Communication Characteristics of Message-Passing Applications, and Impact of RDMA on their Performance

by

Reza Zamani

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
June, 2005

Abstract

With the availability of *Symmetric Multiprocessors* (SMP) and high-speed interconnects, *clusters of SMPs* (CLUMPs) have become the ideal platform for performance computing. The performance of applications running on clusters mainly depends on the choice of parallel programming paradigm, workload characteristics of the applications, and the performance of communication subsystem. This thesis addresses these issues in details.

It is still open to debate whether pure message-passing or mixed MPI-OpenMP is the programming of choice for higher performance on SMP clusters. In this thesis we investigate the performance of the recently released NAS Multi-Zone (NPB-MZ) benchmarks consisting of *BT-MZ*, *SP-MZ,* and *LU-MZ*, and SMG2000 of the ASCI Purple benchmark. Our studies show that the applications studied have a better MPI performance on clusters of small SMPs interconnected by the Myrinet network.

In this thesis, we examine the MPI characteristics of the three applications in the NPB-MZ suite as well as two applications (*SPECseis* and *SPECenv*) in the SPEChpc2002 suite in terms of their point-to-point and collective communications. We also evaluate the impact of different number of processors as well as different problem sizes on the communication characteristics of these applications. Overall, our experiments reveal that the applications studied have diverse communication patterns, and that they are sensitive to the changes in the system size and the problem size.

This thesis presents an in-depth evaluation of the new Myrinet two-port networks at the user-level (GM), MPI-level, and at the *Aggregate Remote Memory Copy Interface* (ARMCI) level. High-performance interconnects such as Myrinet provide a one-sided communication model, referred to as *Remote Direct Memory Access* (RDMA), which is not utilized in many parallel applications, such as NPB-MZ. We realized that non-blocking operations perform better than blocking, and two-port communication outperforms one-port communication. We noticed that for messages larger than 8KB, ARMCI non-blocking *Put* has a better performance than MPI *Send/Receive* operations.

We take on the challenge to utilize these features to convert our two-sided applications in NPB-MZ to one-sided using the ARMCI. Our results indicate communication performance improvement of up to 43%, depending on the workload size and the number of processors involved, can be achieved.

Acknowledgements

I would like to thank continuous support and encouragement of my supervisor Dr. Ahmad Afsahi throughout this work. Without him, this thesis could not have been completed. I am grateful to Queen’s University for granting me the opportunity to complete this work.

I would like to thank my friends at the Parallel Processing Research Laboratory, Nathan R. Fredrickson, Ying Qian and Fan Gao for their extensive help. I appreciate support of staff at the Department of Electrical and Computer Engineering.

Finally, I would like to express my love and appreciation to my family and friends for supporting my endeavours.

Table of Contents

List of Tables

List of Figures

Glossary

3-D	3-Dimensional
ADI	Alternating Direction Implicit
API	Application Programming Interface
ARCO	Atlantic Richfield Corporation
ARMCI	Aggregate Remote Memory Copy Interface
BT	Block Tridiagonal
CDF	Cumulative Distribution Function
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient
CLUMP	Clusters of Symmetric MultiProcessors
EP	Embarrassingly Parallel
FIFO	First In First Out
FT	Fourier Transform
GAMESS	General Atomic and Molecular Electronic Structure System
HPC	High Performance Computing
IS	Integer Sort
LFU	Least Frequently Used
LRU	Least Recently Used
LU	Lower-Upper Diagonal
MCP	Myrinet Control Program
MG	MultiGrid
MPI	Message Passing Interface
MPMD	Multiple-Program Multiple-Data
NIC	Network Interface Cards
NPB	NAS Parallel Benchmark
NPB-MZ	NAS Parallel Benchmark Multi-Zone
P2P	Point-to-Point
POSIX	Portable Operating System Interface

RDMA	Remote Direct Memory Access
RMA	Remote Memory Access
SMP	Symmetric MultiProcessors
SP	Scalar Pentadiagonal
SPEC	Standard Performance Evaluation Corporation
SPMD	Single-Program Multiple-Data
SSOR	Symmetric Successive Over-Relaxation
WRF	Weather Research and Forecasting

Chapter 1 Introduction

1.1 Motivation

Most supercomputing sites in the world are using clusters since they are cheaper and more scalable than other high-performance architectures. As of today, more than 58% of top 500 supercomputers are clusters [48]. Cluster computing provides cost-effective high-performance computing. With the availability of advanced uniprocessors, symmetric multiprocessors (SMPs), and high-speed interconnects, clusters of uniprocessors and clusters of SMPs (CLUMPs) have become the ideal platforms for high-performance computing, as well as supporting the emerging commercial and networking applications.

Many factors influence the performance of an application running on a cluster. However, mainly, the performance is dependant on the type of parallel programming paradigm in use, the communication characteristics of the application, and the performance of the communication subsystem. This thesis addresses these issues in detail.

Firstly, we consider a number of well-known scientific applications. These applications have been written in OpenMP [14], MPI [30], and in the mixed MPI-OpenMP [10, 15] parallel programming paradigms. OpenMP has emerged as the standard for parallel programming on shared-memory systems. Message Passing Interface (MPI) is the de facto standard for parallel programming in clusters. Given the availability of CLUMPs, it is now possible to write applications in mixed-mode. This thesis, investigates which parallel programming paradigm has a better performance on clusters of small multiprocessors interconnected by the Myrinet network.

Communication overhead is one of the most important factors affecting the performance of clusters. The communication characteristics of applications written in MPI, and mixed-mode, as well as the performance of the communication subsystem greatly influence this overhead. Message-passing and mixed-mode applications exhibit a broad range of communication behaviour [54]. Therefore, understanding their behaviour plays a key role in optimizing their performance as well as in designing better communication subsystems. This thesis addresses the spatial and volume communication attributes of the applications under study.

As stated above, communication subsystem including the interconnection network hardware and the communication system software can easily become the bottleneck for an application running on a cluster. Therefore, high-performance clusters use contemporary interconnects to achieve performance. Low communication latency and high communication bandwidth are the two features of these interconnects.

A number of high-performance interconnects are available for cluster computing. They include Quadrics QsNet [39], QsNet II [1], InfiniBand [29], Myrinet [7], GigaNet [52], and Sun Fire Link [40]. Myrinet is one of the popular high performance interconnects for building clusters. Myrinet provides low-latency and high-bandwidth messaging. As of November 2004, more than 38% of top 500 supercomputers [48] use Myrinet as their interconnect of choice.

Message-passing applications run on top of a message-passing library, such as MPI. MPI runs on top of a user-level messaging library which itself runs on top of the network. GM is the user-level message-passing library for the Myrinet networks. Our platform uses Myrinet 2-port (E-Card) network interface cards (NICs). This thesis evaluates the performance of the Myrinet network, at the GM-level under single-port and two-port modelling. We also present the performance of the MPI on top of GM.

Message-passing communication can be done in two different modes: one-sided communication and two-sided communication [21]. Initial Message Passing Interface defined in MPI uses *Send* and *Receive* operations. This model is called two-sided communication. Both sender and receiver are involved in a two-sided communication, and an implicit synchronization is achieved by performing this operation.

In one-sided communication, one process specifies all communication parameters. To ensure the completion of communication, synchronization should be done explicitly. *Get* and *Put* operations are the most common one-sided communication operations. In fact, removing implicit synchronization can enhance the performance of the applications. Reducing data movements and simplifying programming can be addressed as two other major advantages of one-sided communication.

Aggregate Remote Memory Copy Interface (ARMCI) [3, 35] is a library that provides general purpose, efficient and widely portable *Remote Memory Access* (RMA) operations for contiguous and non-contiguous data transfers. User-level libraries and applications that use MPI or PVM [46] can be supported by ARMCI. ARMCI can be built on top of the GM layer. ARMCI *Put* and *Get* operations can be easily used in codes without the hassle of GM one-sided communication.

In this work, we are also interested in evaluating the performance of one-sided communication in ARMCI. Having known the communication profile of the applications under study, and the superior performance of ARMCI over MPI for large messages, we convert our two-sided MPI applications to one-sided using ARMCI.

1.2 Contributions

In this thesis, we study different aspects of application performance on a cluster of SMPs. We use our own cluster in Parallel Processing Research Laboratory at Queen's University. Our evaluation platform consists of eight dual 2.0GHz Intel Xeon MP servers (Dell PowerEdge 2650s). All nodes are connected to a 16-port Myrinet network through the Myrinet two-port "E card" (M3F2-PCIXE-2) network interface cards. Each node is running Red Hat Linux 2.4.24 as its operating system. We use Intel C++/Fortran compiler version 7.1 for 32-bit applications, as well as GCC compiler version 3.2.2. We use the mpich-1.2.5..10 library as the message-passing library, and GM version 2.1.0, Myrinet's messaging library. This thesis makes the following contributions:

- The first contribution of this thesis is in the collection and analysis of the communication characteristics of NAS Multi-Zone (NPB-MZ) [49] and SPEChpc2002 [44] benchmarks. We examine the MPI characteristics of these small to large-scale scientific applications in terms of their point-to-point and collective communications. We evaluate the impact of the problem size and the system size on the communication behaviour of the applications. Locality characteristics of NPB-MZ and SPEChpc2002 applications are gathered. We have used the *First In First Out* (FIFO), *Least Recently Used* (LRU), and *Least Frequently Used* (LFU) locality heuristics to evaluate the locality of message size and message destinations in our applications. It is found that the applications studied have diverse communication characteristics. Those include very small to very large messages, frequent to infrequent messages, various distinct message sizes, set of favourite destinations, and regular versus irregular communication patterns. Some applications are sensitive to the bandwidth of the interconnect, while others are latency-bound as well. To the best of our knowledge, this is the first communication characterization of NPB-MZ and SPEChpc2002.

- The question remains for the research community as to whether pure message-passing or mixed MPI-OpenMP is the programming of choice for higher performance on SMP clusters. This thesis contributes by addressing this question. We gather and analyze the communication characteristics of NPB-MZ and SMG2000 applications in mixed MPI-OpenMP paradigm, along with their performance evaluation. It is shown that for different combinations of processes and threads, pure MPI paradigm outperforms the Mixed MPI-OpenMP paradigm.

- As the third contribution of this work, for the first time, we present an in-depth evaluation of the new Myrinet two-port networks at the user-level (GM), MPI-level, and at the ARMCI-level. We measure the performance of GM basic function calls, such as program initialization, memory allocation, memory deallocation, and program termination. We evaluate and compare the basic latency performance of GM Send/Receive, GM RDMA, MPI Send/Receive, and ARMCI RDMA operations for one- and two-port configuration of the Myrinet network interface card. We realize that, in general, non-blocking operations perform better than blocking, and the two-port communications at the GM, MPI, and ARMCI levels (except for the GM/ARMCI RDMA read) outperform the one-port communications for the bandwidth. We notice that for messages larger than 8KB, ARMCI non-blocking *Put* has a better performance than MPI *Send/Receive* operations.

- The fourth contribution of this thesis is in improving the communication performance of applications using ARMCI RDMA operations. We take on the challenge to convert our two-sided application to one-sided using the ARMCI library. Our performance results indicate that communication performance of NPB-MZ applications improves between -30% to +43%. We also estimated the performance improvement using the communication characteristics of NPB-MZ applications and the basic communication performance of ARMCI and MPI (latency or bandwidth). The expected communication improvement is up to 5.9%. Using both blocking and non-blocking ARMCI *Put* operations improve the communication performance in some cases.

1.3 Thesis Outline

This thesis is presented in seven chapters. In Chapter 2, background of this work is presented. This chapter introduces the message-passing and shared-memory models, along with one-sided and two-sided communications. We introduce communication characteristics of parallel applications, Myrinet network, GM, MPI, and ARMCI.

In Chapter 3, we introduce the different applications that are studied in this thesis. We describe NPB-MZ benchmark suite, SPEChpc2002 suite, and SMG2000.

In Chapter 4, the communication characteristics of NPB-MZ and SPEChpc2002 applications are studied. We examine point-to-point and collective communication characteristics of these applications. We present the frequency, volume, distribution, locality, and other characteristics of message sizes and message destinations. We also compare the communication characteristics of NPB-MZ in mixed MPI-OpenMP with MPI.

In Chapter 5, we evaluate the basic performance of the Myrinet network at different levels. We evaluate the latency/bandwidth performance of the Myrinet network at MPI-level, GM-level, and ARMCI-level. The performance comparisons suggest the potential improvements in applications' performance using ARMCI over MPI.

In Chapter 6, we calculate the expected performance improvement by replacing MPI two-sided operations in NPB-MZ with ARMCI RDMA operations, as well as evaluating the runtime performance improvement. Performance evaluation of NPB-MZ and SMG2000 application under mixed MPI-OpenMP is also presented. We study the effect of using one or two ports of the Myrinet NIC on the performance of NPB-MZ and

SMG2000 applications. Finally, in Chapter 7, we conclude the thesis and present the potential future work.

Chapter 2 Background

Using multiple computational processing units is one of the principles of High Performance Computing (HPC). Distributing a massive workload among a number of processors enables us to gain a large computing power. Multiple processor systems are either directly coupled or connected through an interconnection network. In either type, communication between processing units becomes a key performance factor. Therefore, communication patterns of an application running on a cluster along with the performance characteristics of the interconnection network both become interesting to study.

In this chapter, we introduce the message-passing and shared-memory models as well as one-sided and two-sided communications. MPI [30] and OpenMP [14] are the de facto standards for message-passing and shared-memory programming models, respectively. Mixed MPI-OpenMP programming paradigm is getting a lot of attention with the prominence of SMP clusters. We also give a short introduction to the communication characteristics of message-passing applications. We introduce some of today’s high performance interconnection networks for cluster computing. Specifically, we introduce the Myrinet. Different messaging libraries may be used on top of an interconnection network. We describe available message-passing libraries on top of Myrinet, namely GM, the low-level Myrinet’s messaging library, MPI over GM, and ARMCI.

2.1 Message-Passing and Shared-Memory Model

Message-passing is a model for interaction between processors within a parallel system. A message is constructed by software on one processor and is sent through an interconnection network to another processor. In message-passing model, the data sender process executes a *Send* operation and the data receiver process executes a *Receive* operation accordingly. Once these *Send* and *Receive* operations match each other, data transfer between processes happens. The memory of the processors in message-passing model can be shared or distributed. There is no need for a global memory map in a distributed memory system using message-passing. A schematic of the data transfer between processes in the message-passing model is shown in Figure 2.1. A message-passing communication that involves both the data sender and data receiver parties is called a *two-sided* communication. MPI [30] and PVM [46] are examples of message-passing models.

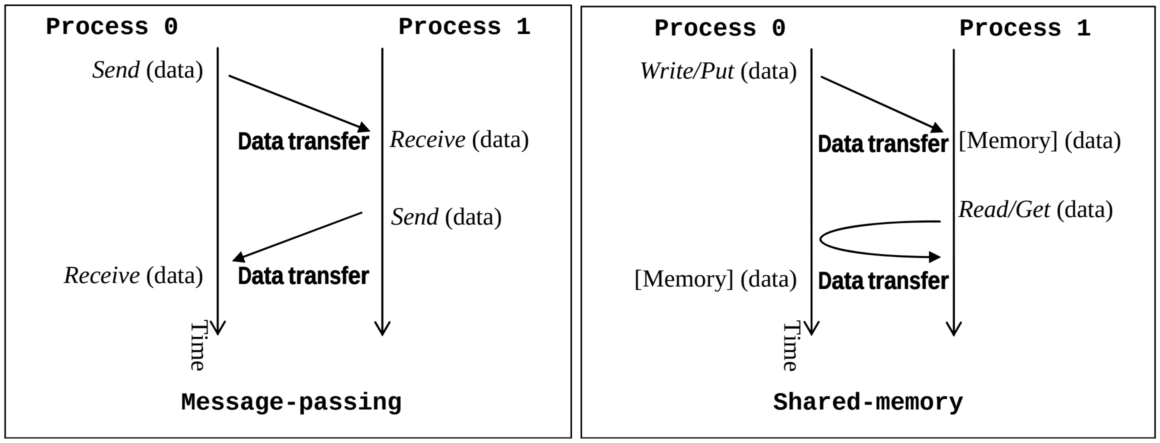


Figure 2.1 Message-passing and shared-memory model data transfer.

Shared-memory model is another model for interactions between processors within a parallel system. In a shared-memory system, each processor has direct access to the memory of every other processor, meaning it can directly load or store any shared address. Multiprocessor systems may physically share a single global memory among their processors. Alternatively, logically shared-memory systems can be implemented on top of distributed memory systems in which each processor has its own local memory. This implementation is done by converting each non-local memory reference into an appropriate inter-processor communication. Shared-memory is generally considered easier to use than message-passing. OpenMP [14] and Pthreads [45] are examples of shared-memory models.

In shared-memory model, the data sender process executes a *Write/Put* operation while the data receiver process does not need to execute any operation. The data will be written to the memory location according to the *Write/Put* operation. On the other hand, a process is able to receive data from other processes by executing *Read/Get* operation. Again, the process that provides the data does not need to perform any action. Once the *Read/Get* operation is issued, the data will be transferred from the memory of data provider process to the memory of the issuer process. A schematic of the data transfer between processes in shared-memory model is shown in Figure 2.1. A communication model that involves either the data sender or the data receiver is called a *one-sided* communication.

The message-passing and shared-memory models each have their own advantages and disadvantages. Developing message-passing parallel applications is more difficult than the shared-memory applications. Message-passing model has more code overhead comparing to the shared-memory model. Message-passing parallel applications are easily portable to different architectures. An application that uses message-passing consists of several concurrent tasks, each with its own data and local memory, using messages to communicate with one another. Message-passing requires the programmer to handle explicitly all parallelism and data distribution. Message-passing programs are inherently parallel, and unless explicitly coordinated by waiting for messages, all processes execute independently. Synchronization among the processes occurs explicitly through message-passing.

Message-passing programs generally take one of two approaches to parallelism: the multiple-program multiple-data (MPMD) approach (also known as the manager/worker approach) or the single-program multiple-data (SPMD) approach. With MPMD, a set of computational worker processes perform work for one or more manager processes. The MPMD method is generally used when little synchronization is required between worker processes [16].

The shared-memory style of programming is convenient because the compiler automatically optimizes computations that can be safely parallelized. The compiler also allows the user to parallelize (by using high-level directives, pragmas, or Pthreads functions [45]) manually the computations that the compiler cannot parallelize automatically. Enabling and disabling many of the compilers’ parallel optimizations is usually supported in the command-line options of the compiler. However, shared-memory model requires the programmer to handle the synchronization in some cases. In addition, although the directives and pragmas allow the user to get better performance from the programs, their functionality is not directly portable to other vendors’ platforms.

2.2 Parallel Programming Paradigms

There are a number of parallel programming paradigms supporting high-performance computing. MPI, OpenMP, and mixed-mode programming are the most common parallel programming paradigms in use today. The applications that we study in this thesis (to be described in Chapter 3) are developed using these parallel programming paradigms. MPI is the de facto standard for the message-passing model. Shared-memory model programs have adapted OpenMP as the standard parallel programming paradigm. The mixed-mode parallel programming paradigm is a combination of message-passing and shared-memory model programming.

2.2.1 MPI

MPI is the standard library for message-passing. MPI is defined by the MPI Forum, which is a group of parallel computer vendors, library writers, and parallel application specialists. Vendors such as IBM, Intel, TMC, Meiko, Cray, Convex, and Ncube, as well as library writers such as PVM, p4, Zipcode, TCGMSG, Chameleon, Express, and Linda participated in the MPI Forum for designing MPI [28]. MPI is designed for parallel computers, clusters, and heterogeneous networks. MPI model was designed because vendor systems were not portable. In addition, available portable systems by research community were incomplete, and did not have the most efficient performance. They were also not supported by vendors.

MPI provides point-to-point message-passing, one-sided communication, and collective (global) operations. A point-to-point communication transfers a message between only two processes. A *Send/Receive* pair is needed for point-to-point message-passing. Collective communication is coordinated among a group of processes. Operations such as *gathering* data from one or more processes and sharing them among all participating processes, or distributing data from one or more processes to a specific group of processes (*scatter*) are examples of collective communication operations. *Broadcasting* data from one process to all other participating processes is another example of collective communication.

2.2.1.1 Blocking and Non-Blocking Point-to-Point Communication

Point-to-point communication is categorized as blocking or non-blocking operations. A blocking communication call means that the program execution will be suspended until the message buffer is safe to use. The common message-passing library standards, such as MPI, specify that a blocking *Send* or *Receive* does not return until the send buffer is safe to reuse (e.g. for *MPI_Send*), or the receive buffer is ready to use (e.g. for *MPI_Recv*). Using blocking communication makes the program more synchronized as blocking *Send* and *Receive* operations have to wait for each other to complete. However, blocking operations will not allow the user to overlap computation with communication while the program is waiting for the return of a blocking operation.

A non-blocking communication call returns immediately after the call is initiated and does not wait to be certain that the communication buffer is safe to use. The programmer must make sure that the send buffer has been copied out before reusing it, or that the receive buffer is full before using it. For the case of MPI, the non-blocking *MPI_Isend* and *MPI_Irecv* are distinguished by the letter I, for immediate return. The syntax and argument list are the same as the blocking versions except for an additional argument, a request handler, which can later be used to wait for, or check on, the completion of the call.

The computation can proceed immediately after a non-blocking communication call without waiting for the call to complete, which improves the program performance. Because the call returns immediately, non-blocking calls allow both communications and computations to proceed concurrently. For the case of MPI library, this is done using the MPI functions *MPI_Test* and *MPI_Wait* with the request handler returned from the non-blocking *Send* and *Receive*.

2.2.1.2 MPI Point-to-point Protocols

Most of the MPI implementations employ a two-level protocol for point-to-point messages. MPI uses *eager* method for sending short messages, while it uses a *rendezvous* mechanism for sending long messages. Eager mechanism improves the latency of messaging while rendezvous mechanism provides a better bandwidth. In the eager mechanism, data is eagerly sent along with the MPI envelope information (context, tag, etc.). This minimizes the interaction of the send operation with the receiver. In a rendezvous implementation, the sender must first send a request and receive an acknowledgment before the data can be transferred. This is to make sure enough buffer space is available for large messages at the receiver side. For large messages, the overhead of the protocol exchange with the receiver is amortized by the transfer of the data [8]. Figure 2.2 illustrates the *eager* and *rendezvous* protocols.

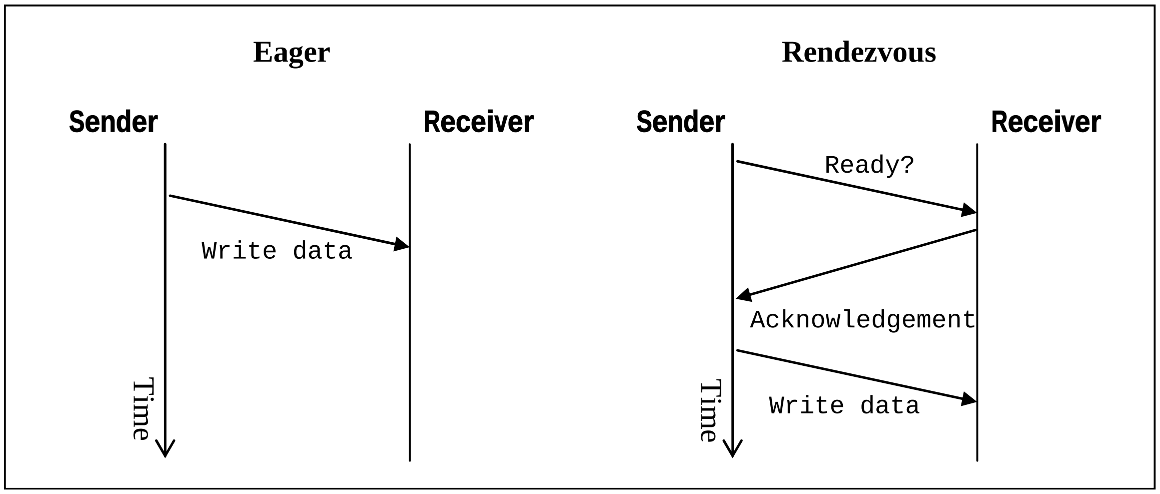


Figure 2.2 MPI *eager* and *rendezvous* messaging protocols.

2.2.1.3 MPI One-sided Communication

One-sided communication is supported in the MPI-2 [30]. Remote Memory Access (RMA) extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. RMA is a one-sided communication operation. Using one-sided communication model facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing.

Two-sided communication (Send/Receive) requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This may require all processes to participate in a time consuming global computation, or to periodically poll for potential communication requests to receive and act upon. RMA communication mechanisms avoid the need for global computations or explicit polling.

MPI-2 provides three one-sided communication calls: *MPI_Put* (remote write), *MPI_Get* (remote read) and *MPI_Accumulate* (remote update). MPI-2 also provides a larger number of synchronization calls that support different synchronization styles. Using RMA functions enables implementers to take advantage of fast communication mechanisms provided by various platforms, such as coherent or non-coherent shared memory, DMA engines, hardware-supported *Put/Get* operations, communication coprocessors, and others.

RMA communications are categorized in two groups: *active target* communication, and *passive target* communication. In active target communication, data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.

In passive target communication, data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, regardless of its location [30].

2.2.1.4 MPI Collective Communication

A collective communication operation is defined as a communication that involves a group of processes. Collective operations can be categorized into three classes: Data movement, synchronization, and collective computation. Non-blocking collective operations are not supported in MPI. Some of the collective functions supported in MPI are listed in Table 2.1.

MPI_Bcast broadcasts from one member to all members of a group. *MPI_Gather* gathers data from all group members to one member. *MPI_Scatter* scatters data from one member to all members of a group. *MPI_Allgather* is a variation on Gather where all members of the group receive the result. *MPI_Alltoall* scatters/gathers data from all members to all members of a group (also called complete exchange). *MPI_Barrier* is a collective operation that blocks until all the associated processes arrive at the barrier. In fact, *MPI_Barrier* synchronizes all the group members. *MPI_Reduce* gets the combined value of the received messages using the operation passed to the function. *MPI_Scan* computes the scan (partial reduction) of data on a collection of processes. Note that, it is possible to have collective operations in a user-defined subset of all processes.

Table 2.1 List of some MPI collective communication operations.

MPI Function	**Type**
MPI_Allgather	Data movement
MPI_Alltoall	Data movement
MPI_Bcast	Data movement
MPI_Gather	Data movement
MPI_Scatter	Data movement
MPI_Barrier	Synchronization
MPI_Reduce	Collective computation
MPI_Scan	Collective computation

2.2.2 OpenMP

OpenMP [14] has emerged as the standard for parallel programming on shared-memory systems. Incremental development of OpenMP codes from the serial version of applications makes it one of the popular parallel programming paradigms. OpenMP is a set of compiler directives and runtime library routines that extend Fortran, C, and C++ to express shared-memory parallelism. OpenMP was designed to exploit certain characteristics of shared-memory architectures (such as directly accessing memory throughout the system with no explicit address mapping) [14]. The OpenMP *application-programming interface* (API) defines parallel regions and work sharing constructs among threads. OpenMP is an explicit programming model, offering the programmer full control over parallelization.

A shared-memory process may consist of multiple threads. OpenMP is based upon the existence of multiple threads in the shared-memory programming paradigm. OpenMP uses the *fork-join* model of parallel execution. All OpenMP programs start as a single process, namely the master thread. The master thread executes sequentially until the first parallel region construct is encountered. When the master thread encounters the parallel region then it creates a team of parallel threads. This is known as *Fork* operation. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads. When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread. This operation is known as *Join* operation. A sample OpenMP C code is given bellow (adapted from [37]):

```
Sequential code …….
/* Fork happens here */
#pragma omp parallel private(var1, var2) shared(var3)
  {
  Parallel section executed by all threads
        .
        .
        .
  At the end of the parallel region, All threads
  join master thread and disband
  }
/* Join happens here */
Sequential code …….
```

2.2.3 Mixed MPI-OpenMP (Mixed-Mode)

Availability of clusters of symmetric multiprocessors has motivated the use of mixed-mode parallel programming. In fact, it is possible to exploit parallelism using a combination of MPI and OpenMP programming paradigms. Prominence of clusters encourages programmers to use MPI in the applications. Using MPI enables the application to be portable and scalable. On the other hand, incremental code development and capability of parallelizing loops easily makes OpenMP desirable for shared-memory programming in an SMP machine. Therefore, clusters of SMPs are suitable platforms for MPI-OpenMP programming, running OpenMP within an SMP node, while running MPI across the nodes.

The mixed MPI-OpenMP programming style is one of the most popular mixed-mode programming paradigms. It provides the application developer a vast flexibility in terms of parallelism and performance tuning. Spreading different combinations of processes and threads over the system nodes enables one to achieve the best performance on an SMP cluster. However, it is still open to debate if pure-MPI or MPI-OpenMP provides the best performance. Some researchers have improved the performance of their applications using mixed-mode paradigm, while this is has not been beneficial for others [15, 10, 42].

2.3 Application Characteristics

Message-passing parallel applications involve a number of processes, where each process may exchange information with the other processes. Message-passing behaviour in parallel applications is important, as it can be crucial to the performance of the applications running on clusters. Application's message-passing characteristics can be studied in terms of its point-to-point communications, collective communications, and locality characteristics. Unless the communications of processes vary by the message arrival time, the MPI characteristics of applications are independent of the experimental platform. This is usually true for the message-passing scientific applications.

Point-to-point communication is the simplest type of communication in message-passing programs. Most of the time, *Send* and *Receive* operations (either blocking or non-blocking) constitutes the point-to-point communication. Although it looks easy to communicate with *Send* and *Receive* functions, their characteristics play a key role in the performance of applications.

Communication properties of message-passing parallel applications can be categorized by the *temporal, volume*, and *spatial* attributes of the communications. The temporal attribute of communications characterizes the rate of message generations, and the rate of computations in the applications. The volume of communications is characterized by the number of messages, and the distribution of message sizes in the applications. The spatial attribute of communications is characterized by the distribution of message destinations. Collective communication is widely used in message-passing applications. Collective communications can be characterized by the type of the collective operation, its frequency, and the payload.

Many message-passing applications follow a repetitive communication pattern. For example, there can be repetition patterns in message size, message destination, or even among send and receive events. Repetitive communication patterns in message-passing applications can be studied using locality heuristics. Locality metrics are useful in message-property prediction schemes for communication latency hiding. Three main locality schemes are widely used, especially in memory replacement policies. These are *First In First Out* (FIFO), *Least Recently Used* (LRU), and *Least Frequently Used* (LFU).

Several researchers have studied some aspects of message-passing characteristics of applications [2, 12, 13, 22, 25, 51, 53]. Vetter and Mueller [51] presented the MPI point-to-point and collective communications as well as floating-point characteristics of some applications in the ASCI Purple suite, and the SAMRAI application. Kim and Lilja [25] quantified the characteristics of some kernels and applications in MPI and PVM as well as their execution times. They also introduced the concept of locality for Send/Receive communication calls using the LRU heuristics. Afsahi and Dimopoulos extended the notion of communication locality to the message destinations, and message reception calls using the LRU, LFU and FIFO policies [2]. They then devised different message predictors. Wong and his colleagues [53] studied the NPB benchmarks. Chodnekar and his associates [12] considered the inter-arrival time of messages, and message volume in message-passing and shared-memory applications. Karlsson and Brorsson [22] compared the communication patterns of some applications in SPLASH and NPB benchmarks under MPI and ThreadMark. Cypher and his colleagues [13] studied some application benchmarks that use explicit communication.

2.4 High-Performance Clusters and Interconnects

The biggest computing challenges are tackled and solved through high performance computing (HPC). Automotive crash test simulations, human genome mapping, meteorological modeling, nuclear blast simulations and many other nowadays' research areas benefit from high performance computing. High performance clusters provide high performance computing at a low-cost.

Interconnection networks enable cluster nodes to communicate with each other. Different interconnection networks may be used for cluster computing. Quadrics QsNet [39], QsNet II [1], InfiniBand [29], Myrinet [7], GigaNet [52], and Sun Fire Link [40] are examples of common interconnects for clusters. Myricom's Myrinet is one of the most popular high performance interconnects used for building clusters [48].

2.4.1 Myrinet Network

Myrinet [7] is a high performance packet communication and switching technology that has become commonplace for connecting clusters of workstations and servers. Clusters in today's computing world are gaining more and more popularity as they can provide high performance computing with lower costs. Myrinet provides high performance, low-latency and high data-rate communication between host processes as well as high availability when faults occur by detecting and isolating them and providing alternative communication paths.

Myrinet supports full-duplex 2+2 Gigabit/second data-rate links, switch ports and interface ports. Error control and flow control is performed on every link. Scalability of the switch networks is up to tens of thousands of hosts. One of the key factors of Myrinet network is offloading the protocol processing from the host processor. Host interfaces execute a firmware to offload protocol processing from the host computer. By bypassing the operating system, the firmware interacts directly with host processes and it interacts directly with the network to send, receive, and buffer packets. Bypassing operating system offers a low-latency communication and therefore achieving higher performance.

Architecture of a node in a Myrinet cluster [6] is shown in Figure 2.3. Each node of the cluster is connected to the network with a Myrinet network interface card. The interface card is connected to host's I/O bus. Each interface card consists of a processor and some fast local memory. The data and control program is stored in its memory. The network interface processor is a fast RISC processor executing the Myrinet control program. There are versatile DMA controllers on the interface to support zero-copy APIs. The network interface used in this study is M3F2-PCIXE-2 E-card Myrinet/PCI-X interface. The E-card has a programmable Lanai-2Xp RISC processor which operates at 333MHz and 2 MB of local memory. The E-card connects to the host with 64-bit 133MHz PCI-X interface. Each port of the interface provides 2+2 Gbps data rate. The E-card provides two ports and therefore supporting data transfer rate of 4+4 Gbps.

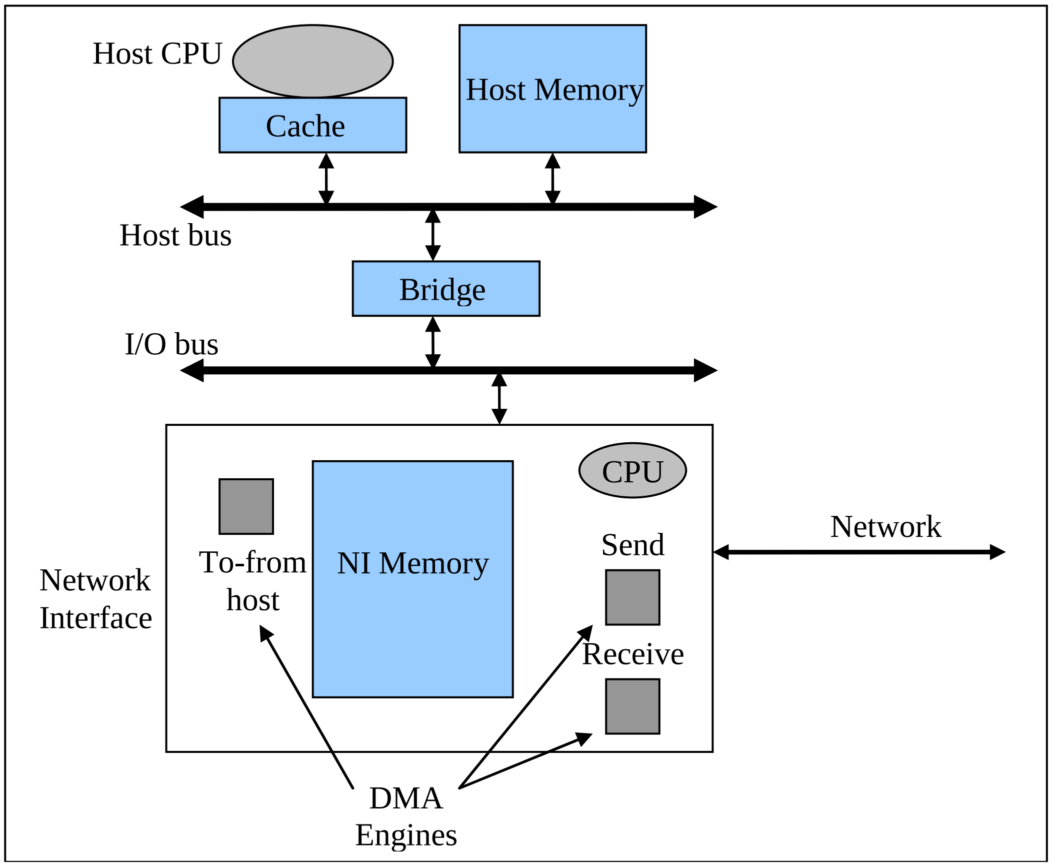


Figure 2.3 Myrinet host and network interface architecture (adapted from [6]).

2.5 Messaging Layers

Myrinet was developed based on packet-switching technology. The packets are wormhole-routed through a network consisting of switching elements and network interface cards (NIC). GM [18] is a messaging library that runs on top Myrinet network. MPICH [19] is a portable implementation of MPI, developed by Argonne National Laboratory. MPICH is popular and highly portable. MPICH-GM [32] is a "port" of MPICH on top of GM (ch_gm) developed and supported by Myricom. We explain GM and MPICH-GM in the following.

2.5.1 GM Messaging Layer

GM [18] is a commercial open source user-level networking protocol from Myricom. It runs on top of Myrinet network. Multiple user processes can share a network interface card simultaneously as GM provides a protected user-level OS-bypass interface to the NIC. GM has a low host-CPU overhead. It provides a connectionless communication model. Communication endpoints in this model are called *ports* (Figure 2.4). GM provides reliable and ordered delivery between these ports. GM consists of a driver, a network mapping program, the GM API library, *Myrinet Control Program* (MCP), and header files. The GM driver provides system services. The mapping program is the Myrinet mapper daemon that maps the network.

GM messages can be delivered with two levels of priority. It allows deadlock-free bounded-memory forwarding. The client software can build a message and send to any port in the network. GM provides ordered message delivery for messages that have the same origin port, the same destination port, and the same priority level. Usually GM applications use only one priority for all the messages so that the order of messages will be preserved. GM supports both *Send/Receive* and RDMA operations. The performance of GM *Send/Receive* is provided in [20, 41, 55] and performance of RDMA in GM is provided in [26, 55].

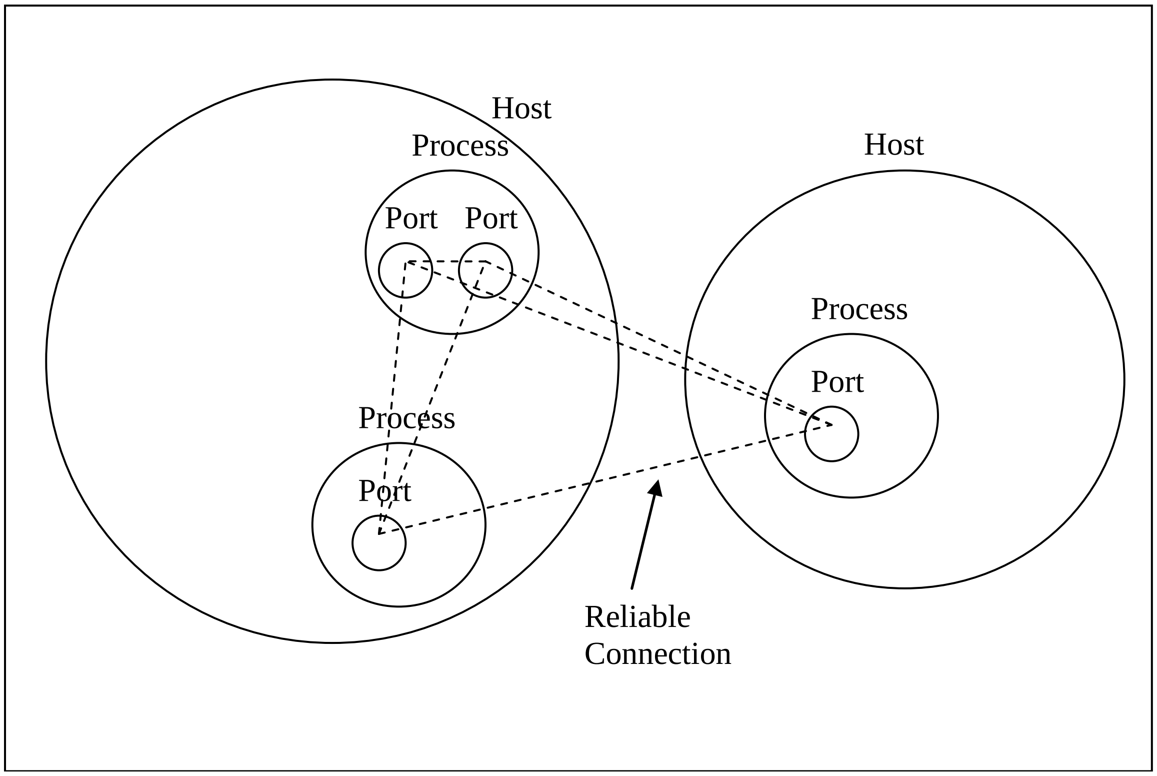


Figure 2.4 GM endpoints (ports) (adapted from [18]).

For sending and receiving messages, GM should first be initialized by the *gm_init()* function; it then should open a port before communication could start. All the buffers used in message-passing must be allocated by calling GM memory allocation/registration functions. GM provides *gm_dma_malloc()* and *gm_dma_free()* function calls for memory allocation and also provides *gm_register_memory()* and *gm_deregister_memory()* to pin and unpin memory on operating systems that support memory registration. The largest message GM can send or receive is limited to 2^{31}-1 bytes. However, because send and receive buffers must reside in DMAable memory, the maximum message size is limited to the amount of DMAable memory the GM driver is allowed to allocate by the operating system.

Both sends and receives in GM are regulated by implicit tokens. These tokens represent the space allocated to the client in various internal GM queues. Internal GM queues for tokens are depicted in Figure 2.5. The client may call certain functions only when possessing an implicit send or receive token. In calling that function, the client implicitly relinquishes the token [18].

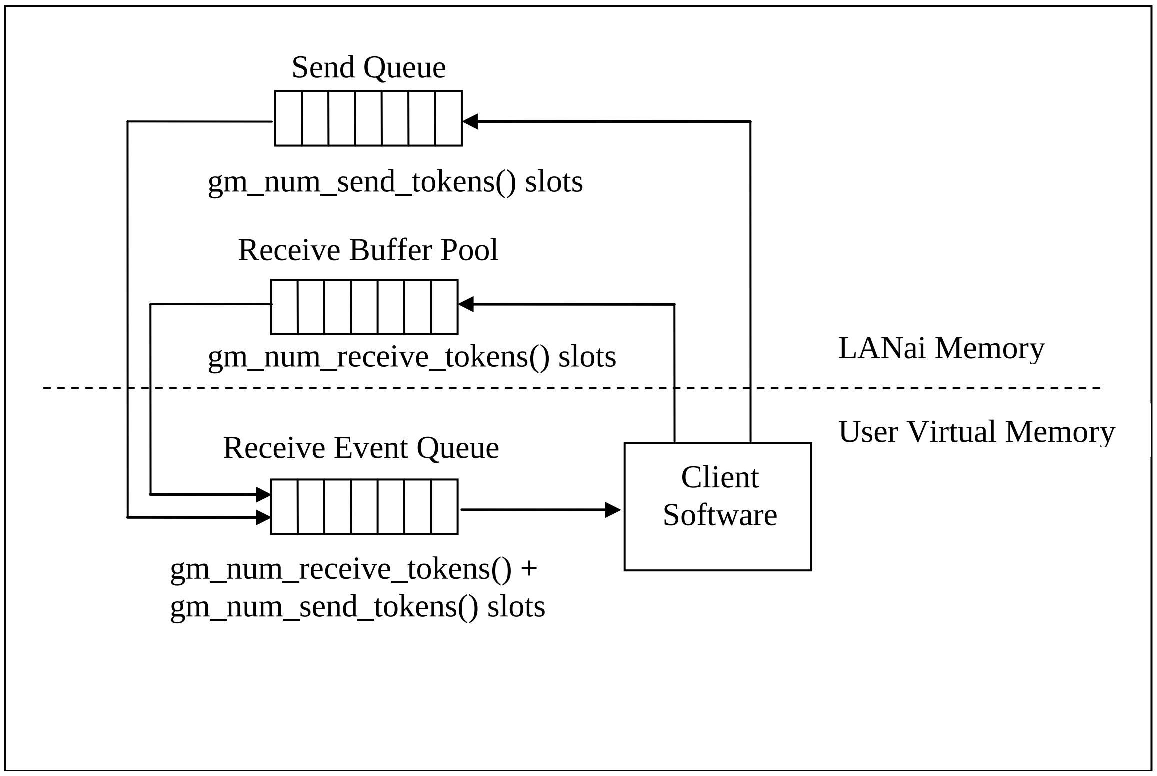

Send Queue
gm_num_send_tokens() slots
Receive Buffer Pool
gm_num_receive_tokens() slots
LANai Memory
User Virtual Memory
Receive Event Queue
Client Software
gm_num_receive_tokens() +
gm_num_send_tokens() slots

Figure 2.5 Internal GM queues for tokens (adapted from [18]).

A client of a port may send a message only when it possesses a send token for that port. Different steps of sending a message in GM are described in Figure 2.6. The client calls a GM API send function, *gm_send_with_callback*. Calling a send function relinquishes a send token. Completion of the send operation is notified to the client by calling a *callback* function and passing a *context* pointer to the client. The client provides a *callback* function and a *context* pointer to the send function. When the send operation completes, GM calls the *callback* function, passes a pointer to the GM port, a pointer to the client's *context*, and a *status* code indicating if the send was successful. Calling the *callback* function implicitly passes back the send token to the client. The *callback* function is only called within a client's call to *gm_unknown()*. The event handler function, *gm_unknown*, is a function that client must call once it receives an unrecognized event.

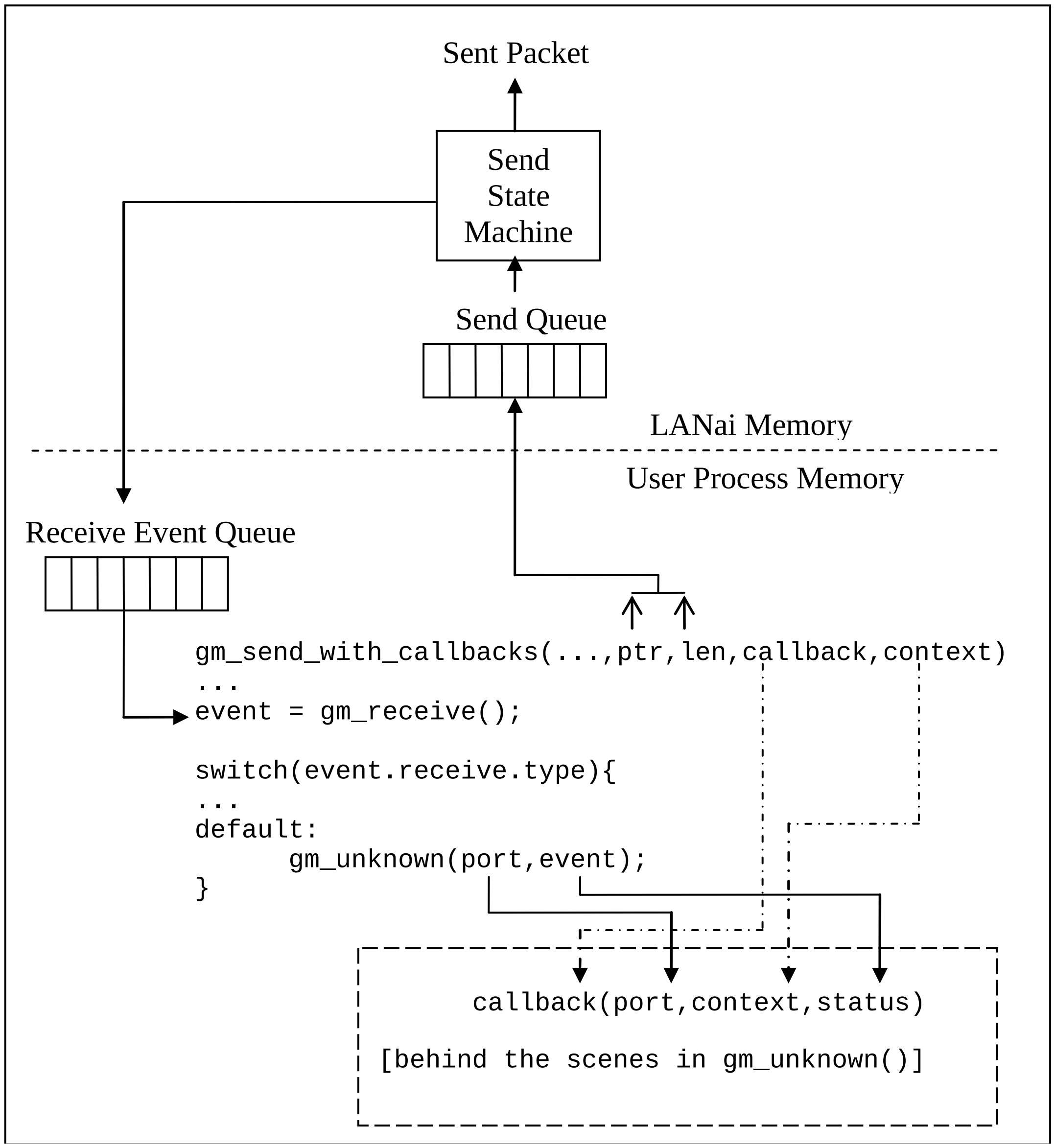


Figure 2.6 GM user token flow (send) (adapted from [18]).

Receiving messages in GM is token-regulated like sending a message. For receiving a message the client provides GM a receive buffer. The client may provide a number of buffers up to the number of receive-tokens that it has. The client provides GM the receive buffer by calling the function *gm_provide_receive_buffer()*. Calling this function implicitly relinquishes a receive-token. The token flow in a receive operation in GM is depicted in Figure 2.7.

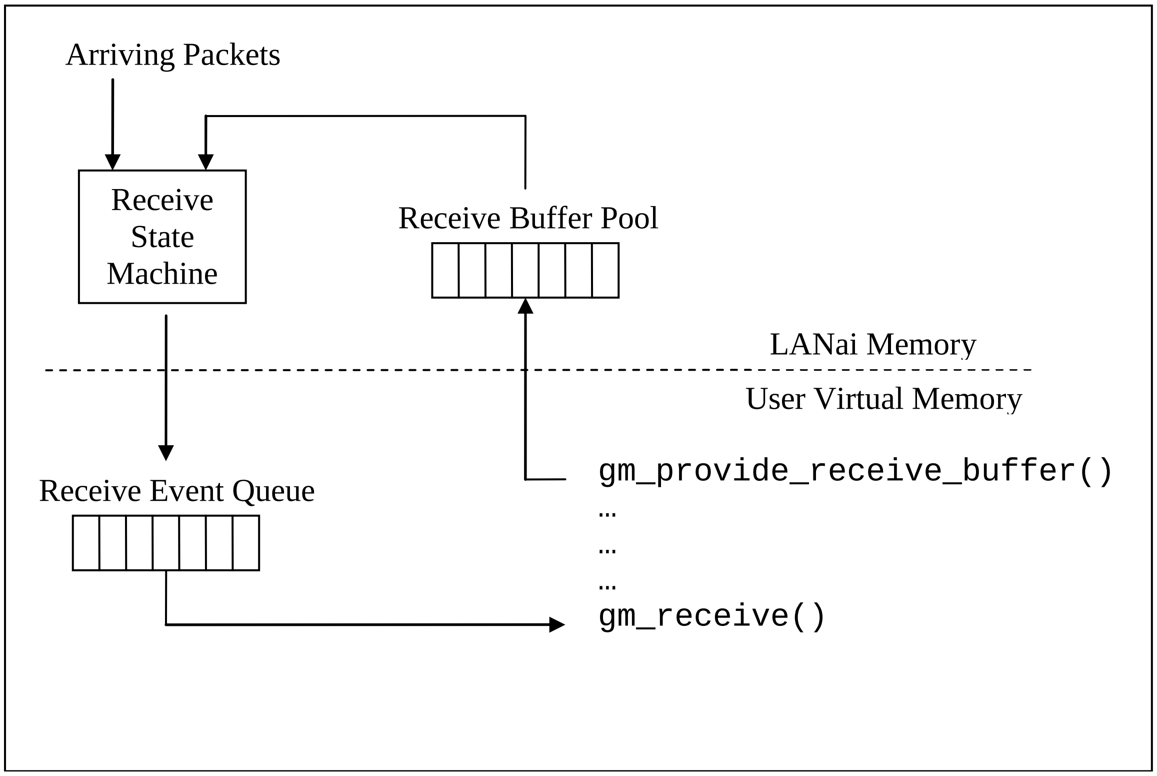


Figure 2.7 GM user token flow (receive) (adapted from [18]).

The client software must provide GM with a receive token before it receives a message of a particular size and priority. The token should match in size and priority with the message. The buffer in which the received message will be stored is specified by the token. After providing such a buffer the client software polls for a receive event. Three different functions maybe used for polling a receive event. The three receive functions used for polling are *gm_receive, gm_blocking_receive,* and *gm_blocking_receive_nospin*. The *gm_receive* function is not blocking. If no receive is pending, an event will be set accordingly. The *gm_blocking_receive* and *gm_blocking_receive_nospin* functions block if necessary. The *gm_blocking_receive* function polls for receives for one millisecond before sleeping, while the *gm_blocking_receive_nospin* function sleeps immediately if no receive is pending.

GM is a lightweight communication layer and it has certain limitations. Sending and receiving messages is only possible into DMAable memory. GM does not support gather and scatter operations. It is not able to register shared memory under Linux. Some of these limitations can be addressed by a heavier layer on top of GM, such as ARMCI [34].

2.5.2 MPICH-GM

MPICH [19] is a portable implementation of the MPI. MPICH over GM (MPICH-GM [32]) is implemented by targeting its Channel Interface to the GM messaging layer. MPICH-GM uses eager protocol for sending small (less than 16K), and control messages via GM send/receive operations. It uses rendezvous protocol for sending large messages via GM one-sided *Put* operation. Note that GM can only send data from registered memory for DMA transfers. In the eager mode, data is copied into a pre-registered buffer to eliminate the overhead of pinning memory for small messages at the expense of a memory copy. To avoid significant overhead in memory copying for long messages, the application message buffers at the source and destination are pinned, and data are transmitted from its original location at the user space to its final destination achieving a zero-copy. In the rendezvous mode, the sender sends a request-to-send to the receiver, and in response, the receiver sends back a clear-to-send as well as the address of the receiver buffer. Then, the sender writes directly to the remote buffer using the GM *Put* operation.

At the receiving side, if a message arrives before a matching message reception call has been posted, MPICH-GM copies the data into a buffer, and adds it to the unexpected queue. When a process calls one of the MPI message reception calls, it first searches the unexpected queue to see if the message has already been arrived. It copies the message into the application buffer space if the matching message is found. Otherwise, a descriptor is posted. It then optionally polls the network device until the corresponding message arrives. MPICH-GM by default uses the polling method; however, it provides the ability to change the behaviour. Three modes are supported: *polling*, *blocking*, and *hybrid*. These modes are supported though the function calls *gm_receive()*, *gm_blocking_receive()*, and *gm_blocking_receive_nospin()*, respectively.

2.5.3 ARMCI

GM is a low-level messaging library. However, it is not portable. MPICH-GM [32] is a portable library on top of GM. However, it does not support the *Remote Memory Access* (RMA) features of MPI-2 (there are reported research works in supporting MPI-2 one-sided communication in MPICH-2 [27, 31]). Aggregate Remote Memory Copy Interface [3] is a library that provides general purpose, efficient and widely portable RMA operations for contiguous and non-contiguous data transfers. A list of ARMCI operations is provided in Table 2.2.

Table 2.2 ARMCI remote operations description.

Operation	**Description**
ARMCI_Put, _PutV, _PutS	Contiguous, vector and strided versions of put
ARMCI_Get, _GetV, _GetS	Contiguous, vector and strided versions of get
ARMCI_Acc, _AccV, _AccS	Contiguous, vector and strided versions of atomic accumulate
ARMCI_Fence	Blocks until outstanding operations targeting specified process complete
ARMCI_AllFence	Blocks until all outstanding operations issued by calling process complete
ARMCI_Rmw	Atomic read-modify-write
ARMCI_Malloc	Memory allocator, returns array of addresses for memory allocated by all processes
ARMCI_Free	Free memory allocated by ARMCI_Malloc
ARMCI_Lock, _Unlock	Mutex operations

ARMCI provides data transfer operations including *put*, *get* and *accumulate*. It also provides synchronization operations such as local and global fence and atomic *read-modify-write*. Utility operations such as memory allocation and deallocation and error handling are supported in ARMCI too. ARMCI only supports communication that targets remote memory allocated via the provided memory allocator routine, *ARMCI_Malloc()*.

In scientific computing, it is popular to store data in arrays. If the desired data is stored in different parts of the array or in general if the data is stored in several locations, this type of data is called non-contiguous data. Therefore using ARMCI in scientific applications helps improving the communication performance. Remote copy APIs that only support contiguous data transfer, require multiple contiguous data transfers to send non-contiguous data. ARMCI, however, is optimized for non-contiguous data transfer. It is meant to be used primarily by library implementers rather than application developers. Example libraries that ARMCI is targeting include Global Array [36], P++/Overture [5], and Adlib PCRC run-time system [11]. User-level libraries and applications that use MPI [19], PVM [46] or TCGMSG [28] can be supported by ARMCI.

Tipparaju and others [47] have used ARMCI to improve the performance of message-passing applications. They used ARMCI to evaluate effectiveness of the RMA communication on several popular scientific benchmarks and applications such as NAS CG and MG. They have achieved 12-49% overall improvement over MPI on 128 processors. CG and MG are kernel applications and are not compute intensive.

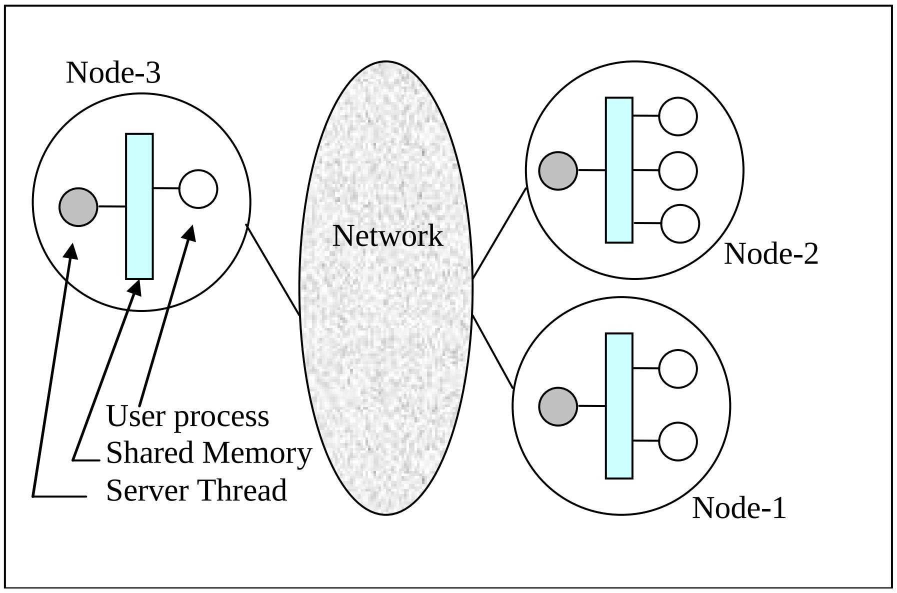


Figure 2.8 ARMCI client-server architecture (adapted from [9]).

ARMCI uses client-server architecture in clusters of workstations using GM [35]. Each node of the cluster has a server thread that handles remote memory operations for each of the user processes running on the node. When a user process wants to perform a remote memory operation, it sends a request to the server thread at the node where the remote process is running. Each user process shares a memory region with the server thread. When the server thread receives a request, it performs the operation on the memory region for that process [9]. The client-server architecture of ARMCI is depicted in Figure 2.8.

2.6 Summary

In this chapter, we introduced the status of high performance clusters and interconnects. Shared-memory and message-passing models were described, as well as one-sided and two-sided communications. We showed the importance of communication characterization of parallel applications. This chapter introduced the Myrinet interconnection network and presented its architecture. We explained user-level messaging layer of Myrinet (GM), along with MPI built on top of GM. The Aggregate Remote Memory Copy Interface library, which supports one-sided communication on top of GM, is introduced in this chapter, as well.

In the next chapter, we introduce the parallel applications studied in this thesis. In chapter 4, we gather the communication characteristics of our applications. Later on, we analyze the communication characteristics of the applications and propose using the ARMCI one-sided communication instead of MPI two-sided communications, to improve their communication performance.

Chapter 3 Parallel Applications

Exploiting parallelism is a natural way to boost performance of applications. Parallel applications are designed to run on multiprocessor systems. Traditionally, performance of parallel applications on multiprocessor systems is evaluated by some well-known scientific or engineering *benchmarks*. Such benchmarks imitate the execution of the large applications. In this chapter, we introduce the parallel applications and benchmarks used in this thesis. They include the NPB-MZ benchmark suite, SPEChpc2002 benchmark suite, and SMG2000 of ASCI purple suite. For our study, we have used the most recently released versions of these benchmark applications. An overview of these sophisticated applications is presented in Table 3.1.

Table 3.1 Overview of application benchmarks.

Application	Field	Language	#Lines
BT-MZ	Computational fluid dynamics; Block-Tridiagonal systems	Fortran	4700
SP-MZ	Computational fluid dynamics; Scalar Pentadiagonal systems	Fortran	4200
LU-MZ	Computational fluid dynamics; Lower-Upper symmetric Gauss-Seidel	Fortran	4600
SPECenv	Weather research and forecasting model	Fortran and C	180000
SPECseis	Computing time and depth migrations used to locate gas and oil deposits	Fortran and C	23000
SMG2000	Solver for the linear systems	C	27000

3.1 NPB-MZ (Multi-Zone) 3.0

The NAS Parallel Benchmark (NPB) [49] is a set of eight programs designed at the NASA Ames Research Center to help evaluate the performance of parallel supercomputers. The NPB benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three pseudo-applications. Kernel applications are *conjugate gradient (CG)*, *multigrid (MG)*, *3-D fast-Fourier Transform (FT)*, *Integer Sort (IS)*, and *Embarrassingly Parallel (EP)*. Pseudo-applications are *Block Tridiagonal (BT), Scalar Pentadiagonal (SP)*, and *Lower-Upper Diagonal (LU).*

In BT, network bandwidth and instruction cache is tested. In SP, memory bandwidth is tested, while in LU network latency and cache instruction is studied. NAS Multi-Zone benchmark suite (NPB-MZ) [49] is an extension of the NPB suite that involves solving the application benchmarks LU-MZ, BT-MZ and SP-MZ on collections of loosely coupled discretization meshes. NPB-MZ 3.0 was first released in summer 2003. Each of these three applications is described in more detail later in the text.

NPB consists of eight programs. These programs exhibit mostly fine-grain exploitable parallelism, and are almost all iterative, requiring multiple data exchanges between processes between iterations. Implementations in MPI, Java, High Performance Fortran, and OpenMP all take advantage of this fine-grain parallelism. However, many important scientific problems feature several levels of parallelism, and this property is not reflected in NPB. To remedy this deficiency, the NPB-MZ versions were created. The solutions on the meshes are updated independently, but after each time step, they exchange boundary value information. This strategy, which is common among structured-mesh production flow solver codes in use at NASA Ames and elsewhere, provides relatively easily exploitable coarse-grain parallelism between meshes. Since the individual application benchmarks also allow fine-grain parallelism themselves, this NPB extension, named NPB Multi-Zone (NPB-MZ), is a good candidate for testing hybrid and multi-level parallelization tools and strategies (e.g., clusters of multiprocessors).

NPB-MZ benchmarks are serial and parallel implementations of Multi-Zone benchmarks based on the original single-zone NPB 3.0. They are meant for testing the effectiveness of multi-level and hybrid parallelization paradigms and tools. The parallel implementation uses hybrid parallelism: MPI for the coarse-grain parallelism and OpenMP for the loop-level parallelism.

Problem sizes and verification values are given for benchmark classes S, W, A, B, C, and D. Problem size S, W, and A are fairly small for evaluating performance of large parallel architecture and are usually used to test the benchmark. Class B, C and D problem sizes are suitable for measuring the performance of a large scale system. The larger the problem size is, the better the system performance is evaluated. Class D has the biggest problem size and class C has a larger problem size than class B. In this thesis, we have used our applications with the problem size class B, and C. Class D is too big to run on our cluster.

3.1.1 NAS BT-MZ

Block Tridiagonal (BT) is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (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 [49]. BT-MZ is written in FORTRAN and has around 4700 lines of code.

3.1.2 NAS SP-MZ

Scalar Pentadiagonal (SP) is a simulated CFD application that has a similar structure to BT. The finite difference solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension [49]. SP-MZ is written in FORTRAN and has around 4200 lines of code.

3.1.3 NAS LU-MZ

Lower-Upper Diagonal (LU) is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems [49]. LU-MZ is written in FORTRAN and has around 4600 lines of code.

3.2 SPEChpc2002

In December 2002, SPEC (Standard Performance Evaluation Corporation) organization introduced *SPEC HPC2002* suite [44], which improved upon and replaced the SPEC HPC96 benchmark suite. The benchmarks in the *SPEChpc2002 V1.0* suite are derived from real HPC applications, and measure the overall performance of high-end computer systems, including the processors, the interconnection networks (shared or distributed memory), the compilers, the MPI and/or OpenMP parallel library implementation, and the input/output system. Serial, OpenMP, MPI, and combined MPI-OpenMP parallelisms are supported. SPEChpc2002 supports shared memory, distributed memory and cluster architectures.

SPEChpc2002 consists of three different benchmarks: *SPECchem*, *SPECenv*, and *SPECseis*. Each of the three benchmarks has Small (S) and Medium (M) workload classes. In this thesis, we have studied SPECseis and SPECenv, both with small and medium classes. SPECchem and SPECenv support MPI, OpenMP and combined MPI-OpenMP. SPECseis supports MPI and OpenMP parallelism. SPEC HPC2002 runs on a UNIX or Linux system (Windows is not yet supported) with minimum 2 GB of memory, up to 100GB of disk, and a set of compilers.

3.2.1 SPEChpc2002 – SPECenv

SPECenv is based on a weather research and forecasting model called WRF. WRF is a state-of-the-art non-hydrostatic mesoscale weather model. The SPECenv class M metric expresses the performance of a computing system in simulating the weather over the continental United States for a 24 hour period at a 22km resolution using the WRF Model [44].

3.2.2 SPEChpc2002 – SPECseis

SPECseis represents an industrial application that performs time and depth migrations used to locate gas and oil deposits and originally developed at Atlantic Richfield Corporation (ARCO). SPECseis includes more than 23,000 lines of FORTRAN and C code. Computational codes are written in FORTRAN while file Input/Output, data partitioning, synchronization primitives and message-passing layer are written in C [44].

3.2.3 SPEChpc2002 – SPECchem

SPECchem is based on a quantum chemistry application called GAMESS (General Atomic and Molecular Electronic Structure System) which is an improved version of programs that originated in the Department of Energy's National Resource for Computations in Chemistry. Many of the functions found in GAMESS are duplicated in commercial packages used in the pharmaceutical and chemical industries for drug design and bonding analysis. SPECchem includes more than 120,000 lines of FORTRAN and C code [44].

3.3 SMG2000

SMG2000 is one of the applications in the ASCI compact benchmark suite [4]. SMG2000 is a parallel semi-coarsening multi-grid solver for the linear systems arising from finite differences, finite volume, or finite element discretizations of the diffusion equation $\nabla\cdot(D\nabla u) + \sigma u = f$ on logically rectangular grids. It solves both 2-D and 3-D problems. For solving problems of radiation diffusion and flow in porous media, such solver is needed. The parallelism in SMG2000 is achieved by data decomposition. SMG2000 is a memory-access bound application and memory-access speed has a large effect on performance.

There is no standard problem size for SMG2000. In fact, the problem size scales with the number of processes. The problem size of SMG2000 is equal to the input problem size multiplied by number of processes; hence, as the number of processes increases, the problem size increases proportionally. In order to run SMG2000 with fixed problem size, the input value of problem size has to be decreased proportional to the increase in the number of processes; in other words, the multiplication of input problem size and number of processes should remain constant.

3.4 Summary

In this chapter, we introduced the different applications and benchmarks that we have studied in this thesis. We described the NPB-MZ benchmark suite, SPEChpc2002 benchmark suite and SMG2000 application. These popular benchmarks are used to evaluate the performance of multiprocessor systems.

Communication characteristics of parallel applications can be crucial on their performance. In order to have a better understanding of their performance on our platform, in chapter 4 we study the message-passing behaviour of these parallel applications.

Chapter 4 Application Characteristics

Message-passing parallel applications involve a number of processes, where each process may exchange information with the other processes. Message-passing behaviour of parallel applications is important, as it can be crucial to their performance running on clusters. In this chapter, we discuss different message-passing behaviour of our applications introduced in chapter 3. We study the MPI point-to-point communications, collective communications, and locality characteristics of these applications. We have written our own profiling code for NPB-MZ and SPEChpc2002 applications, using wrapper facility of MPI. However, we used Vampir/Vampirtrace too [38] for the MPI analysis of SMG2000. The information provided in this chapter will provide the HPC users, programmers and system designers with a better understanding of parallel applications and their communication characteristics impact on the performance. It is noteworthy that the MPI characteristics of the applications are independent of the platform we used in our experiments.

4.1 Evaluation Platform

We recall that our evaluation platform consists of eight dual 2.0GHz Intel Xeon MP Servers (Dell PowerEdge 2650s). All nodes are connected to a 16-port Myrinet network through the Myrinet two-port "E card" (M3F2-PCIXE-2) network interface cards. Each node is running Red Hat Linux 9 with Kernel 2.4.24 as its operating system. We use Intel C++/Fortran Compiler version 7.1 for 32-bit applications, as well as GCC compiler version 3.2.2. We have used the mpich-1.2.5..10 library as the message-passing library, and GM version 2.1.0, Myrinet’s messaging library.

4.2 Point-to-point Communications

Communication properties of message-passing parallel applications can be categorized by the *temporal*, *volume*, and *spatial* attributes of the communications [12, 25]. The temporal attribute of communications characterizes the rate of message generations, and the rate of computations in the applications. We do not discuss the temporal attribute in this thesis. The volume of communications is characterized by the number of messages, and the distribution of message sizes in the applications. The spatial attribute of communications is characterized by the distribution of message destinations.

Point-to-point (P2P) communication is the simplest type of communication among processes in a message-passing programming paradigm. Major events in P2P communication are sends and receives that either of them can be blocking or non-blocking. We quantify metrics such as number of sends (either blocking or non-blocking), average message size per message, total message size transferred per process, message size cumulative distribution function (CDF), number of unique message destinations per process, and destination distribution of messages of the root process (process zero). Knowing these metrics assists one in choosing the proper interconnection network for a cluster. For example, if a program is sending many short messages then latency of interconnection network becomes very important for the performance of the application. If the size of the messages is very large then the bandwidth of the network becomes an issue.

Vetter and Mueller [51] presented the MPI point-to-point and collective communications as well as floating-point characteristics of some applications in the ASCI Purple suite, and the SAMRAI application. Kim and Lilja [25] quantified the characteristics of some kernels and applications in MPI and PVM as well as their execution times. Wong and his colleagues [53] studied the NPB benchmarks. Chodnekar and his associates [12] considered the inter-arrival time of messages, and message volume in message-passing and shared-memory applications. Karlsson and Brorsson [22] compared the communication patterns of some applications in SPLASH and NPB benchmarks under MPI and ThreadMark. Cypher and his colleagues [13] studied some application benchmarks that use explicit communication. In this thesis, we study the new application benchmarks that have not been studied before. Comparison of message-passing characteristics of applications under mixed MPI-OpenMP and pure MPI is another aspect of this chapter that is not addressed by others before.

4.2.1 Message Frequency

Message frequency of an application is the simplest P2P metric. In this study, we count the number of send calls, either blocking or non-blocking, for each process in an application. Minimum, average and maximum number of send calls is calculated per process and presented in Figure 4.1. The X-axis in Figure 4.1 shows the class size and the number of processes for each case. For instance, “B2” and “M4” correspond to class B running with two processes, and class M running with four processes, respectively.

It is evident that the number of messages sent in the BT-MZ, SP-MZ, and LU-MZ is decreasing (except for some of the cases where the number of processes is two) with the increasing number of processes. This trend is consistent for both classes B, and C. However, contrary to the NPB-MZ benchmarks, the number of messages sent for the SPECenv and SPECseis applications shows a different trend, where they actually increase when the number of processes increases. This trend is consistent for both small and medium workloads.

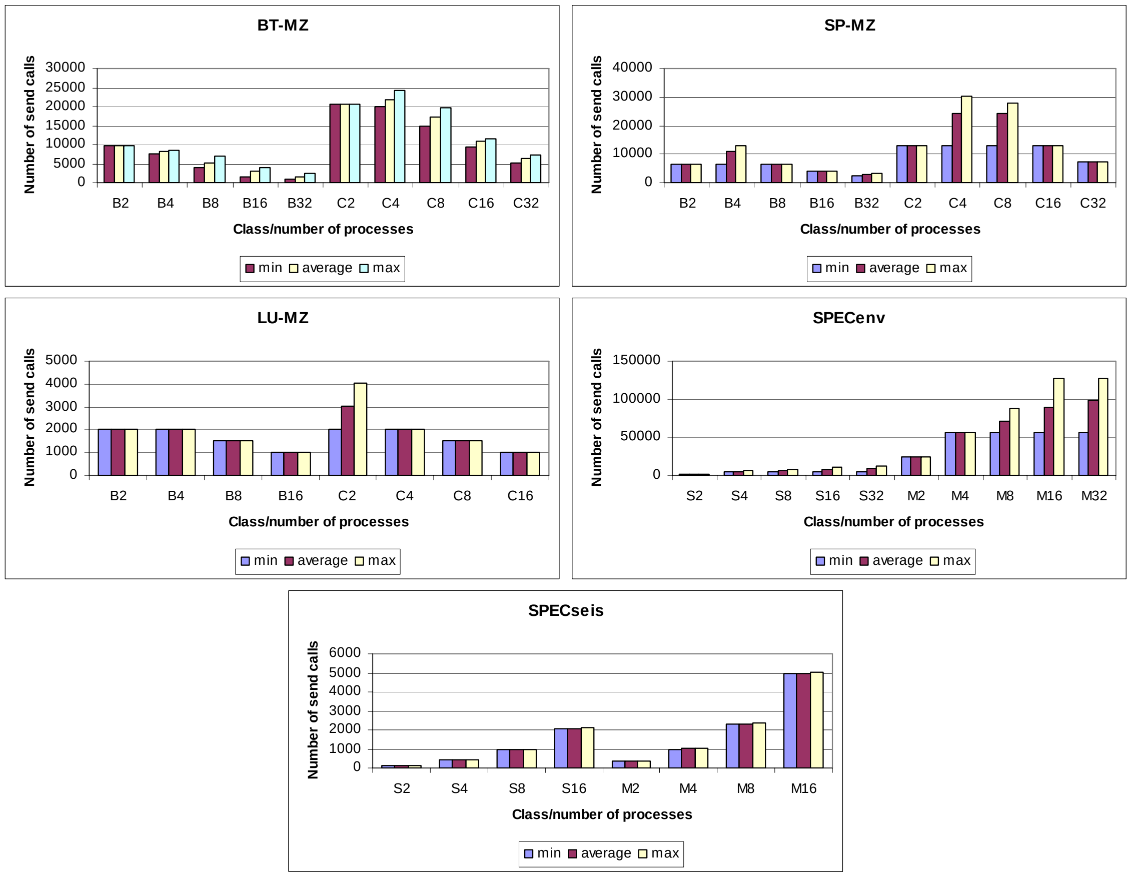


Figure 4.1 Number of messages sent per process.

An interesting observation is that the processes in the LU-MZ (except for C2) and SPECseis send equal number of messages to their destinations, where this is not the case for the other applications. We can see a large difference between the minimum and the maximum number of messages sent among processes of SPECenv under class M. Among the five applications, SPECenv has the largest number of messages sent per process. For instance, with 16 processes, each process in SPECenv sends 90,000 messages on average, while each process in the SP-MZ, BT-MZ, SPECseis, and LU-MZ sends roughly 13,000, 10,000, 5,000 and 1,000 messages, respectively. This shows that latency of the interconnection network will affect performance of SPECenv the most, but will affect the performance of LU-MZ the least.

4.2.2 Average Message Size

In this section, we quantify the average message size of all messages sent in each application. This will give us an understanding of size of the messages exchanged in the application and will help us understand the bottlenecks of the system due to the bandwidth of the interconnection network. It is interesting to know if the message sizes of an application are regular for different number of processes. We present the average message sizes of each benchmark in Figure 4.2. Note that the average message size is presented in Kilobyte (1024 bytes).

One can easily observe that the average message size for the SPEC applications, for both small and medium workloads, becomes smaller as the number of processes increases. In contrast, the average message size for the NPB-MZ application benchmarks increases. BT-MZ and SP-MZ roughly use the same sort of message sizes; between 15KB to 25KB for the BT-MZ, and between 10KB to 21KB for the SP-MZ. However, LU-MZ uses larger message sizes, especially for the larger class C with message sizes between 80KB and 100KB.

In order to be able to compare the average message size sent in different applications, average of message sizes over all different classes and workloads, and different number of processes for each application is calculated and presented in Figure 4.3. NAS applications, especially for the small class B, use the smallest average message size among the all applications. It means the impact of network bandwidth on these applications will not be big compared to other applications. An overall observation is that the SPECenv, SPECseis, and LU-MZ are more bandwidth bound than the other applications.

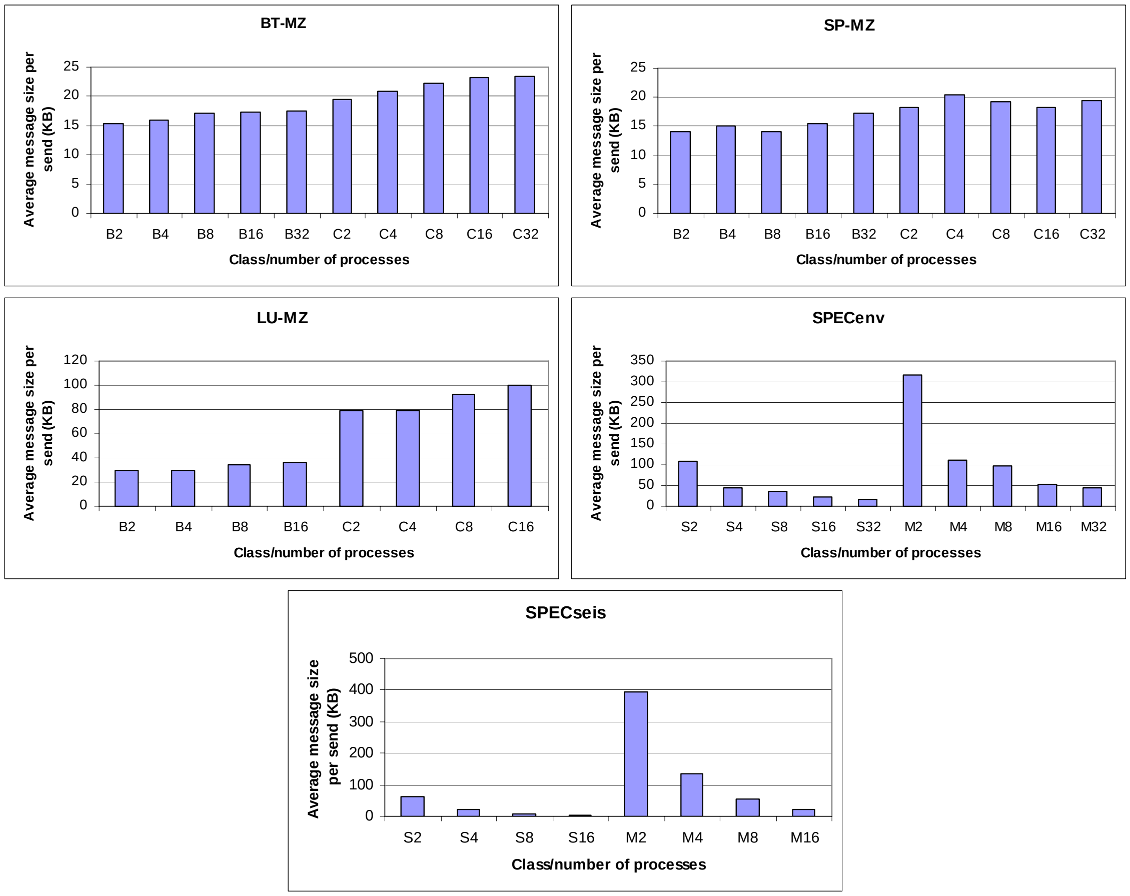


Figure 4.2 Average message size per send.

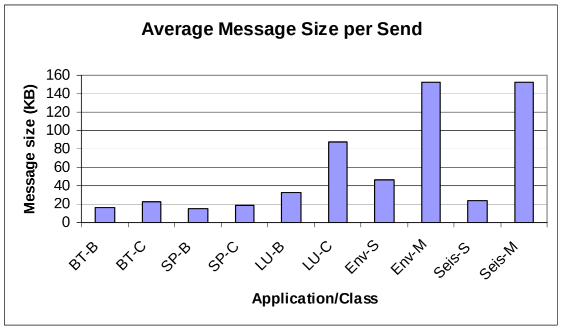


Figure 4.3 Comparison of average message sizes of benchmarks

4.2.3 Message Volume

It is interesting to know how much traffic each process generates on the interconnection network. The total message volume that a process sends over the network is roughly equal to the average message size per send times the number of messages sent per process. We have quantified and presented the minimum, average and the maximum number of bytes that each process sends over the network in Figure 4.4.

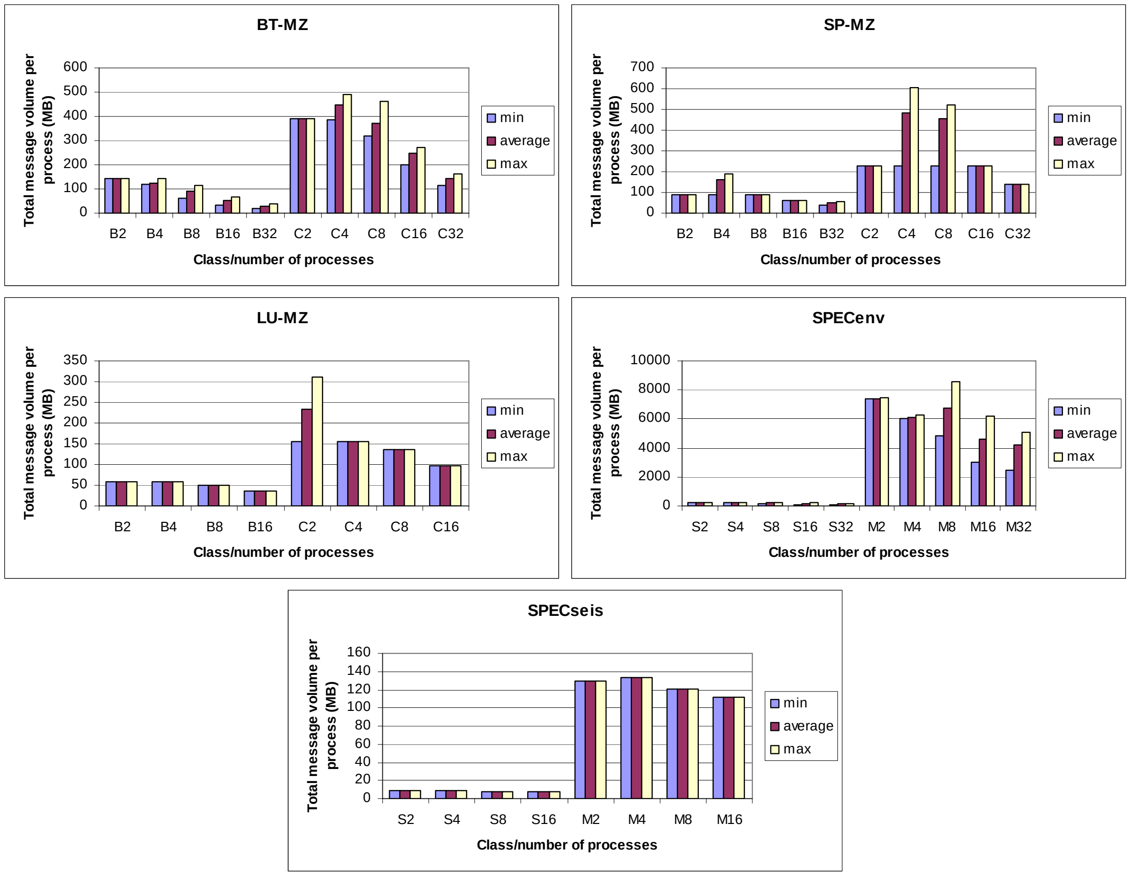


Figure 4.4 Total message volume per process

The total message volume sent by each process in the SPEChpc2002 applications is very different for small and medium classes. However, this is not the case for the classes B and C in the NPB-MZ. SPECenv, medium class, has the largest message traffic per process on the network, roughly between 4000MB to 7000MB. However, for the small class, each process sends roughly 140MB to 230MB. SPECseis has message traffic of 110MB to 130MB per process on the network for the medium class, and around 8MB for the small class. For the C class, each process in BT-MZ, SP-MZ, and LU-MZ sends

130MB to 440MB, 130MB to 480MB, and 97MB to 232MB, respectively. In all of the five applications, we can notice that there is a big difference between small/B and medium/C classes in terms of total message volume per process.

As shown in Figure 4.4, in most cases there is not a big difference among processes in terms of total number of bytes sent. One can see big differences among some processes for LU-MZ in C2, SPECenv in M8, M16 and M32, SP-MZ in C4 and C8, and BT-MZ in C4 and C8. SPECseis seems to be the most regular application in terms of number of sent bytes per process.

4.2.4 Message Size Cumulative Distribution Function

The *cumulative distribution function* (CDF) of message sizes provides more detail about the different message sizes sent in an application. Figure 4.5 presents the CDF of the message sizes for the applications under different system and problem sizes. Note that the horizontal axis for the SPECseis is in logarithmic scale. From the graphs, it can be seen that the BT-MZ and SPECenv use a large number of different message sizes while the other applications use only a few message sizes. BT-MZ uses up to 21 different message sizes in class C (16 in class B). The shortest and the longest messages are 5KB, and 55KB, respectively. SPECenv uses up to 70 different messages sizes in class S (50 in class M). It uses both short messages (as small as 4 bytes) and very long messages (as large as 3129KB for the class M). Thus, SPECenv is very much sensitive to both latency and bandwidth.

The distribution of the size of messages sent by the SP-MZ, and LU-MZ are bimodal. There are only two different message sizes used in these applications. Message sizes for these two applications suggest they are very much bandwidth-bound. SPECseis uses five different message sizes each for both classes. It uses small messages (including zero-byte messages) as well as very large messages (up to 32768KB). This shows that SPECseis is more sensitive to the bandwidth than to the latency of the interconnect. It is easily seen that message size range is very different for all these applications. We discovered that minimum message sizes for benchmarks are 5KB, 14KB, 29KB, 4 bytes and zero byte, and maximum message sizes are 55KB, 28KB, 79KB, 3129KB and 32768KB for BT-MZ, SP-MZ, LU-MZ, SPECenv and SPECseis, respectively.

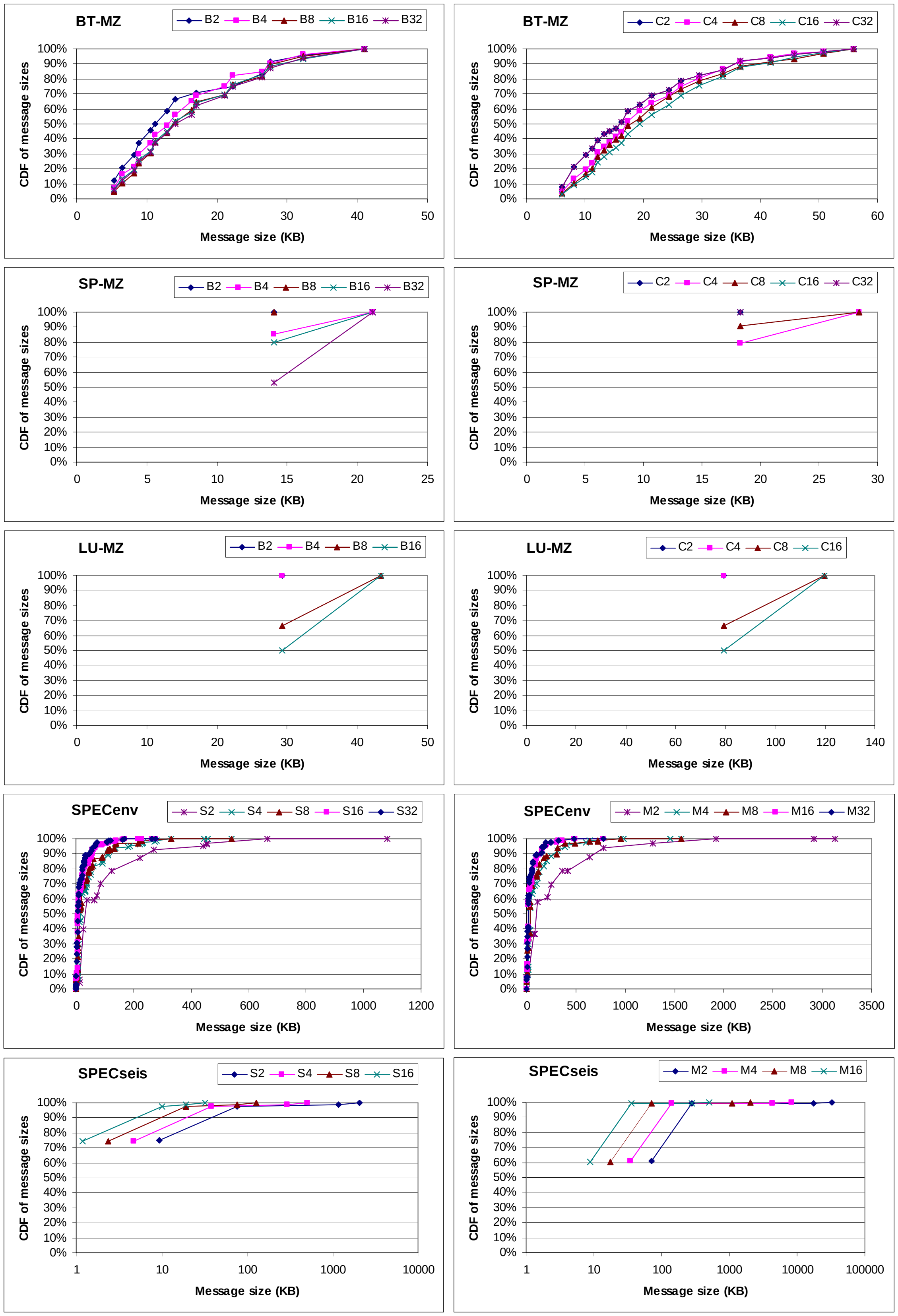


Figure 4.5 Message size CDF of NPB-MZ and SPEChpc2002 applications.

4.2.5 Message Destinations

Spatial behaviour is characterized by the distribution of message destinations [25], [12]. We have studied the number of message destinations for each process in the applications, as shown in Figure 4.6. From the graphs, the number of message destinations per process does not change with the workload for the LU-MZ, and SPEC applications. Processes in the LU-MZ, SP-MZ, and SPECenv (except for some of the processes) have a few favourite communication partners. This is consistent with previous results for other applications [25, 51].

However, processes in BT-MZ (especially the C class) and SPECseis communicate with most of the remaining processes. SPECenv has a very diverse range of number of destinations. For instance in S32, SPECenv has 8 destinations per process on average. However, some processes have 30 destinations and some only have 4 destinations. We can see that except for LU-MZ and SPECseis, other benchmarks have some irregularity in terms of the number of destinations among processes.

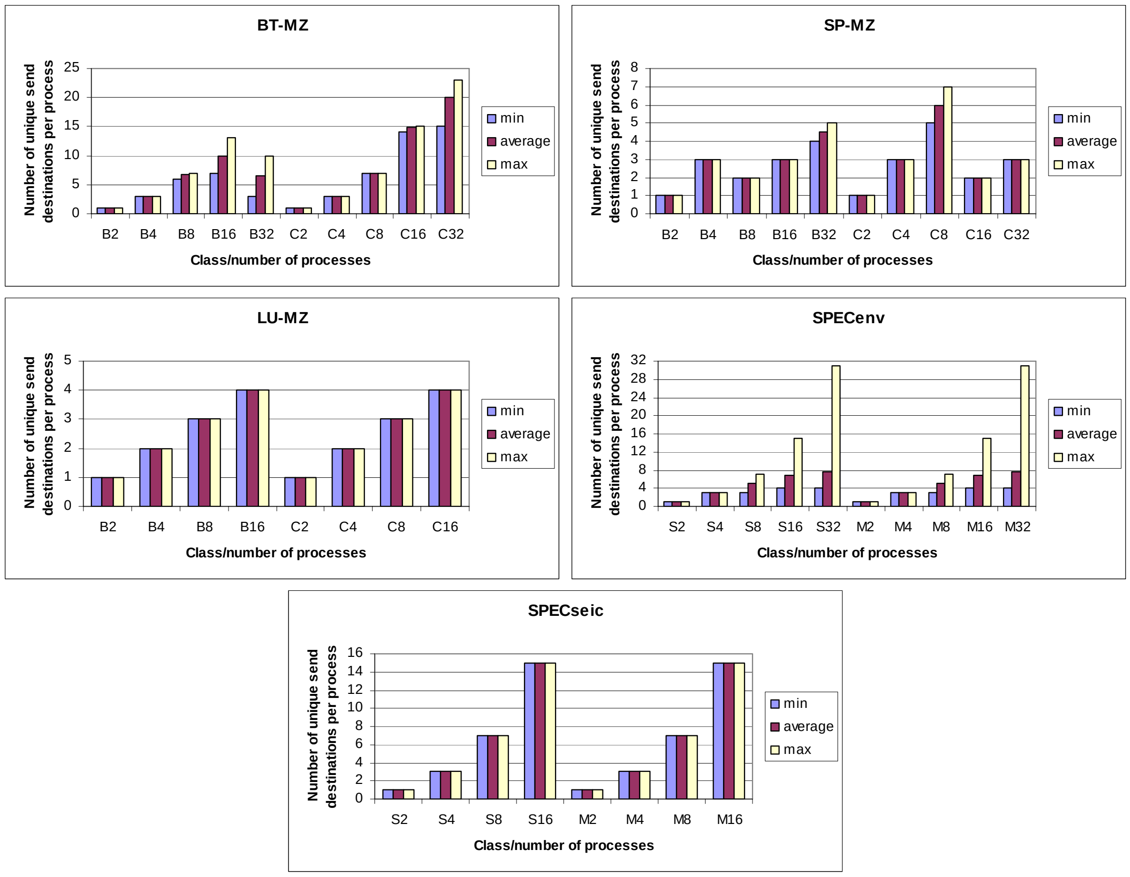


Figure 4.6 Number of destinations per process.

4.2.6 Destination Distribution

As discussed earlier, not all processes in the applications communicate with all other processes. Usually, process zero (root process) is responsible for distributing the data and verifying the results. This makes it a favourite destination for other processes. However, it is interesting to discover the set of destinations for process zero. Figure 4.7 shows the distribution of message destinations for process zero in the applications running with 16 processes.

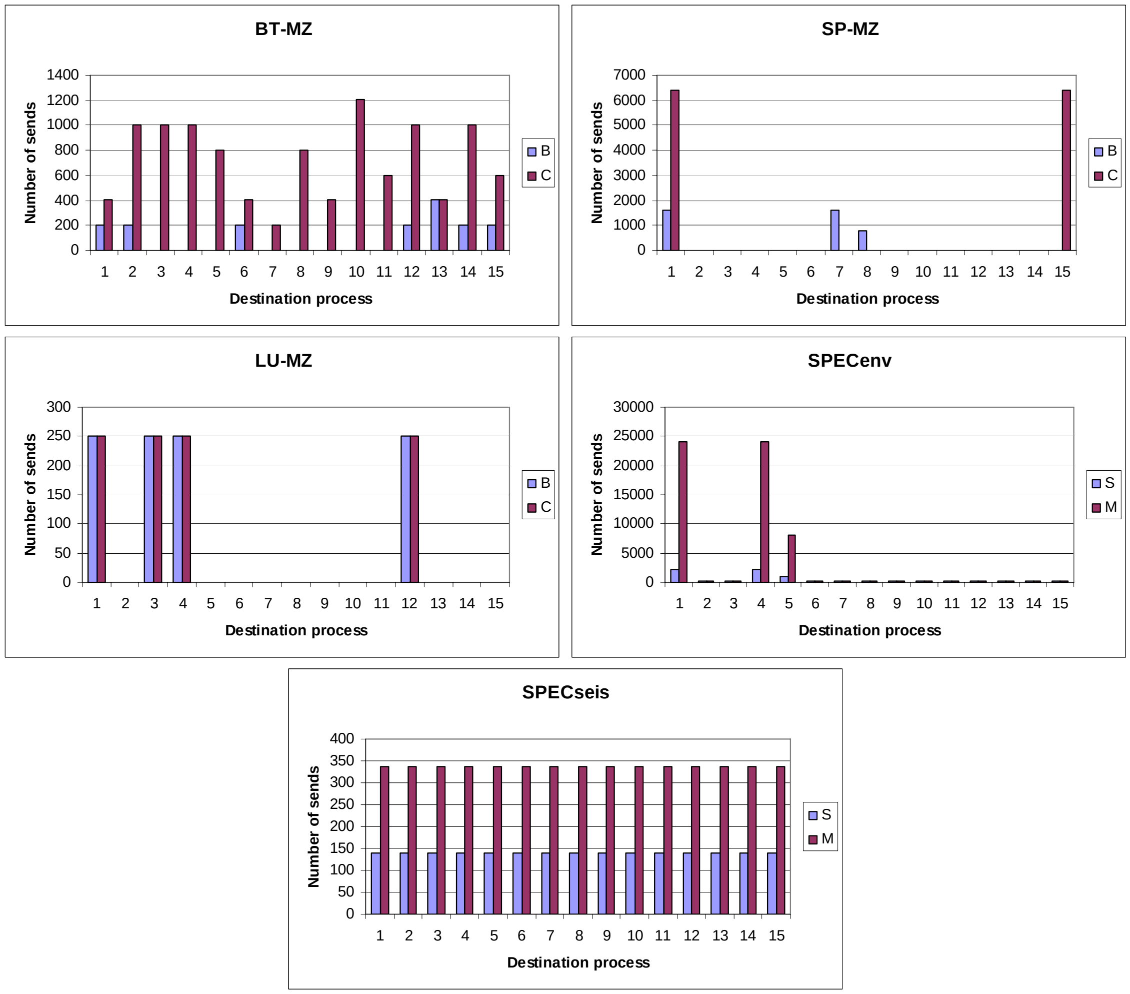


Figure 4.7 Destination distribution of process 0 (16 processes).

Process zero in the SPEC applications, and the BT-MZ (class C) communicates with all other processes. Process zero in the SPECenv communicates infrequently with all other processes, and it has its own favourite partners. Interestingly, process zero in the

SPECseis has a uniform communication pattern. In all other cases, process zero communicates with a subset of all other processes. In SP-MZ and LU-MZ, process zero communicates with two to three other processes and four processes, respectively.

4.3 Collective Communications

Collective communication is widely used in message-passing applications. Collective communication involves more than two processes. Data distribution and synchronization is easier to implement using collective communication. Collective communications can be characterized by the type of the collective operation, frequency of collective operations, and their payload.

Quantitative study of collective communications in our applications is presented in Table 4.1. This table presents the type, frequency, and the payload (in bytes) of the collective operations used in the NPB-MZ and SPEChpc2002 applications. Broadcast, barrier and reduce are the only collective primitives used in these applications. The reduce primitive in the SPECenv uses the “sum” operation. NPB-MZ applications use the “sum”, as well as the “max” operation. SPECenv uses a large number of broadcast operations with very large payloads. NPB-MZ applications use a few number of collective operations with small payload, while SPECenv uses a large number of collective operations with large payloads. SPECseis uses collective operations more frequently than NPB-MZ applications with larger payload. In contrast to the applications Vetter [51] has studied, SPECenv has significant collective payload.

Table 4.1 Collective communications of NPB-MZ and SPEChpc2002 (16 processes)

Application	Class	Number of processes	Number of Broadcasts and payload (bytes)	Barrier	Number of Reduces and payload (bytes)
BT-MZ	B, C	2-32	3 (12)	2	3 (88)
SP-MZ	B, C	2-32	3 (12)	2	3 (88)
LU-MZ	B, C	2-16	7 (64)	2	4 (96)
SPECenv	S	2-32	946 (6631224)	-	1 (16)
SPECenv	M	2-32	2247(102597148)	-	1 (16)
SPECseis	S, M	2-16	38 (23312)	20	-

4.4 Locality Characteristics

It is interesting to discover if message-passing applications exhibit any repetitive communication patterns. For example, there might be repetition patterns in message size, message destination, or even among the send and receive events. Locality metrics are useful in message-property prediction schemes and communication latency hiding techniques. Locality is also useful in predicting the buffer requirements ahead of time to hide the communication latency. It can also be used to set up the communication path in circuit-switch networks and optical networks [2]. Locality can be defined in different ways. In general, locality, regardless of its definition, shows the probability of repetition of a pattern in future based on that definition. It is important to choose a good model of locality to be able to predict next events easier. As locality is a probability metric, its value is between zero and one. The sooner the locality gets to value one, the better it can predict the next events.

Three major locality schemes are widely used, especially in memory replacement policies. These are *First In First Out* (FIFO), *Least Recently Used* (LRU), and *Least Frequently Used* (LFU). These models are studied for message sizes and message destinations of our benchmarks. FIFO, LRU and LFU heuristics all maintain a set of *k* (window size) unique message identifier. If the next message event is already in the set then a *hit* is counted, otherwise a *miss* is counted. If a miss occurs, based on the heuristic the new identifier will be moved to the set.

FIFO is the simplest heuristic. The last *n* unique identifiers are already in the set and if the next identifier is already in the set then number of hits will be increased by one; otherwise, the number of misses will be increased by one. The locality for FIFO is defined as "*hit ratio*"; that is the number of hits divided by the total number of hits and misses. The locality is presented in percentage format. Once a new identifier joins the set, the first identifier that has joined in *k -1* steps ago will be moved out of the set. The youngest member replaces the oldest member of the set.

LRU and LFU heuristics are similar to FIFO. The replacement scheme in LRU is such that the member of the set that the new member replaces with is the one that is least recently used. While applying LRU scheme, if there is a hit then the hit member will be moved from its place to the top of the set and all the other members will be pushed down.

If there is a miss then the last member of the set, which is the least recently used member will be moved out of the set and the new member will be placed at the top of the set and all other member will be pushed down. In the LFU heuristic, usage frequency of all the members is noted and if a miss occurs in the set then the member that has to be moved out is the one that has the least usage frequency. The new member will replace the least used member of the set.

Kim and Lilja [25] introduced the concept of locality for Send/Receive communication calls using the LRU heuristics. Afsahi and Dimopoulos extended the notion of communication locality to the message destinations, and message reception calls using the LRU, LFU and FIFO policies [2]. They then devised different message predictors.

4.4.1 Message Size Locality

As mentioned earlier, FIFO is our simplest locality heuristic. Localities of message sizes based on FIFO heuristic for the five benchmarks are shown in Figure 4.8. Each application has been run with different number of processes. For a given number of processes, message size locality is the average of message size locality for each process. Message size locality varies as the number of processes in the application changes. Message size locality shown here is for window sizes 1 to 16. As the window size gets larger, the probability of a *hit* becomes larger; therefore, localities increase and get closer to the value one. It is clear that applications that do not have many different unique message sizes will reach the maximum value one quickly. In short, number of unique message sizes and window size has the largest effect on locality. BT-MZ and SPECenv have many different unique message sizes and therefore their locality curves approach the value one more smoothly and more slowly than the other applications that have only two or three different unique message sizes.

We realized from Figure 4.8 that for BT-MZ class B, the more processes you have the better locality curve you get. This means that locality grows faster and gives a larger probability of repetition based on FIFO heuristic for larger number of processes. For example, BT-MZ class B reaches 80% locality with 32 processes with the window size of four while the same benchmark with two processes only reaches 70% locality with window size of 13. For BT-MZ class C, the difference between locality curves of different number of processes is less than class B.

The fact that larger number of processes leads to a better locality curve is true for SPECenv as well (except for number of processes equal to two). With two processes, surprisingly, SPECenv has a better locality curve than all the other curves for large number of window sizes. FIFO locality curve of SPECenv class S reaches 100% with window size of nine and for class M it reaches the value one with window size of eight while locality of other number of processes does not even get to the value one with window size of 16. This confirms that SPECenv has a large number of different unique message sizes and each unique message size is not locally repeated in a small number of consecutive sends very often.

Studying the locality of message sizes of LU-MZ and SP-MZ shows that almost more that 50% of the time, the next message sent in the application has the same message size as the previous message. For SPECseis, this percentage is very close to 100% and it means that most of the time messages with the same size are sent consecutively.

Locality of message sizes for LRU heuristics are shown in Figure 4.9. LRU heuristic performance is very similar to FIFO heuristic for our benchmarks. For BT-MZ class B and C, the larger the number of processes, the larger the probability of repetition patterns. For SPECenv class S and M, number of processes equal to two has the highest locality in message size and after that, 32, 16, 8 and 4 processes have higher localities, respectively. Locality of message sizes for LFU heuristics are shown in Figure 4.10. LFU heuristic performs slightly different from FIFO and LRU. The same trend among different number of processes exists that applications running with a higher number of processes have a higher locality (except for a few cases, such as SPECenv with 2 processes).

4.4.2 Message Destination Locality

Each process in a parallel application might have a different style in communicating with other processes. A process might communicate only with one other process, a few other processes, or all other processes. Sequence of communication with other processes might also be different between applications. In this section, we study the message destination locality of *MPI Send/ISend* primitives in NPB-MZ and SPEChpc2002 benchmarks. FIFO, LRU and LFU heuristics are once again used for this study.

FIFO heuristic locality results for message destinations are shown in Figure 4.11. In general, as the window size increases, there is a better chance for a *hit* rather than a *miss*. The larger number of processes in the application means that there is larger number of possible destinations and therefore variety of destinations can decrease their locality. For example, in a two-process application, there are only two possible destinations hence the locality of destinations for window sizes of larger than one would be equal to one while locality for a 32-process application is usually less than 5% in our applications in small window sizes.

Although for large number of processes there are many different possibilities as a message destination, as mentioned earlier not necessarily each process will communicate will all the other processes. This will eliminate some of destinations and will decrease the actual number of destinations for a process; therefore, the locality of message destinations will grow faster than a regular application in terms of message destinations. For example for 32 processes in BT-MZ class B, SP-MZ class B, SP-MZ class C, SPECenv class S and SPECenv class M, message destination locality will be around 100% for window sizes of larger than 9, 5, 2, 7, and 7, respectively for the FIFO heuristic. This shows that on average each process communicates only with 10, 6, 3, 8, and 8 other processes, respectively. We can see that these numbers (except for BT-MZ class B) match very well with the number of message destinations presented in Figure 4.6. In short, BT-MZ and SPECseis processes communicate with many other processes, while SP-MZ and LU-MZ only communicate with a few favourite processes, and SPECenv is somewhere between these two groups.

Message destination localities using the LRU heuristic are presented in Figure 4.12. Similar to message size results, LRU heuristic results for message destinations are very similar to FIFO. SP-MZ and LU-MZ have high message destination localities according to the LRU heuristic, while BT-MZ and SPECseis have low message destination localities. One can say that SPECenv has a medium locality comparing to theses two groups.

Message destination locality results for LFU heuristic is shown in Figure 4.13. The same trend in the locality with LFU heuristic can be seen that BT-MZ and SPECseis have low locality, SPECenv has a medium locality, and SP-MZ and LU-MZ have high locality in message destination. The LFU results are slightly different from FIFO and LRU. In the next section, we compare these heuristic to figure out which one could predict the next identifiers better.

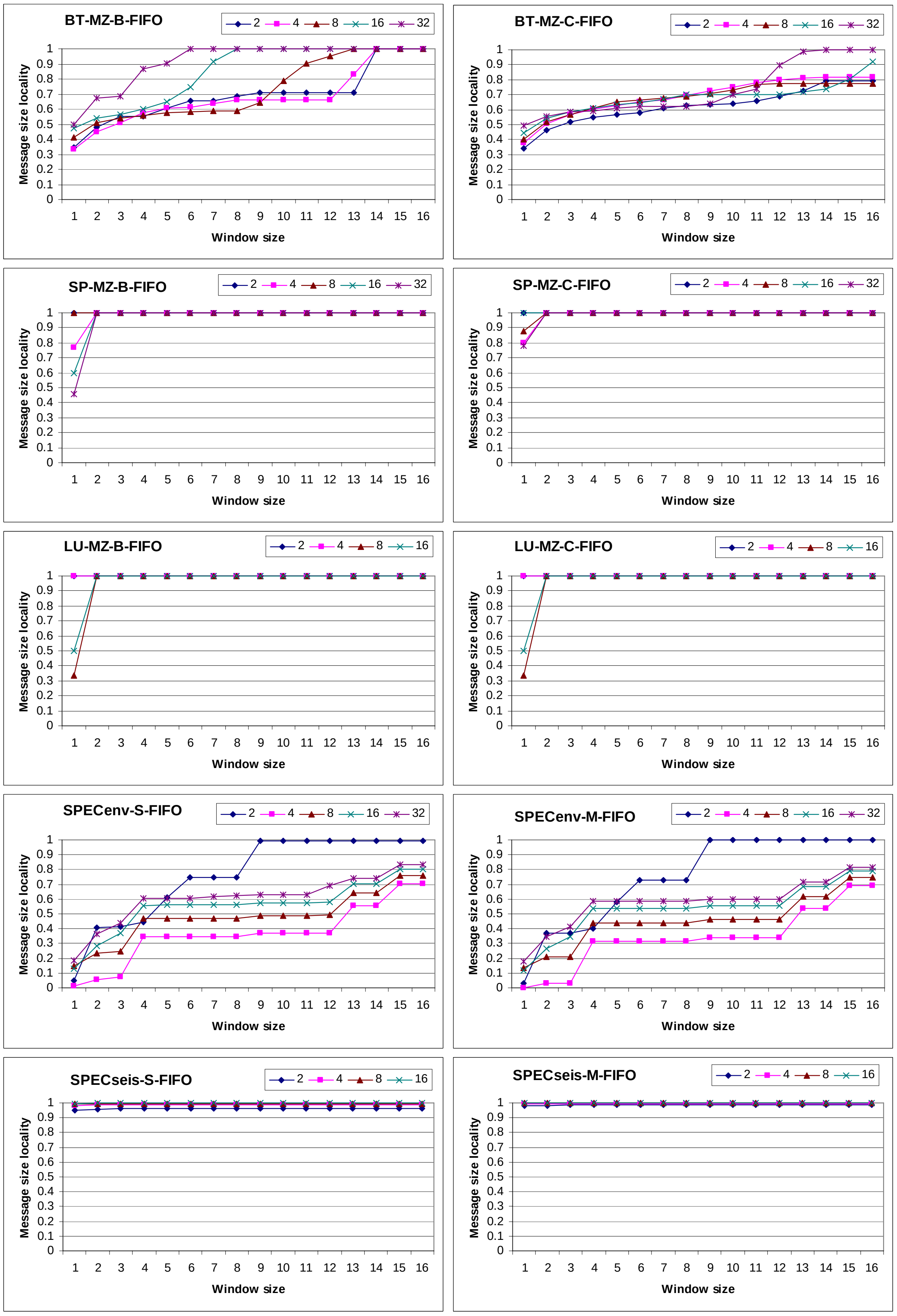


Figure 4.8 Message size locality (FIFO heuristic)

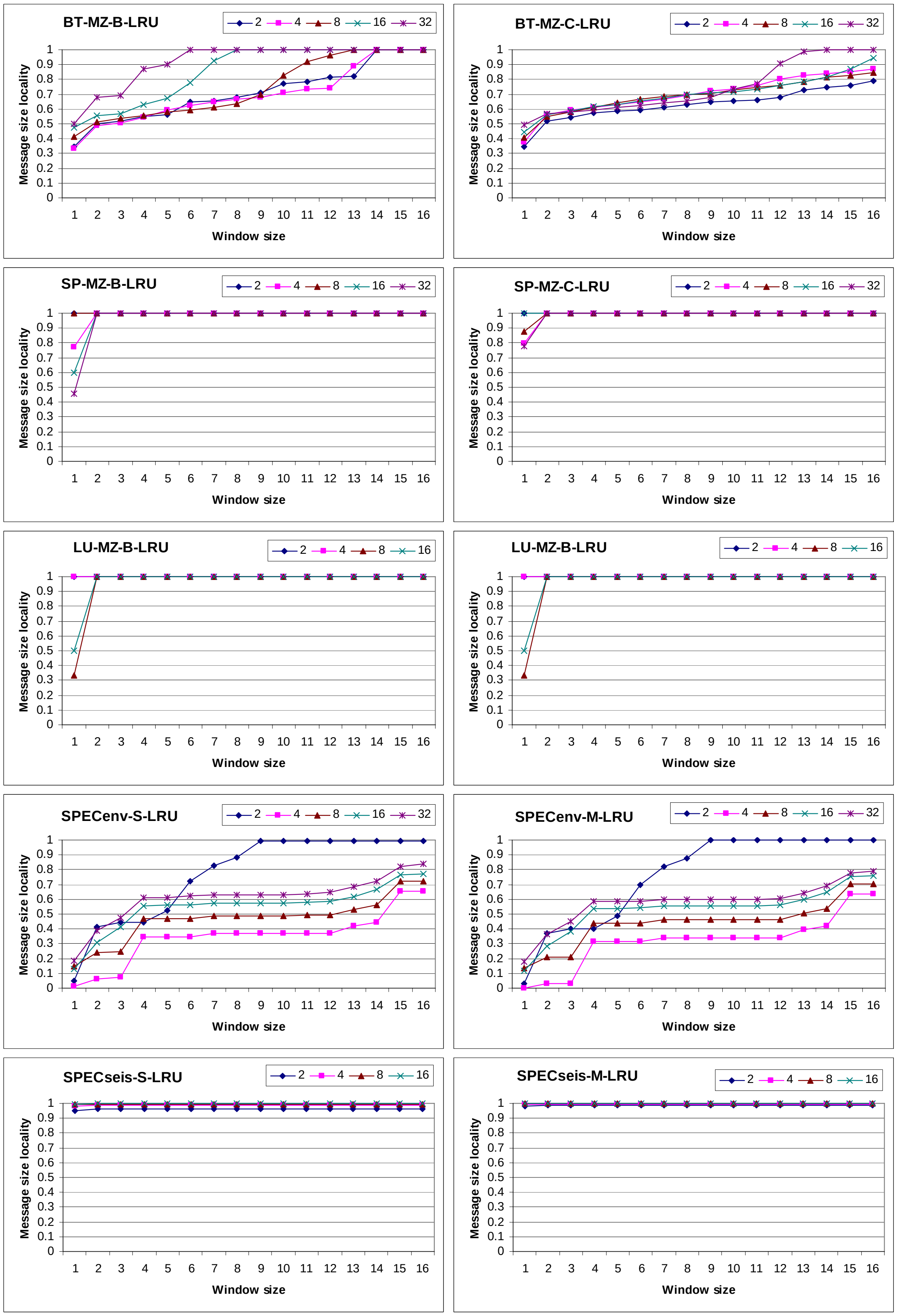


Figure 4.9 Message size locality (LRU heuristic)

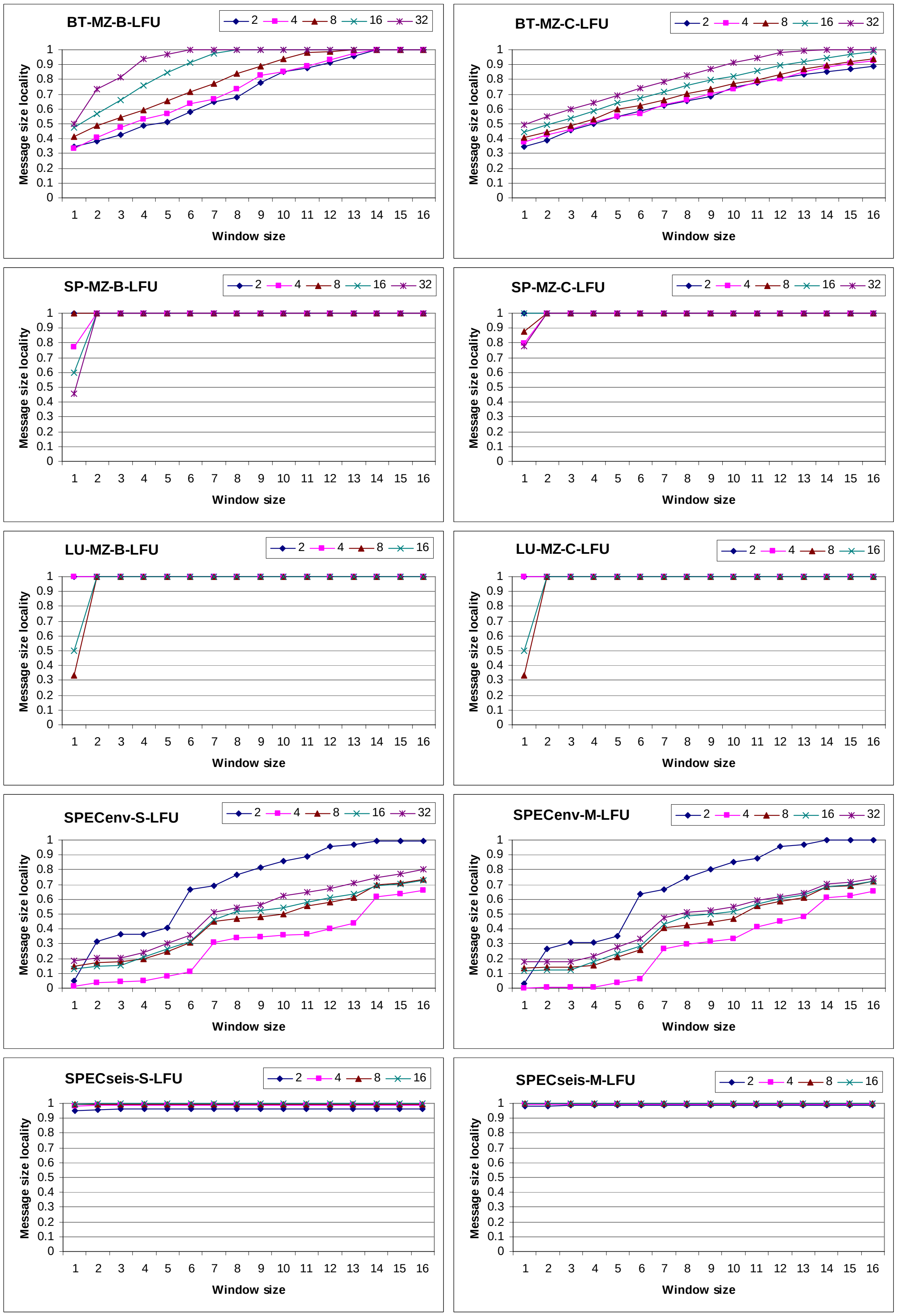


Figure 4.10 Message size locality (LFU)

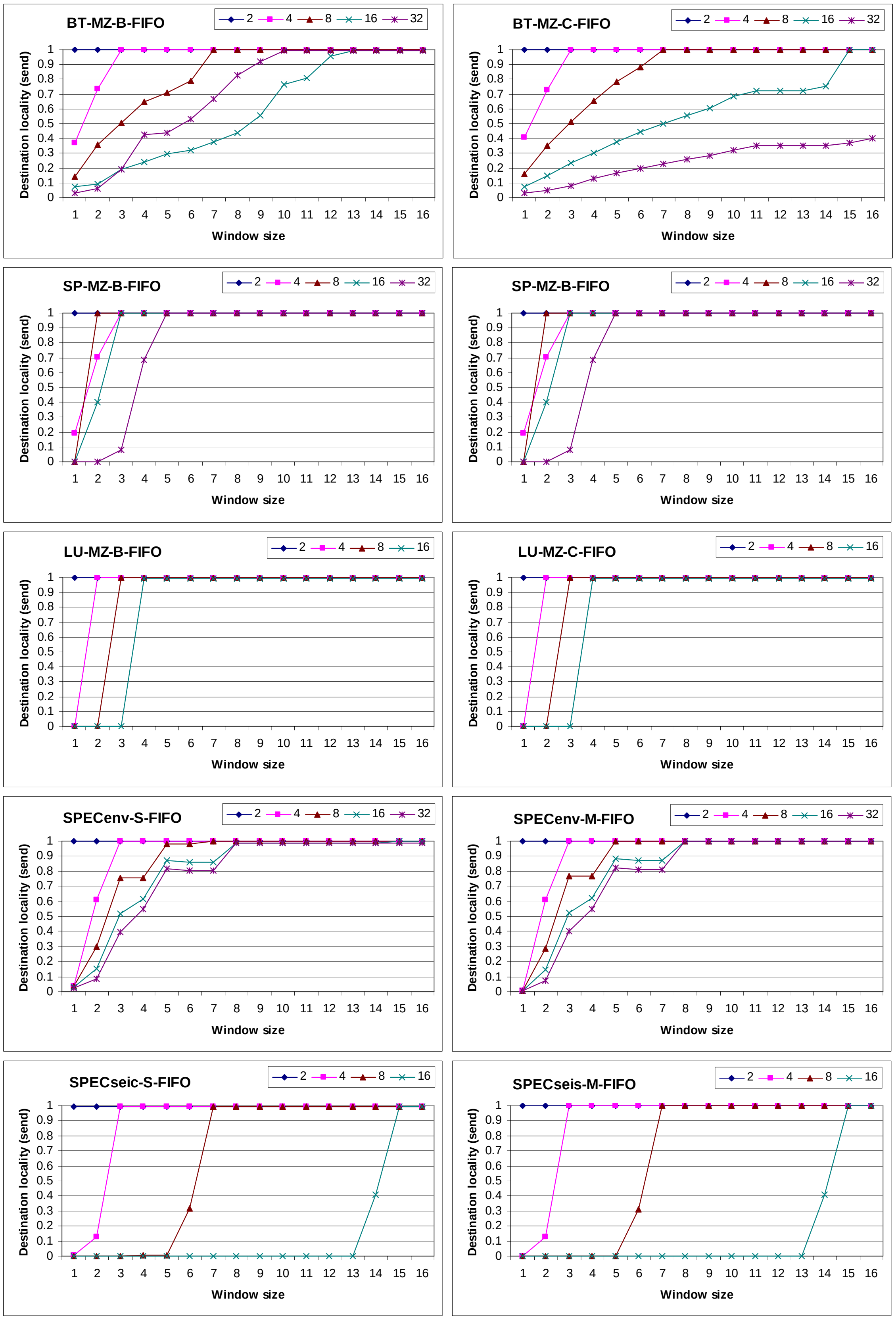


Figure 4.11 Message destination locality (FIFO)

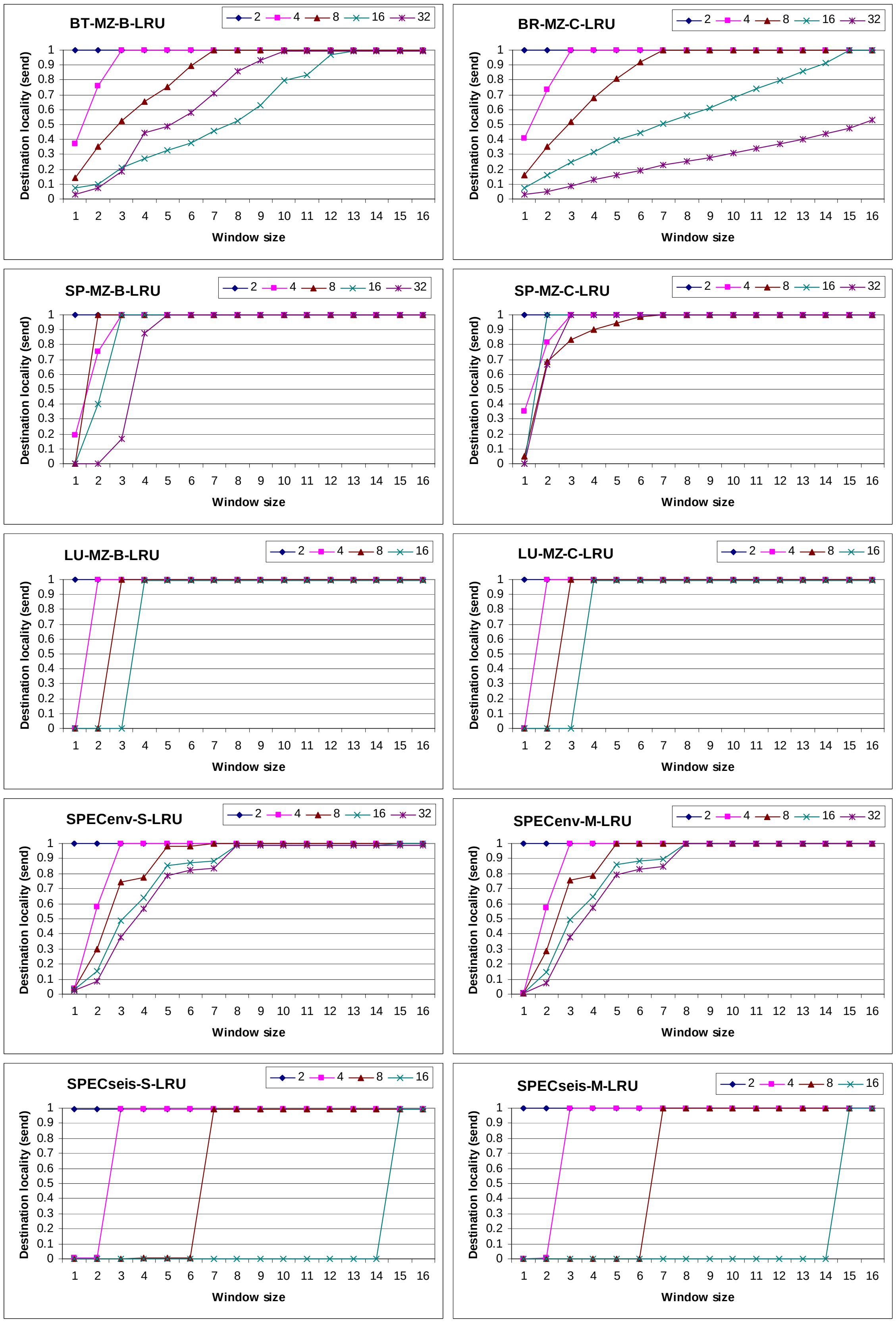


Figure 4.12 Message destination locality (LRU)

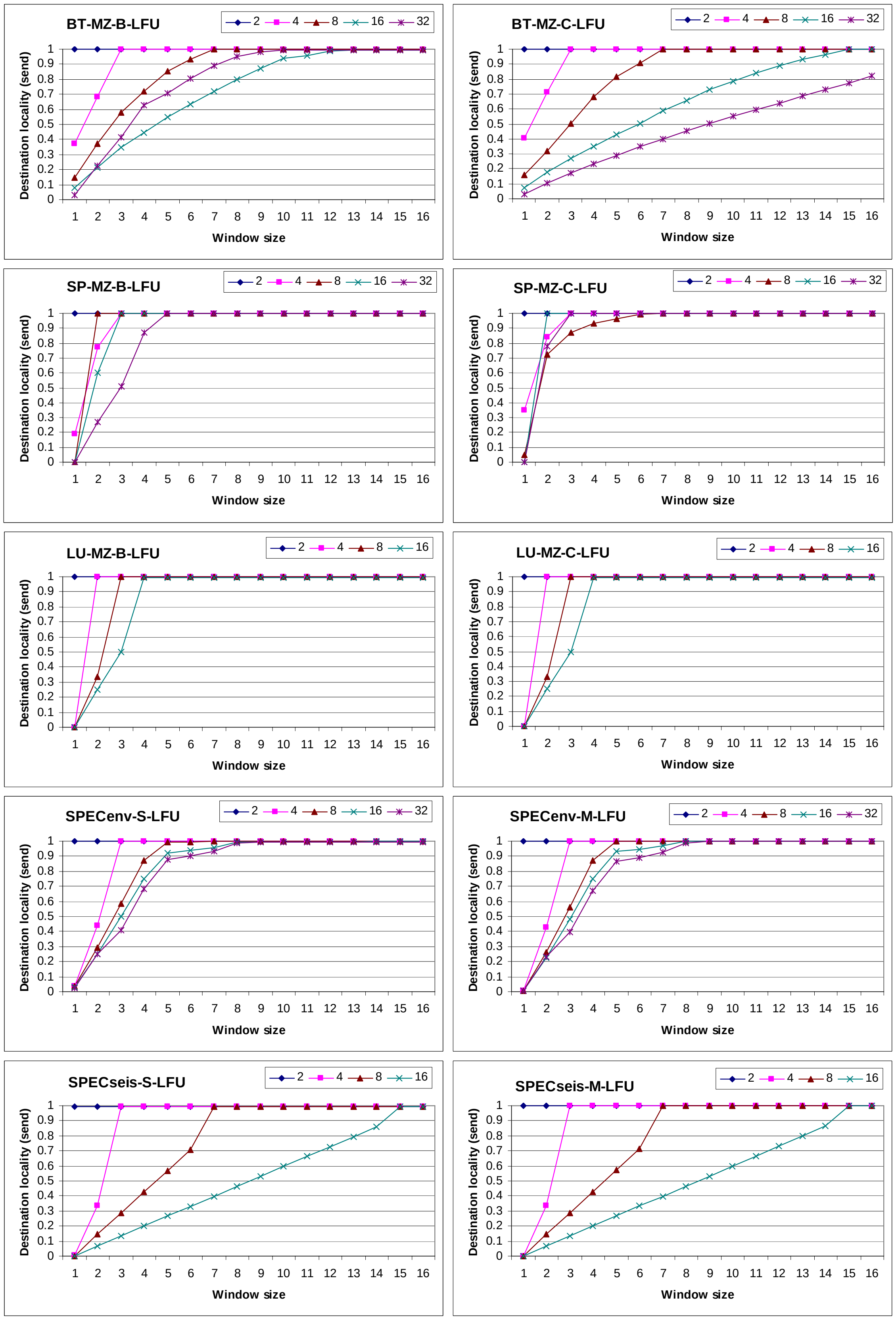


Figure 4.13 Message destination locality (LFU)

4.4.3 Comparison of Localities

In this section, we compare message destination locality and message size locality of all five applications under different classes, while running with only 16 processes. We want to find out which heuristic performs better. The comparison of FIFO, LRU and LFU heuristics for message destinations and message size locality of applications are shown in Figure 4.14.

For BT-MZ class B and C, LFU performs better than LRU and FIFO, both in message size locality and message destination locality. SP-MZ and LU-MZ have a relatively high locality and all three heuristics perform almost the same both for message size and message destination locality. The SPECenv message destination locality is almost the same for all three heuristics, while for message size locality there are small differences between them. For message size locality, LRU and FIFO perform almost the same and better than LFU for both classes of SPECenv. The case is completely the opposite for SPECseis. LFU performs better than LRU and FIFO for message destination locality and all of them perform similarly for message size locality, as SPECseis does not use many unique message sizes.

In summary, LRU and FIFO have a very similar performance. LFU for some applications outperforms LRU and FIFO and sometimes shows a poorer performance. It can be concluded that dependant on the type of the application, different locality schemes should be used to get the most out of prediction schemes.

4.5 Mixed-Mode Communication Characteristics

It is possible to exploit parallelism using a combination of different parallel programming paradigms. Parallel programs that use multiple parallel programming paradigms concurrently are called mixed-mode programs. Prominence of clusters encourages programmers to use MPI in the applications. Meanwhile, the relatively easy programming style in OpenMP and its scalability on shared-memory system makes OpenMP a desirable parallel programming paradigm for SMP nodes. The mixed MPI-OpenMP programming style is therefore one of the most promising parallel programming paradigms for SMP clusters.

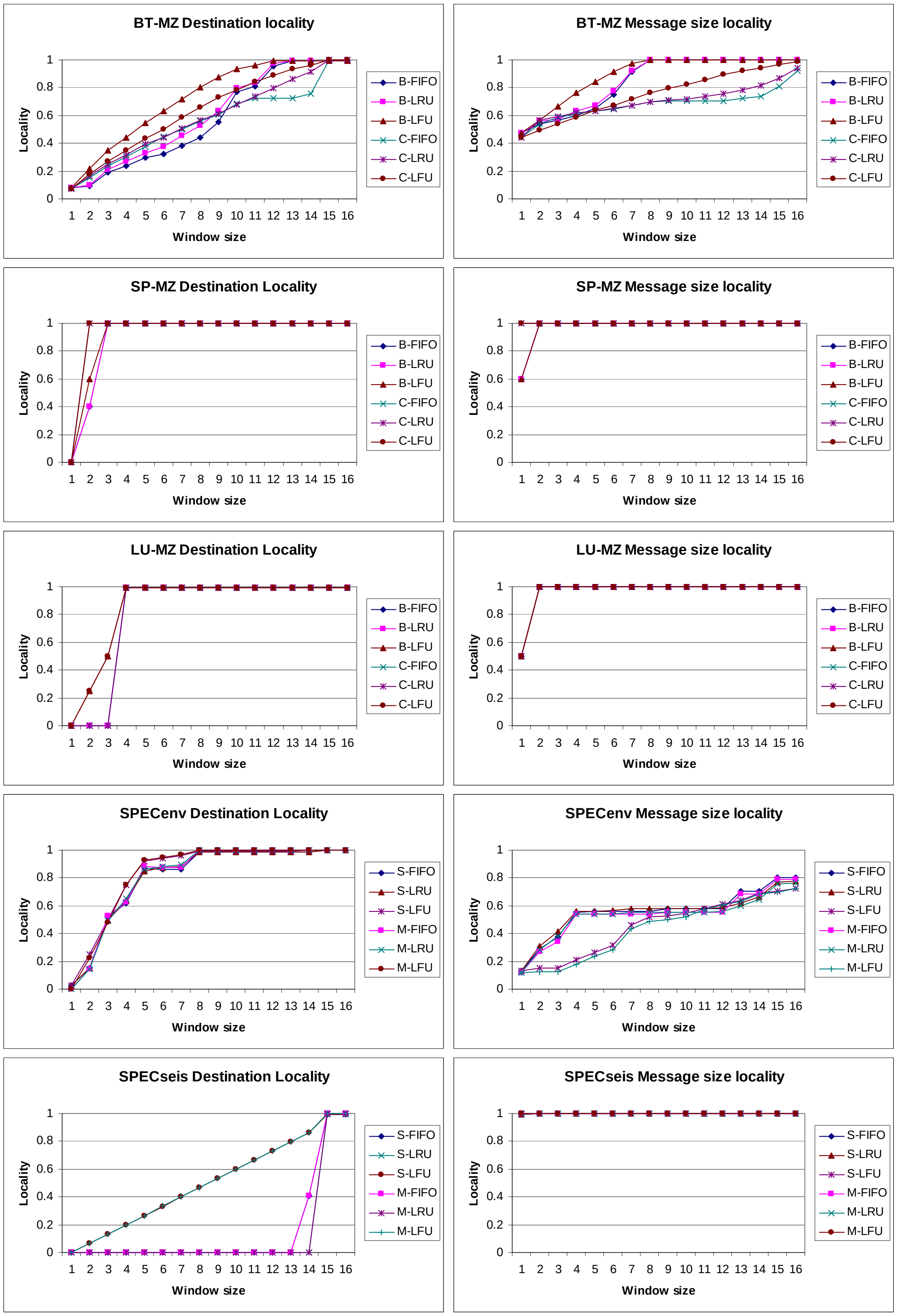


Figure 4.14 Comparison of different locality heuristics (16 processes).

NPB-MZ benchmark is a mixed-mode benchmark and it is possible to use both MPI and OpenMP concurrently. In this section, we look into the communication characteristics of our mixed-mode applications.

4.5.1 Message Frequency

It is possible to run different combinations of threads and processes in mixed-mode applications across a cluster. In this section, we present the message frequency characteristics of NPB-MZ applications for different combinations. For example, some of the possible combinations of processes and threads that one can run on an eight dual-node cluster (16 processors) are 1P16T, 2P8T, 4P4T, 8P4T, and 16P1T. For instance, "8P2T" means that there are eight processes evenly divided among four nodes of the cluster, where each process has two threads running on its respective node. 1P16T is the pure OpenMP case, where there are no MPI communications, thus, we do not consider this case in our figures. 16P1T is the pure MPI case, where there is no OpenMP parallelization. This case is the one we have studied so far in this chapter. In the following, we compare the message frequency of different combinations of processes and threads.

We count the number of send calls, either blocking or non-blocking, for each process in an application. We calculate the average send calls per process for each process and thread combination such as 2P8T, 4P4T, 8P4T, and 16P1T. Figure 4.15 shows the message frequency of NPB-MZ in mixed-mode. It is evident that the number of messages sent in the BT-MZ, SP-MZ, and LU-MZ is decreasing (except for some of the cases where the number of processes is two) with the increasing number of processes. This trend is consistent for both classes B, and C. Total number of exchanged messages in the application is roughly equal to the number of processes times the average number of messages per process. Although number of sent messages per process is decreasing in this trend, the total number of messages exchanged in the applications is increasing.

An interesting observation is that, on average, the SP-MZ-C sends equal number of messages per process when running with 4P4T and 8P2T, where this is not the case for the other applications and process-thread combinations. Among the three applications, SP-MZ-C and BT-MZ-C have the largest number of messages sent per process, respectively. SP-MZ-C sends around 24000 messages per process when running with 4P4T and 8P2T. BT-MZ-C sends approximately 21000 messages per process when running with 2P8T and 4P4T.

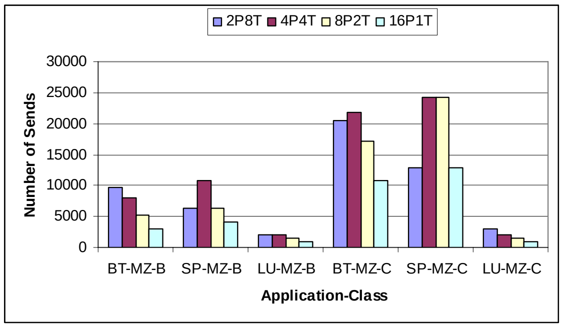


Figure 4.15 Number of messages sent per process in NPB-MZ (mixed-mode).

4.5.2 Average Message Size

In this section, we quantify the average message size of all messages sent in the NPB-MZ applications for different process/thread combinations. This helps us tune our applications when there is a bandwidth limitation in our interconnection network. The average message size of NPB-MZ applications in mixed-mode is presented in Figure 4.16.

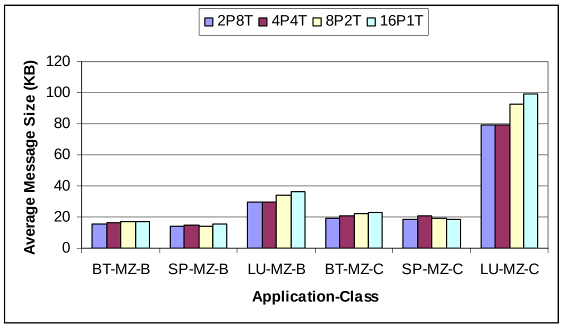


Figure 4.16 Average message size of NPB-MZ applications (mixed-mode).

The average message size for the NPB-MZ application benchmarks do not vary significantly for different combinations of processes and threads, except for LU-MZ that has up to 25% average message size increase when running with 16P1T and 8P2T. For the studied process/thread cases, BT-MZ, SP-MZ, and LU-MZ-B have average message sizes of between 15KB and 40 KB, while LU-MZ-C has average message sizes of between 80KB and 100KB. An overall observation is that the LU-MZ-C is more bandwidth bound than the other NPB-MZ applications.

4.5.3 Message Volume

The total message volume that a process sends over the network is roughly equal to the average message size per send times the number of messages sent per process. We present the average of number of bytes that each process sends over the network, for different combinations of processes and threads, in Figure 4.17.

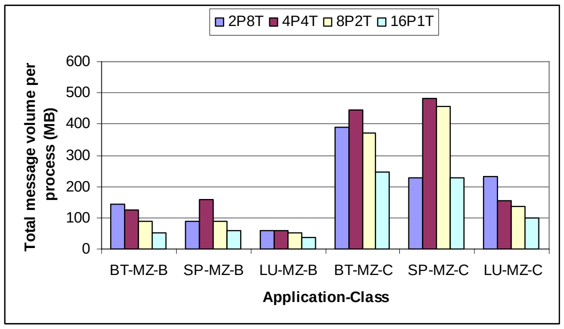


Figure 4.17 Message volume of NPB-MZ applications (mixed-mode).

The general trend, when increasing the number of processes and decreasing the number of threads, is that message volume gets smaller. The total message volume sent by each process in the SP-MZ and BT-MZ-C, when running with 4P4T, is larger than other studied combinations of processes and threads. For the C class, each process in BT-MZ, SP-MZ, and LU-MZ sends 130MB to 440MB, 130MB to 480MB, and 97MB to 232MB, respectively. The total amount of exchanged bytes in the application is equal to the number of processes times the average message volume per process. Although the average message volume per process is decreasing with more processes than threads, the total amount of exchanged bytes in the applications is increasing.

4.5.4 Comparison of MPI and Mixed-Mode Characteristics

By carefully looking at the results presented in previous sections, we notice that the basic characteristics data of NPB-MZ is independent of the number of running threads. By that, we mean that if NPB-MZ applications are running with P processes, then number of threads, T, does not affect the communication characteristics of the application, while P is constant. For example, communication characteristics of 4P1T, 4P2T, and 4P4T will be the same. We investigated this, and found out that there is no MPI communication operations inside the OpenMP parallel regions, thus the OpenMP loop parallelization will not affect the MPI communication characteristics.

However, if a certain number of parallel entities (either processes, or threads in each process) is desirable, then different combinations of threads and processes can be used to run the application in mixed-mode across a cluster. Even by utilizing the same number parallel entities, different process/thread combinations of NPB-MZ have different communication characteristics, as we showed in the previous sections. Based on the application communication characteristics and system latency/bandwidth limitations, one process/thread combination may be more advantageous than the other combinations. Of course, it also depends on the availability of the SMP nodes with sufficient number of processors to support execution of the threads.

4.6 SMG2000 Characteristics

The communication characteristics of SMG2000 have been studied by other researchers [51]. We have used VAMPIR [38] to extract some basic communication characteristics of SMG2000 in this work, as shown in Table 4.2. As there is no standard problem size for SMG20000, we chose 128x64x64 input size for its serial version. If the input problem size is not changed, the total application problem size is proportional to the number of processes in the application. Therefore, we have scaled down the input size proportionally with the number of processes to keep the total problem size constant.

SMG2000 uses short message sizes (smaller than 1KB) compared to NPB-MZ and SPEChpc2002 applications. As the number of processes increases, the average message size becomes smaller. SMG2000 sends more messages when running larger number of processes. SMG2000 uses few collective operations, such as *All_reduce*, *Barrier*, *All_gather*, and *All_gatherv*. The number of collectives used in SMG2000 does not change according to the number of process (except for *All_reduce* with two and four processes).

Table 4.2 MPI Characteristics of SMG2000.

#processes	**2**	**4**	**8**	**16**
#Send calls	44303	51941	52423	50827
Average message size (KB)	0.92	0.54	0.32	0.32
#All_reduce	15	14	14	14
#Barrier	1	1	1	1
#All_gather	1	1	1	1
#All_gatherv	1	1	1	1

4.7 Summary

In this chapter, we have examined the MPI characteristics of small to large-scale scientific applications in terms of their point-to-point and collective communications. We quantified metrics such as message frequency, average message size per message, total message volume per process, message size cumulative distribution function, number of unique message destinations per process, and destination distribution of messages of the root process.

For collective communications, we presented the type, frequency, and the payload. We also evaluated the impact of the problem size and the system size on the communication behaviour of the applications. We found that the applications studied have diverse communication characteristics. Those include very small to very large messages, frequent to infrequent messages, various distinct message sizes, set of favourite destinations, and regular versus irregular communication patterns. Some applications are sensitive to the bandwidth of the interconnect, while others are latency-bound as well. Our evaluation also revealed that most applications are sensitive to the changes in the system size and the problem size. We discovered all applications use only a few collective operations. However, SPEC applications use them frequently with very large payloads. The applications studied have static communication characteristic that do not change between multiple runs.

Overall, the information provided in this chapter will help system designers, application developers, and library/middleware designers to better understand the current and future communication workloads of parallel applications. This study verifies that message-passing applications communicate intensively. Therefore, they will benefit from improvements in the interconnect hardware and their features as well as the communication system software and libraries. Collective communications such as broadcast, barrier, and reduce are expensive operations. Thus, it is essential to optimize their implementation in hardware and/or software in the future computer systems.

We have also gathered the locality characteristics of NPB-MZ and SPEChpc2002 applications. We used the FIFO, LFU, and LRU locality heuristics to evaluate the locality of message size and message destinations in our applications. We found out that LRU and FIFO have a very similar performance. LFU for some applications outperforms LRU and FIFO and sometimes shows a poorer performance. We realized that for BT-MZ class B and C, LFU heuristic performs better than LRU and FIFO heuristics, both in message size locality and message destination locality. SP-MZ and LU-MZ have a relatively high locality and all three heuristics (FIFO, LFU, and LRU) perform almost the same both for message size and message destination locality.

We noticed that all three heuristics have similar results for the SPECenv message destination locality and message size locality. SP-MZ and LU-MZ show a high locality for both message size and message destination. SPECseis message size also shows a high locality, while its message destination locality grows linearly as the window size is increased. Message destination locality of BT-MZ and SPECenv, as well as the message size locality of BT-MZ, show a medium locality compared to the other applications. SPECenv message size locality is low compared to the other applications.

In this chapter, we have also compared the communication characteristic of NPB-MZ applications in the mixed-mode. We found out that different process/thread combinations change the communication characteristics of NPB-MZ. We also realized that MPI communication characteristics of NPB-MZ are independent from the number of threads.

To evaluate the message-passing performance of applications, we will evaluate the basic performance of our Myrinet network in the next chapter. We evaluate the performance of GM, MPI over GM, and ARMCI to see how these messaging libraries really affect the communication performance.

Chapter 5 Myrinet Performance Evaluation

Performance of applications running on clusters mainly depends on the programming paradigm of choice, communication characteristics of the applications, and most importantly on the performance of the communication subsystem. So far, we have studied the parallel applications and their communication characteristics. In this chapter, we assess the performance of the Myrinet interconnect at different layers; that is at the GM level, MPI level and ARMCI level. We measure the performance of the GM basic function calls. We evaluate the latency/bandwidth performance of GM *Send/Receive*, GM RDMA, MPI *Send/Receive*, and ARMCI RDMA operations for one- and two-port configurations of the Myrinet network card interface. This chapter helps in a better understanding of the impact of communication subsystem on the application performance.

5.1 GM Basic Performance

GM [18] is a commercial open source user-level networking protocol from Myricom. GM runs on top of the Myrinet network. Multiple user processes can share a network interface card (NIC) simultaneously as GM provides a protected user-level interface to the NIC. GM provides a connectionless communication model. Communication endpoints in this model are called *ports*. GM provides reliable and ordered delivery between these ports.

GM consists of a driver, a network-mapping program, the GM API library, *Myrinet-Interface Control Program* (MCP), and header files. The NICs that we used in this study, two-port “E-card” Myrinet/PCI-X interface, have been introduced recently by Myricom. They have two ports instead of one as in previous models. It means that each NIC is equipped with two uplink optical fibres and two downlink optical fibres. Having two links for each up- or down-link communication avoids large latency and provides a better bandwidth in case of traffic on one link. The concepts of ports on the NICs and ports in GM should not be mistaken. Ports in GM are software concepts while ports on the NIC are physical fibre links. GM depending on its version has eight or more ports where some of them are reserved for internal use. In this chapter, we compare the performance aspects of the Myrinet networks when the two ports of NIC are operating versus when only one of them is working. All the measurements in this section are done using our own code.

In a program that uses GM, some initializations need to be done. *GM_Init()* has to be called and a GM port for communication should be opened with *GM_Open()*. *GM_Allow_remote_memory_access()* should be called if a program wants to expose and area of its memory so that other programs can write into that memory, with RDMA calls such as *GM_Put()*. *GM_Provide_receive_buffer()* should be called to provide a buffer in case of receiving of a message. At the termination point of the program *GM_Close()* is used to close the used ports. To allocate and pass buffers to these functions, memory should be allocated using *GM_Malloc(),* and should be freed using *GM_Free()*. The overhead of these function calls are presented in Table 5.1. The table shows that terminating a GM program is a very costly operation (*GM_Close*). Exposing memory to other GM processes, as well as providing receive buffers are not very time-consuming operation. Opening GM ports is a time-consuming operation compared to other initialization operations, such as *GM_Init*.

Table 5.1 Timing of Basic GM function calls.

GM Function	**Time (μs)**
GM_Init	57.45
GM_Open	1682.81
GM_Allow_remote_memory_access	0.24
GM_Provide_receive_buffer	0.38
GM_Close	20863.90

Figure 5.1 illustrates the execution time of *GM_Free* for different message sizes. Freeing the allocated memory in GM is not a costly operation and it takes approximately one to two microseconds for various message sizes. In both the Send/Receive and RDMA communication models, communication buffers must be registered/deregistered in the physical memory using *gm_register_memory()* and *gm_deregister_memory()* at both ends to enable DMA transfer in and out of those regions. Performance of these functions for different message sizes is presented in Figure 5.2. Up to 64KB message sizes, registration

time is shorter than deregistration. In general, registration and deregistration are costly operations.

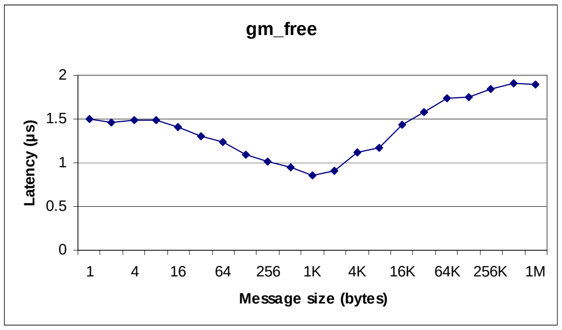


Figure 5.1 *GM_Free* execution time for different message sizes.

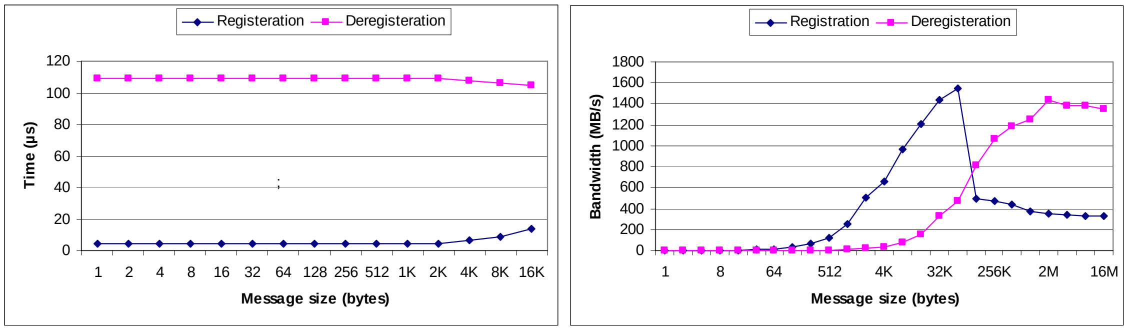


Figure 5.2 GM memory registration and deregistration cost and bandwidth.

5.1.1 GM Send/Receive Performance

In the following, we first define the latency and bandwidth. Then we describe our experimental framework. The message latency and bandwidth are two important metrics for many parallel and distributed computations. 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. Bandwidth is reported as the total number of bytes per unit time delivered during the time measured. In the *unidirectional* latency/bandwidth test, the sender transmits a message repeatedly to the receiver, and then waits for the last message to be acknowledged. The *bidirectional* test is the *ping-pong* test where the sender sends a message and the receiver upon receiving the message, immediately replies with the same message size. This is repeated sufficient number of times to eliminate the transient conditions of the network. In the *both-way* test, both the sender and receiver send data simultaneously. This test puts more pressure on the communication subsystem, and the PCI-X bus.

In the two-sided communication model, the data sender process executes a *Send* operation and the data receiver process executes a *Receive* operation accordingly. Once these *Send* and *Receive* operations match, data transfer between processes completes. GM supports the channel communication model, where it is possible to transfer data using *Send* and *Receive* operations. We measure the unidirectional, bi-directional and both-way latency/bandwidth with different message sizes for GM send/receive. Figure 5.3 shows the latency of GM Send/Receive operation for different messaging schemes using one and two ports of the Myrinet NIC. The short message latency of one- and two-port configurations are not very different; unidirectional messaging has the smallest latency and both-way messaging shows the largest latency.

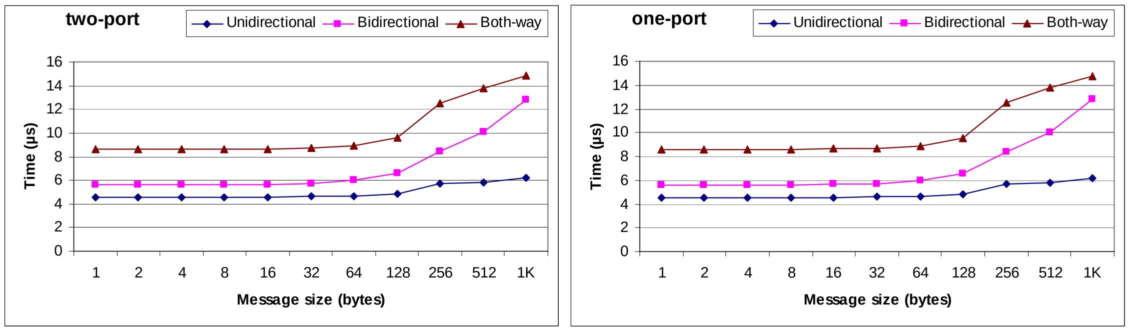


Figure 5.3 GM *Send/Receive* latency in unidirectional, bidirectional and both-way messaging (one/two-port).

Figure 5.4 compares the GM *Send/Receive* bandwidth under three different messaging schemes (Unidirectional, Bidirectional and Both-way) using one and two ports of the NIC. Unidirectional GM *Send/Receive* achieves the maximum bandwidth of 495 MB/s using two ports of the NIC while its bandwidth is 247 MB/s for one port. Bidirectional GM *Send/Receive* reaches the bandwidth of 490 MB/s using two ports of the NIC and 246 MB/s using one port. Both-way messaging has the best bandwidth as both parties are sending messages at the same time. Using two ports of the NIC, GM *Send/Receive* achieves a bandwidth of 766 MB/s and bandwidth of 493 MB/s while using one port. As expected, the bandwidth performance of NIC with the two ports active is much better than the one port. We believe the bandwidth drop for 4KB messages may be due to a protocol change in GM or the fact that GM uses 4KB packets for messaging. We suggest designers of Myrinet to improve this issue in their future systems.

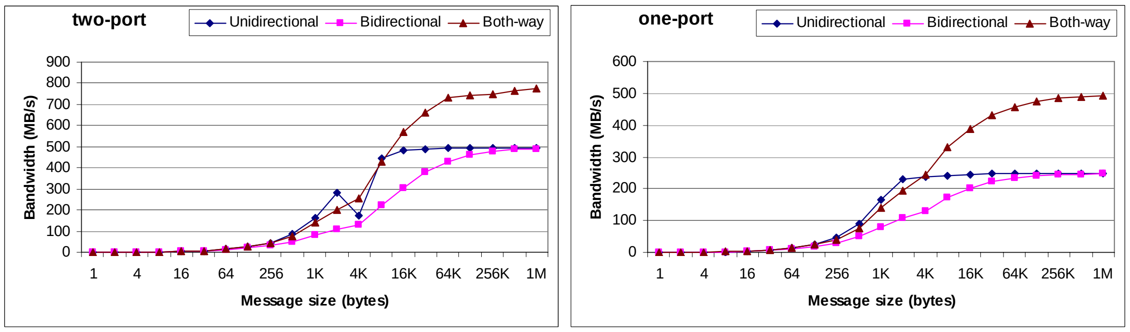


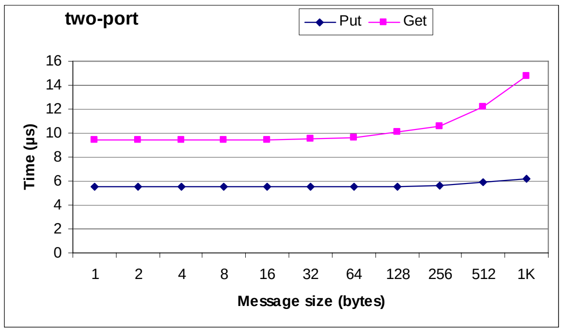
Figure 5.4 GM *Send/Receive* bandwidth in unidirectional, bidirectional, and both-way messaging (one/two-port).

5.1.2 GM RDMA Performance

In one-sided communication model, data transfer is done with participation of only one party, while in two-sided communication model both communication parties have to participate in transferring data. In one-sided communication model, the data sender process executes a *Write/Put* operation and the data receiver process does not need to execute any operation. The data will be written to the memory location according to the *Write/Put* operation. On the other hand, a process is able to receive data from other processes by executing *Read/Get* operation. Again, the process that provides the data does not need to perform any action. Figure 5.5 shows the unidirectional communication latency comparison of GM *put* and GM *get* operations using one and two ports of the NIC. GM *Put* and *Get* operations on one-port configuration have smaller latency than two-port configuration.

Figure 5.6 shows the bandwidth comparison of GM *Put* and *Get* using one- and two-port configurations. GM *Put* achieves bandwidth of 493 MB/s and 247 MB/s for two- and one-port, respectively, while GM *Get* achieves bandwidth of 137 MB/s and 247 MB/s for two- and one-port configurations, respectively. GM P*ut* and *Get* operations have similar latency and bandwidth using one port of the NIC.

In the two-port configuration, the minimum latency of GM *Send/Receive*, *Get*, and *Put* operations are 4.57, 5.50, and 9.40 microseconds, respectively. For message size of larger than 512 bytes, latency of GM *Send/Receive* and *Get* operations are very close to each other (except for message size of 4Kbytes). Latency of GM *Get* operation is larger than GM *Send/Receive* and *Get* operations for all the message sizes of one byte to one megabyte.



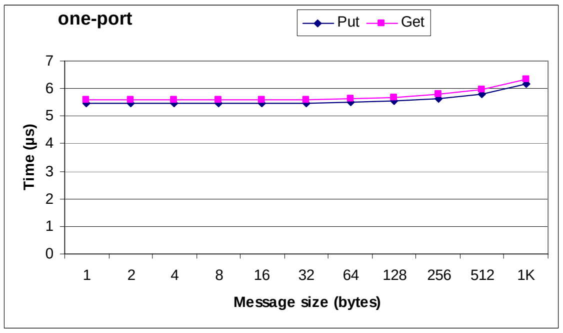
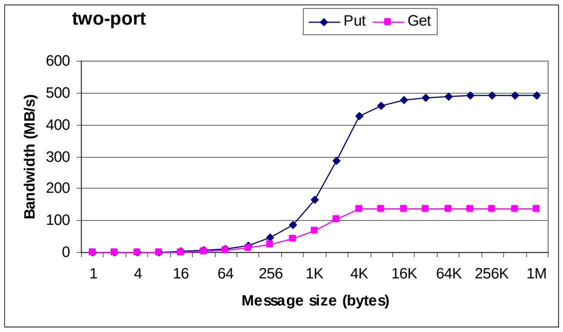


Figure 5.5 GM *Put* and *Get* unidirectional latency.



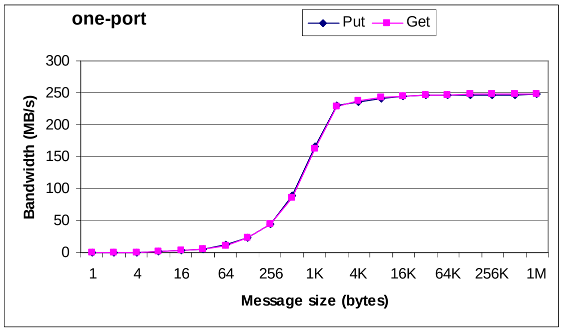


Figure 5.6 GM *Put* and *Get* unidirectional bandwidth.

In the one-port configuration of the system, the minimum latency of GM *Send/Receive*, *Get*, and *Put* operations are 4.57, 5.47, and 5.59 microseconds, respectively. For message sizes of one byte to one megabyte, latency of GM *Send/Receive*, *Get*, and *Put* operations are very close to each other. According to these results, we do not suggest using GM *Get* operation when the system is using two ports of the Myrinet NIC. GM *Get* operation in a two-port configuration has a larger latency than the GM *Get* operation in a one-port configuration. However, due to promising performance of both GM *Send/Receive* and GM *Put*, we recommend using them in GM user-level programming for either one- or two-port configured systems.

5.2 MPI over GM Basic Performance

Many of the HPC applications use MPI as the message-passing library. Finding performance of MPI helps us get a better understanding of communication time of the application. MPI provides point-to-point message-passing and collective operations. One-sided communication is supported in the MPI-2 [30], but we do not address the performance of MPI-2 one-sided operations in this thesis (it is not yet available on top of the Myrinet network). The MPI we used, MPICH-GM, is built on top of the GM layer. All the measurements in this section are performed using our own MPI codes.

Figure 5.7 shows the latency in sending messages with unidirectional, bidirectional, and both-way schemes. The short message latency of MPI *Send/Receive* for different messaging schemes does not differ a lot between one-port and two-port configurations. MPI blocking *Send/Receive* operation has the minimum latency of 5.4 μs while non-blocking *Send/Receive* operation has the latency of 6.0 μs. Among different messaging schemes, bidirectional-messaging using blocking *Send/Receive* operations, shows the largest latency (for messages larger than 128 bytes). As it uses blocking operations and ping-pong synchronization is performed for every message, it is expected that bidirectional-messaging scheme shows a larger latency than the others. In fact, both-way non-blocking *Send/Receive* test shows the smallest latency, because there is no synchronization between the messages and both communication parties send message without waiting for the other one. Using non-blocking operations enhances the latency too.

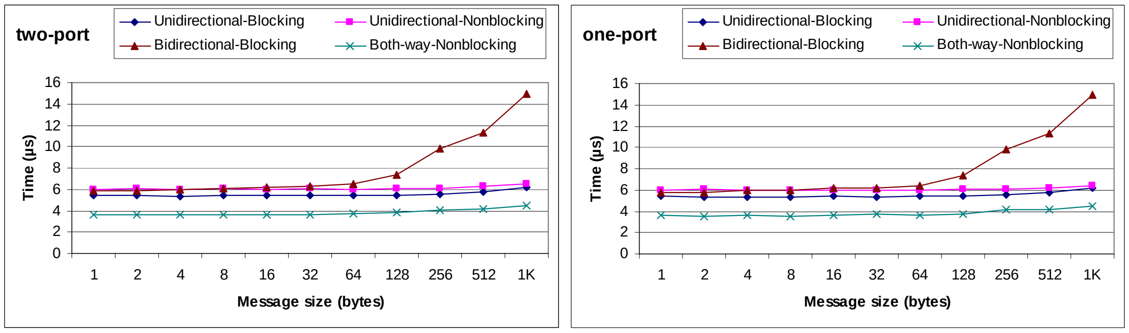


Figure 5.7 MPI blocking and non-blocking *Send/Receive* unidirectional, bidirectional, and both-way latency (one/two-ports).

Figure 5.8 shows the bandwidth under unidirectional, bidirectional, and both-way messaging schemes using one- and two-port configurations. In the two-port configuration, the system reaches the bandwidth of 445 MB/s, 479 MB/s, 443 MB/s, and 742 MB/s for unidirectional blocking, unidirectional non-blocking, bidirectional blocking, and both-way non-blocking messaging schemes, respectively. By using one port of the Myrinet NIC, we achieved bandwidth of 234 MB/s, 243 MB/s, 234 MB/s, and 483 MB/s for unidirectional blocking, unidirectional non-blocking, bidirectional blocking, and both-way non-blocking messaging schemes, respectively. This shows that using both ports of the Myrinet NIC provides a higher bandwidth (50-90% more than one-port). As the MPI library is built on top of the GM layer, we do not expect it surpass the GM performance. GM *Send/Receive* has a higher bandwidth than MPI *Send/Receive*. GM *Send/Receive* achieved 495 MB/s and 247 MB/s for one- and two- port configuration, while MPI achieved only 479 MB/s and 243 MB/s.

Most of the MPI implementations employ a two-level protocol for point-to-point messages. MPI uses *eager* method for sending short messages, while it uses a *rendezvous* mechanism for sending long messages. Eager mechanism improves the latency of messaging while rendezvous mechanism provides a better bandwidth. In Figure 5.8, one can see the bandwidth suddenly drops at around 16KB messages. It then increases until reaching its maximum. This is due to the protocol change in MPI as explained above.

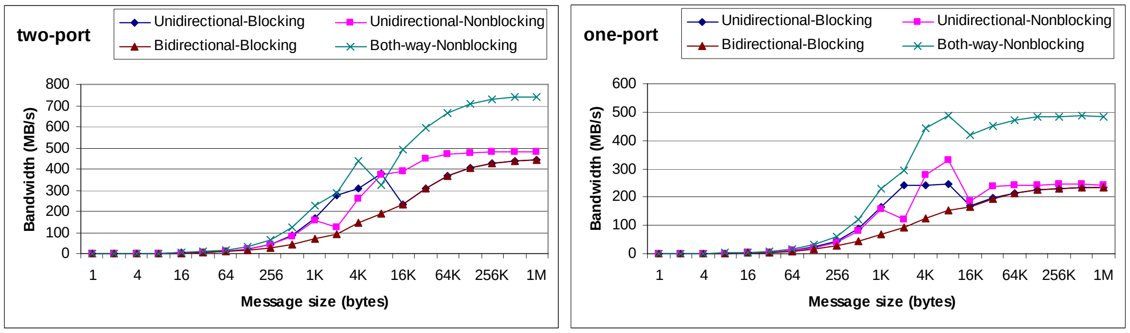


Figure 5.8 MPI blocking and non-blocking *Send/Receive* unidirectional, bidirectional, and both-way bandwidth (one/two-ports).

5.3 ARMCI Basic Performance

ARMCI [3] is a library that provides general purpose, efficient and widely portable RMA operations for contiguous and non-contiguous data transfers. If data is stored in only one location in the memory, it is called contiguous data. On the other hand, if data is stored in multiple locations in the memory, it is called non-contiguous data. Using multidimensional arrays in programs and sending data from different parts of it to another process, is a simple example of a non-contiguous message-passing. In scientific computing using non-contiguous data is popular. The performance comparison of contiguous and non-contiguous ARMCI operations is done by other researchers [33]. In this chapter, we focus on raw performance of the messaging libraries. Therefore, we assess the performance of contiguous data transfer of ARMCI. All the measurements in this part are performed using our ARMCI codes for measuring latency and bandwidth.

As we mentioned earlier, ARMCI uses client-server architecture in clusters of workstations using GM [35]. Each node of the cluster has a server thread that handles remote memory operations for each of the user processes running on the node. In order to assess the best performance of the system in our measurements, we leave one processor dedicated to the ARMCI server thread.

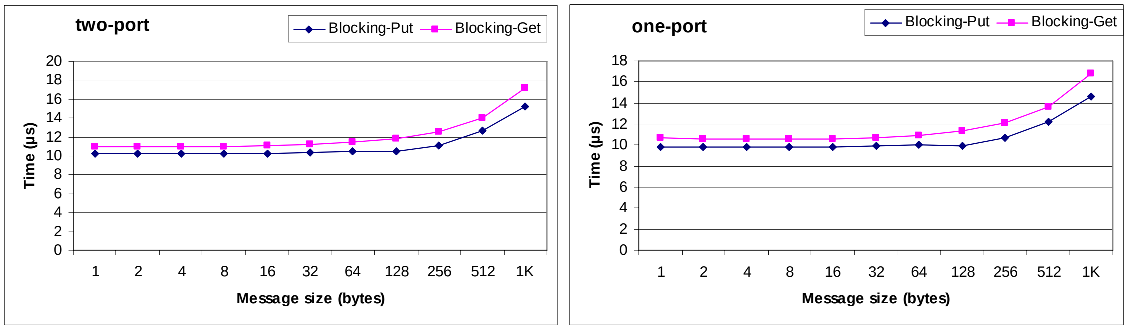


Figure 5.9 ARMCI blocking *Put* and *Get* latency (one/two-port).

ARMCI provides data transfer operations including *Put*, *Get* and *Accumulate*. Blocking and non-blocking one-sided communication is supported in ARMCI. Figure 5.9 compares the latency of ARMCI blocking *Put* and blocking *Get* operations for both one- and two-port configurations of the Myrinet NIC. ARMCI blocking *Put* operation has smaller latency than ARMCI blocking *Get* operation using either one or two ports of the Myrinet NIC. However, we do not see a big difference between their latencies. The smallest latency of ARMCI blocking *Put* is 9.8 microseconds using one port of the Myrinet NIC, and similarly 10.6 microseconds for ARMCI blocking *Get*. When using two ports of the Myrinet NIC, the smallest latency of ARMCI blocking *Put* is 10.3 microseconds and similarly 11.0 microseconds for ARMCI blocking *Get*.

Figure 5.10 compares the latency of ARMCI non-blocking *Put* and non-blocking *Get* operations for both one- and two-port configurations of the Myrinet NIC. Similar to blocking operations of ARMCI, non-blocking *Put* operation has smaller latency than non-blocking *Get* operation using either one or two ports of the Myrinet NIC. Non-blocking *Put* and *Get* latencies have very close latencies. The smallest latency of ARMCI non-blocking *Put* is 5.4 microseconds using one port of the Myrinet NIC, and similarly 5.5 microseconds for ARMCI non-blocking *Get*. When using two ports of the Myrinet NIC, the smallest latency of ARMCI non-blocking *Put* is 5.4 microseconds and similarly 5.6 microseconds for ARMCI non-blocking *Get*.

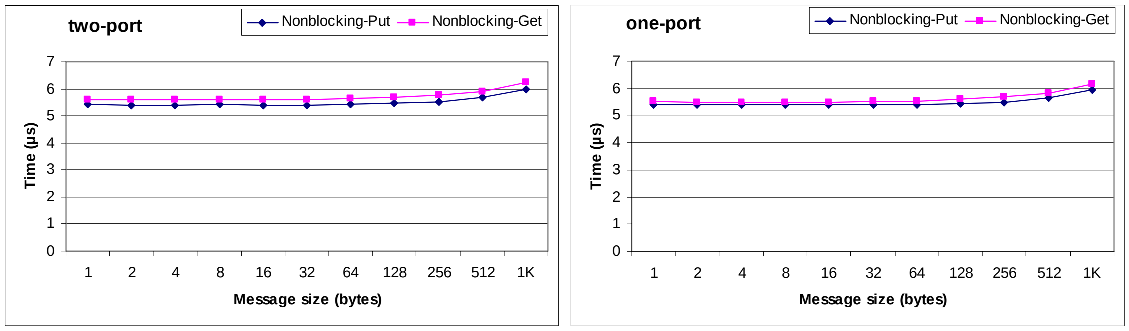


Figure 5.10 ARMCI non-blocking *Put* and *Get* latency (one/two-port).

Figure 5.11 presents the bandwidth comparison of ARMCI blocking *Put* and *Get* on one- and two-port configuration of the Myrinet NIC. Bandwidth of blocking *Put* and *Get* operations on the one-port configuration are very similar. Maximum bandwidth of 246 MB/s is reached with blocking *Put* and *Get* operations on the one-port configuration. When using two ports of the Myrinet NIC, ARMCI blocking *Put* outperforms the ARMCI blocking *Get* operation and it reaches to a bandwidth of 489 MB/s while the bandwidth of ARMCI blocking *Get* is 133 MB/s for message sizes between 32 KB and 1MB. ARMCI blocking *Get* reaches the bandwidth of 190 MB/s at the message size of 8KB. We investigated the blocking bandwidth drop at 16KB messages under two-port configuration, but it is not quite clear whether this drop is due to the ARMCI library or the GM implementations.

Figure 5.12 presents the bandwidth comparison of ARMCI non-blocking *Put* and *Get* on one- and two-port configuration of the Myrinet NIC. Similar to the blocking operations, bandwidth of non-blocking *Put* and *Get* operations on the one-port configuration are very close. Maximum bandwidth of 247 MB/s is reached with non-blocking *Put* and *Get* operations on the one-port configuration. Similar to blocking operations, when using two ports of the Myrinet NIC, ARMCI non-blocking *Put* outperforms the ARMCI non-blocking *Get* operation and it reaches to a bandwidth of 494 MB/s while the bandwidth of ARMCI non-blocking *Get* is 133 MB/s for message sizes between 32 KB and 1MB. ARMCI non-blocking *Get* reaches the bandwidth of 362 MB/s at the message size of 4KB.

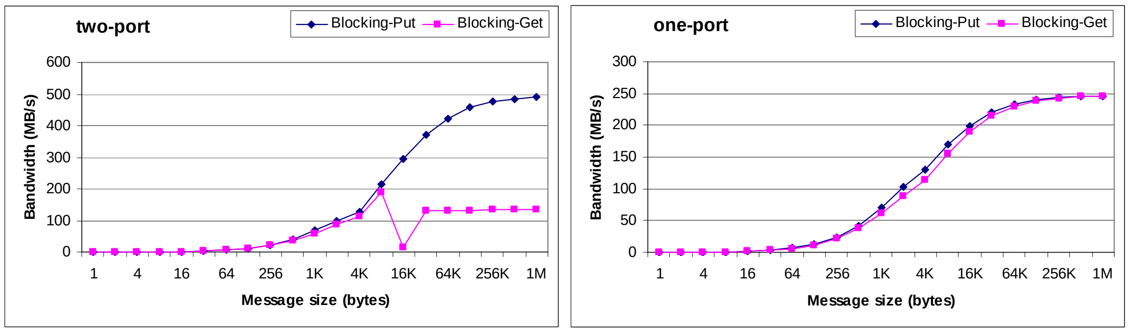


Figure 5.11 ARMCI blocking *Put* and *Get* bandwidth (one/two-port).

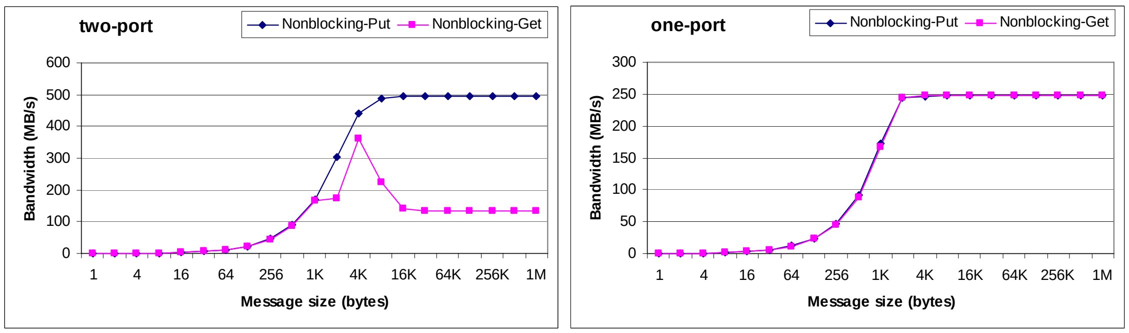


Figure 5.12 ARMCI non-blocking *Put* and *Get* bandwidth (one/two-port).

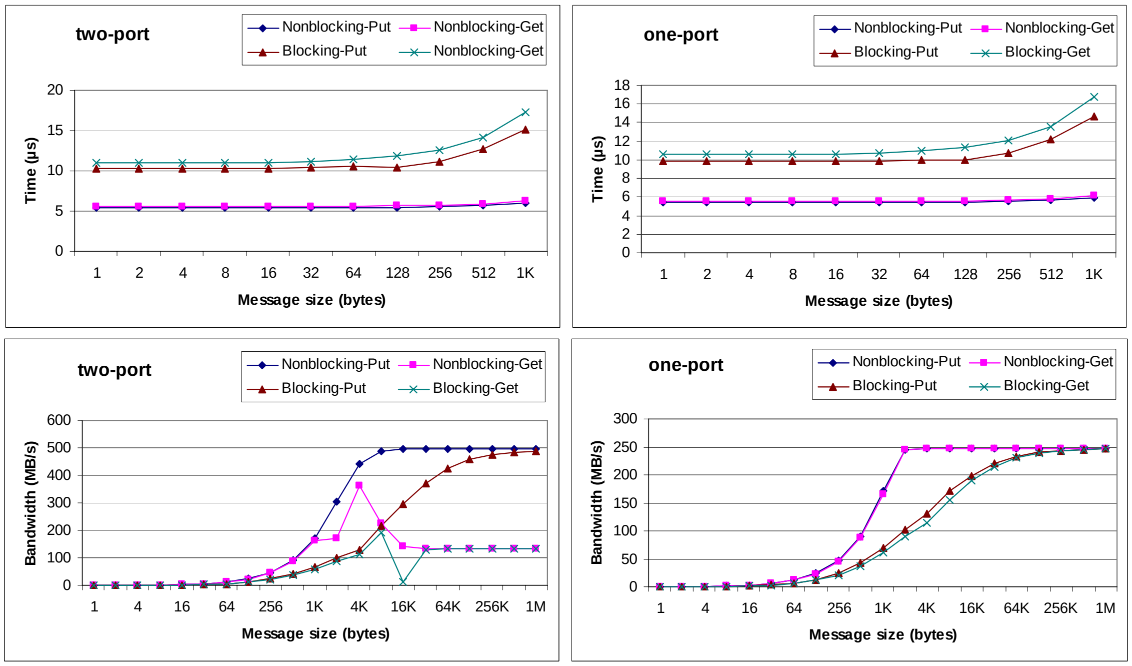


Figure 5.13 Latency and bandwidth comparison of ARMCI blocking and non-blocking *Put* and *Get*.

Obviously, the non-blocking ARMCI operations show a better performance than the blocking ones. For both blocking and non-blocking ARMCI operations, *Put* performs better, when both ports of the Myrinet NIC is utilized, while *Get* operation performs better on a one-port configuration of the Myrinet NIC. Figure 5.13 compares ARMCI blocking and non-blocking RDMA operations together. It is evident that non-blocking *Put* has the best performance among blocking and non-blocking RDMA operations, while blocking *Get* shows a poor performance. *Get* operations are very inefficient compared to two-port configuration of the Myrinet NIC.

5.4 Overall Performance Comparison

It is possible to utilize different communication libraries in the parallel applications. We evaluated the basic performance of three different communication libraries: GM, MPI, and ARMCI, in the previous sections. It is important to understand the performance differences of these libraries in order to tune the performance of applications on clusters. In this section, we compare the latency and the bandwidth of

MPI *Send/Receive*, GM *Send/Receive*, GM *Put* and *Get*, and ARMCI *Put* and *Get* operations.

Table 5.2 compares the latency of MPI *Send/Receive*, GM *Send/Receive,* GM *Put* and *Get*, and ARMCI non-blocking *Put* and *Get* operations. We showed in the previous section that ARMCI non-blocking *Put* and *Get* operations outperform the ARMCI blocking *Put* and *Get* operations, therefore we do not include those in the table. Short message latency of MPI blocking and non-blocking *Send/Receive*, GM *Send/Receive,* GM RDMA *Put*, and ARMCI non-blocking RDMA *Put* and *Get* operations are very close and between 4.6 and 6.0 µs on one- and two-port utilized NICs. GM RDMA *Get* shows a larger short message latency of 9.4 µs compared to the other operations when using two ports of the Myrinet NIC, while it has 5.6 µs latency on the one-port configuration.

When utilizing two ports of the Myrinet NIC, GM *Put* shows the best bandwidth amongst the MPI, GM, and ARMCI operations. MPI blocking and non-blocking *Send/Receive*, GM *Send/Receive*, GM *Put*, and ARMCI non-blocking *Put* operations achieve bandwidth of 424 MB/S, 457 MB/S, 472 MB/S, 470 MB/S, and 471 MB/S, respectively. Both GM and ARMCI *Get* operations perform poor compared to the *Put* operations or two-sided message-passing operations. GM and ARMCI *Get* operations achieve bandwidth of 130 MB/S and 127 MB/S, respectively. The bandwidths calculated above are for the message size of one megabyte.

When using one-port configuration of the Myrinet NIC, interestingly, MPI non-blocking *Send/Receive*, GM *Send/Receive*, GM *Put* and *Get*, and ARMCI non-blocking *Put* and *Get* operations all achieve a bandwidth between 232 MB/S and 236 MB/S. MPI blocking *Send/Receive* achieves bandwidth of 223 MB/S for one megabyte message sizes.

Table 5.2 Latency of MPI, GM, and ARMCI operations in μs (one- and two-port).

two-port	MPI (Send/Receive)		GM			ARMCI (Nonblocking)	
Message	Blocking	Nonblocking	Send/Recv	Put	Get	Put	Get
1	5.4	6.0	4.6	5.5	9.4	5.4	5.6
2	5.4	6.1	4.6	5.5	9.4	5.4	5.6
4	5.4	6.0	4.6	5.5	9.4	5.4	5.6
8	5.4	6.0	4.6	5.5	9.4	5.4	5.6
16	5.4	6.0	4.6	5.5	9.5	5.4	5.6
32	5.4	6.0	4.6	5.5	9.5	5.4	5.6
64	5.4	6.0	4.7	5.5	9.6	5.4	5.6
128	5.5	6.1	4.9	5.6	10.1	5.5	5.7
256	5.6	6.1	5.7	5.7	10.6	5.5	5.8
512	5.8	6.3	5.8	5.9	12.2	5.7	5.9
1K	6.2	6.5	6.2	6.2	14.8	6.0	6.2
2K	7.5	16.6	7.2	7.1	19.6	6.7	11.9
4K	13.4	15.9	23.4	9.5	30.2	9.3	11.3
8K	21.6	22.1	18.5	17.8	60.2	16.8	36.6
16K	70.6	41.9	34.1	34.3	119.8	33.2	116.6
32K	106.7	73.5	67.1	67.5	239.1	66.3	246.1
64K	179.6	138.9	133.3	133.8	477.9	132.6	491.9
128K	325.1	274.3	265.5	266.5	955.5	265.0	983.9
256K	611.1	545.0	530.0	531.9	1910.7	530.2	1967.5
512K	1197.3	1090.6	1059.0	1062.7	3820.7	1060.3	3934.7
1M	2355.2	2185.1	2117.1	2124.3	7641.2	2120.5	7868.3
one-port							
1	5.4	6.0	4.6	5.5	5.6	5.4	5.5
2	5.4	6.0	4.6	5.5	5.6	5.4	5.5
4	5.4	6.0	4.6	5.5	5.6	5.4	5.5
8	5.4	6.0	4.6	5.5	5.6	5.4	5.5
16	5.4	6.0	4.6	5.5	5.6	5.4	5.5
32	5.4	6.0	4.6	5.5	5.6	5.4	5.5
64	5.4	6.0	4.7	5.5	5.6	5.4	5.5
128	5.5	6.1	4.8	5.5	5.7	5.4	5.6
256	5.6	6.1	5.7	5.6	5.8	5.5	5.7
512	5.7	6.2	5.8	5.8	6.0	5.6	5.8
1K	6.2	6.5	6.2	6.2	6.3	6.0	6.2
2K	8.4	17.2	9.0	8.9	8.9	8.4	8.4
4K	16.8	14.7	17.4	17.3	17.3	16.6	16.6
8K	33.3	24.9	33.8	33.9	33.8	33.2	33.1
16K	98.0	88.0	66.9	67.0	66.9	66.3	66.2
32K	167.3	139.0	133.0	133.2	132.9	132.5	132.3
64K	306.1	271.0	265.2	265.7	265.1	265.0	264.5
128K	583.8	538.9	529.6	530.6	529.4	530.0	528.9
256K	1139.3	1074.6	1058.4	1060.4	1057.9	1059.9	1057.9
512K	2251.8	2150.9	2116.3	2120.1	2115.0	2119.7	2115.7
1M	4472.6	4304.5	4231.8	4239.5	4229.3	4239.4	4231.4

5.4.1 Observations

As GM provides low-level functions, it is not suitable for developing applications. ARMCI provides higher-level functions than GM. However, ARMCI functions are mostly used in implementing communication libraries, such as Global Array [36]. We look into the potential benefits of using ARMCI functions in applications. Figure 5.14 compares the latency and bandwidth of MPI *Send/Receive*, GM *Send/Receive*, and ARMCI blocking and non-blocking *Put* on one- and two-port configurations of the Myrinet NIC. One can see that after message size of 8KB, ARMCI operations perform more efficiently than the MPI blocking functions (ARMCI Nonblocking operations outperform for some messages shorter than this too). This improvement potential opens up the debate whether replacing MPI functions with ARMCI functions will improve the performance of communication or the applications. We address this question in the next chapter.

To have an idea of how much performance improvement we may gain by moving away from MPI functions to ARMCI functions in the applications, in Table 5.3 we present the communication latency difference of MPI and ARMCI. Latency differences for both blocking and non-blocking operations are evaluated on one- and two-port utilized configurations of the Myrinet NIC. The negative numbers in the table show that MPI outperforms ARMCI, while positive numbers show the better performance of ARMCI. In order to locate easily the message sizes that ARMCI outperforms MPI, we have shaded the corresponding boxes.

ARMCI blocking *Put* outperforms the MPI blocking and non-blocking *Send/Receive* for messages larger than 8KB, and 256KB, respectively (under one/two-port). Latency of ARMCI non-blocking *Put* operations is equal or smaller than MPI blocking/non-blocking *Send/Receive* for all the messages sizes between 1B-1MB (expect for MPI non-blocking with messages size of 4KB and 8KB under one-port). The maximum communication performance improvement of ARMCI non-blocking *Put* over MPI blocking is 53%, and 32% under two-port and one-port, respectively (with 16KB messages). The maximum improvement of ARMCI non-blocking *Put* over MPI non-blocking is 59%, and 51% under two-port and one-port, respectively (with 2KB messages).

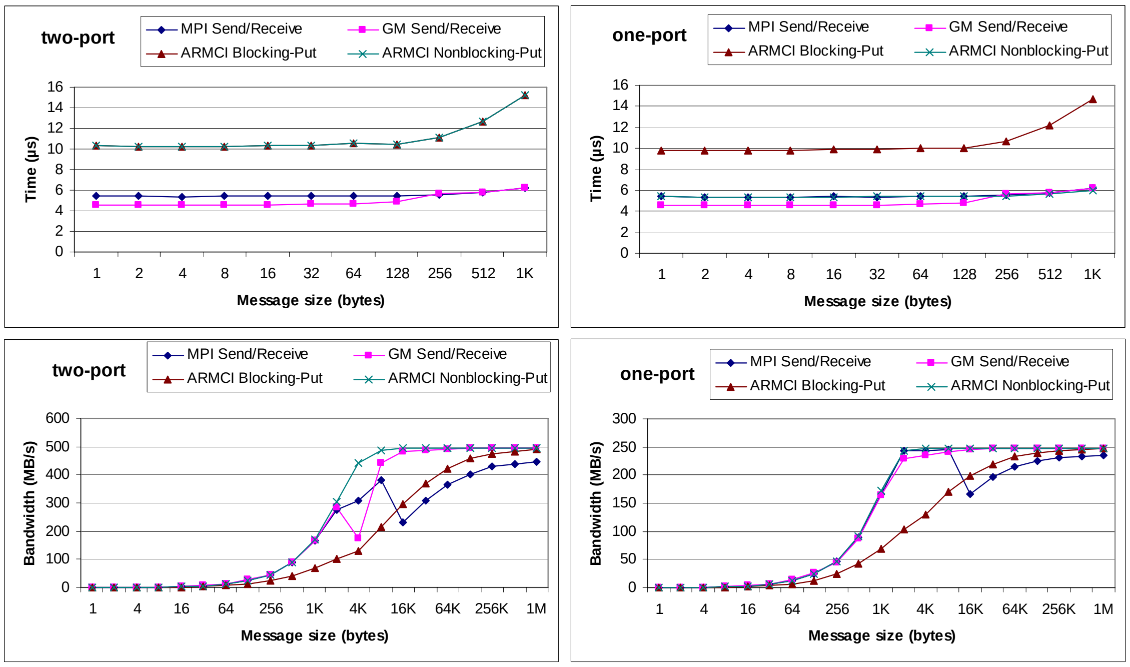


Figure 5.14 Latency and bandwidth comparison of MPI blocking *Send/Receive*, GM *Send/Receive*, and ARMCI blocking and non-blocking *Put* (one/two-port).

When using one port of the Myrinet NIC, not only ARMCI *Put* operation shows superior performance to MPI, but also performance of ARMCI *Get* operation shows the similar improvement potential. Blocking and non-blocking ARMCI operations (*Put/Get*) performs up to 216-241 µs faster than MPI blocking *Send/Receive*. MPI non-blocking *Send/Receive* operations have larger latency of up to 48-73 µs than ARMCI operations.

We believe that the better performance of MPI in short messages is due to its *eager* protocol. MPI uses *eager* method for sending short messages, and *rendezvous* mechanism for sending long messages. *Eager* mechanism improves the latency of messaging while rendezvous mechanism provides a better bandwidth. In short, using ARMCI *Put* (blocking/non-blocking) operations is promising to achieve a better performance in parallel applications. We investigate this, in more detail, in the next chapter.

Table 5.3 Performance advantage/disadvantage of ARMCI over MPI (in μs).

two-port	MPI Blocking		MPI Nonblocking		MPI Blocking		MPI Nonblocking	
	ARMCI Blocking				ARMCI Nonblocking			
Message	Put	Get	Put	Get	Put	Get	Put	Get
1	-4.9	-5.6	-4.3	-5.0	0.0	-0.2	0.6	0.4
2	-4.9	-5.6	-4.2	-5.0	0.0	-0.2	0.7	0.5
4	-4.9	-5.6	-4.3	-5.0	0.0	-0.2	0.6	0.4
8	-4.9	-5.6	-4.2	-5.0	0.0	-0.2	0.6	0.4
16	-4.9	-5.6	-4.3	-5.1	0.0	-0.2	0.6	0.4
32	-5.0	-5.8	-4.3	-5.2	0.0	-0.2	0.6	0.4
64	-5.1	-6.0	-4.5	-5.4	0.0	-0.2	0.6	0.4
128	-5.0	-6.3	-4.4	-5.7	0.0	-0.2	0.6	0.4
256	-5.5	-7.0	-5.0	-6.4	0.1	-0.2	0.6	0.4
512	-7.0	-8.3	-6.5	-7.8	0.1	-0.1	0.6	0.4
1K	-9.0	-11.0	-8.7	-10.7	0.2	-0.1	0.5	0.3
2K	-13.0	-16.1	-3.9	-7.0	0.7	-4.5	9.9	4.7
4K	-18.7	-23.3	-16.1	-20.7	4.1	2.1	6.6	4.6
8K	-16.5	-21.4	-16.0	-20.9	4.8	-15.0	5.3	-14.5
16K	15.3	-1167.6	-13.4	-1196.4	37.4	-46.0	8.7	-74.7
32K	18.2	-145.0	-15.0	-178.2	40.4	-139.4	7.2	-172.7
64K	24.8	-317.7	-15.8	-358.3	46.9	-312.3	6.2	-353.0
128K	37.9	-663.1	-12.8	-713.8	60.0	-658.9	9.3	-709.6
256K	58.9	-1359.7	-7.2	-1425.8	80.8	-1356.5	14.7	-1422.6
512K	115.0	-2737.8	8.2	-2844.6	137.0	-2737.4	30.3	-2844.1
1M	212.7	-5510.1	42.5	-5680.2	234.7	-5513.1	64.6	-5683.2
one-port								
1	-4.4	-5.3	-3.8	-4.7	0.0	-0.1	0.6	0.5
2	-4.4	-5.2	-3.8	-4.5	0.0	-0.1	0.7	0.6
4	-4.5	-5.2	-3.9	-4.6	0.0	-0.1	0.6	0.5
8	-4.4	-5.2	-3.8	-4.6	0.0	-0.1	0.6	0.5
16	-4.4	-5.1	-3.9	-4.6	0.0	-0.1	0.6	0.5
32	-4.5	-5.3	-3.9	-4.7	0.0	-0.1	0.6	0.5
64	-4.6	-5.5	-4.0	-4.9	0.0	-0.1	0.6	0.5
128	-4.5	-5.8	-3.9	-5.2	0.1	-0.1	0.6	0.5
256	-5.1	-6.5	-4.6	-6.0	0.1	-0.1	0.6	0.4
512	-6.5	-7.9	-5.9	-7.4	0.1	-0.1	0.6	0.4
1K	-8.5	-10.6	-8.2	-10.3	0.2	0.0	0.5	0.3
2K	-11.5	-14.6	-2.8	-5.8	0.0	0.1	8.8	8.9
4K	-14.7	-19.3	-16.8	-21.5	0.2	0.3	-1.9	-1.9
8K	-14.8	-19.6	-23.2	-28.0	0.1	0.2	-8.3	-8.2
16K	15.3	11.7	5.3	1.6	31.7	31.9	21.7	21.8
32K	18.2	14.5	-10.1	-13.8	34.8	35.1	6.5	6.8
64K	24.4	21.0	-10.7	-14.1	41.1	41.6	6.0	6.5
128K	37.3	34.3	-7.7	-10.6	53.9	54.9	8.9	9.9
256K	62.7	60.8	-1.9	-3.9	79.4	81.4	14.7	16.7
512K	115.5	115.5	14.6	14.6	132.0	136.1	31.1	35.2
1M	216.8	220.7	48.7	52.7	233.2	241.3	65.1	73.2

5.5 Summary

In this chapter, we have evaluated the basic performance of different message-passing libraries on top of the Myrinet Network. We measured the performance of GM basic function calls, such as program initialization, memory allocation, memory deallocation, and program termination. We assessed and compared the basic communication latency performance of GM *Send/Receive*, GM RDMA, MPI *Send/Receive*, and ARMCI RDMA operations for one- and two-port configurations of the Myrinet network interface card.

We realize that, in general, non-blocking operations perform better than blocking, and the two-port communication at the GM, MPI, and ARMCI levels (except for the RDMA read) outperforms the one-port communication for the bandwidth. We notice that for messages larger than 8KB, ARMCI blocking *Put* performs better than MPI (under one/two-port). By using ARMCI operations instead of MPI, we argue there is potential in improving the communication performance in the parallel applications. This may also affect the application performance if the communication/computation ratio is large enough. In the next chapter, we will look into replacing MPI calls with ARMCI RDMA calls in order to gain a better communication performance in real applications.

Chapter 6 Application Performance and Impact of RDMA

It is important to systematically assess the features and performance of the new interconnects for high performance clusters. We presented the performance of the two-port Myrinet networks at the GM, MPI, and ARMCI layers using a complete set of micro-benchmarks in the previous chapter. We also presented the communication characteristics of the NAS Multi-Zone benchmarks in detail, and communication characteristics of the SMG2000 application briefly in chapter four. In this chapter, we show the performance of these applications under the MPI and MPI-OpenMP programming paradigms, and two-port and one-port configuration of the Myrinet NIC.

Our experiments presented in the previous chapter show that the two-port communications at the GM, MPI, and ARMCI levels (except for the RDMA read) outperform the one-port communication for the bandwidth. In this chapter, we investigate if this translates in a considerable improvement for our applications.

In the previous chapter, we also showed that ARMCI one-sided operations, for certain message sizes, outperform the MPI two-sided operations. In the second part of this chapter, we look into communication performance enhancement of NPB-MZ, using one-sided communications instead of two-sided MPI communications. We use the communication characteristics of NPB-MZ applications presented in chapter four and the basic communication performance of ARMCI and MPI (latency or bandwidth) from chapter five, to calculate the expected messaging performance improvement of NPB-MZ. In order to measure the run-time messaging performance improvement of NPB-MZ, we replace the MPI two-sided communications with ARMCI one-sided operations.

6.1 Mixed-Mode Application Performance

In this section, we first show the speedup of the MPI version of NPB-MZ and SMG2000 applications in Figure 6.1. Interestingly, the speedup, from one to 16 processes, for the MPI version of the NPB-MZ benchmarks is linear. Applications reach almost perfect speedup. However, the speedup for SMG2000, MPI version, is not as good as the NPB-MZ. The speedup with 2, 4, 8, and 16 processes are 1.6, 3.5, 6.3, and 6.5, respectively.

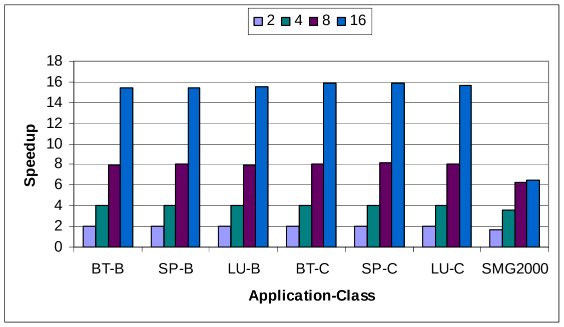


Figure 6.1 Speedup of MPI version of NPB-MZ and SMG2000 (two-port).

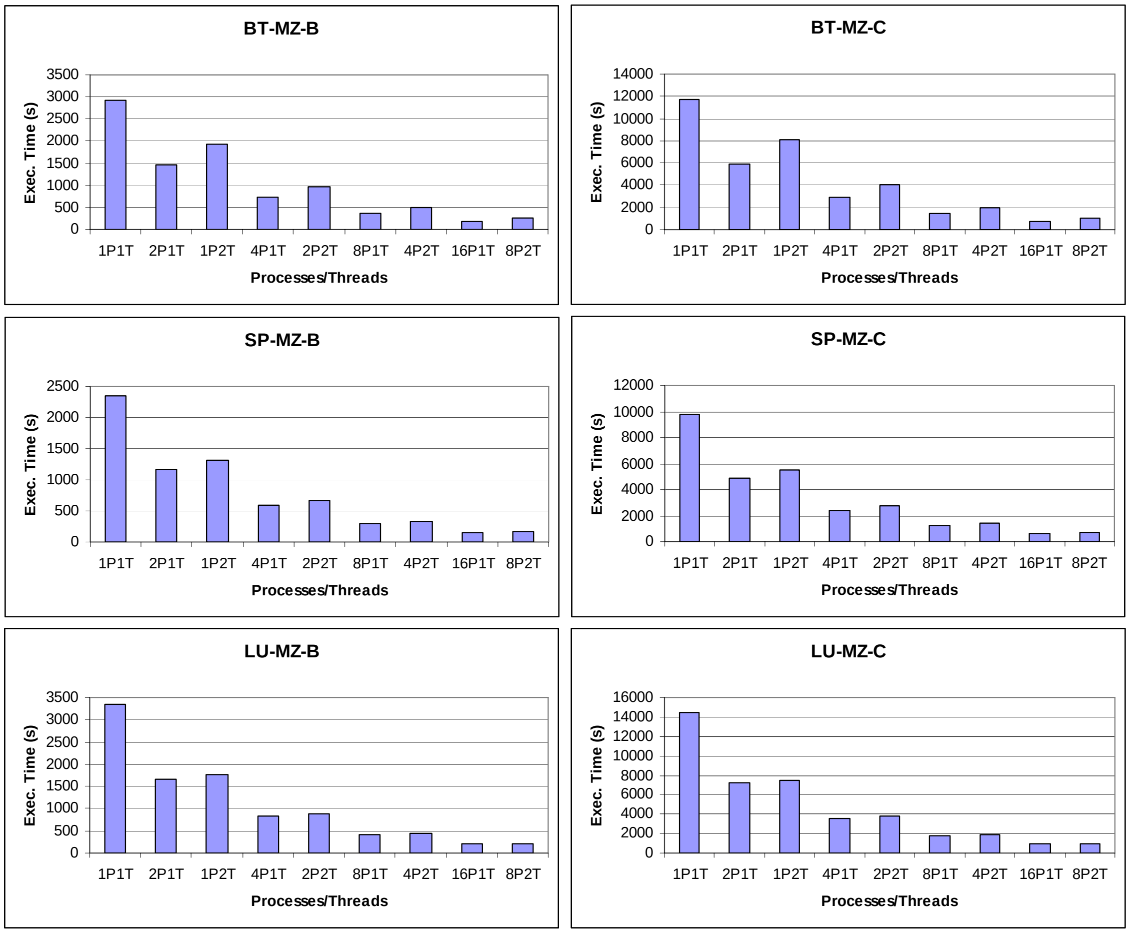


Figure 6.2 Execution time of NPB-MZ applications on Myrinet network (two-port).

It is possible to run different combinations for the number of threads and processes in mixed-mode applications across a cluster. Jin and his colleague [23] have examined the effectiveness of hybrid parallelization paradigms in NPB-MZ on three different parallel computers. In this section, we present the performance of NPB-MZ mixed-mode applications, as well as SMG2000 of the ASCI purple suite with different combinations of number of threads and processes. We would like to know which parallel programming paradigm, MPI or MPI-OpenMP, gives the best performance on our cluster. Figure 6.2 and Figure 6.3 present the execution time of BT-MZ, SP-MZ, LU-MZ, and SMG2000 applications on our platform. We chose the input size of 32x32x32 for the serial version of SMG2000. We scale it down proportionally with the number of processes to keep the total problem size constant for all runs.

The X-axis in Figure 6.2 and Figure 6.3 shows the number of processes and threads for each case. For instance, “4P2T” means that there are four processes evenly divided among four nodes of our cluster, where each process has two threads running on its respective node. By using this approach, we are able to compare the performance of the applications under pure MPI, and the mixed MPI-OpenMP. From the results, one can claim that the MPI version of the applications performs better than their mixed-mode versions. For instance, for BT-MZ with class C (BT-C), 2P1T runs faster than 1P2T; 4P1T runs faster than 2P2T; and so on. The same is true for the SMG2000. In the next section, we will discuss the application performance difference, when using one-port or two-port of the Myrinet NIC.

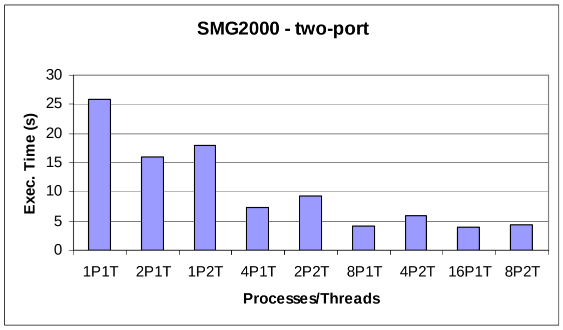


Figure 6.3 Execution time of SMG2000 on Myrinet Network (two-port).

6.2 Two-port Myrinet Card Application Performance

In the previous chapter, we studied the communication performance of the Myrinet network using one or two ports of the Myrinet NIC. It is important to discover if the bandwidth gain offered by the two-port NIC is actually beneficial to the application layer. In order to study the effect of port utilization of the two-port Myrinet cards, we have evaluated the performance of NPB-MZ and SMG2000 using one and two ports of the Myrinet NIC (for both MPI and MPI-OpenMP). The difference in performance is minimal, with at most 3% improvements for the two-port cases. It is noteworthy to mention that these applications are compute-bound. The communication time is always less than 5% of the total execution time. Therefore, the improvement in communication cannot translate to application performance.

As we showed in chapter four, the message sizes for SMG2000 are short (less than 1KB) so the two-port Myrinet network cannot offer any improvement over the one-port. However, this is not the case for the NPB-MZ applications. Having a closer look at the distribution of message sizes for the NPB-MZ applications in chapter four, it reveals that the BT-MZ uses a large number of different message sizes. It uses up to 21 different message sizes in class C (16 in class B). The shortest and the longest messages are 6KB and 55KB in class C, respectively. In class B, 4KB and 41KB are the shortest and the longest messages, respectively. The distribution of message sizes sent by the SP-MZ, and LU-MZ are bimodal (14KB, and 21KB for SP-B; 18KB, and 28KB for SP-C; 29KB, and 43KB for LU-B; and 79KB, and 119KB for LU-C). Message sizes and the number of messages sent for the NPB-MZ applications suggest these applications are bandwidth-bound. Therefore, the two-port Myrinet network should improve the performance over the one-port. A setback in performance improvement could be associated with the fact that the one-port NIC can offer a better computation/communication overlap than the two-port [55].

6.3 ARMCI One-Sided vs. MPI Two-Sided

MPI send and receive operations provide the programmer with a two-sided communication model. In the two-sided communication, both the data sender and the data receiver parties have to call the corresponding API functions. In MPI two-sided communication, the data sender process executes an *MPI_Send* operation and the data receiver process executes an *MPI_Receive* operation accordingly.

ARMCI provides one-sided communication operations. In the ARMCI one-sided operations, processes involved in communication have direct access to the memory of their peers. Only one of the communication peers needs to execute an operation. For example, the data sender process executes a *Write/Put* operation while the data receiver process does not need to execute any operation. The data will be written to the memory location according to the *Write/Put* operation. On the other hand, a process is able to receive data from other processes by executing *Read/Get* operation. Again, the process that provides the data does not need to perform any action. Once the *Read/Get* operation is issued, the data will be transferred from the memory of data provider process to the memory of the issuer process.

6.3.1 Expected per Message Communication Improvement

In the previous chapter, we showed that ARMCI one-sided operations outperform MPI two-sided operations for certain message sizes. In this section, we use this feature in enhancing the communication performance of NPB-MZ by utilizing ARMCI one-sided communications instead of MPI two-sided communications. We use the communication characteristics of NPB-MZ applications from chapter four and the basic communication performance of ARMCI and MPI (latency or bandwidth) presented in chapter five, to calculate the expected messaging performance improvement in NPB-MZ. NPB-MZ applications use non-blocking MPI *Send*/*Receive* operations.

In chapter five, we presented the communication latency of ARMCI and MPI operations for some certain message sizes (powers of two from one byte to one Megabyte). However, NPB-MZ applications exhibit different message sizes. Therefore, we measure the ARMCI and MPI communication latency of different message sizes used in the NPB-MZ applications.

Table 6.2, Table 6.3, Table 6.1, Table 6.4, and Table 6.5 show the expected difference in latency of non-blocking MPI *Send*/*Receive* operations and ARMCI blocking and non-blocking *Put* operations, when using one or two ports of the Myrinet NIC for different message sizes in NPB-MZ. The negative numbers in the tables show that MPI outperforms ARMCI, while positive numbers show otherwise. In order to observe easily the message sizes that ARMCI outperforms MPI, we have shaded the corresponding boxes. It is noteworthy to mention that these latency differences are calculated per message. The number of times that each message size is used in each application is also shown in the tables. Later in section 6.3.2, we will determine the total communication latency difference in the application.

Table 6.1 Estimated communication improvement of BT-MZ-C per message.

						Estimated performance difference (μs)			
Message size (KB)	Number of Messages/Application					ARMCI Blocking		ARMCI Nonblocking	
	C-2	C-4	C-8	C-16	C-32	2-port	1-port	2-port	1-port
6.1	3216	4422	4824	5628	6030	-15.7	-20.5	2.5	-6.7
8.1	5628	7236	9246	10452	12864	-13.7	-20.0	3.3	-5.7
10.2	3216	5628	8844	9246	12864	-16.3	-17.0	4.3	-2.8
11.2	1608	3618	4824	5628	6030	-18.4	-15.3	4.1	-0.1
12.2	2412	6432	10452	11256	12060	-13.8	-16.4	4.1	-2.2
13.2	1608	2814	6030	6030	6432	-14.5	-15.0	4.3	-0.7
14.2	804	2814	5226	5628	6432	-15.0	-14.6	4.6	-0.4
15.2	804	3216	5226	5226	6030	-16.6	-18.8	4.5	-2.2
16.3	1608	2814	3216	5628	6030	-5.5	24.5	14.2	38.5
17.3	3216	6432	9246	10452	12060	-7.4	-7.9	13.2	6.5
19.3	1608	5628	6834	11658	11658	-10.0	-10.4	14.5	6.4
21.3	2412	4824	9246	10452	12864	-14.8	-7.8	5.7	6.4
24.4	1608	4020	10050	11658	12060	-8.8	-7.9	12.6	6.4
26.4	2412	5628	7236	9648	12462	-7.8	-8.0	14.0	6.7
29.5	1608	4824	7638	11658	12060	-14.9	-7.8	5.5	6.6
33.5	1608	5226	6030	11256	12864	-7.7	-7.3	13.3	6.9
36.6	2412	4422	6834	10050	12864	-13.3	-7.4	4.7	7.0
41.6	804	2412	4422	5226	6432	-7.6	-7.5	13.4	6.9
45.7	804	2010	2814	6432	6432	-14.8	-7.3	5.5	6.9
50.8	804	1206	4422	5226	6030	-8.1	-8.1	14.2	6.9
55.9	804	1608	4422	4422	6432	-15.1	-10.3	6.9	6.7

Table 6.2 Estimated communication improvement of BT-MZ-B (2 processes) per message.

		Estimated performance difference (µs)			
Message size (KB)	Number of Messages/ Application	ARMCI Blocking		ARMCI NonBlocking	
		2-port	1-port	2-port	1-port
5.3	2412	-14.6	-18.8	1.7	-5.4
6.4	1608	-16.2	-18.9	2.6	-4.5
8.2	1608	-13.8	-14.4	3.2	-0.3
8.8	1608	-14.7	-15.7	3.9	-1.2
10.5	1608	-17.1	-15.6	4.3	-0.9
11.1	804	-18.5	-16.4	4.1	-1.2
12.9	1608	-14.3	-15.9	4.2	-1.5
14.1	1608	-14.8	-16.2	4.8	-1.8
17.0	804	-7.2	-7.9	13.1	6.5
22.3	804	-13.8	-8.1	6.3	6.7
26.4	1608	-7.8	-8.0	14.0	6.7
27.5	1608	-10.1	-10.3	14.7	6.5
32.2	804	-9.0	-7.5	12.6	6.8
41.0	804	-7.4	-7.5	13.1	6.9

Table 6.3 Estimated communication improvement of BT-MZ-B (4-32 processes) per message.

					Estimated performance difference (µs)			
Message size (KB)	Number of Messages/Application				ARMCI Blocking		ARMCI NonBlocking	
	B-4	B-8	B-16	B-32	2-port	1-port	2-port	1-port
5.3	2412	2010	3216	3216	-14.6	-18.8	1.7	-5.4
6.4	2814	2412	3216	3216	-16.2	-18.9	2.6	-4.5
8.2	1608	2814	3216	3216	-13.8	-14.4	3.2	-0.3
8.8	2814	2814	3216	3216	-14.7	-15.7	3.9	-1.2
10.5	2412	2814	2814	3216	-17.1	-15.6	4.3	-0.9
11.1	1608	3216	3216	3216	-18.5	-16.4	4.1	-1.2
12.9	2010	2412	3216	3216	-14.3	-15.9	4.2	-1.5
14.1	2412	3216	3216	3216	-14.8	-16.2	4.8	-1.8
16.4	2814	3216	3216	3216	-7.4	-7.8	12.7	6.5
17.0	1206	2412	2814	3216	-7.2	-7.9	13.1	6.5
21.1	2010	2010	2814	3216	-14.4	-7.9	5.6	6.6
22.3	2412	2814	3216	3216	-13.8	-8.1	6.3	6.7
26.4	804	2412	3216	3216	-7.8	-8.0	14.0	6.7
27.5	1608	3216	2814	3216	-10.1	-10.3	14.7	6.5
32.2	2010	2412	2412	3216	-9.0	-7.5	12.6	6.8
41.0	1206	2010	3216	3216	-7.4	-7.5	13.1	6.9

Table 6.4 Estimated communication improvement of SP-MZ per message.

	Message size (KB)	Number of Messages/ Application	Estimated performance difference (µs)			
			ARMCI Blocking		ARMCI Nonblocking	
			2-port	1-port	2-port	1-port
SP-B-2	14.1	12832	-14.8	-16.2	4.8	-1.8
SP-B-4	14.1	36892	-14.8	-16.2	4.8	-1.8
	21.1	6416	-14.4	-7.9	5.6	6.6
SP-B-8	14.1	51328	-14.8	-16.2	4.8	-1.8
SP-B-16	14.1	51328	-14.8	-16.2	4.8	-1.8
	21.1	12832	-14.4	-7.9	5.6	6.6
SP-B-32	14.1	51328	-14.8	-16.2	4.8	-1.8
	21.1	44912	-14.4	-7.9	5.6	6.6
SP-C-2	18.3	25664	-7.7	-8.0	14.0	6.6
SP-C-4	18.3	76992	-7.7	-8.0	14.0	6.6
	28.4	20050	-13.1	-7.8	4.9	6.5
SP-C-8	18.3	176440	-7.7	-8.0	14.0	6.6
	28.4	17644	-13.1	-7.8	4.9	6.5
SP-C-16	18.3	205312	-7.7	-8.0	14.0	6.6
SP-C-32	18.3	205312	-7.7	-8.0	14.0	6.6
	28.4	25664	-13.1	-7.8	4.9	6.5

Table 6.5 Estimated communication improvement of LU-MZ per message.

	Message size (KB)	Number of Messages/ Application	Estimated performance difference (µs)			
			ARMCI Blocking		ARMCI Nonblocking	
			2-port	1-port	2-port	1-port
LU-B-2	29.3	4016	-15.1	-7.8	5.4	6.5
LU-B-4	29.3	8032	-15.1	-7.8	5.4	6.5
LU-B-8	29.3	8032	-15.1	-7.8	5.4	6.5
	43.4	4016	-10.0	-9.8	14.6	7.0
LU-B-16	29.3	8032	-15.1	-7.8	5.4	6.5
	43.4	8032	-10.0	-9.8	14.6	7.0
LU-C-2	79.2	6024	-14.6	-9.9	6.7	6.9
LU-C-4	79.2	8032	-14.6	-9.9	6.7	6.9
LU-C-8	79.2	8032	-14.6	-9.9	6.7	6.9
	119.8	4016	-12.8	-8.2	9.0	8.7
LU-C-16	79.2	8032	-14.6	-9.9	6.7	6.9
	119.8	8032	-12.8	-8.2	9.0	8.7

One can observe that the ARMCI blocking *Put* does not have a better performance than the MPI *Send*/*Receive* operations for the NPB-MZ message sizes. ARMCI non-blocking *Put,* when using two ports of the Myrinet NIC, shows a smaller latency than the MPI non-blocking *Send*/*Receive* for all of the message sizes used in BT-MZ, SP-MZ, and

LU-MZ. In this case, the expected time difference for message sizes in NPB-MZ is up to 14μs per message. When using the one-port configuration of the Myrinet NIC, ARMCI non-blocking *Put* shows a smaller latency than the MPI non-blocking *Send*/*Receive* for most of the message sizes used in BT-MZ, and SP-MZ. This is true, for all the message sizes used in LU-MZ as well. The latency difference is up to 38μs for this case.

6.3.2 Expected Overall Communication Improvement

To determine the total communication time improvement that an application can benefit by using ARMCI instead of MPI operations, we multiply the number of messages by the estimated latency difference per message, and then divide them by the number of running processes. Table 6.6, Table 6.7, and Table 6.8 present the total communication time improvement for NPB-MZ applications when replacing their MPI non-blocking *Send*/*Receive* calls with the ARMCI blocking and non-blocking *Put* operations. When using two ports of the Myrinet NIC, using ARMCI operations is beneficial for different number of processes of NPB-MZ applications, while using one port of the NIC, ARMCI does not improve the communication performance of the SP-MZ-B (except for 32 processes). One can easily notice the positive effect of using ARMCI non-blocking *Put* operations on the communication performance of the applications.

The expected communication performance improvement for BT-MZ is between 12-173 milliseconds under two-port NICs, and between 4-79 milliseconds under one-port. For SP-MZ, it is between 16-319 milliseconds, and between 6-160 milliseconds, under two-port and one-port, respectively; and finally, for LU-MZ, it is between 8-20 milliseconds and between 7-21 milliseconds under two-port and one-port, respectively. It is interesting to observe that the performance gain is larger for larger classes with smaller number of processes. Looking back at number of messages per process for NPB-MZ applications in Figure 4.1, we see that number of sent messages per process decreases as the number of processes increase. In addition, larger class size has larger number of messages per process.

Table 6.6 Estimated improvement of BT-MZ communication time (ms).

		blocking		non-blocking	
Class	Processes	2-port	1-port	2-port	1-port
B	2	-128	-134	63	6
B	4	-105	-104	55	13
B	8	-67	-65	40	12
B	16	-39	-38	23	6
B	32	-20	-20	12	4
C	2	-257	-235	151	62
C	4	-269	-239	173	79
C	8	-212	-188	136	62
C	16	-131	-110	90	48
C	32	-76	-64	51	27

Table 6.7 Estimated improvement of SP-MZ communication time (ms).

		blocking		non-blocking	
Class	Processes	2-port	1-port	2-port	1-port
B	2	-95	-104	31	-12
B	4	-160	-162	53	-6
B	8	-95	-104	31	-12
B	16	-59	-58	20	-1
B	32	-44	-37	16	6
C	2	-99	-103	179	85
C	4	-214	-194	293	160
C	8	-199	-194	319	160
C	16	-99	-103	179	85
C	32	-60	-58	94	48

Table 6.8 Estimated improvement of LU-MZ communication time (ms).

		blocking		non-blocking	
Class	Processes	2-port	1-port	2-port	1-port
B	2	-30	-16	11	13
B	4	-30	-16	11	13
B	8	-20	-13	13	10
B	16	-13	-9	10	7
C	2	-44	-30	20	21
C	4	-29	-20	13	14
C	8	-21	-14	11	11
C	16	-14	-9	8	8

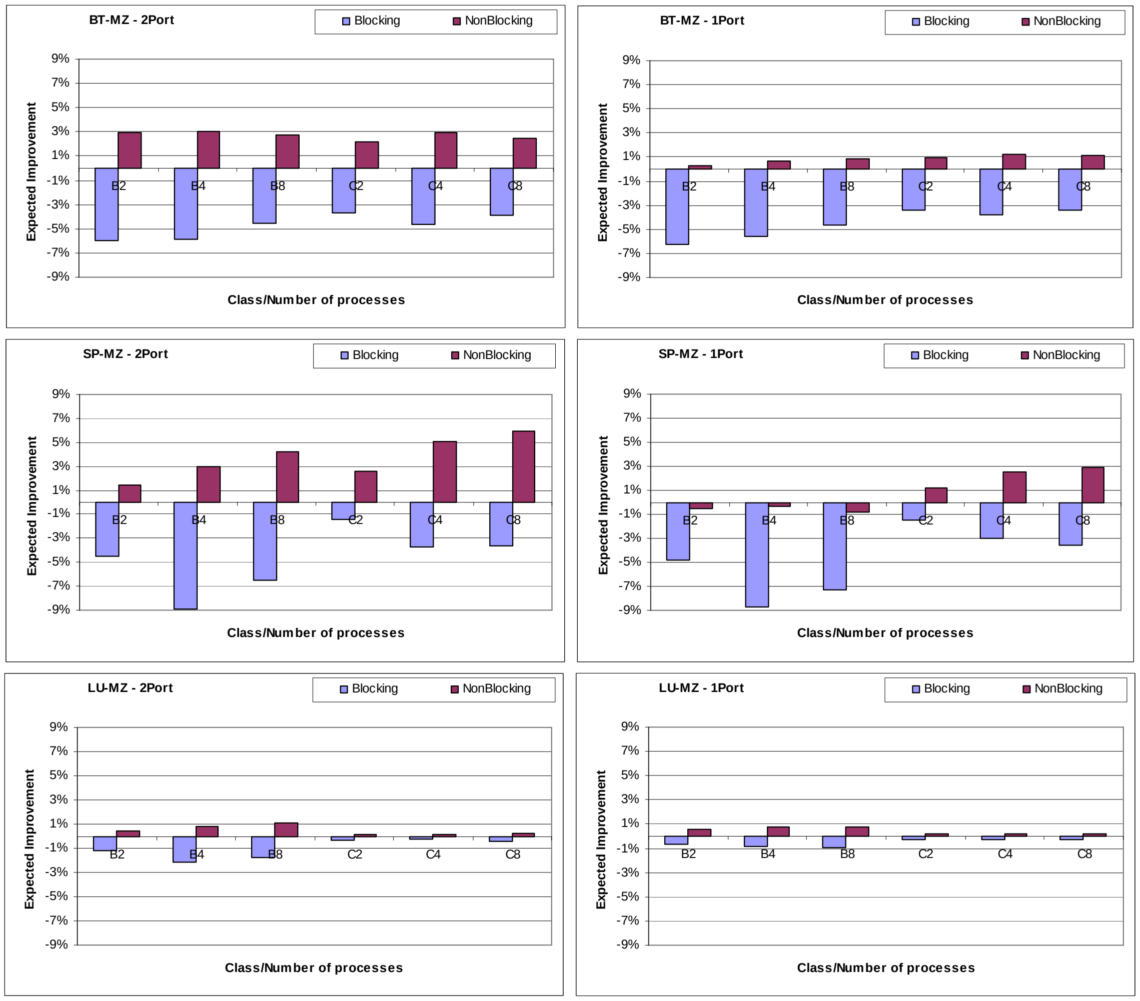


Figure 6.4 Expected Messaging Improvement of Blocking and Non-Blocking ARMCI over MPI for BT-MZ, SP-MZ, and LU-MZ.

The expected communication time improvements (percentage) using ARMCI are shown in Figure 6.4. Our estimations show 2.2-3.1% improvement for BT-MZ communication time when using two ports of the Myrinet NIC, and 0.3-1.2% when using only one port. SP-MZ shows an improvement potential of 1.4-5.9% when using two ports, and 1.2-2.9% when using one port. LU-MZ does not seem to have a good improvement potential. Our estimations show 0.1-1.1% expected improvement for LU-MZ when using two ports of the NIC, and 0.2-0.7% when utilizing only one port of the Myrinet NIC.

6.3.3 NPB-MZ Communication Patterns

In this section, we describe the communication patterns of NPB-MZ applications. NPB-MZ applications have two phases of execution: computation and communication. For each process, these two phases are separate from each other. By that, we mean one process does not enter the computation phase until it finishes the communication phase, and vice versa. It is noteworthy to mention that it is possible the computation phase of one process overlaps with the communication phase of another process.

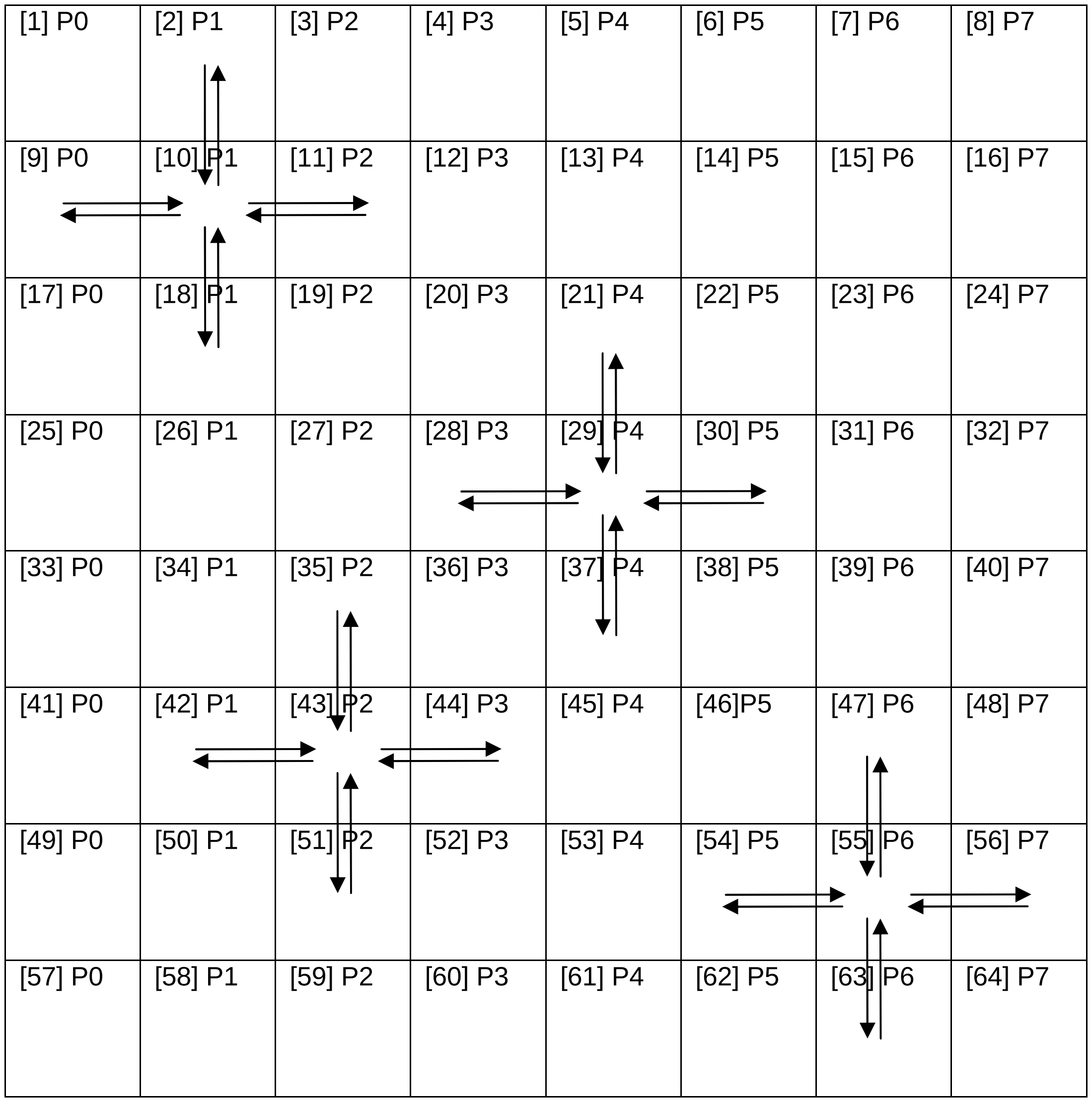


Figure 6.5 Example of message exchange among zones in NPB-MZ.

The workload in NPB-MZ is split into a mesh of zones, as illustrated in Figure 6.5. At the beginning of the program, all the zones are evenly distributed among the processes. For example if the running class size of the application has 64 zones, and the application is running with 8 processes, 8 zones are assigned to each process. After finishing computation of each zone, the owner process of the zone updates the neighbouring zones with the new values. Once one process finishes the computation of all its zones, it enters the communication phase and updates all the neighbouring zones (by sending messages to their owner process). The communication phase completes when message exchange with all of the neighbouring zones is finished. Once the communication phase is completed, each process continues with another computation phase.

```
Do loop (for all the zones belonging to the process)
{
       Computing zone IDs
       Receiving message from the zone west  (MPI_IRecv)
       Sending message to the zone west      (MPI_ISend)
       Receiving message from the zone east  (MPI_IRecv)
       Sending message to the zone east      (MPI_ISend)
       Receiving message from the zone south (MPI_IRecv)
       Sending message to the zone south     (MPI_ISend)
       Receiving message from the zone north (MPI_IRecv)
       Sending message to the zone north     (MPI_ISend)
}
Wait for completion of all the non-blocking operations (MPI_Waitall)
```

Figure 6.6 Pseudo code for NPB-MZ communication.

The message exchange algorithm with the neighbouring zones for a few zones is illustrated in Figure 6.5. The processes are shown with P0-P8 and zone numbers are written inside the bracket. Each zone performs the depicted message exchange with its neighbours. To have a better understanding of the message exchange algorithm in NPB-MZ, pseudo code of the NPB-MZ communication is presented in Figure 6.6. This message exchange algorithm leads to a loose synchronization among all the zones (and their owner processes). First, each process computes its zone IDs. The process sends a message to the four neighbours of its zones (west, east, south, and north). Each zone receives a message from each of its four neighbours as well. At the end, the program assures that the non-blocking message exchange is completed.

6.3.4 Converting MPI Communications to ARMCI

BT-MZ, SP-MZ, and LU-MZ use MPI non-blocking communication for exchanging data among processes. To verify our expectations as claimed in the previous section, and to see the impact of replacing two-sided communication operations with one-sided operations, we modified the NPB-MZ codes and replaced their MPI non-blocking *Send*/*Receive* communications with ARMCI blocking and non-blocking *Put* operations. As we used different APIs in our code, some codes had to be added to make the code change possible. However, the code overhead in our applications was minimal.

Replacing MPI calls with ARMCI functions is not an easy task. NPB-MZ applications are written in FORTRAN language, while ARMCI supports C programming language. ARMCI provides data transfer operations including *put*, *get* and *accumulate*. Utility operations such as memory allocation and deallocation and error handling are supported in ARMCI. However, ARMCI only supports communication that targets remote memory allocated via the provided memory allocator routine, *ARMCI_Malloc()*. The address of the allocated memory region by ARMCI is stored in pointers. However, C pointers are not compatible with FORTRAN pointers.

Jin and Jost [22] have evaluated the feasibility of RMA programming on shared memory parallel computers. They have discuss different RMA based implementations of selected CFD application benchmark kernels, such as BT, SP, and LU of NPB, and have compared them to corresponding message passing based codes. They have used MPI for the message-passing implementation, and shared memory parallelization library (SMPlib) and the MPI-2 extension to the MPI Standard for the RMA based implementations. They have found the RMA programming more scalable than MPI programming.

A pseudo code of the converted code using ARMCI blocking and non-blocking operations is presented in Figure 6.8 and Figure 6.7, respectively. Instead of sending messages by *MPI_ISend*, we use ARMCI blocking/non-blocking *Put* operations. As the data receiver is not notified in one-sided communication, a notification message is needed for the data receiver process. We use the notification functions provided by ARMCI

(*ARMCI_Notify/Waitnotify*) to notify the data receiver process. When using non-blocking ARMCI operations, we use *ARMCI_Waitall* function to assure the completion of the operations.

```
Do loop (for all the zones belonging to the process)
{
      Computing zone IDs
      Writing message to the zone west    (ARMCI_NBPut)
      Writing message to the zone east    (ARMCI_NBPut)
      Writing message to the zone south   (ARMCI_NBPut)
      Writing message to the zone north   (ARMCI_NBPut)
}
/* Making sure that ARMCI Non-blocking operations are completed */
ARMCI_Waitall
Do loop (for all the zones belonging to the process)
{
      Computing zone IDs
      /* Notifying the peer that the ARMCI_NBPut is completed */
      Sending notification to the zone west  (ARMCI_notify)
      Sending notification to the zone east  (ARMCI_notify)
      Sending notification to the zone south (ARMCI_notify)
      Sending notification to the zone north (ARMCI_notify)
}
Do loop (for all the zones belonging to the process)
{
      Computing zone ids
      /* Making sure that the ARMCI_NBPut of the peer is completed */
      Waiting for notification of the zone west  (ARMCI_WaitNotify)
      Waiting for notification of the zone east  (ARMCI_WaitNotify)
      Waiting for notification of the zone south (ARMCI_WaitNotify)
      Waiting for notification of the zone north (ARMCI_WaitNotify)
}
```

Figure 6.7 Pseudo code for ARMCI nonblocking version of NPB-MZ communication.

```
Do loop (for all the zones belonging to the process)
{
       Computing zone IDs
       Writing message to the zone west       (ARMCI_Put)
       /* Notifying the peer that the ARMCI_Put is completed */
       Sending notification to the zone west  (ARMCI_notify)
       Writing message to the zone east       (ARMCI_Put)
       Sending notification to the zone east  (ARMCI_notify)
       Writing message to the zone south      (ARMCI_Put)
       Sending notification to the zone south (ARMCI_notify)
       Writing message to the zone north      (ARMCI_Put)
       Sending notification to the zone north (ARMCI_notify)
}
Do loop (for all the zones belonging to the process)
{
       Computing zone IDs
       /* Making sure that the ARMCI_Put of the peer is completed */
       Waiting for notification of the zone west  (ARMCI_WaitNotify)
       Waiting for notification of the zone east  (ARMCI_WaitNotify)
       Waiting for notification of the zone south (ARMCI_WaitNotify)
       Waiting for notification of the zone north (ARMCI_WaitNotify)
}
```

Figure 6.8 Pseudo code for ARMCI blocking version of NPB-MZ communication.

6.3.5 Observed Communication Improvement

As stated earlier, ARMCI uses client-server architecture in clusters of workstations using GM. Each node of the cluster has a server thread that handles remote memory operations for each of the user processes running on the node. As our cluster consists of eight dual nodes, we assessed the communication performance of ARMCI version of NPB-MZ from two to eight processes (each on one node), in order to keep one processor dedicated to the ARMCI server thread.

We have instrumented the NPB-MZ codes by adding timers to measure the time each process spends in communication. Processes do not have equal communication time. We measure the communication time of each process by running the instrumented code. Figure 6.9 shows the communication times of each process in SP-MZ class C. We instrumented three versions of the code: original MPI version, ARMCI blocking version, and ARMCI non-blocking version. In the original MPI version, we only inserted appropriate timers to measure the communication time. In the ARMCI blocking version, in addition to the inserted timers, we replaced the MPI *Send*/*Receive* calls with ARMCI blocking *Put* operations. The ARMCI non-blocking version is similar to the blocking version, except that we used ARMCI non-blocking *Put* instead of blocking operations. It is clear that for most of the processes, the code with ARMCI non-blocking calls takes less amount of time than MPI and ARMCI blocking codes. Figure 6.9.b shows the average communication time of processes. We ran the instrumented codes a number of times and calculated the average communication time.

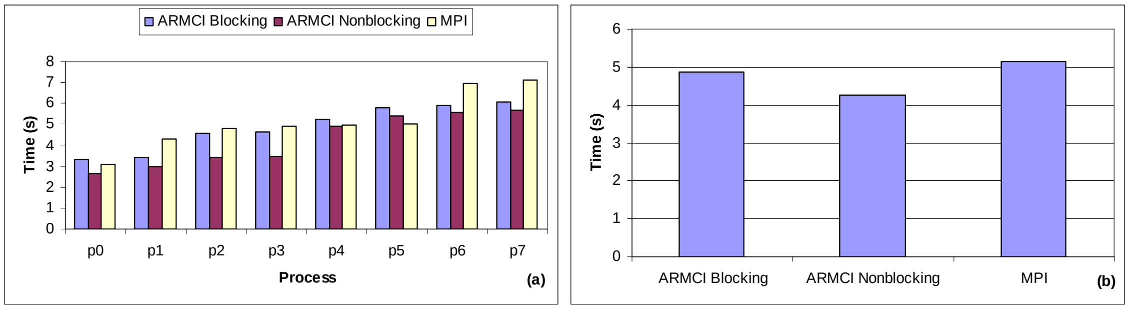


Figure 6.9 (a) Communication time per process, and (b) average communication time per process for SP-MZ-C (two-port).

Using the above method, we measured the average communication time per process of the instrumented codes. Figure 6.10 shows the communication time improvement of NPB-MZ applications using ARMCI functions in percentage format. Although blocking ARMCI results are not promising, we achieved average improvement for some cases. When using both ports of the Myrinet NIC, BT-MZ class C shows communication performance improvement of 7-40%, using ARMCI blocking *Put*, whereas non-blocking *Put* shows performance improvement of 12-35%. BT-MZ does not show significant improvement when using blocking operations for class B; however, the communication improves 2-8% for non-blocking operations. When utilizing only one port of the Myrinet NIC, BT-MZ shows communication improvement of 6-28% and 7-27% by replacing non-blocking and blocking operations with MPI, respectively.

SP-MZ was the most promising application among NPB-MZ applications for communication improvement. SP-MZ communication time decreases 4-35% using blocking operations, and 13-21% using non-blocking operations on two-port configuration of the Myrinet NIC. We achieved 6-27% better communication performance by using ARMCI blocking *Put*, and 5-21% improvement using ARMCI non-blocking *Put*, under one port of the Myrinet NIC.

Our estimations in the previous section do not show a good improvement potential for LU-MZ. However, using ARMCI blocking operations communication improves between 25-40% for some cases under two-port, while using non-blocking operations decreases the communication time by 3-43% for class C. When using one port of the Myrinet card, for some cases, LU-MZ communication time improves 1-17% and 2-16% by using blocking and non-blocking operations, respectively.

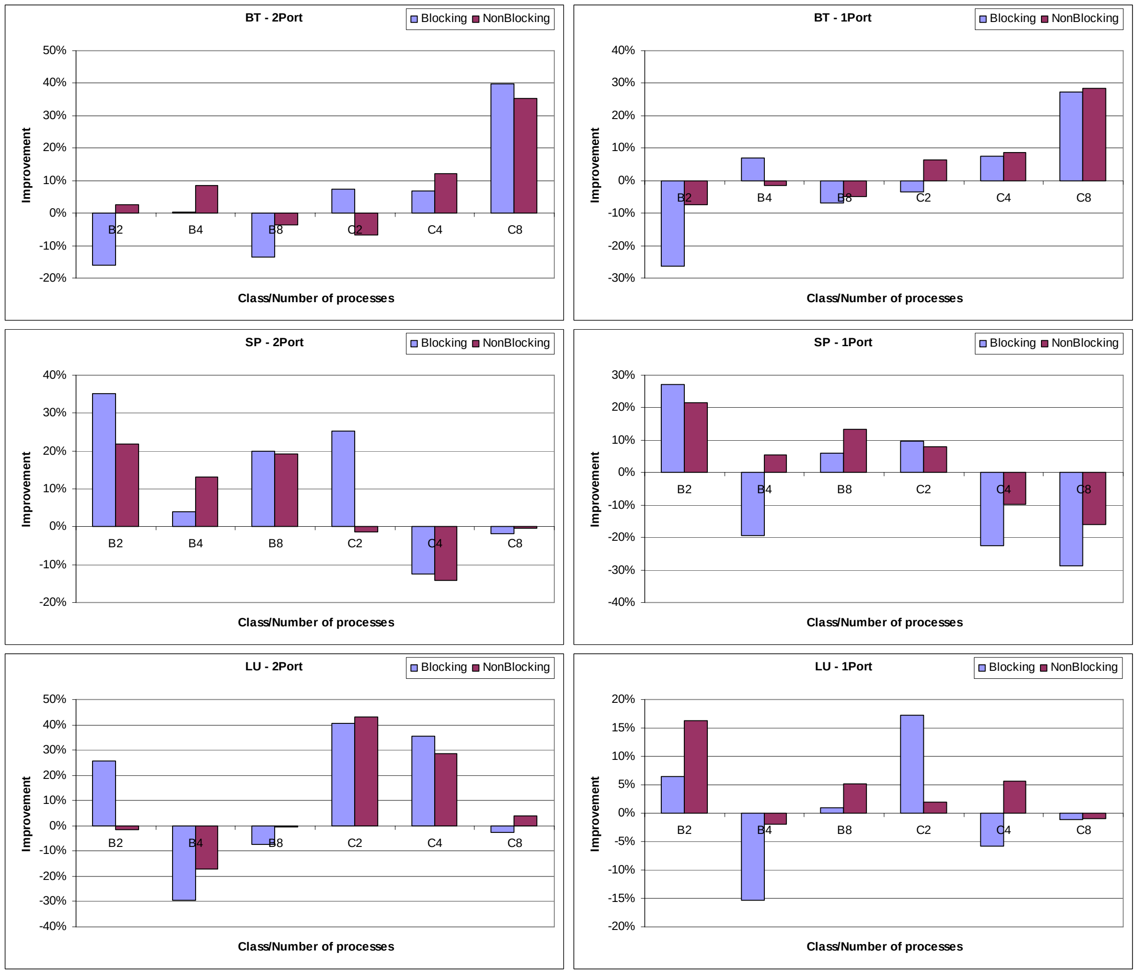


Figure 6.10 Messaging Improvement of blocking and non-blocking ARMCI over MPI for BT-MZ, SP-MZ, and LU-MZ.

The broad range of performance results hints there might be many other factors affecting our communication timing. We can name operating system load, synchronization of the nodes, and network traffic contention as some of the possible reasons. The performance improvement varies across different classes and different number of processes. The more communication traffic, the more likely improvement in performance using ARMCI. We investigated the source of broad range of performance, and we found out it is due to the communication pattern of NPB-MZ applications. The communication time of each process consists of different timings, such as sending messages, sending notifications, and waiting for the notification from other zones. The ARMCI superior performance enhances sending the messages, while the wait for receiving notifications from other zones (processes) is not directly affected by ARMCI. To cancel the effect of wait time in the measured communication time of processes, we choose the process with the shortest communication time in each run. We run the applications several times. Then, we average the communication time of processes with the minimum wait. In fact, we measure the communication time with the least wait time in each run in order to observe the effect of ARMCI on communication time. The improvement results calculated using this method is presented in Figure 6.11. Using this method, our results are more consistent and we do not get a broad range of results for different runs. The minimum and maximum of the measurement are shown by bars in Figure 6.11.

Using ARMCI non-blocking improves the BT-MZ performance for all of the classes and number of processes (except for C8) under one and two-port. Its communication performance improvement is between 23-43% under two-port and between 4-16% under one-port. ARMCI blocking also improves the BT-MZ communication under two-port up to 18%.

ARMCI non-blocking operations improve the performance of SP-MZ class B under two-port by up to 57%. SP-MZ class C with 8 processes also presents 11% of improvement. SP-MZ-B with eight processes is the only case that shows improvement under one-port (34%). ARMCI blocking operations did not improve the SP-MZ communication time.

LU-MZ is not improved by using ARMCI under two-port (except for B8). LU-MZ class C under one-port can benefit in communication time up to 44% and 29% by using ARMCI blocking and non-blocking operations, respectively.

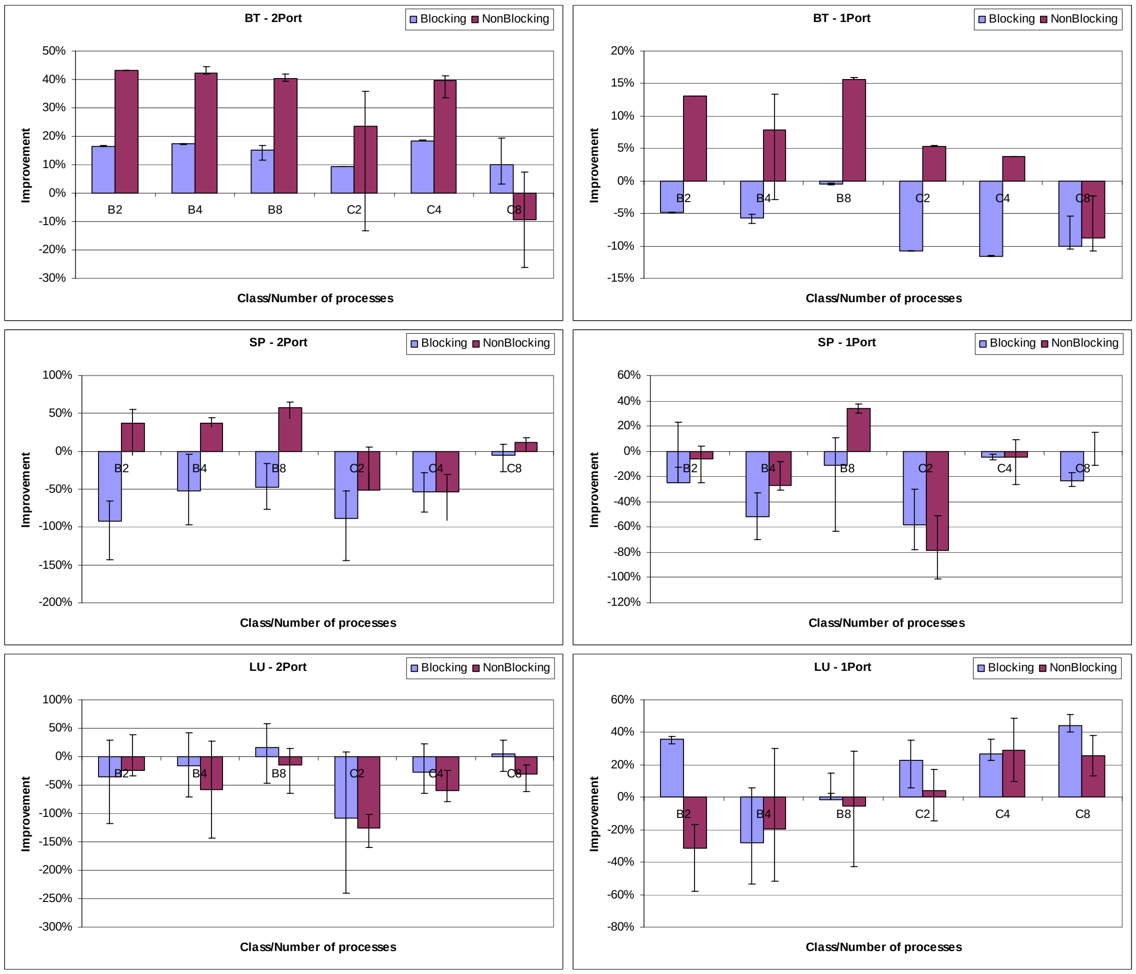


Figure 6.11 Messaging Improvement of Blocking and Non-Blocking ARMCI over MPI for BT-MZ, SP-MZ, and LU-MZ (Minimum process time).

Our expected results did not show any potential for improvement by using blocking operations; however, our empirical results show improvement for many classes and number of processes of NPB-MZ. We believe this is due to the synchronized communication pattern of NPB-MZ applications. As explained in the previous section, each process of NPB-MZ reaches a communication phase after a computation phase. The communication phase is blocking and the process does not exit this phase until the communications finish. This blocking structure of the communication phase of the code makes all the processes synchronize with each other to some extents every time they reach this part of the program. Using non-blocking operations in the part of the code that itself is blocking, imposes extra synchronization in that section. We think that this extra synchronization in the code may be the reason for the reduced performance when using non-blocking operations. In fact, non-blocking operations normally show better performance in applications where computation and communication can overlap. In a blocking communication part of the code without any computation, there is not much gain to use non-blocking operations.

6.4 Summary

In this chapter, we evaluated the performance of NPB-MZ applications under the MPI and MPI-OpenMP programming paradigms. We showed that for different combinations of number of processes and threads, pure MPI paradigm outperforms the Mixed MPI-OpenMP paradigm. This is also true for SMG2000 benchmark from ASCI purple suite. NPB-MZ scalability is very good on our cluster and it almost reaches the linear scalability. However, SMG2000 does not show a perfect scalability and it achieves speedup of 6.5 running on sixteen processors.

We also compared the performance of NPB-MZ and SMG2000 applications on two-port and one-port configuration of the Myrinet NIC. Our experiments presented in the previous chapter show that the two-port communication at the GM, MPI, and ARMCI levels (except for the RDMA read) outperforms the one-port communication for the bandwidth. We investigated if the performance can be optimally used at the application layer. However, this did not translate in a considerable improvement at least for our applications. The difference in performance was minimal, with at most 3% improvements for the two-port cases. All three of these applications are very compute-bound application, where they spend most of the time computing rather than communicating. The time spent in communication is very small compared to the time they spend in computation.

In this chapter, we looked into whether the performance gain of ARMCI one-sided operations over MPI two-sided operations can translate in performance for the applications. We tried to enhance the communication performance of NPB-MZ, using one-sided communications instead of two-sided communications. We estimated the performance improvement utilizing the communication characteristics of NPB-MZ applications, and the measured communication latency difference of ARMCI and MPI for NPB-MZ message sizes. We discovered that ARMCI non-blocking *Put* operations could enhance the communication performance of the NPB-MZ applications.

To verify our speculations, we replaced the MPI two-sided communications in the NPB-MZ applications with ARMCI one-sided operations. The empirical performance of the modified codes shows performance improvement of up to 43%. Using either ARMCI blocking and non-blocking *Put* operations improved the communication performance in some cases. In some other cases, NPB-MZ did not show any performance improvement.

Chapter 7 Conclusion and Future Work

As network computing becomes commonplace, the interconnection networks and the communication system software become critical in achieving high performance. Clusters of SMPs have become the ideal platform for high performance computing, as well as supporting the emerging commercial and networking applications. The choice of parallel programming paradigm, workload characteristics of the applications, and the performance of communication subsystem mainly affect the performance of the applications running on clusters. In this thesis, we have discussed these issues in detail.

OpenMP has emerged as the standard for parallel programming on shared-memory systems. Message-passing, particularly the Message Passing Interface, is the de facto standard for parallel programming in network-based computing systems. However, it is still open to debate whether pure message-passing or mixed MPI-OpenMP is the programming of choice for higher performance on SMP clusters. To address this question, we evaluated the performance of NPB-MZ applications under the MPI and MPI-OpenMP programming paradigms. We showed that for different combinations of processes and threads, pure MPI paradigm outperforms the Mixed MPI-OpenMP paradigm. We showed MPI performs better than mixed MPI-OpenMP in NPB-MZ applications, and SMG2000 benchmark from ASCI purple. NPB-MZ scalability is very good on our cluster and it almost reaches the linear scalability. However, SMG2000 does not show a perfect scalability and it achieves speed up of 6.5 running on sixteen processors.

To help having a better understanding of the applications' performance on clusters, it is important to understand their communication behaviour. We examined the MPI characteristics of small to large-scale scientific applications, including the NPB-MZ benchmark suite, SPEChpc2002 benchmark suite, and SMG2000 application, in terms of their point-to-point and collective communications. We quantified metrics such as number of sends, average message size per message, total message volume per process, message size cumulative distribution function (CDF), number of unique message destinations per process, and destination distribution of messages of the root process (process zero).

For collective communications, we presented the type, frequency, and the payload. We also evaluated the impact of the problem size and the system size on the communication behaviour of the applications. We found that the applications studied have diverse communication characteristics. Those include very small to very large messages, frequent to infrequent messages, various distinct message sizes, set of favourite destinations, and regular versus irregular communication patterns. Some applications are sensitive to the bandwidth of the interconnect, while others are latency-bound as well. Our evaluation also revealed that most applications are sensitive to the changes in the system size and the problem size. We discovered all applications use only a few collective operations. However, SPEC applications use them frequently with very large payloads.

This thesis presents the locality characteristics of NPB-MZ and SPEChpc2002 applications. We used FIFO, LFU, and LRU locality heuristics to evaluate the locality of message size and message destinations in our applications. We found out that LRU and FIFO have a very similar performance. LFU for some applications outperforms LRU and FIFO and sometimes shows a lower performance. We realized that some applications show good locality of message size or message destinations. We compared the communication characteristic of NPB-MZ applications in MPI-OpenMP with pure MPI. We found out that different process/thread combinations change the communication characteristics of NPB-MZ. We also realized that MPI communication characteristics of NPB-MZ are independent from the number of threads.

Overall, the information provided in this thesis will help system designers, application developers, and library/middleware designers to understand better the current and future communication workloads of parallel applications. This study verifies that message-passing applications communicate intensively. Therefore, they will benefit from improvements in the interconnect hardware and their features as well as the communication system software and libraries. Collective communications such as broadcast, barrier, and reduce are expensive operations. Thus, it is essential to optimize their implementation in hardware and/or software in the future computer systems.

To examine how network really affects the communication performance of the applications, we evaluated the basic performance of different message-passing libraries on Myrinet Network. In this thesis, we presented an in-depth evaluation of the new Myrinet two-port networks at the user-level (GM), MPI-level, and at the Aggregate Remote Memory Copy Interface (ARMCI) level. We measured the performance of GM basic function calls, such as program initialization, memory allocation, memory deallocation, and program termination. We assessed and compared the basic latency performance of GM Send/Receive, GM RDMA, MPI Send/Receive, and ARMCI RDMA operations for one- and two-port configuration of the Myrinet network card interface. We realized that, in general, non-blocking operations perform better than blocking, and the two-port communication at the GM, MPI, and ARMCI levels (except for the RDMA read) outperforms the one-port communication for the bandwidth. We noticed that for certain messages sizes, ARMCI performs better than MPI.

We compared the performance of NPB-MZ and SMG2000 applications on two-port and one-port configuration of the Myrinet NIC. We investigated if the superior bandwidth performance of the two-port NIC can be optimally used at the application layer. However, this did not translate in a considerable improvement at least for our applications. The difference in performance was minimal, with at most 3% improvements for the two-port cases.

Most of the parallel applications written in MPI, including the NPB-MZ applications, use a two-sided communication model based on send and receive operations. In this model, communication involves both the sender and the receiver side. Synchronization is achieved implicitly through communication operations. High-performance interconnects such as Myrinet provide a one-sided communication model referred to as Remote Direct Memory Access (RDMA). One-sided operations allow data transfer directly between user-level buffers on remote nodes without the active participation of the receiver. This does not incur software overhead at the receiving end.

In this thesis, we showed the potential improvement in communication time of parallel applications by using ARMCI one-sided operations instead of MPI two-sided calls. We looked into whether the performance gain of ARMCI one-sided operations over

MPI two-sided operations can be translated for the applications. We took on the challenge to convert our two-sided applications to one-sided.

The empirical performance of the modified communication codes shows performance improvement of up to 43%. Using either ARMCI blocking and non-blocking *Put* operations improved the communication performance in some cases. In some other cases, NPB-MZ did not show any performance improvement.

Finally, we reiterate the programmers, users, and system designers should continually consider the impact of communications on the overall performance of their applications. The choice of the communication hardware and the supporting software, parallel programming paradigms, messaging libraries, and communication algorithms are some of the noteworthy issues in achieving high performance in clusters.

7.1 Future Work

We would like to extend our study on the communication characteristic of other scientific, engineering, and commercial message-passing application and benchmarks. We would like to include communication/computation timing comparisons in our study. As some of the applications are mixed-mode applications, they can run under MPI, OpenMP, and MPI-OpenMP. Thus, it is interesting to characterize OpenMP directives of these applications as well.

We intend to evaluate the performance of MPI-2 (MPI-2/MPICH-2) one-sided operations on our cluster, and examine its impact on the applications. Our plan is to compare the performance of the MX, the new alternative messaging library for Myrinet (not yet available), with GM. We intend to extend our study on other interconnection networks, such as Quadrics and InfiniBand to evaluate the impact of RDMA.

Algorithms for one-sided communication and two-sided communication models are very different. It is interesting to revise the algorithms, and implement different applications with the appropriate algorithm for one-sided communications, rather than just replacing the two-sided communications.

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,* August 2003.

[2] A. Afsahi and N.J. Dimopoulos, "Efficient Communication Using Message Prediction for Clusters of Multiprocessors", *Concurrency and Computation: Practice and Experience*, 14(10) 2002, pp. 859-883.

[3] ARMCI. Available : http://www.emsl.pnl.gov/docs/parsoft/armci/

[4] ASCI Purple, SMG2000 Readme File, (http://www.llnl.gov/asci/purple).

[5] F. Bassetti, D. Brown, K. Davis, W. Henshaw, and D. Quinlan, "Overture: an object-oriented framework for high performance scientific computing", *Proceedings of the 1998 ACM/IEEE conference on Supercomputing,* pp. 1-9, 1998.

[6] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal, "User-Level Network Interface Protocols", *IEEE Computer*, Nov. 1998, pp. 53-60.

[7] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and Su Wen-King, "Myrinet A Gigabit-per-Second Local-Area Network" *IEEE Micro*, February 1995, pp. 29-36.

[8] R. Brightwell, and K. Underwood, "Evaluation of an Eager Protocol Optimization for MPI", Workshop Paper, *EuroPVM/MPI 2003*, September 2003.

[9] D. Buntinas, A. Saify, D.K. Panda, and J. Nieplocha, "Optimizing Synchronization Operations for Remote Memory Communication Systems", *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'03*, 2003.

[10] F. Cappello and O. Richard, "Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model", *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on,* 12-16 Oct. 1999 pp. 108 – 116.

[11] B. Carpenter, G. Zhang, and Y. Wen, "NPAC PCRCruntime kernel definition", Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997.

[12] S. Chodnekar, V. Srinivasan, A. Vaidya, A. Sivasubramanian, and C. Das, "Towards a Communication Characterization Methodology for Parallel Applications",

Proceedings of 3rd International Conference on High Performance Computer Architecture, 1997, pp. 310- 321.

[13] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "Architectural Requirements of Parallel Scientific Applications with Explicit Communication", *Proceedings of 20th International Symposium on Computer Architecture*, 1993.

[14] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming", *Computational Science and Engineering, IEEE* Volume 5, Issue 1, Jan.-March 1998 pp. 46 – 55.

[15] N. Drosinos, and N. Koziris, "Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters", *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International,* 26-30 April 2004 pp. 15.

[16] Exemplar Programming Guide for HP-UX Systems, First Edition, 1997. Available: http://docs.hp.com/en/B6056-96002/ch07s03.html

[17] N.R. Fredrickson, A. Afsahi, and Y. Qian, "Performance Characteristics of OpenMP Constructs, and Application Benchmarks on a Large Symmetric Multiprocessor", *Proceedings of the 17th Annual ACM International Conference on Supercomputing,* ICS'03, San Francisco, CA, USA, June 23-26, 2003, pp. 140-149.

[18] GM reference manual 2.0.6. Myricom.

[19] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", *Parallel Computing*, 22(6), pp. 789-828, 1996.

[20] J. Hsieh, T. Leng, V. Mashayekhi, and 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] W. Jiang, L. Jiuxing, J. Hyun-Wook, D.K. Panda, W. Gropp, and R. Thakur, "High Performance MPI-2 One-Sided Communication over InfiniBand", *In Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004). IEEE,* 2004, pp. 531-538.

[22] H. Jin and G. Jost, "Performance Evaluation of Remote Memory Access (RMA) Programming on Shared Memory Parallel Computers", NASA Technical Reports

NAS-03-001, January 2003.

[23] H. Jin, R.F. Van der Wijngaart, “Performance characteristics of the multi-zone NAS parallel benchmarks”, *Proceedings of 18th International Parallel and Distributed Processing Symposium*, 2004. 26-30 April, pp. 6.

[24] S. Karlsson, and M. Brorsson, “A Comparative Characterization of Communication Patterns in Applications using MPI and Shared Memory on an IBM SP2”, *Proceedings of CANPC’98, Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, 1998.

[25] J. Kim and D.J. Lilja, “Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs”, *Proceedings of CANPC’98 Workshop on Communication, Architecture, and Applications for Network-based Parallel computing*, 1998.

[26] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D.K. Panda, and P. Wyckoff, “Microbenchmark Performance Comparison of High-Speed Cluster Interconnects”, *IEEE Micro*, 2004, Vol. 24, No.1, pp.42-51.

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

[28] T. G. Mattson, “Programming environments for parallel computing: a comparison of CPS, Linda, P4, PVM, POSYBL, and TCGMSG”, *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences, Software Technology* Volume 2, 4-7 Jan. 1994 pp.586 – 594.

[29] Mellanox Technologies, Available: http://www.mellanox.com.

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

[31] MPICH-2 homepage. Available: http://www-unix.mcs.anl.gov/mpi/mpich2/.

[32] MPICH-GM. Available: http://www.myri.com/scs/

[33] J. Nieplocha and B. Carpenter, “ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems”, *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99,* San Juan, Puerto

Rico, April 1999.

[34] J. Nieplocha, E. Apra, J. Ju, and V. Tipparaju, “One-sided Communication on Clusters with Myrinet”, *Cluster Computing* 6, pp. 115-124, 2003.

[35] J. Nieplocha, J. Ju, and E. Apra, “One-sided Communication on Myrinet-based SMP Clusters using the GM Message-Passing Library”, *in Proceedings of CAC01 Workshop, 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, 2001.

[36] J. Nieplocha, R.J. Harrison, and R.J. Littlefield, “Global Arrays: A nonuniform memory access programming model for high-performance computers”, *Journal of Supercomputing* 10, pp. 197-220, 1997.

[37] OpenMP Tutorial from Lawrence Livermore National Laboratory, available: http://www.llnl.gov/computing/tutorials/openMP

[38] Pallas GmbH. Pallas: Think Parallel. http://www.pallas.com.

[39] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie, “Performance Evaluation of the Quadrics Interconnection Network”, *Journal of Cluster Computing*, 2003, pp. 125-142.

[40] Y. Qian, A. Afsahi, N.R. Fredrickson, and R. Zamani, “Performance Evaluation of the Sun Fire Link SMP Clusters”, *18th International Symposium on High Performance Computing Systems and Applications, HPCS 2004*, May 2004, pp. 145-156.

[41] Y. Qian, A. Afsahi, and R. Zamani, "Myrinet Networks: A Performance Study", *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications, IEEE NCA04*, Cambridge, MA, USA, August 30 - September 1, 2004, pp. 323-328.

[42] R. Rabenseifner, “Comparison of Parallel Programming Models on Clusters of SMP Nodes”, *In proceedings of the 45th Cray User Group Conference, CUG SUMMIT* 2003, May 12-16, Columbus, Ohio, USA.

[43] D. Seed, A. Sivasubramaniam, and C.R. Das, “Communication in Parallel Applications: Characterization and Sensitivity Analysis”, In *Proceedings of the International Conference on Parallel Processing*, pages 446-453, August 1997.

[44] SPEC HPC2002 Benchmark suite, (http://www.spec.org/hpc2002/).

[45] Standard for information technology - portable operating system interface (POSIX). System interfaces. IEEE Std 1003.1, 2004 Edition.

[46] V. Sunderam, J. Dongarra, A. Geist, and R Manchek, "The PVM Concurrent Computing System: Evolution, Experiences, and Trends", *Parallel Computing*, Vol. 20, No. 4, April 1994, pp. 531-547.

[47] V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, and D. K. Panda, "Exploiting Nonblocking Remote Memory Access Communication in Scientific Benchmarks on Clusters", *International Conference on High Performance Computing, HiPC 2003*, Bangalore, India.

[48] TOP500 Supercomputer Sites. Available: http://www.top500.org

[49] R.F. Van der Wijngaart, and H. Jin, "NAS Parallel Benchmarks, Multi-Zone Versions", NAS Technical Report NAS-03-010, July 2003.

[50] R.F. Van der Wijngaart, Rupak Biswas, Michael Frumkin, and Huiyu Feng, "Beyond the NAS Parallel Benchmarks: Measuring Performance of Dynamic and Grid-oriented Applications", NASA Ames Research Center.

[51] J.S. Vetter and F. Mueller, "Communication Characteristics of large-scale scientific applications for contemporary cluster architectures", *Journal of Parallel and Distributed Computing* 63, 2003, pp. 853-865.

[52] W. Vogels, D. Follett, J. Hsieh, D. Lifka, and D. Stern, "Tree-saturation control in the AC3 velocity cluster", *Hot Interconnect 8*, 2000.

[53] F.C. Wong, R.P. Martin, R.H. Arpaci-Dusseau and D.E. Culler, "Architectural requirements and scalability of the NAS parallel benchmarks", *Proceedings of 1999 ACM/IEEE conference on Supercomputing*, 1999.

[54] R. Zamani and A. Afsahi, "Communication Characteristics of Message-Passing Scientific and Engineering Applications", *16th IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS 2004*, MIT, Cambridge, MA, USA, November 9-11, 2004.

[55] R. Zamani, Y. Qian, and A. Afsahi, "An Evaluation of the Myrinet/GM2 Two-Port Networks", *3rd IEEE Workshop on High-Speed Local Networks, HSLN 2004*, held in conjunction with the 29th Annual IEEE Conference on Local Computer Networks, LCN 2004, Tampa, FL, USA, November 16-18, 2004, pp. 734-742.