bull, "Java for High performance Computing".

[37] F. Wong, R. Martin, R. Arpaci-Dusseau, D. Culler, "Architectural Requirements and Scalability of the NAS Parallel Benchmarks", *Supercomputing Conference (SC'99) on High Performance Networking and Computing*, Portland, OR. Nov. 1999.

[38] InfiniBand Architecture. Available: http://www.infinibandta.org.

[39] Message Passing Interface Forum: MPI, A Message Passing Interface Standard, Version 1.2, 1997.

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

[41] Myricom. Available: http://www.myrinet.com.

[42] NAS Parallel Benchmarks homepage. Available: http://www.nas.nasa.gov/.

[43] OpenMP C/C++ Application Programming Interface, Version 2.0, March 2002.

[44] POSIX Threads programming. Available:

http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html.

[45] Remote Shared Memory API for Sun Cluster Systems. Available: http://docs-pdf.sun.com/817-4415/817-4415.pdf.

[46] Sun Fire Link System Overview. Available: http://docs.sun.com/db/doc/816-0697-11.

[47] Sun HPC ClusterTools 5 Software Performance Guide [Online 2003]. Available: http://docs-pdf.sun.com/817-0090-10/817-0090-10.pdf.

[48] Sun™ MPI 6.0 Software Programming and Reference Manual.

[49] Top500 supercomputer sites. http://www.top500.org.

[50] The GM API. Available: http://www.myri.com.

[51] The Stream Benchmark. Available: http://www.streambench.org.

VITA

Name:	Ying Qian
Place and Year of Birth:	Shanghai, China P.R. 1978
Education:	Shanghai Jiao Tong University, China P.R.
	1994-1998
	BSc. (Electrical Engineering) 1998
Experience:	Software Engineer
	Shanghai Communications Technologies Center
	1998-2001

Publications:

1. Ying Qian, Ahmad Afsahi, Nathan R. Fredrickson, and Reza Zamani, "Performance Evaluation of the Sun Fire Link SMP Clusters", *18th International Symposium on High Performance Computing Systems and Applications, HPCS 2004*, Winnipeg, Canada, May 16-19, 2004,145-156.

2. Ahmad Afsahi and Ying Qian, "Remote Shared Memory over Sun Fire Link Interconnect", *15th IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS 2003,* Marina del Rey, CA, USA, November 3-5, 2003, pp. 381-386.

3. Nathan R. Fredrickson, Ahmad Afsahi, and Ying Qian, "Performance Characteristics of OpenMP Constructs, and Application Benchmarks on a Large Symmetric Multiprocessor", *17th Annual ACM International Conference on Supercomputing, ICS'03*, San Francisco, CA, USA, June, 2003, pp. 140-149.