Performance Analysis And Optimal Utilization Of Inter-Process Communications On A Commodity Cluster

Saqib Ali Khan
North Carolina Agricultural and Technical State University

PERFORMANCE ANALYSIS AND OPTIMAL UTILIZATION OF INTER-PROCESS COMMUNICATIONS ON A COMMODITY CLUSTER

by

Saqib Ali Khan

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Department: Electronics, Computer, and Information Technology
Major: Information Technology
Major Professor: Dr. Yilli Tseng

North Carolina A & T State University
Greensboro, North Carolina
2011

School of Graduate Studies
North Carolina Agricultural and Technical State University

This is to certify that the Master's Thesis of

Saqib Ali Khan

has met the thesis requirements of
North Carolina Agricultural and Technical State University

Greensboro, North Carolina
2011

Approved by:

Dr. Yilli Tseng Major Professor	Dr. DeWayne R. Brown Committee Member
Dr. Qing-An Zeng Committee Member	Dr. Clay Gloster Department Chairperson

Dr. Sanjiv Sarin
Dean of Graduate Studies

DEDICATION

I dedicate this work to my parents, teachers and any one from whom I learned anything in my life.

BIOGRAPHICAL SKETCH

Saqib Ali Khan was born in a small village in Pakistan on October 1974. He Received the Bachelor of Science degree in Electrical Engineering from University of Engineering and Technology Peshawar, Pakistan. He is a candidate for Master of Science Degree in Information Technology from North Carolina A&T State University in Greensboro, North Carolina.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Khan, Saqib Ali. PERFORMANCE ANALYSIS AND OPTIMAL UTILIZATION OF INTER-PROCESS COMMUNICATIONS ON A COMMODITY CLUSTER. **(Major Advisor: Dr. Yilli Tseng)**, North Carolina Agricultural and Technical State University.

Classical science is based on theory, observation and physical experimentation. Contemporary science is characterized by theory, observation, experimentation and numerical simulation. With the use of hardware and software we can simulate lots of phenomenon. This saves time, money and physical resources.

Simulation of a certain phenomenon requires lots of computing power. Answer to these computational power needs is high performance computer. High performance computers consist of numerous processors working on same task in parallel. In the past, high performance computers were very expensive and affordable by few institutions. After Message Passing Interface library is ported to PC platform, commodity clusters can be built of inexpensive PCs and afforded by any researcher.

Lots of performance analyses have been conducted on high-end supercomputers. None has been done on commodity clusters. In this thesis, experiments for six major MPI communication functions were performed on eight different configurations of clusters. Performance analyses were then conducted on the results. Based on the results, methods for optimal utilization of inter-process communications on commodity clusters were proposed.

CHAPTER 1

INTRODUCTION

1.1 The Need for High Performance Computing

Classical science is based on observation, theory, and physical experimentation. In contrast, with the utilization of computer hardware and software, contemporary science is characterized by observation, theory, experimentation, and numerical simulation [1]. Numerical simulation is a mathematical modeling on a discrete model. It may represent a discrete approximation of the continuum partial differential equations, or it may represent a statistical representation of the microscopic model [2]. Numerical simulation is an increasingly important tool for scientists and engineers, who often cannot use physical experimentations to test because they may be too expensive or time-consuming, because they may be unethical, or because they are impossible to perform [1]. Numerical simulation requires computers to carry out calculations. Many important scientific and engineering problems are so complex that solving them via numerical simulation requires extraordinary powerful computers. Those complex problems are often called grand challenges for science and require high performance computers to perform numerical simulations [2].

1.2 High Performance Computer

High Performance Computers (HPC) refer to parallel computers or multiprocessor computers. HPCs gain high throughput by having plenty of processors working in parallel. Categorized by communication models, parallel computers are classified into two major categories, namely shared-memory platform and message-passing platform [3] - [5].

1.2.1 Shared Memory Platform

A typical architecture of shared-memory platform is shown in Figure 1.1. As the name suggest, it has shared-memory which is accessible to all processor in the system. All processes interact through accessing shared variables stored on the shared-memory. Although the programming model of shared-memory machines is much easier than message-passing platform, it requires more costs to build shared-memory hardware. The hardware complexity also prevents it from implementing as many processors as message-passing systems do.

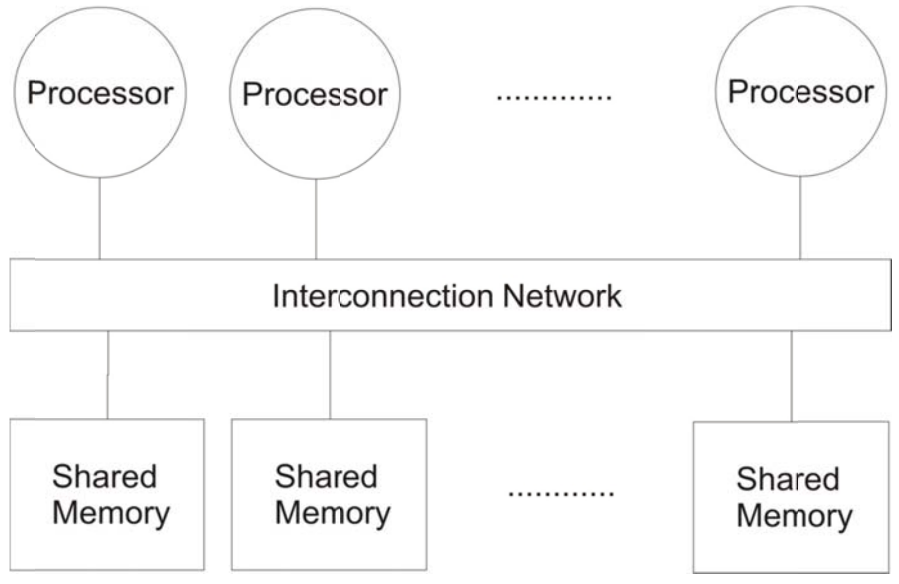

Figure 1.1. Architecture of Shared-Memory Platform

1.2.2 Message-Passing Platform

A typical architecture of message-passing platform is shown in Figure 1.2. Multiple processors are connected by networking switches to form a cluster. As the name suggests, each node of a message-passing platform interacts with one another through passing messages. Because of the simplicity of its architecture, it is very easy for message-passing platform to scale to plenty of processors. It becomes the most popular high performance computer architecture. On www.top500.org, it shows more than 82.2% of the top 500 supercomputers in the world are clusters as of June, 2011. Message-Passing Interface (MPI) is the most popular standard developed for the programming paradigm for message-passing platform [1] - [4], [6] - [8].

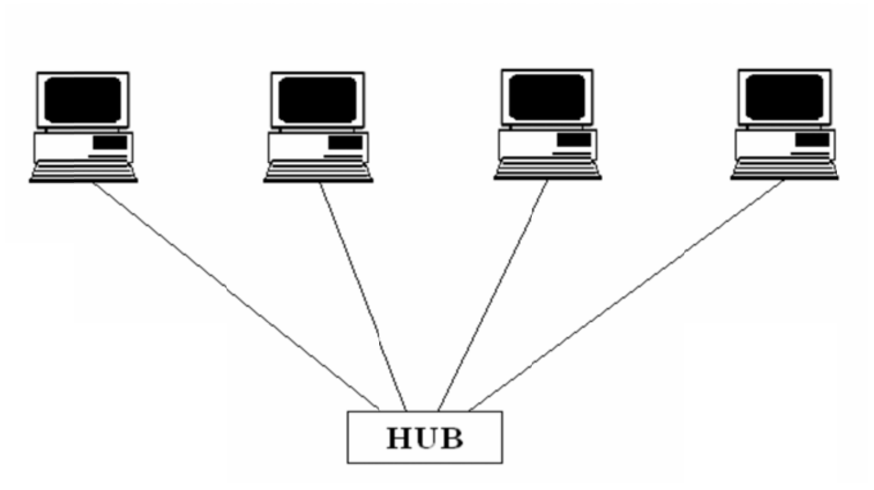

Figure 1.2. Architecture of Message-Passing Platform

1.3 Dilemma of High Performance Computing and Resolution

Although numerical simulation becomes an important tool for many science and engineering disciplines, it relies on the availability of HPCs. In the past, HPCs are very expensive and cannot be afforded by most researchers and institutions. The high costs were the barrier to wide adoption of HPCs. Nonetheless, due to the simplicity of message-passing platform's architecture and thanks to the porting of MPI library to personal computer (PC) platform by open-source software developers, now researchers and institutions can connect inexpensive PCs with generic networking and install MPI library on them to build a commodity cluster [9] - [12]. Even retired PCs can be used to build a commodity cluster [13]. That paves a way for all researchers who are interested in numerical simulations to own an effective and affordable tool to perform research through numerical simulations. The commodity clusters still have decent computing power to solve smaller-scale numerical simulations.

1.4 Related Work

Although lots of performance analyses have been conducted for HPCs and high performance networking systems in the past, they were all performed on high-end HPCs. Xu et al. [14] and Miguel et al. [15] addressed IBM SP2. Muelder et al. [16] and Graham et al. [17] measured Cray XT. High-end clusters are the focus of the papers of Luecke et al. [18], Grbovic et al. [19], and Shipman et al. [20]. Dongarra et al. [21] evaluated multiple systems, but all of them were high-end HPCs. Doerfler et al. [22] and Graham et al. [23] investigated high performance networks. InfiniBand is the target network system of the papers of Shipman et al.[24], Hoefler et al.[25], and Hursey et al.[26]. No performance analysis has been done on commodity clusters. As they use generic parts instead of high performance parts employed by high-end supercomputers, for example Gigabit Ethernet networking versus InfiniBand networking, commodity cluster could exhibit different characteristics than those of supercomputers. Because they are more popular and adopted than before, performance analysis should be conducted on commodity clusters to investigate their characteristics in executing MPI communication functions in order to carry out optimal utilization of them. This thesis begins with introducing installation and configuration of a typical commodity cluster; then executes and measures six major MPI communication functions on eight different configurations of commodity clusters; finally analyzes performance of commodity clusters and proposes optimal utilization of major MPI communication functions.

CHAPTER 2

INSTALLATION AND CONFIGURATION OF A COMMODITY CLUSTER

In this Chapter, the general components and procedures required to install and configure a commodity cluster is covered [13].

2.1 Hardware Requirements

Any PC with 128MB RAM or more and an Ethernet network interface card (NIC) can work as a node. PCs with Pentium III 500MHz CPUs work smoothly at author's institution. All nodes have to be connected with a network hub or switch. Generic Ethernet NICs and switches can be acquired with very little cost.

2.2 Operating System and Software Packages

Among all operating systems, only Linux can be acquired for free. Also, Linux operating systems come with plenty of hardware drivers which cover almost all legacy and new hardware, that further makes it the ideal OS for commodity clusters. Although several implementations of open source Linux operating systems are available, not all of them work well with the MPI library. After extensive experiments, the author chose and installed CentOS Linux which is a clone of commercial Red Hat Enterprise Linux and downloadable at www.centos.org. The following actions should be done for all nodes in the cluster. Firewall and SELinux should be turned off during installation as they cause difficulty for communications among nodes which are required for executing MPI programs. Fortunately, security is not a concern as long as the cluster is not connected with other networks. The editor, gcc C++ and FORTRAN have to be installed to as they are required for MPI programming process. One node should be selected as the server node and the following software should be installed on the node during installation: Network Information Service (NIS) server and Network File System (NFS) server configuration tool. After Linux is installed, networking should be correctly configured so that all nodes of the cluster can communicate among one another. In general, one user account with the same name needs to be created on each node because a parallel program is dispatched to each node under the same user account. NIS which is addressed in later subsections can take care of this and other issues.

2.3 MPI library

The next major step is to install the MPI library. Again, there are several implementations of MPI, such as MPICH, LAM/MPI, Open MPI, etc. Nevertheless, only Open MPI is still under active development and growing more powerful. Therefore, Open MPI is the best choice. The steps to install Open MPI are quite straightforward. They are described as follows [6]. First, download the latest version of the library from Open MPI's website, www.openmpi.org.Log into the root account to install it. Copy the compressed file to the /tmp directory. Uncompress the file with the command line or double click to invoke GUI uncompress software:

```
shell$ gunzip -c openmpi-1.4.3.tar.gz | tar xf –
```

Then change to the directory openmpi-1.3. Configure and make Open MPI with the commands below [6]. Replace the directory after prefix option if you want to install into another directory.

```
shell$ cd openmpi-1.3
shell$ ./configure --prefix=/usr/local
shell$ make all install
```

The make process may take up to one hour on an old PC.

2.4 NFS and NIS

Before we run a parallel program on our cluster, we need to dispatch a copy of the program's executable file onto every node under the same account. Manually copying the executable to all nodes is impractical. NFS is the solution to help us to complete this mission. With NFS, the program's executables only need to be saved into the shared directory of the NFS and a copy of the program is automatically copied to all other nodes. While NFS can solve the copying problem, there are some potential security problems. Each user has access to the shared directory, meaning any user can run, correct, save, and delete others' programs. To remedy this problem, NIS was used to create accounts on the server node. All users can login from any node and manage their own shared directory. You have to login as root to perform all following setups and reboot all nodes to take effect. Do not reboot any client node until the server node completes its boot-up process, otherwise other nodes cannot read the correct configuration information from the server and perform normally.

2.4.1 Set Up the NFS Server

1) From the *NFS Server Configuration* window, click *File → Add Share*. The *Add NFS Share* window appears. In the *Add NFS Share* window Basic tab, type the following information [27]:

- Directory – Type the name of the directory you want to share. Type "/home" which is the parent directory to all user directories.
- Host(s) – Enter one or more host names to indicate which hosts can access the shared directory. Type "*" to let all nodes access NFS server.
- Basic permissions – Click *Read/Write* to let remote computers mount the shared directory with read/write access.

2) To permanently turn on the NFS service, type:

```
shell$ chkconfignfs on
shell$ chkconfignfslock on
```

2.4.2 Set Up the NFS Client

To set up an NFS file system to mount automatically each time you start your Linux system, you need to add an entry for that NFS file system to the /etc/fstab file. The /etc/fstab file contains information about all different kinds of mounted (and available to be mounted) file systems for your Linux system [27].

The format for adding an NFS file system to your local system is the following:

```
host:directory  mountpoint     options          0         0
```

The first item "host:directory" identifies the NFS server computer and shared directory. Mountpoint is the local mount point on which the NFS directory is mounted, followed by the file system type “nfs”. Any options related to the mount appear next in a comma separated list. For our system, we add the following NFS entries to /etc/fstab:

```
kingtiger1:/home /home nfsrsize=8192,wsize=8192 0
```

2.4.3 Set Up the NIS Client

1) Defining an NIS domain name:

   Our NIS domain name is kingtiger, we can set it by typing the following as the root user from the shell:

   ```
   shell$ domainnamekingtiger
   ```

   To make the NIS domain name permanently, you need to have the domainname command run automatically each time your system boots. We can do it by adding the command line to a run-level script that runs before the ypbind daemon is started. We edited the /etc/init.d/network file and added the following lines just after the first set of comment lines [27], [28].

   ```
   Domain name kingtiger
   ```

2) Setting up the /etc/yp.conf file:

   We had an NIS domain called kingtiger and a server called kingtiger1, we should have the following entries in our /etc/yp.conf file:

   ```
   Domain kingtiger server kingtiger1
   Domain kingtiger broadcast
   ypserver kingtiger1
   ```

3) Configuring NIS client daemons:

We need set up an existing run-level script called ypbind to start automatically at boot time. To do this, you can run the following command:

```
shell$ chkconfig ypbind on
```

4) Using NIS maps:

   For the information being distributed by the NIS server to be used by the NIS client, you must configure the /etc/nsswitch.conf file to include nis in the search path for each file you want to use. In most cases, the local files are checked first"(files", followed by "nis". The following are examples of how some entries should be changed:

```
passwd:    files nis
shadow:    files nis
group:  files nis
hosts:   files nis dns
```

   As soon as the /etc/nsswitch file is changed, the data from NIS maps are accessible. No need to restart the NIS service.

2.4.4 Set Up the NIS Server

1) To configure your Linux system as an NIS server, you should first configure it as an NIS client and reboot the system [27], [28].
2) Creating NIS maps:

   To create NIS maps so that your Linux system can be an NIS server, start from the /var/yp directory from a Terminal window as root user. In that directory, a Makefile enables you to configure which files are being shared with NIS. All default configurations in Makefile are ok for our purposes, so we don't need change them.

3) Configuring access to maps:

   In the /etc/ypserv.conf file, you can define rules regarding which client host computers have access to which maps. For our purposes we just need add the following line into /etc/ypserv.conf to allow all hosts access to all maps:

```
        *       :       *       :       *       :   none
```

4) Configuring NIS serve daemons:

   We can use the following chkconfig command to set ypserv and yppasswdd scripts to start automatically at boot time.

```
        shell$ chkconfig ypserv on
        shell$ chkconfig yppasswdd on
```

5) Updating the NIS maps:

   If you modify the sources for NIS maps (for example if you create a new user by adding the account to the passwd file), you need to regenerate the NIS maps. This is done by a simple

```
        make –C /var/yp
```

   This command will check which sources have changed, creates the maps new and tell ypserv that the maps have changed.

2.5 Disabling Password Authentication

As Open MPI is configured by default to use ssh (secured shell) to dispatch parallel tasks, it is ssh that asks for password to authenticate the connection. Extra steps below will prevent ssh from requesting passwords [6]. Because the measure should only work for the user account which intends to run MPI applications, log into the user account instead of root account of server node, to configure. First, generate the private and public key for the user account by executing:

```
shell$ ssh-keygen -t dsa
```

Next, ssh to all other nodes respectively by typing:

```
shell$ ssh host_name
```

That will generate the hidden .ssh directory on each node with the necessary attribute. Remember to go back to the server node by typing "exit" to finish the remaining procedures. Change into the .ssh directory under the user account and do the following.

```
shell$ cd/home/user/.ssh
shell$ cp id_dsa.pub authorized_keys
```

Next, repeat the following steps for all other nodes in the cluster. Replace "user" with account names which you want to disable requesting passwords.

```
shell$ scp authorized_keyshost_name:/home/user/.ssh
shell$ scp id_dsahost_name:/home/user/.ssh
shell$ scp id_dsa.pub host_name:/home/user/.ssh
```

With these procedures done, all private, public, and authorized keys are duplicated on each node. They will be used for authentication for all future connections without passwords being requested from other nodes. Now the MPI applications can be executed on multiple nodes of this cluster without being asked for passwords.

CHAPTER 3

INTER-PROCESS COMMUNICATION EXPERIEMNTS

3.1 MPI Library

Since commodity clusters are based on message-passing platform, they rely on passing messages to other processes to perform parallel tasks. The most popular communication protocol for message-passing platform is MPI which was made a standard in 1994. MPI has several different implementations since then. Currently, the most popular MPI library is Open MPI which is developed by Open MPI team at Indiana University. The Open MPI library used in this experiment is version 1.3.

3.2 Tested MPI Functions

MPI communication functions consist of two major categories: point-to-point communication and collective communication. Point-to-point communication functions are used to send messages from one process to another. Collective communication functions involve all processes participating in a parallel program [2], [7], [8].

3.2.1 Point-to-Point Communication Functions

The point-to-point communication functions tested in the experiments were MPI_Send and MPI_Recv. MPI_Send is called by a process to send a message to another process. It must be paired by a MPI_Recv function called by the receiving process. Both functions are blocking functions which do not return until the message is successfully passed from sending process to receiving process. Therefore, the performance of both functions is essential to MPI programs.

3.2.2 Collective Communication Functions

The five collective communication functions tested in the experiments were MPI_Bcast, MPI_Reduce, MPI_Gather, MPI_Scatter, and MPI_Alltoall. A communicator is defined as all processes participating in a parallel program. MPI_Bcast is called by a process to broadcast the same message to all processes in the same communicator, including itself. MPI_Reduce collects a message from each process in the same communicator, including itself, and reduce them with a specified operation to a variable in a specified process. MPI_Gather collects a message from each process in the communicator, including itself, and stores them in an array in the order of the rank of each process. MPI_Scatter splits an array in a process and distributes one segment to one process in the communicator in the order of the rank of each process. MPI_Alltoall is equivalent to all processes in the communicator calling MPI_Gather or MPI_Scatter [3], [7], [8]. The five collective communication functions are most frequently used in and hence are fundamental to MPI programs.

3.3 Methodologies

In the experiments, different MPI communication functions were executed with different sizes of messages ranging from 4 bytes, the length of an integer variable, to 1 Mbytes on different cluster configurations. The message length is multiplied by a factor of 2 for next sample. For the point-to-point communication functions, the well-known ping-pong method was utilized [14], [18], [29]. In ping-pong method, process 0 calls MPI_Send to send a message to process 1 while process 1 uses MPI_Recv to receive the message. Process 1 immediately sends the same message back to process 0. The latency of ping-pong operation is measured and then divided by two to determine the one-way point-to-point communication time. The procedure is repeated for 1000 times. In reality, MPI_Wtime is called before the first iteration and after the 1000th iteration to dilute the overhead of calling MPI_Wtime. The average latency is calculated and used as the result. As for collective communication functions, the function is called and followed by calling MPI_Barrier to ensure the collective communication is synchronized on all processes before marking ending time [18], [30]. The procedure is repeated for 1000 iterations. Likewise, MPI_Wtime is called before and after the 1000 iterations and average time is calculated as the result. For all functions, three iterations are executed to “warm up” the communication channels and procedures before the beginning time is marked. The number of iterations used was 20 in the methodology of Xu et al. [14] while it was 1000 in the paper of Miguel et al. [15] and OSU Benchmarks [30]. The results from Luecke et al. [18] show that 15000 iterations do not improve the accuracy significantly. Hence, 1000 iterations were used in the experiments.

3.4 Configurations of Tested Clusters

Three physical clusters were used in the experiments: Four nodes equipped with AMD Phenom X4 2.5GHz CPU and 1GB RAM each, four nodes equipped with Intel Pentium 4 2.4GHz CPU and 256MB RAM each, and eight nodes equipped with Intel Pentium III CPU and 256MB RAM each. Eight different configurations were created by changing network interface cards, adjusting number of nodes participating the testing program, and dispatching different number of processes to be executed by each node. The detailed information of each configuration is listed in Table 3.1.

Table 3.1. Configurations of Tested Commodity Clusters

Configuration	CPU	Networking	Number of Node
1	AMD Quadcore 2.5GHz	Internal Channels	4 cores
2	AMD Quadcore 2.5GHz	Gigabit Ethernet	4
3	Intel Pentium 4 2.4GHz	Gigabit Ethernet	4
4	Intel Pentium 4 2.4GHz	100Base-T Ethernet	4
5	Intel PIII 700MHz	Gigabit Ethernet	4
6	Intel PIII 700MHz	100Base-T Ethernet	4
7	Intel PIII 700MHz	Gigabit Ethernet	8
8	Intel PIII 700MHz	100Base-T Ethernet	8

CHAPTER 4

PERFORMANCE ANALYSIS ON RESULTS AND OPTIMAL UTILIZATION

Performance analyses were conducted on the results collected from the experiments. In high performance computing field, performance of a HPC is compared as a whole system with other HPCs, not on individual processor or networking equipment. Therefore, the performance of all configurations of clusters was compared for each MPI communication function first. Then each collective communication function was compared against MPI_Send for each configuration of cluster individually to determine whether the specific function performs better than equivalent number of MPI_Send on a specific configuration of cluster. Lastly, methods for optimal utilization of inter-process communications were proposed based on the performance analyses. As the size of message length is multiplied by a factor of two for next sample, it grows exponentially and is difficult to be displayed and distinguished on the same chart due to non-uniform scale. Therefore, all results for each test are divided into four groups and are charted as four graphs so that the scale can be properly shown and all samples can be distinctly read.

4.1 Comparison of Performance of All Clusters

In this section, results from configurations with 4 nodes for the same MPI communication function are plotted on same chart so they can be compared. The latency of results from the configuration of PIII/100Base-T/4 nodes is too big when compared with readings from the other five configurations. Hence, they are removed so that other results can be displayed with proper scale and are distinguishable. Charts for MPI_Send, MPI_Bcast, MPI_Reduce, MPI_Gather, MPI_Scatter, and MPI_Alltoall were plotted and displayed from Figure 4.1. to Figure 4.6.

In Figure 4.1., which displays the performance of MPI_Send, it shows that, from message length of 16 Kbytes up, the latency for different cluster from low to high is in the following order:

1. **Quadcore/internal channels**
2. **Quadcore/Gigabit**
3. **P4/Gigabit**
4. **PIII/Gigabit**
5. **P4/100Base-T**
6. **PIII/100Base-T**

Although the ranking of latency for message length under 16 Kbytes is not in the same order, the latency ranking for all other MPI communication functions tested for all message lengths are quite in this order except for MPI_Scatter and MPI_Alltoall with large messages over 32 Kbytes. The only difference is that latency of PIII/Gigabit is better than that of P4/Gigabit for those large file sizes. All other four clusters are still in the same order. That shows that generally the performance of those configurations is in this ranking order. It demonstrates that Gigabit NICs perform significantly better 100Base-T NICs. In turn, internal channels outperform Gigabit networking.

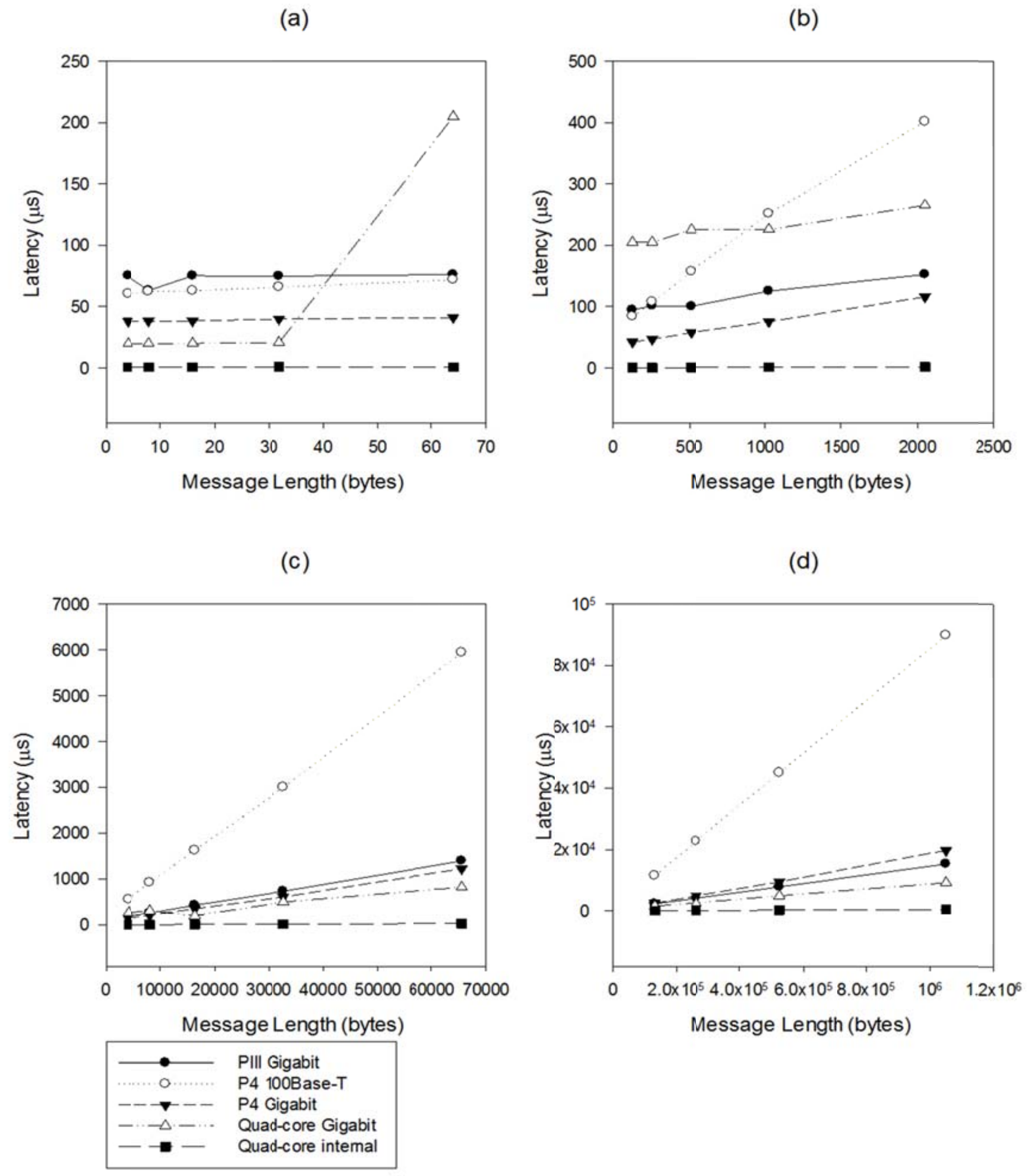

Figure 4.1. Comparison of MPI_Send Latency on All Clusters

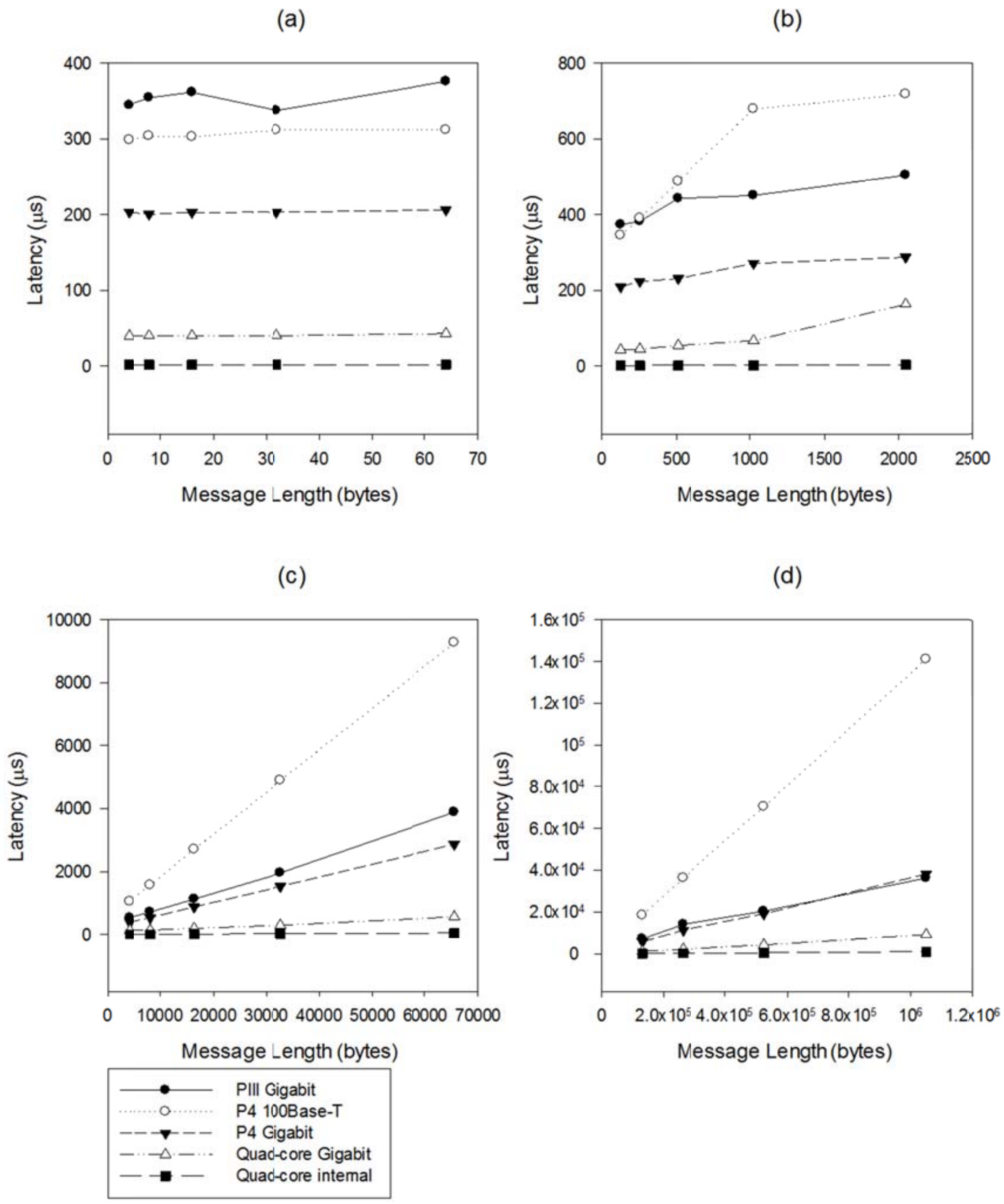

Figure 4.2. Comparison of MPI_Bcast Latency on All Clusters

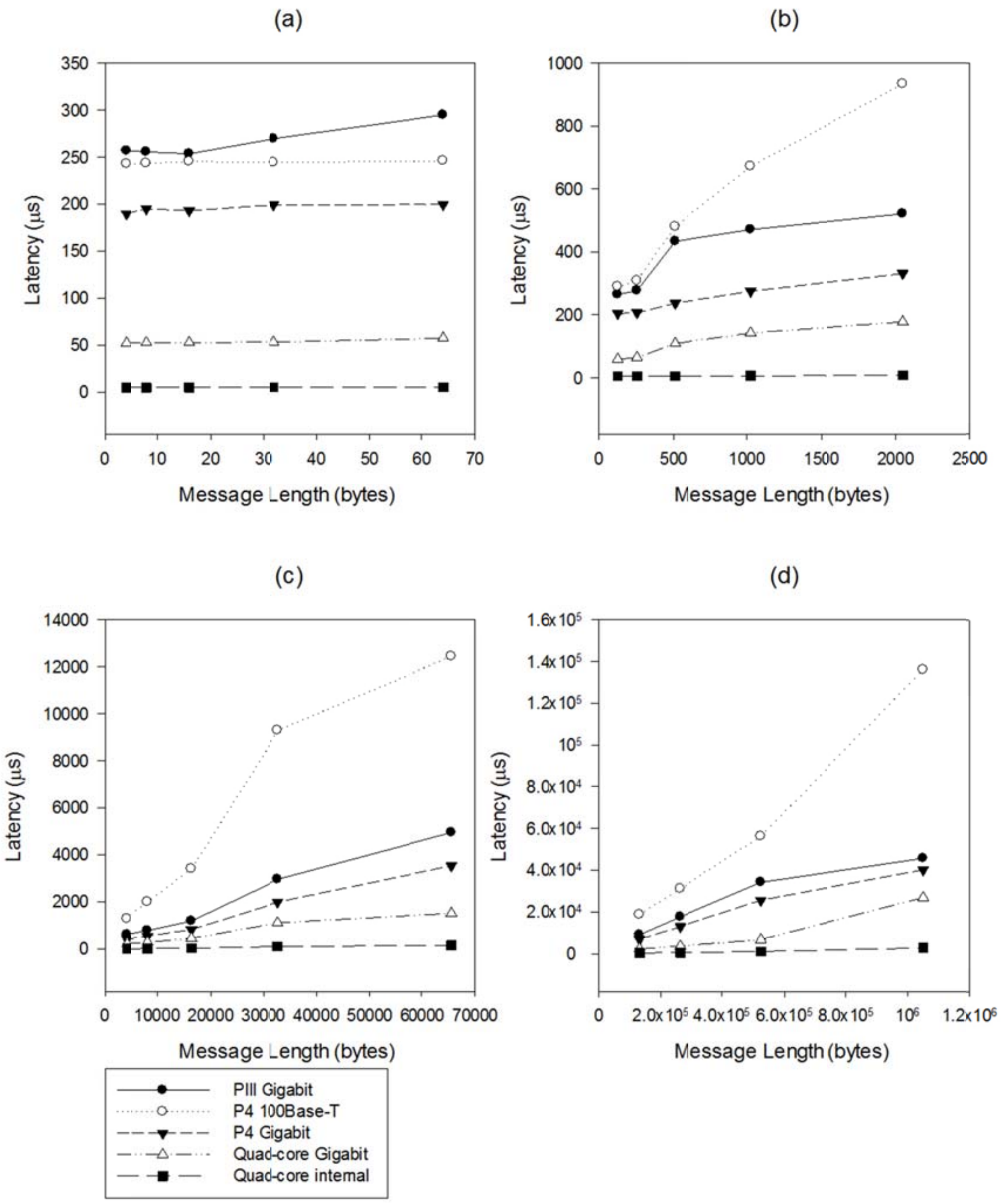

Figure 4.3. Comparison of MPI_Reduce Latency on All Clusters

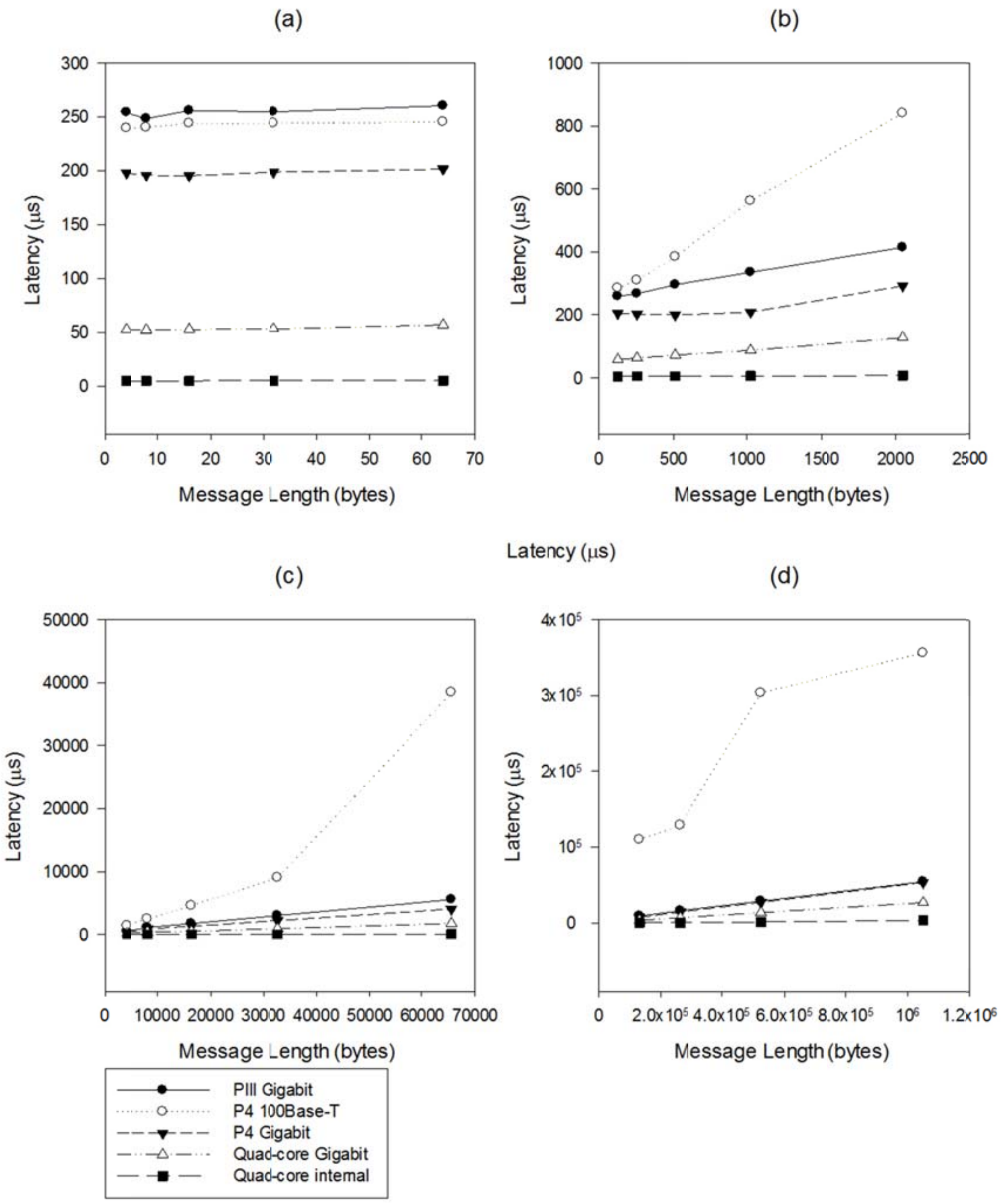

Figure 4.4. Comparison of MPI_Gather Latency on All Clusters

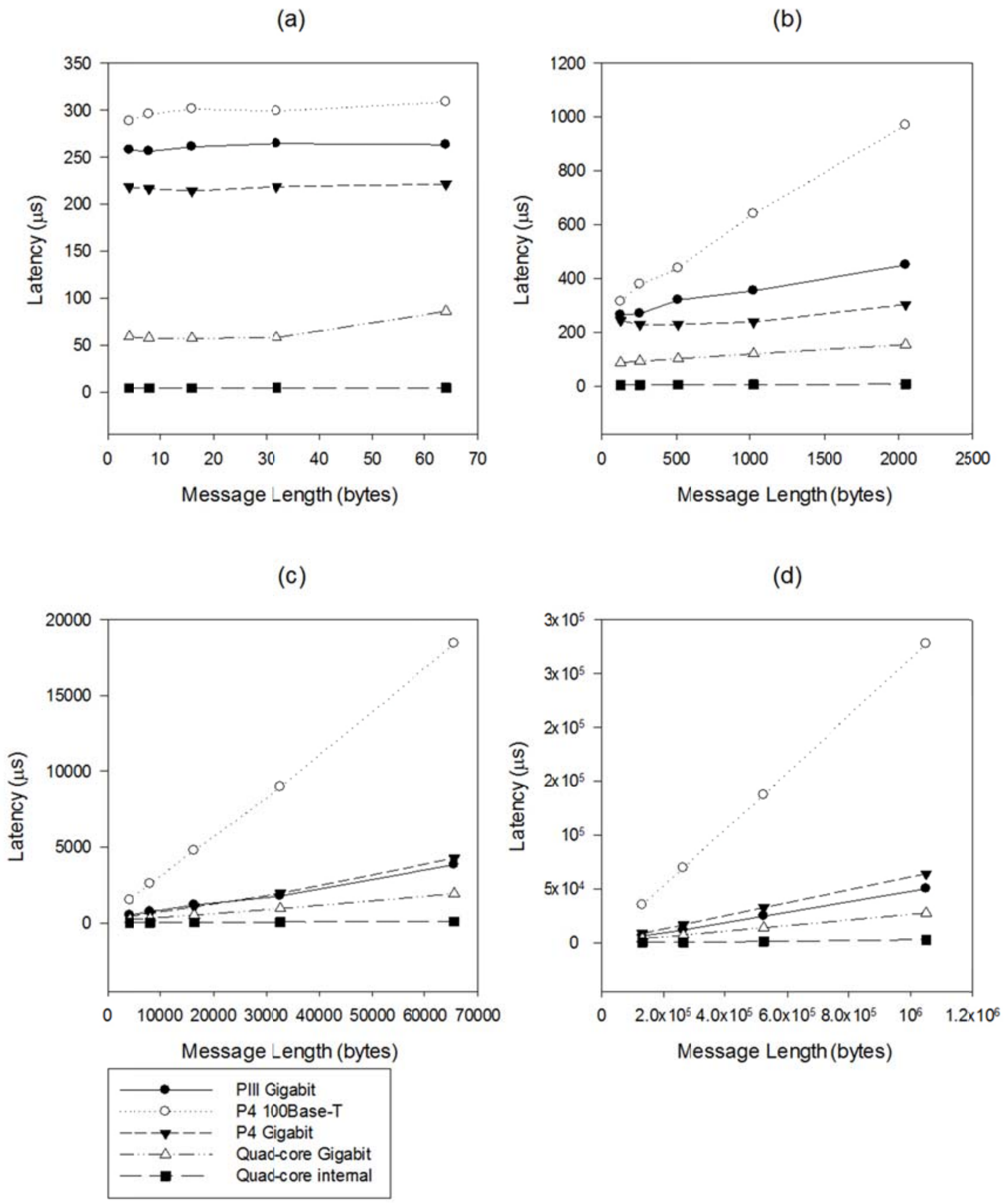

Figure 4.5. Comparison of MPI_Scatter Latency on All Clusters

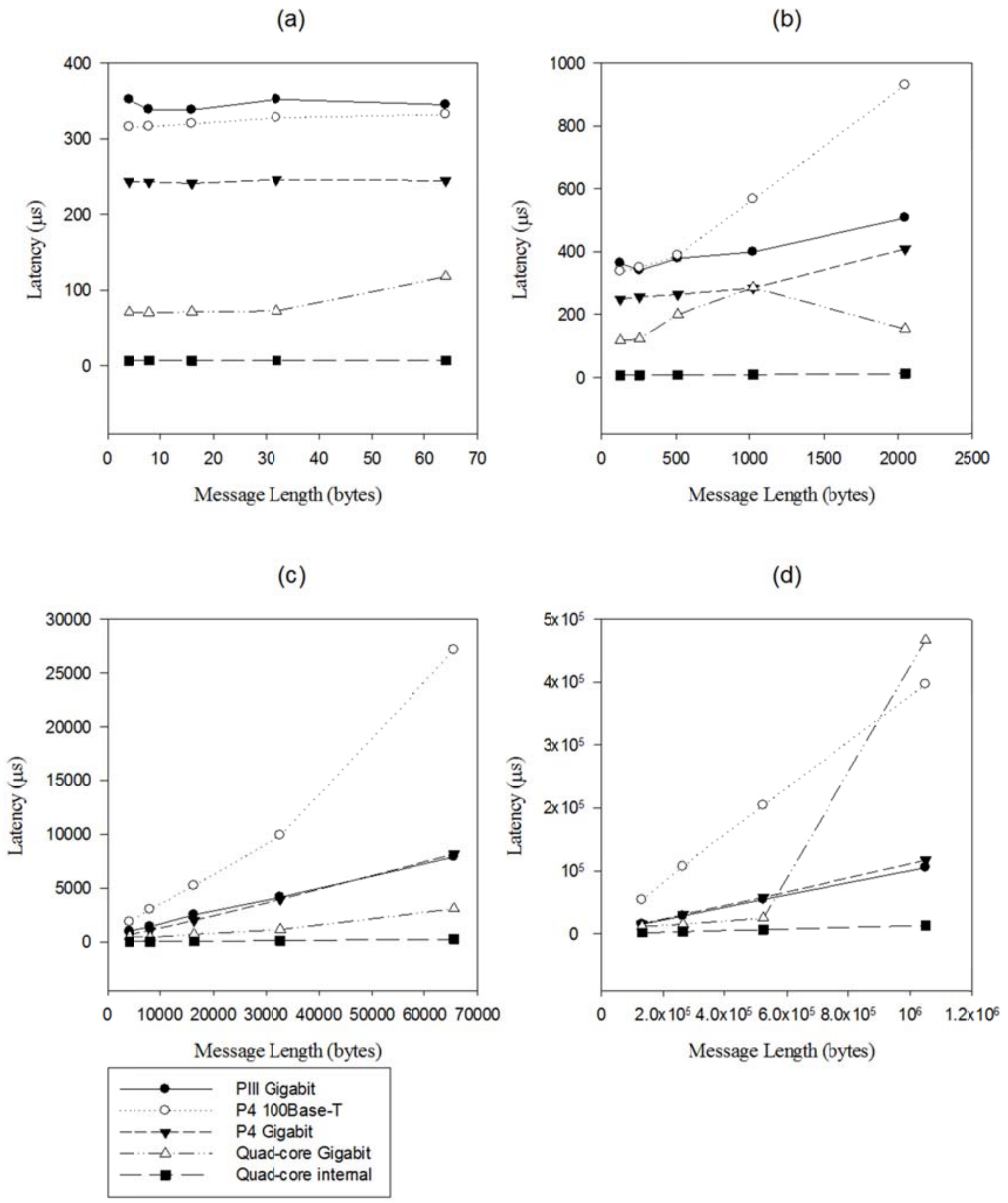

Figure 4.6. Comparison of MPI_Alltoall Latency on All Clusters

4.2 Performance Analyses on Collective Communication Functions

As MPI collective communication functions send out messages to all processes in the same communicator, the following performance analyses compare the latency of each MPI collective communication function to the aggregate latency of equivalent number of MPI_Send. For example, calling MPI_Bcast in a parallel program which is executed with four processes is equivalent to calling MPI_Send three times to send the same message to the other three processes because the overhead of sending the same message to itself is negligible. The latency of executing MPI_Bcast with four processes is compared with the aggregate latency of executing MPI_Send three times. The latency of executing MPI_Alltoall with four processes is compared with the aggregate latency of executing MPI_Send twelve times because each process of the four processes need to collect one message from the other three processes.

4.2.1 Scenarios

There are four possible scenarios when the performance of a collective function is compared against that of MPI_Send:

1. The latency of a collective communication function is higher than that of MPI_Send for all message lengths. Under that circumstance, multiple MPI_Send should be used instead of the specific collective communication function. An example is shown in Figure 4.7.
2. The latency of a collective function is lower than that of MPI_Send for all message lengths. Under that circumstance, the specific collective communication function should be used instead of multiple MPI_Send. An example is shown in Figure 4.8.

3. The latency of a collective communication function is higher than that of MPI_Send initially for messages with shorter lengths. Above a certain message size, the latency of the specific collective communication function becomes lower than that of MPI_Send for all messages lengths. Under that circumstance, multiple MPI_Send should be used instead of the specific collective communication function to send messages shorter than that certain size. An example is shown in Figure 4.9.
4. The latency of a collective communication function is lower than that of MPI_Send initially for messages with shorter lengths. Above a certain message size, the latency of the specific collective communication function becomes higher than that of MPI_Send for all messages lengths. Under that circumstance, multiple MPI_Send should be used instead of the specific collective communication function to send messages longer than that certain size. An example is shown in Figure 4.10.

4.2.2 Notations

Latencies of all collective communication functions were compared against their corresponding number of MPI_Send on each configuration of cluster. The results were charted for individual cluster. All the charts are included in Appendix. The observations based on the four scenarios are tabulated for each collective communication function and listed in following sections. The notation H means the latency of the collective communication function is higher than the aggregate latency of MPI_Send. The notation L means the latency of the collective communication function is lower than the aggregate latency of MPI_Send. For scenarios 3 and 4, the length where the change happens is denoted as changing length.

4.3 Observations

The observations on MPI_Bcast against MPI_Send are listed in Table 4.1. The latency of MPI_Bcast on PIII/100Base-T/4 nodes is higher than the aggregate latency of MPI_Send for all message lengths. That means using three callings to MPI_Send is going to take less time than using MPI_Bcast. Contrarily, the latency of MPI_Bcast on Quadcore/internal/4 nodes and Quadcore/Gigabit/4 nodes is both lower than aggregate latency of MPI_Send for all message lengths. That means it is worthless to replace MPI_Bcast with three MPI_Send callings in the program. For the case of P4/Gigabit/4 nodes, the latency of MPI_Bcast is originally higher than aggregate latency of MPI_Send for messages with shorter lengths. Then the latency of MPI_Bcast becomes lower than that of MPI_Send when message size is over 2048 bytes. Hence, it will reduce latency if use MPI_Send three times instead of MPI_Bcast when the message size is shorter than 2048 bytes. As for the case of PIII/100Base-T/8 nodes, it will reduce latency to use seven MPI_Send callings instead of MPI_Bcast when the message size is or over 16384 bytes. The same interpretations work for the observations for the other four MPI communication functions. Their observations are listed from Table 4.2. to Table 4.5.

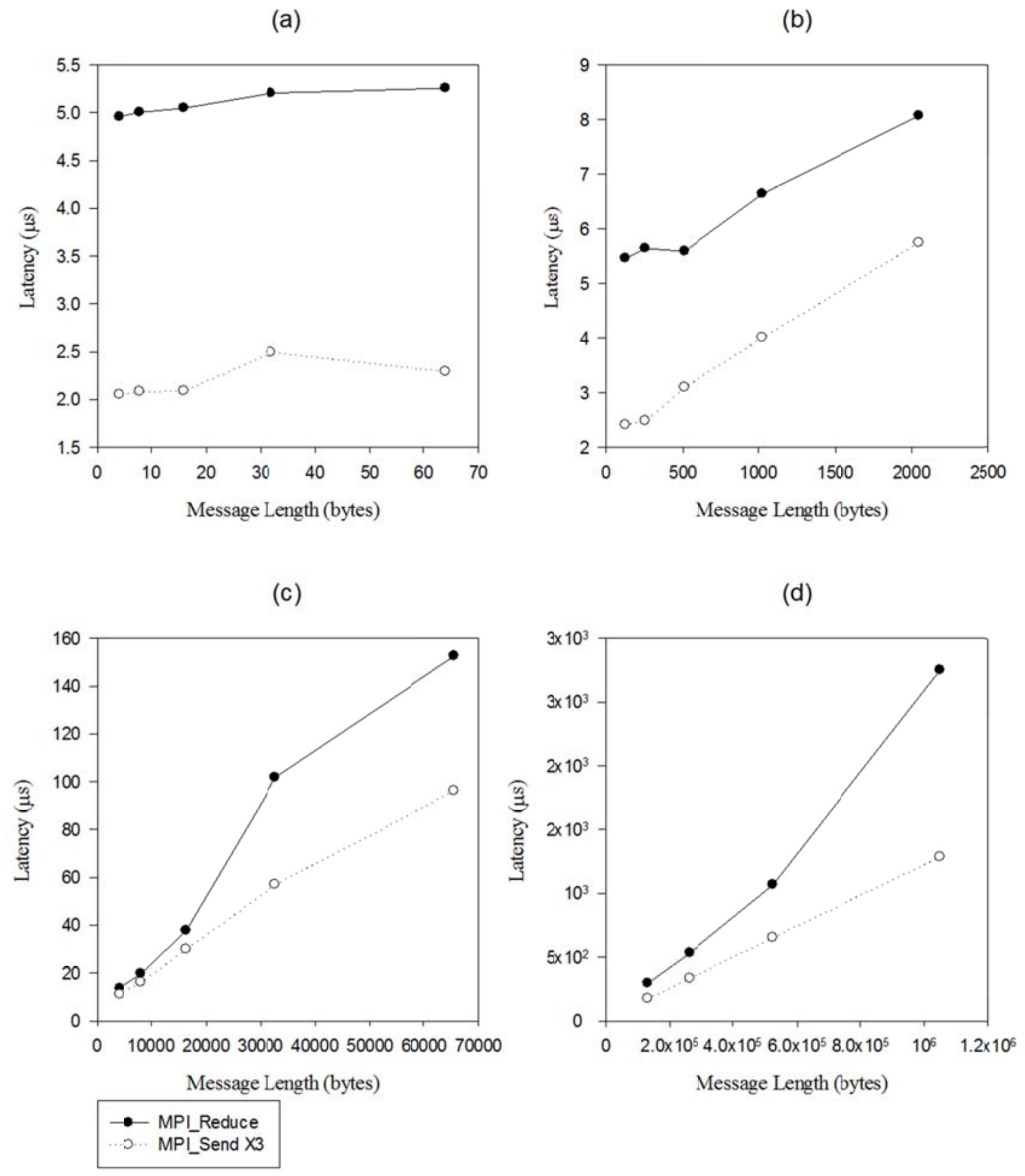

Figure 4.7. MPI_Reduce versus MPI_Send on Quadcore/Internal Channels/4 nodes

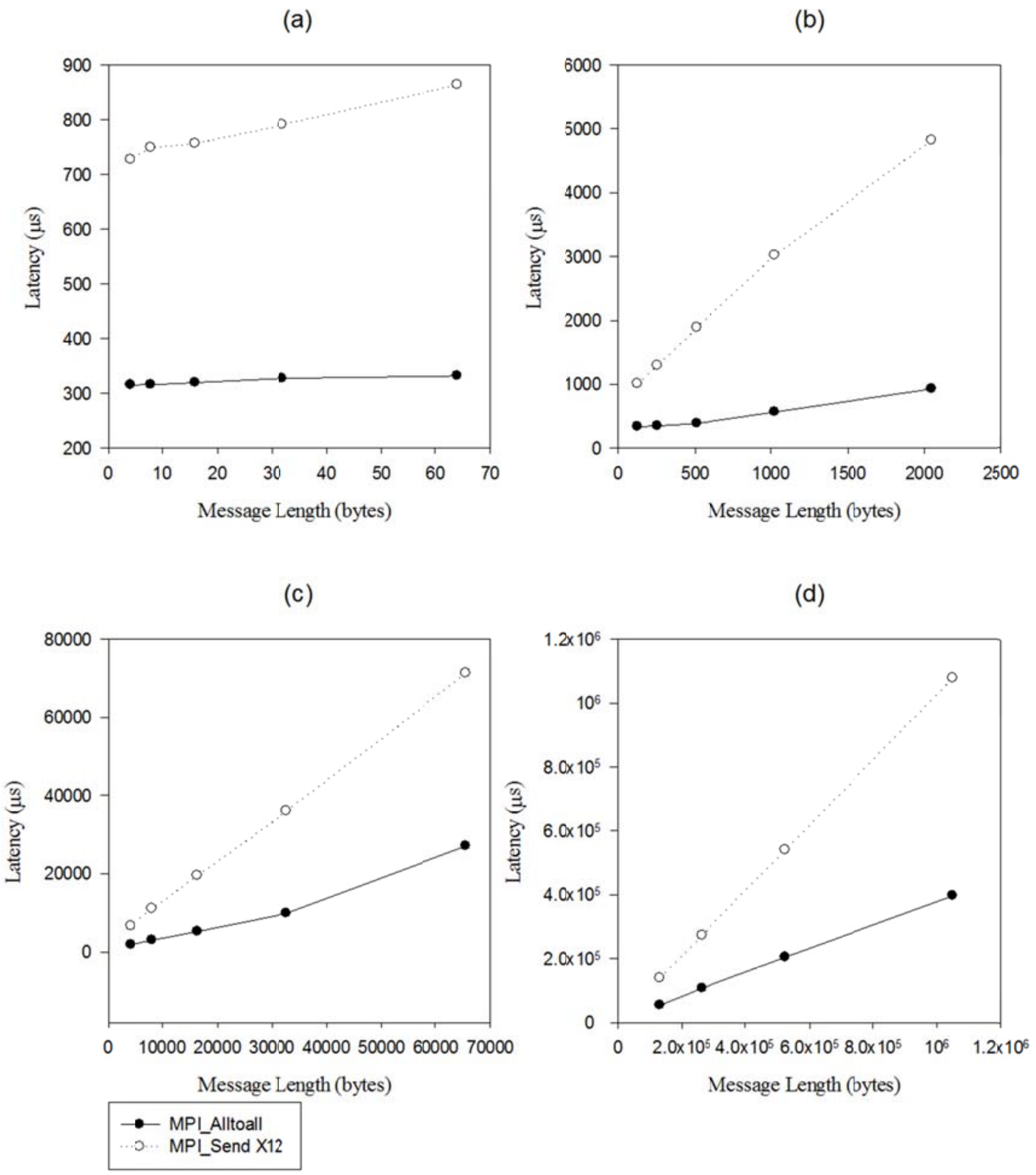

Figure 4.8. MPI_Alltoall versus MPI_Send on P4/100Base-T/4 nodes

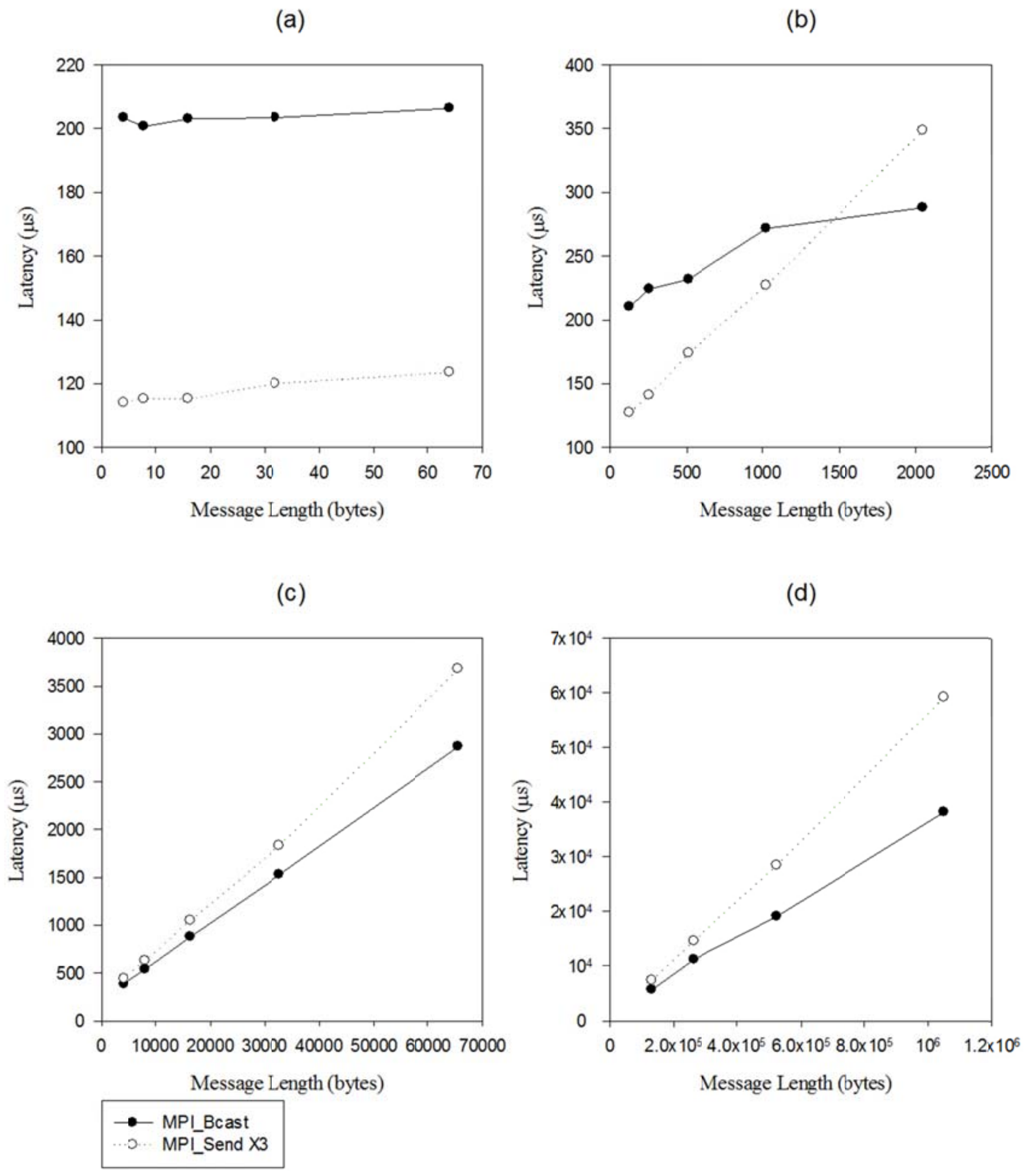

Figure 4.9. MPI_Bcast versus MPI_Send on P4/Gigabit/4 nodes

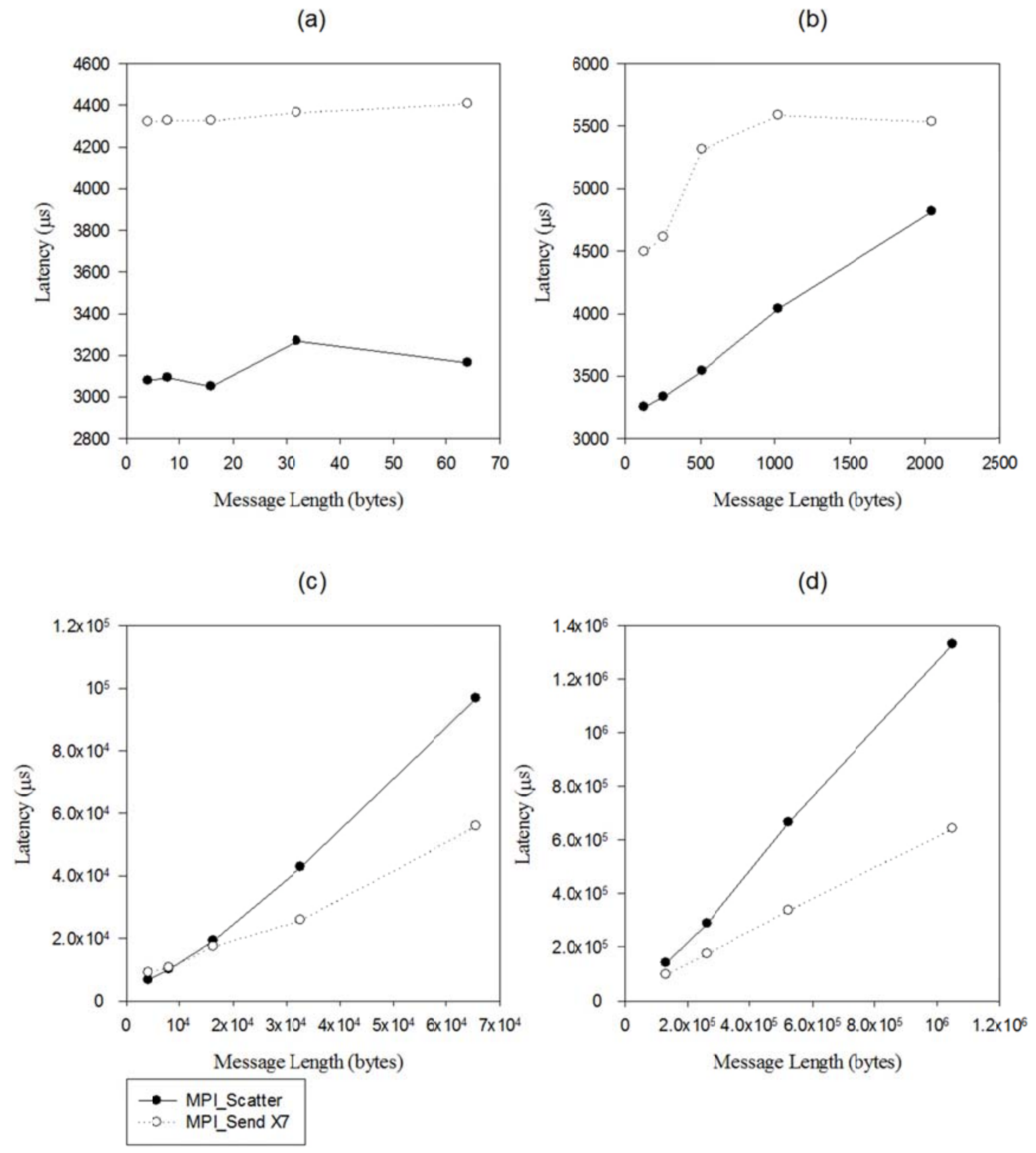

Figure 4.10. MPI_Scatter versus MPI_Send on PIII/100Base-T/8 nodes

Table 4.1. Observations on MPI_Bcast versus MPI_Send

Configuration	Short Message Latency	Long Message Latency	Changing Length (bytes)
Quadcore 2.5GHz/Internal Channels/4 nodes	L	L	
Quadcore 2.5GHz/Gigabit Ethernet/4 nodes	L	L	
P4 2.4GHz/Gigabit Ethernet/4 nodes	H	L	2048
P4 2.4GHz/100Base-T Ethernet /4 nodes	L	H	1024
PIII 700MHz/Gigabit Ethernet/4 nodes	H	L	262144
PIII 700MHz/100Base-T Ethernet/4 nodes	H	H	
PIII 700MHz/Gigabit Ethernet/8 nodes	H	L	128
PIII 700MHz/100Base-T Ethernet /8 nodes	L	H	16384

Table 4.2. Observation on MPI_Reduce versus MPI_Send

Configuration	Short Message Latency	Long Message Latency	Changing Length (bytes)
Quadcore 2.5GHz/Internal Channels/4 nodes	H	H	
Quadcore 2.5GHz/Gigabit Ethernet/4 nodes	L	L	
P4 2.4GHz/Gigabit Ethernet/4 nodes	H	L	2048
P4 2.4GHz/100Base-T Ethernet /4 nodes	H	L	1024
PIII 700MHz/Gigabit Ethernet/4 nodes	H	H	
PIII 700MHz/100Base-T Ethernet/4 nodes	H	H	
PIII 700MHz/Gigabit Ethernet/8 nodes	H	L	128
PIII 700MHz/100Base-T Ethernet /8 nodes	L	H	16384

Table 4.3. Observation on MPI_Gather versus MPI_Send

Configuration	Short Message Latency	Long Message Latency	Changing Length (bytes)
Quadcore 2.5GHz/Internal Channels/4 nodes	H	H	
Quadcore 2.5GHz/Gigabit Ethernet/4 nodes	L	L	
P4 2.4GHz/Gigabit Ethernet/4 nodes	H	L	262144
P4 2.4GHz/100Base-T Ethernet /4 nodes	L	H	65536
PIII 700MHz/Gigabit Ethernet/4 nodes	H	H	
PIII 700MHz/100Base-T Ethernet/4 nodes	H	H	
PIII 700MHz/Gigabit Ethernet/8 nodes	L	H	8192
PIII 700MHz/100Base-T Ethernet /8 nodes	L	H	8192

Table 4.4. Observation on MPI_Scatter versus MPI_Send

Configuration	Short Message Latency	Long Message Latency	Changing Length (bytes)
Quadcore 2.5GHz/Internal Channels/4 nodes	H	H	
Quadcore 2.5GHz/Gigabit Ethernet/4 nodes	L	L	
P4 2.4GHz/Gigabit Ethernet/4 nodes	H	H	
P4 2.4GHz/100Base-T Ethernet /4 nodes	H	H	
PIII 700MHz/Gigabit Ethernet/4 nodes	H	H	
PIII 700MHz/100Base-T Ethernet/4 nodes	L	H	1024
PIII 700MHz/Gigabit Ethernet/8 nodes	L	L	
PIII 700MHz/100Base-T Ethernet /8 nodes	L	H	16384

Table 4.5. Observation on MPI_Alltoall versus MPI_Send

Configuration	Short Message Latency	Long Message Latency	Changing Length (bytes)
Quadcore 2.5GHz/Internal Channels/4 nodes	L	H	131072
Quadcore 2.5GHz/Gigabit Ethernet/4 nodes	L	H	1048576
P4 2.4GHz/Gigabit Ethernet/4 nodes	L	L	
P4 2.4GHz/100Base-T Ethernet /4 nodes	L	L	
PIII 700MHz/Gigabit Ethernet/4 nodes	L	L	
PIII 700MHz/100Base-T Ethernet/4 nodes	L	H	8192
PIII 700MHz/Gigabit Ethernet/8 nodes	L	L	
PIII 700MHz/100Base-T Ethernet /8 nodes	L	H	16384

4.4 Methods for Optimal Utilization of Inter-Process Communications

Based on the previous observations and performance analysis, the following methods are recommended for optimal utilization of inter-process communications on commodity clusters:

A. **Use faster CPUs** – CPUs with faster clock speeds can execute MPI communication functions faster than CPUs with slower clock speeds, but the speedup is not to the ratio of their clock speeds.

B. **Adopt multicore CPUs to utilize the internal communication channels** – The main benefit of using multicore CPUs is the extremely low overhead of communication through internal communication channels on the chip.

C. **Employ Gigabit networking** – Upgrading 100Base-T networking to Gigabit networking significantly improves the communication performance. The communication performance of the cluster equipped with PIII and Gigabit network is better than the cluster equipped with Pentium 4 and 100Base-T networking.

D. **Run tests of different MPI communication command against MPI_Send** – As the results presented in previous sections show, the latency of all MPI communication functions are not always lower than that of aggregate latency of equivalent MPI_Send on commodity clusters. However, the characteristics are not uniform. Therefore, it recommended that researchers run benchmark programs to determine which MPI communication functions perform better than aggregate MPI_Send under what conditions on targeted commodity cluster. Then apply MPI_Send for conditions when MPI communication functions do not perform better.

CHAPTER 5

CONCLUSIONS

Classical science is based on observation, theory, and physical experimentation. In contrast, with the utilization of computer hardware and software, contemporary science is characterized by observation, theory, experimentation, and numerical simulation. With the advancement of computer hardware and software, numerical simulation becomes an important tool for researchers. Nonetheless, complicated numerical simulations rely on HPC. In the past, HPCs were very expensive and not affordable by most researchers and institutions.

Due to the simplicity of message-passing platform and the porting of MPI library to PC platform by open-source software developers, commodity clusters can be built out of inexpensive PCs. They provide an effective tool for all researchers to take advantage of numerical simulations. Although a lot of performance analyses have been conducted HPCs, they are all on high-end HPCs. No performance analysis has been conducted on commodity clusters before. In order to take the most out of commodity clusters, performance evaluations should be performed on them, especially when commodity clusters utilize generic components which exhibit different characteristics than those of high performance components employed by supercomputers.

Experiments which executed six major MPI communication functions were performed on eight different configurations of clusters. Performance analyses were conducted on results from those experiments. It was observed that performance of networking system still plays the key role. A cluster equipped with PIII CPUs and Gigabit NICs outperforms a cluster consisting of Pentium 4 CPUs and 100Base-T NICs. Gigabit NICs are very cheap to acquire nowadays. The cost/performance ratio of upgrading to Gigabit networking is well worth of it. It was also observed that collective communication functions may not perform better than utilizing multiple MPI_Send for certain configuration of cluster with certain size of message. Recommendations for optimal utilization of inter-process communications based on observations on experiment results were proposed.

REFERENCES

[1] M. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.

[2] G. Karniadakis and R. Kirby, *Parallel Scientific Computing in C++ and MPI*,Cambridge University Press, 2003.

[3] P. Pacheo , *Parallel Programming with MPI*, Morgan-Kaufmann, 1996.

[4] A. Grama, A. Gupta, G. Karypis and V. Kumar, *Introduction to Parallel Computing*, Addison-Wesley, 2003.

[5] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, and Programmability*, McGraw Hill, New York, 1993.

[6] Open MPI: Frequently Asked Questions. http://www.openmpi.org/faq.

[7] W. Gropp, E. Lusk and A. Skjellum, *Using MPI*, MIT Press, 1999.

[8] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2*, MIT Press, 1999.

[9] J. Sloan, *High Performance Linux Clusters*, O'Reilly, 2005.

[10] C. Bookman, *Linux Clustering*, New Riders, 2003.

[11] A. Vrenios, *Linux Cluster Architecture*, Sams, 2002.

[12] R. Lucke, *Building Clustered Linux Systems*, Prentice Hall, 2005.

[13] Y. Tseng. and C. Hargrove, "Construction of Parallel Machines for Engineering Courses with Retired Personal Computers," ASEE Southeastern Section Annual Conference, 2008.

[14] Z. Xu and K Hwang, "Modeling Communication Overhead: MPI and MPLPerformance on the IBM SP2,"*IEEE parallel & distributed Technology,* Vol. 4, No.1, Spring 1996, pp. 9-23.

[15] J. Miguel and A. Arruabarrena, "Assessing the Performance of the New IBM SP2 Communication Subsystem," *IEEE Parallel & Distributed technology.*1996.

[16] C. Muelder, F. Gygi, and K. Ma, “Visual Analysis of Inter-Process Communication for Large-Scale Parallel Computing,” *IEEE Transactions on Visualization and Computer Graphics,* Vol. 15, NO. 6, Nov/Dec 2009.

[17] R.Graham, R. Brightwell, B. Barrett, Ge. Bosilca and J. Grbovic, “An Evaluation of Open MPI's Matching Transport Layer on the Cray XT,” EuroPVM/MPI 2007, 2007.

[18] G. Luecke, J. Coyle, J. Hoekstra, and M. Kraeva, “Measuring MPI Latency and Communication Rates for Small Messages,” Shanghai Supercomputing Center, 2009.

[19] J. Grbovic, G. Fagg, T. Angskun, G. Bosilca and J. Dongarra, “MPI Collective Algorithm Selection and Quadtree Encoding,” Euro PVM/MPI 2006, 2006.

[20] G. Shipman, S. Poole, P. Shamis, and I. Rabinovitz, “X-SRQ - Improving Scalability and Performance of Multi-Core InfiniBand Clusters,” EuroPVM/MPI 2009, 2009.

[21] J.J Dongarra and T. Dunigan, “Message Passing Performance of Various Computers,” Tech. Report UT-CS-95-229, Computer Science Dept, Univ. of Tennessee,1995, available at http://www.cs.utk.edu/~library/1995.html.

[22] D. Doerfler and R. Brightwell, “Measuring MPI Send and Receive Overhead and Application Availability in High Performance Network Interfaces,” Euro PVM/MPI 2006, 2006.

[23] R. Graham, S. Timothy and M. Jeffrey, “Open MPI: A Flexible High Performance MPI,”6th International Conference on Parallel Processing and Applied Mathematics in Poznan, 2005.

[24] G. Shipman, T. Woodall, R. Graham, A. Maccabe and P. Bridges, “InfiniBand Scalability in Open MPI,” IEEE Parallel and Distributed Processing Symposium (IPDPS), 2006.

[25] T. Hoefler, P. Gottschling, W. Rehm and A. Lumsdaine, “Optimizing a Conjugate Gradient Solver with Non Blocking Collective Operations,” Euro PVM/MPI 2006, 2006.

[26] J. Hursey, T. Mattox and A. Lumsdaine, “Interconnect agnostic checkpoint/restart in Open MPI,” 18th ACM international symposium on High Performance Distributed Computing (HPDC 2009), 2009.

[27] M. Sobell, *A Practical Guide to Red Hat Linux,* Third Edition, Prentice-Hall, 2006.

[28] Linux NIS-HOWTO: http://www.linux-nis.org/nis-howto/.

[29] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham and G. Bosilca, “Analysis of the Component Architecture Overhead in Open MPI,” Euro PVM/MPI 2005, 2005.

[30] The Ohio State University Benchmarks: http://nowlab.cse.ohio-state.edu/projects/mpi-iba/benchmarks.html.

[31] Intel’s MPI Benchmarks 3.1: http://www.intel.com/cd/software/products/asmo-na/eng/cluster/cluster/clustertoolkit/219848.html.

[32] T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett and A. Lumsdaine, “A High-Performance, Scalable,Multi-Network Point-to-Point Communications Methodology,” Euro PVM/MPI 2004, 2004.

[33] J. Hursey, J Squyres, T. Mattox and A. Lumsdaine, “The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI,” International Parallel and Distributed Processing Symposium (IPDPS 2007), 2007.

APPENDIX

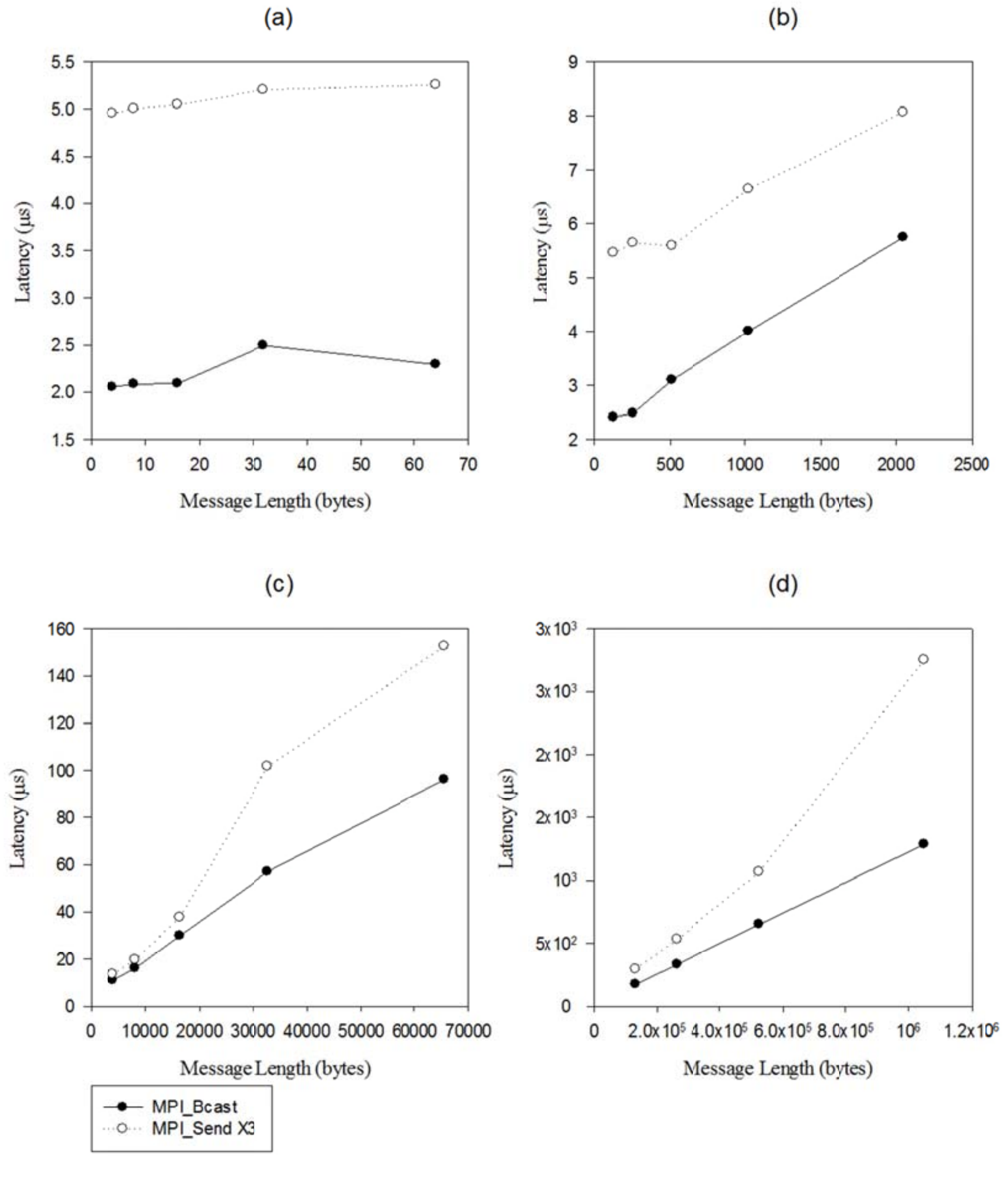

Figure A.1. MPI_Bcast versus MPI_Send on Quadcore/Internal Channels/4 nodes

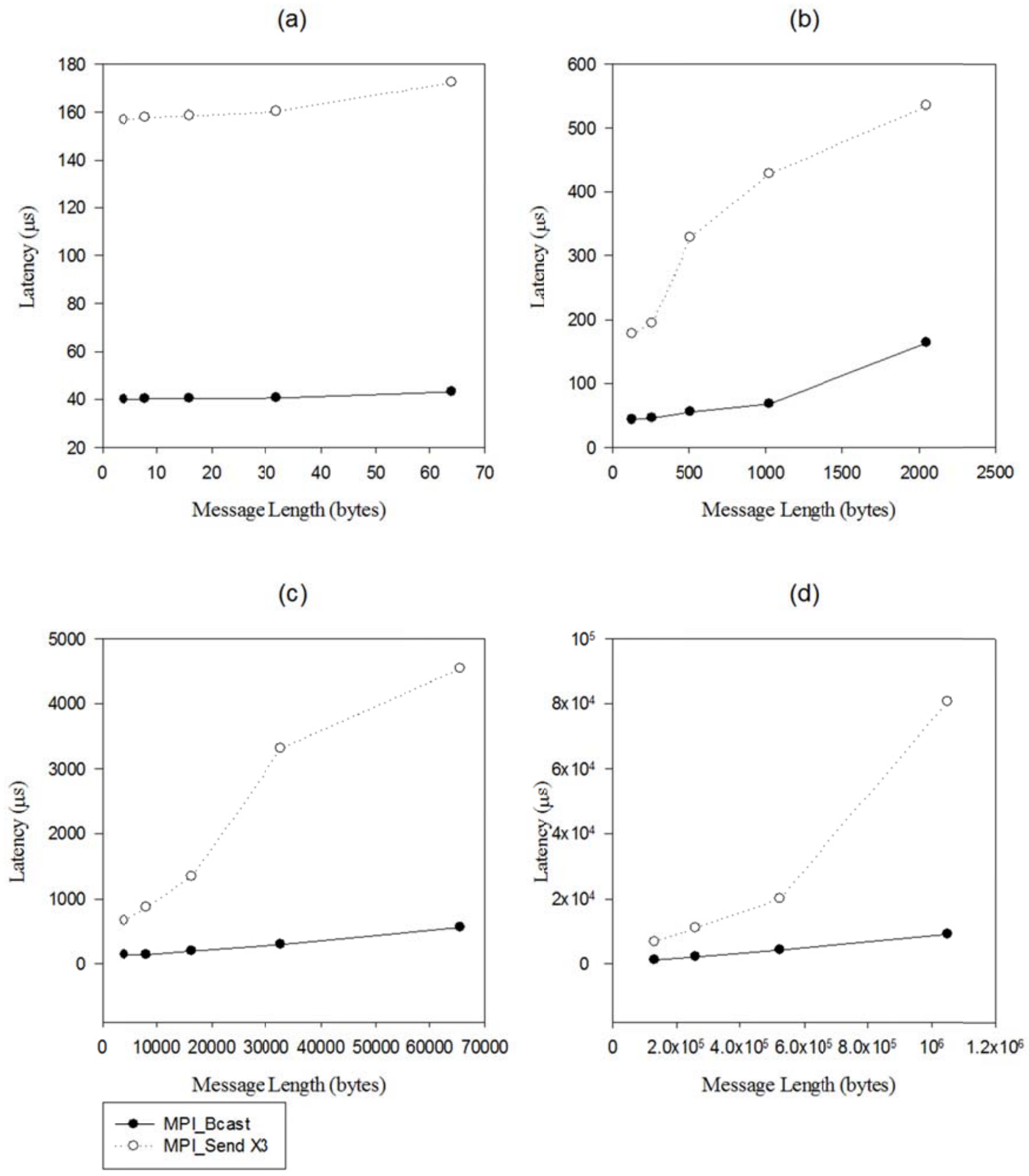

Figure A.2. MPI_Bcast versus MPI_Send on Quadcore/Gigabit/4 nodes

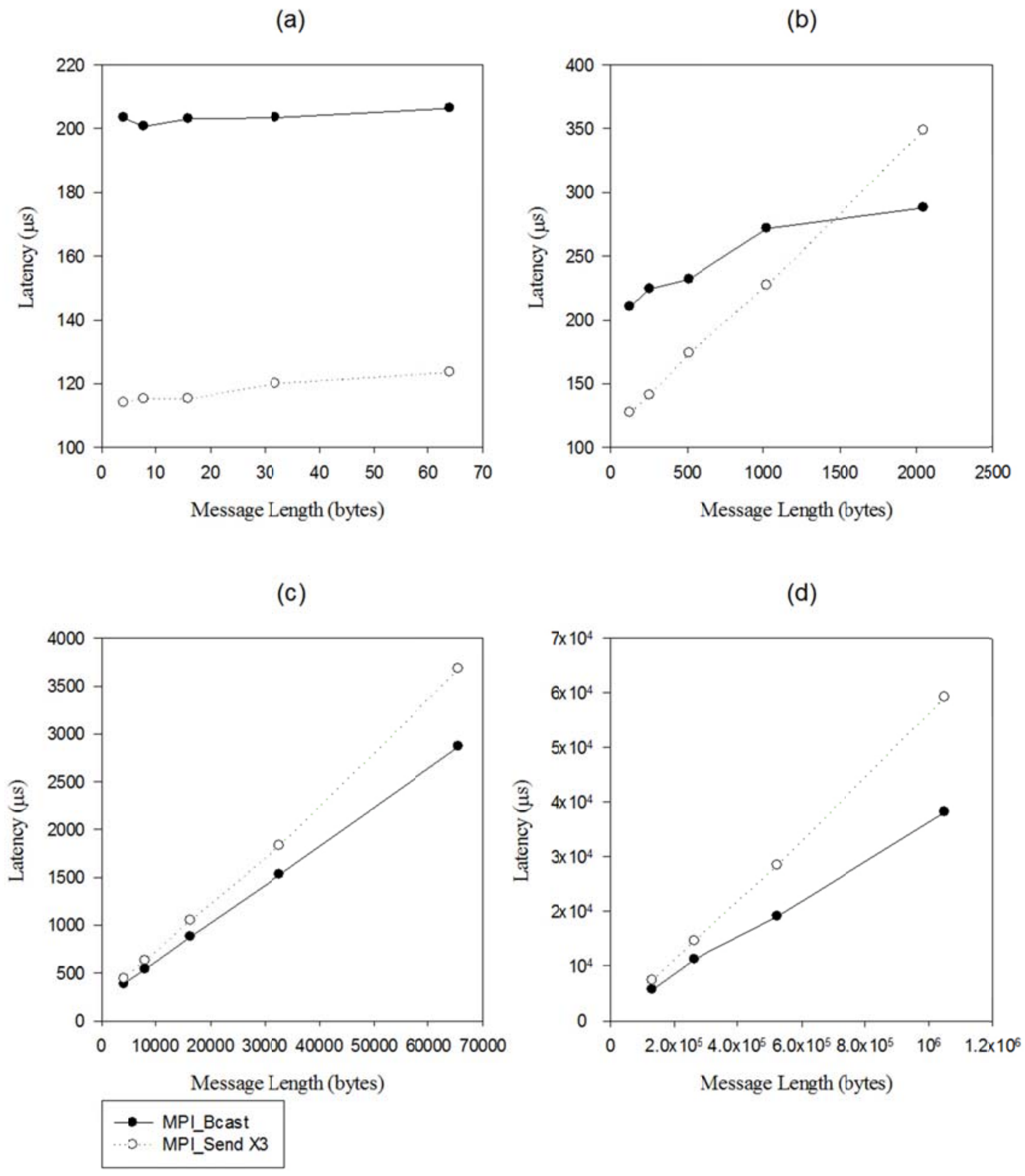

Figure A.3. MPI_Bcast versus MPI_Send on P4/Gigabit/4 nodes

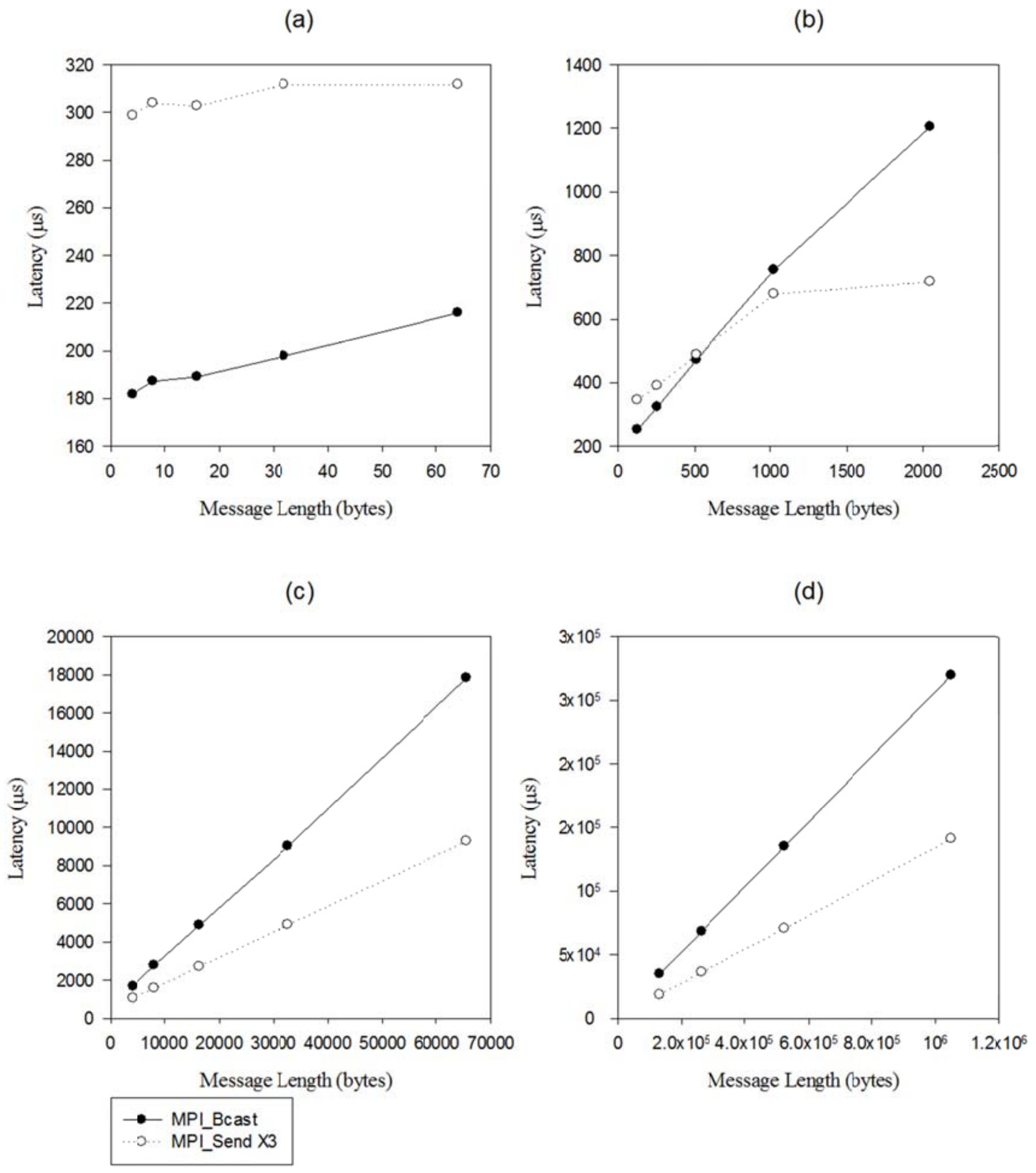

Figure A.4. MPI_Bcast versus MPI_Send on P4/100Base-T/4 nodes

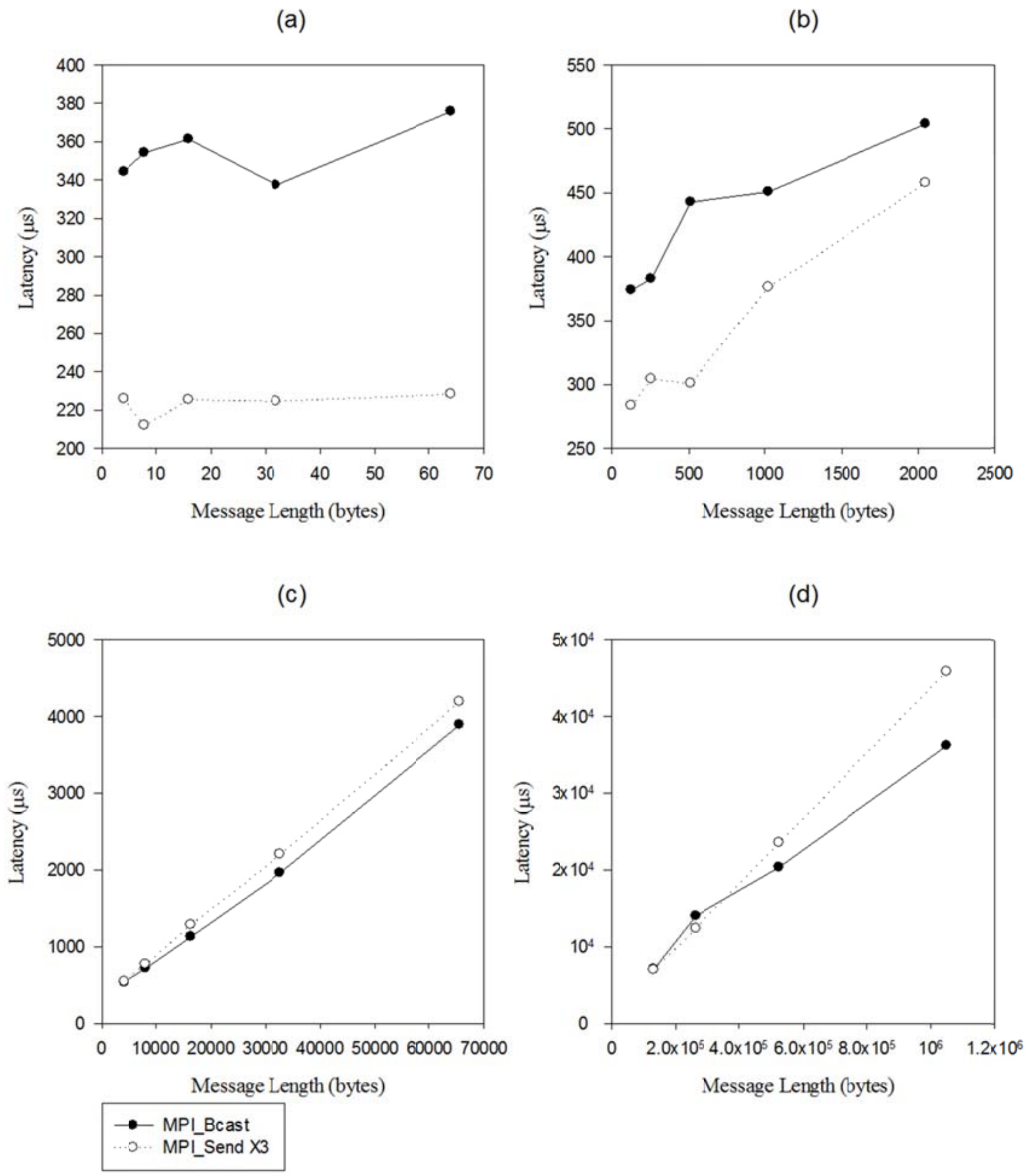

Figure A.5. MPI_Bcast versus MPI_Send on PIII/Gigabit/4 nodes

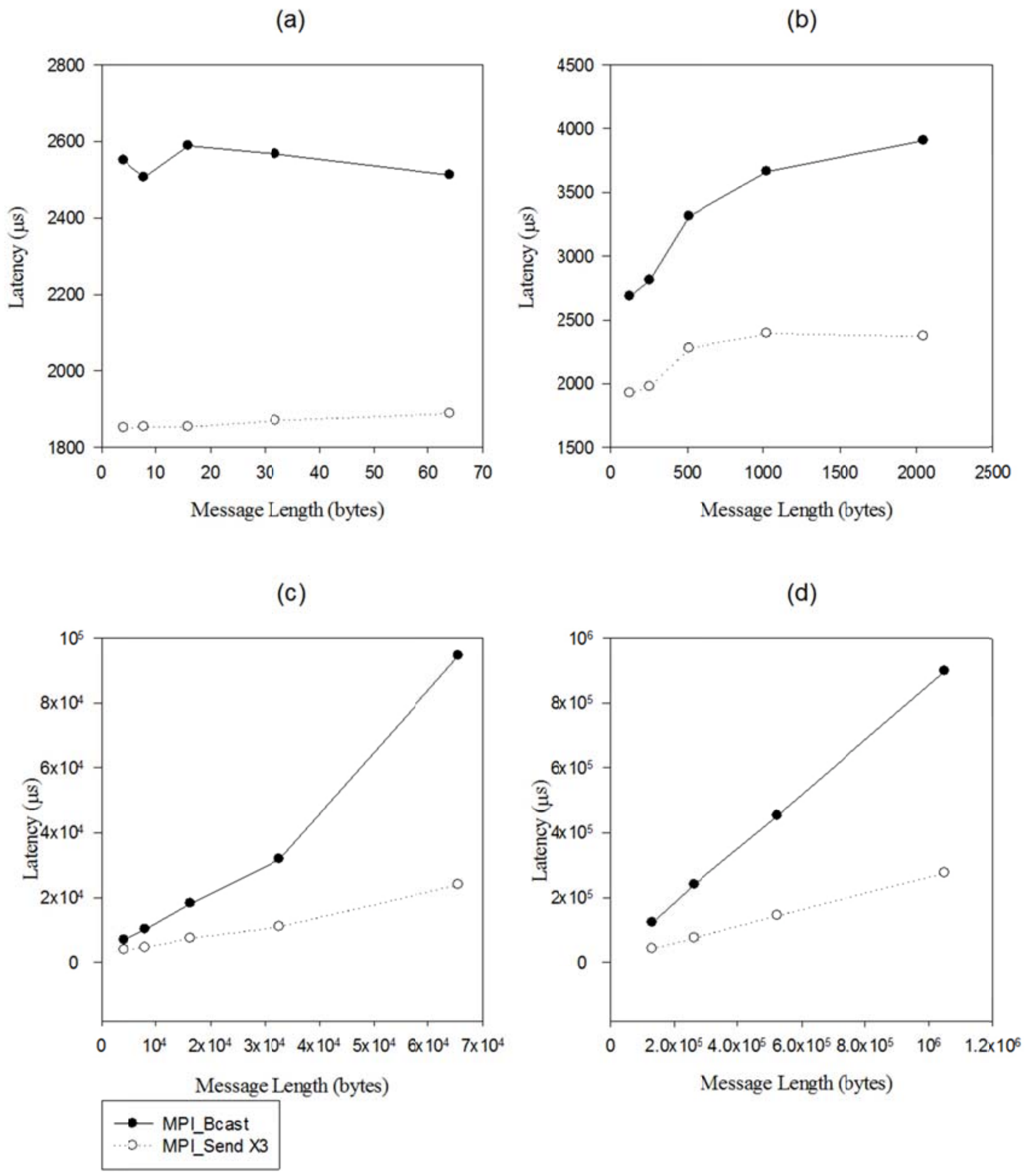

Figure A.6. MPI_Bcast versus MPI_Send on PIII/100Base-T/4 nodes

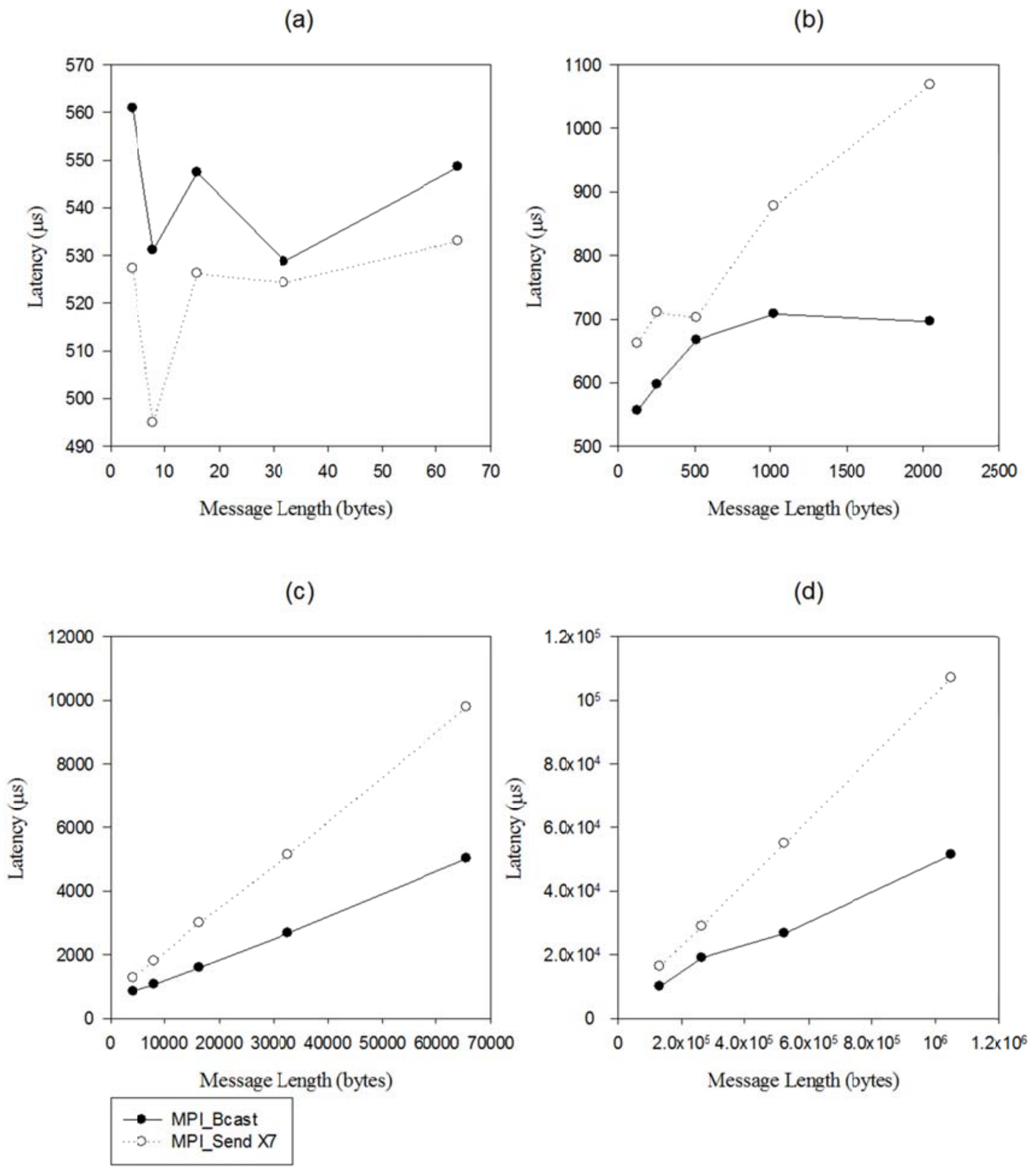

Figure A.7. MPI_Bcast versus MPI_Send on PIII/Gigabit/8 nodes

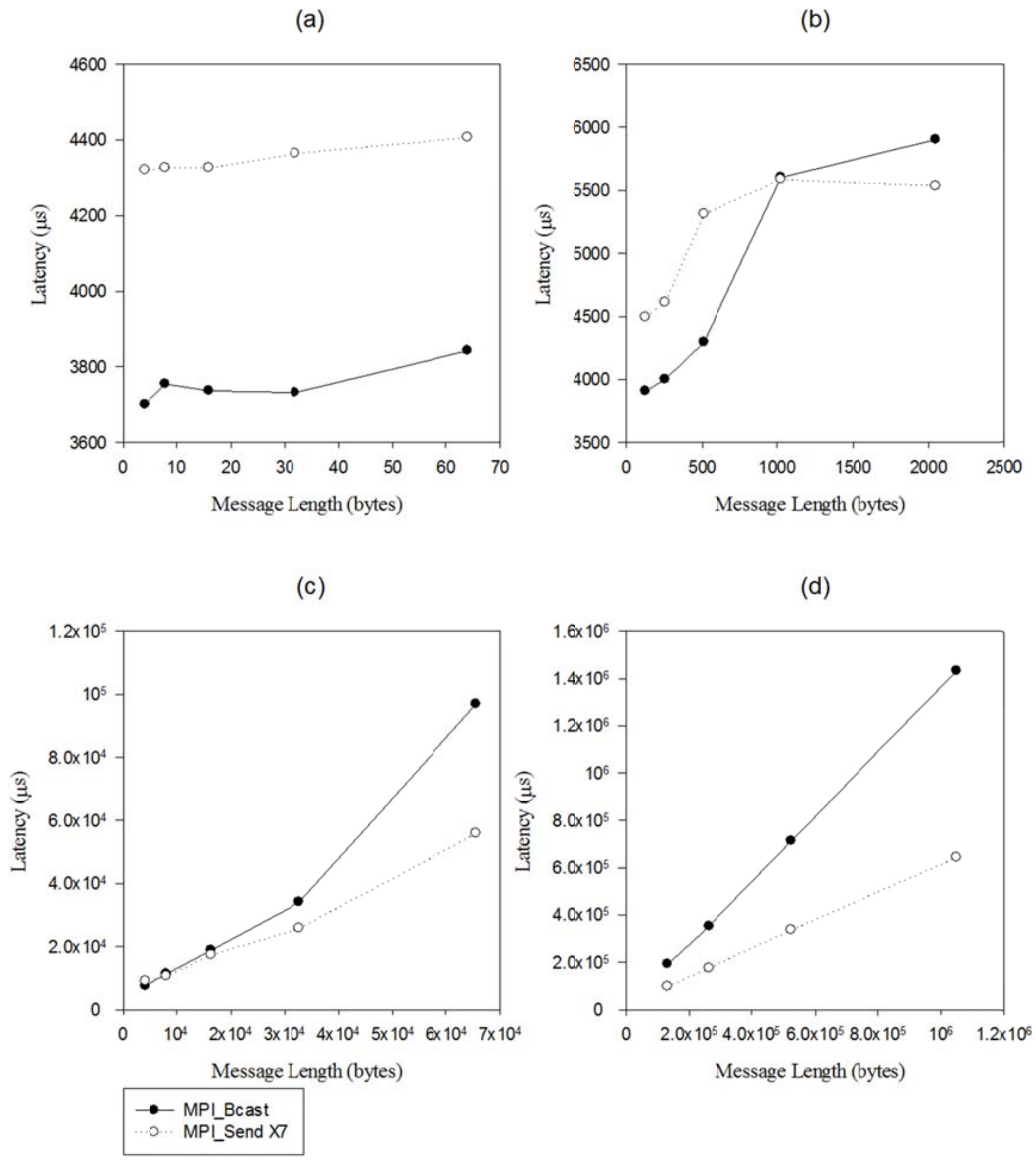

Figure A.8. MPI_Bcast versus MPI_Send on PIII/100Base-T/8 nodes

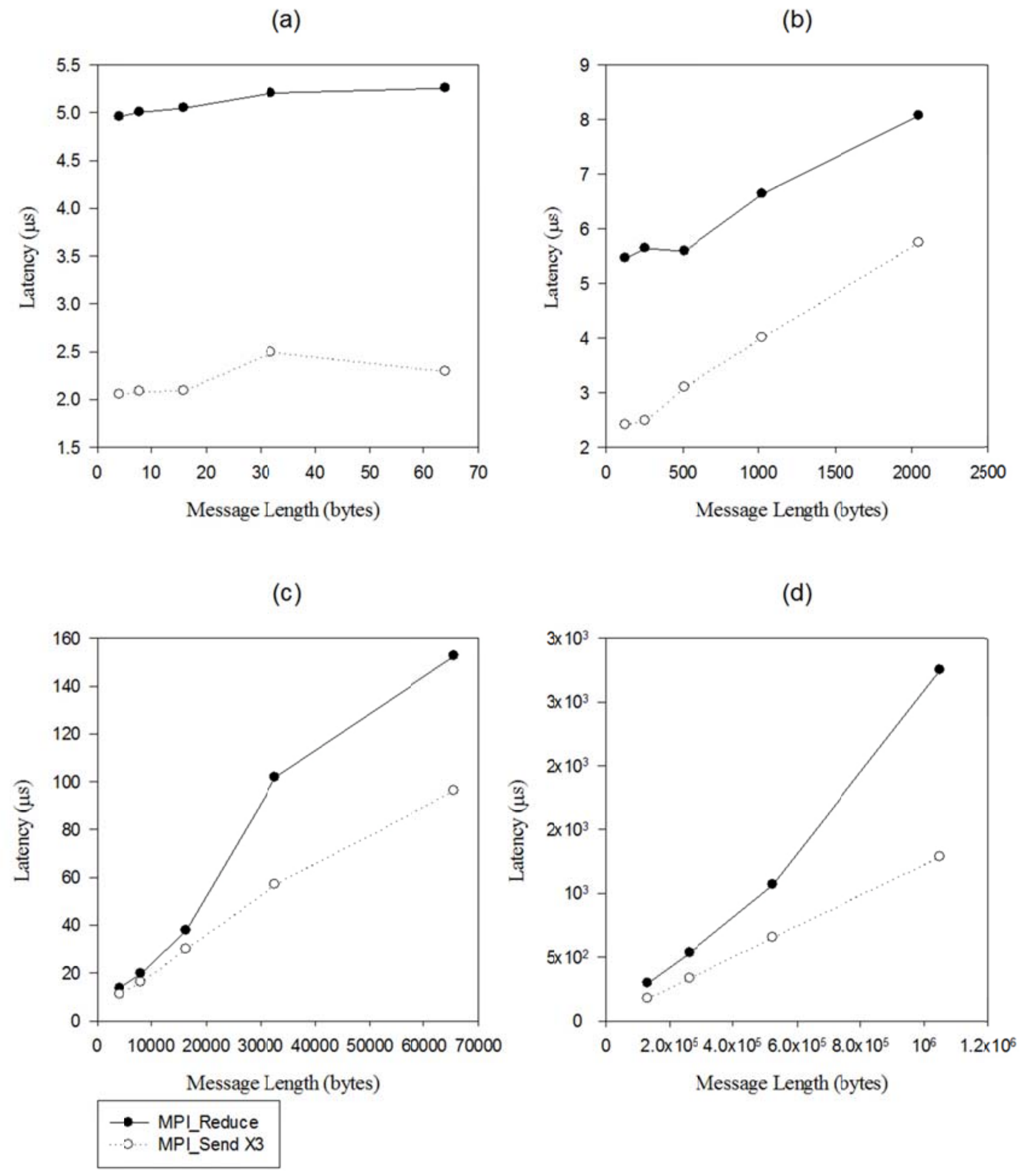

Figure A.9. MPI_Reduce versus MPI_Send on Quadcore/Internal Channels/4 nodes

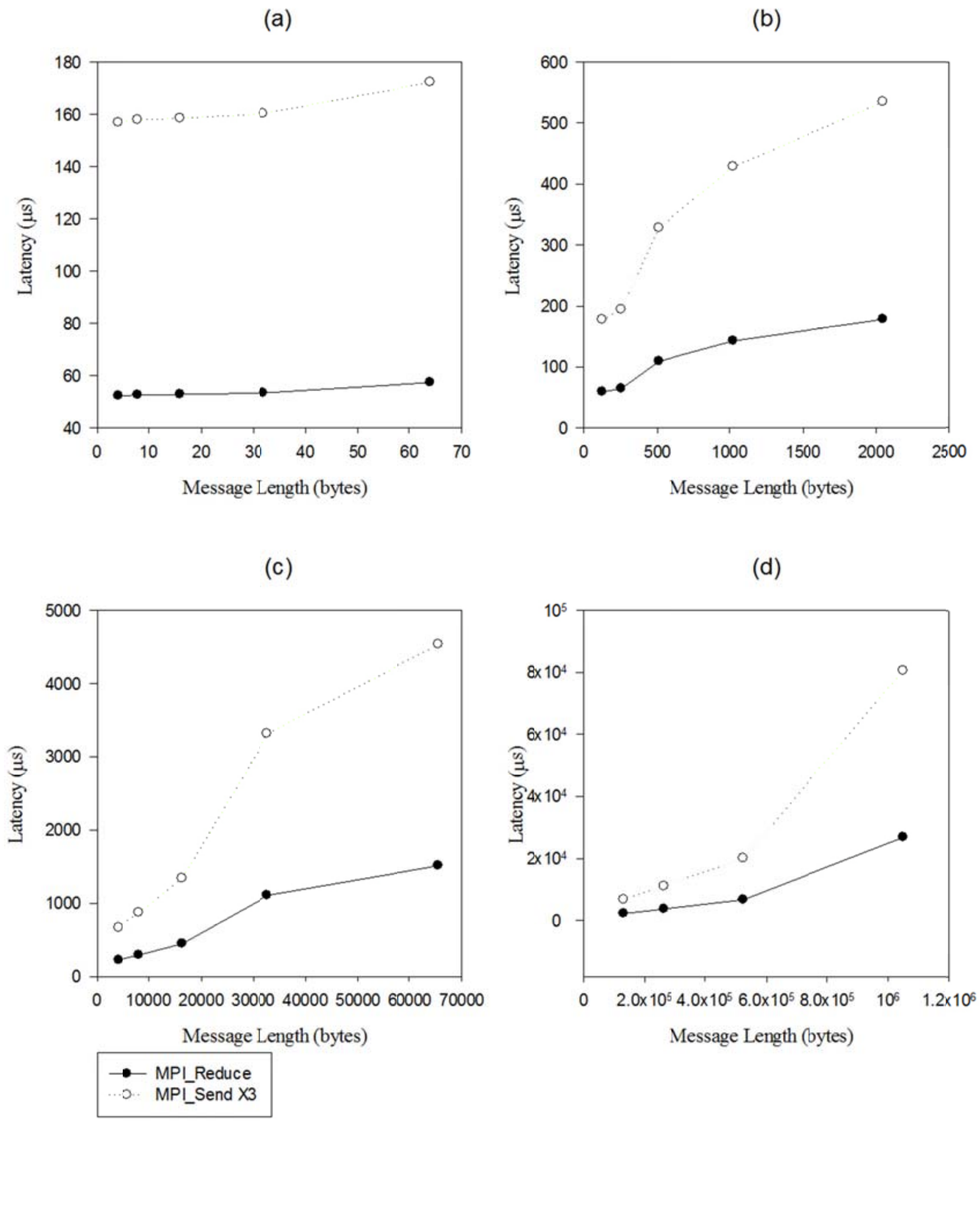

Figure A.10. MPI_Reduce versus MPI_Send on Quadcore/Gigabit/4 nodes

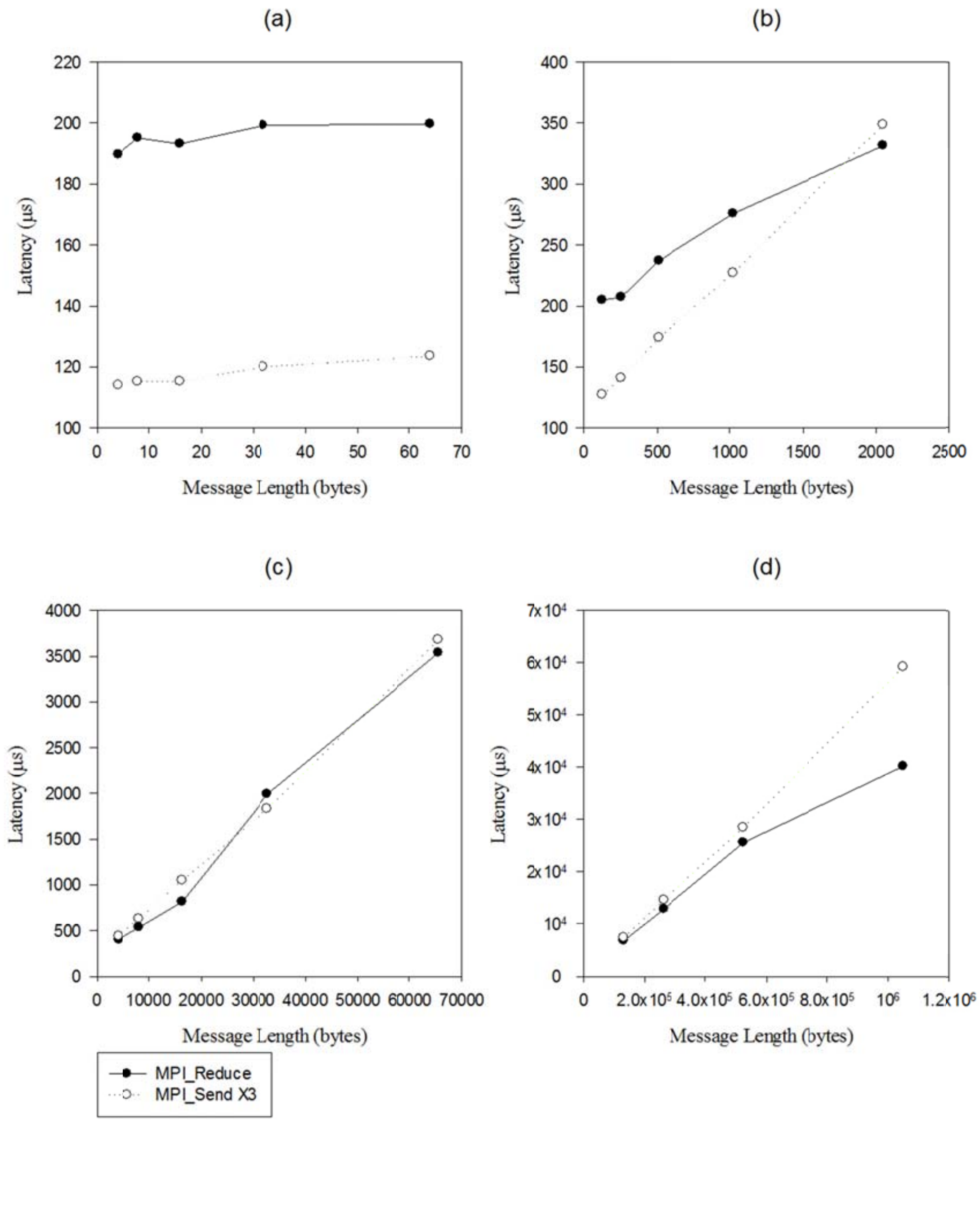

Figure A.11. MPI_Reduce versus MPI_Send on P4/Gigabit/4 nodes

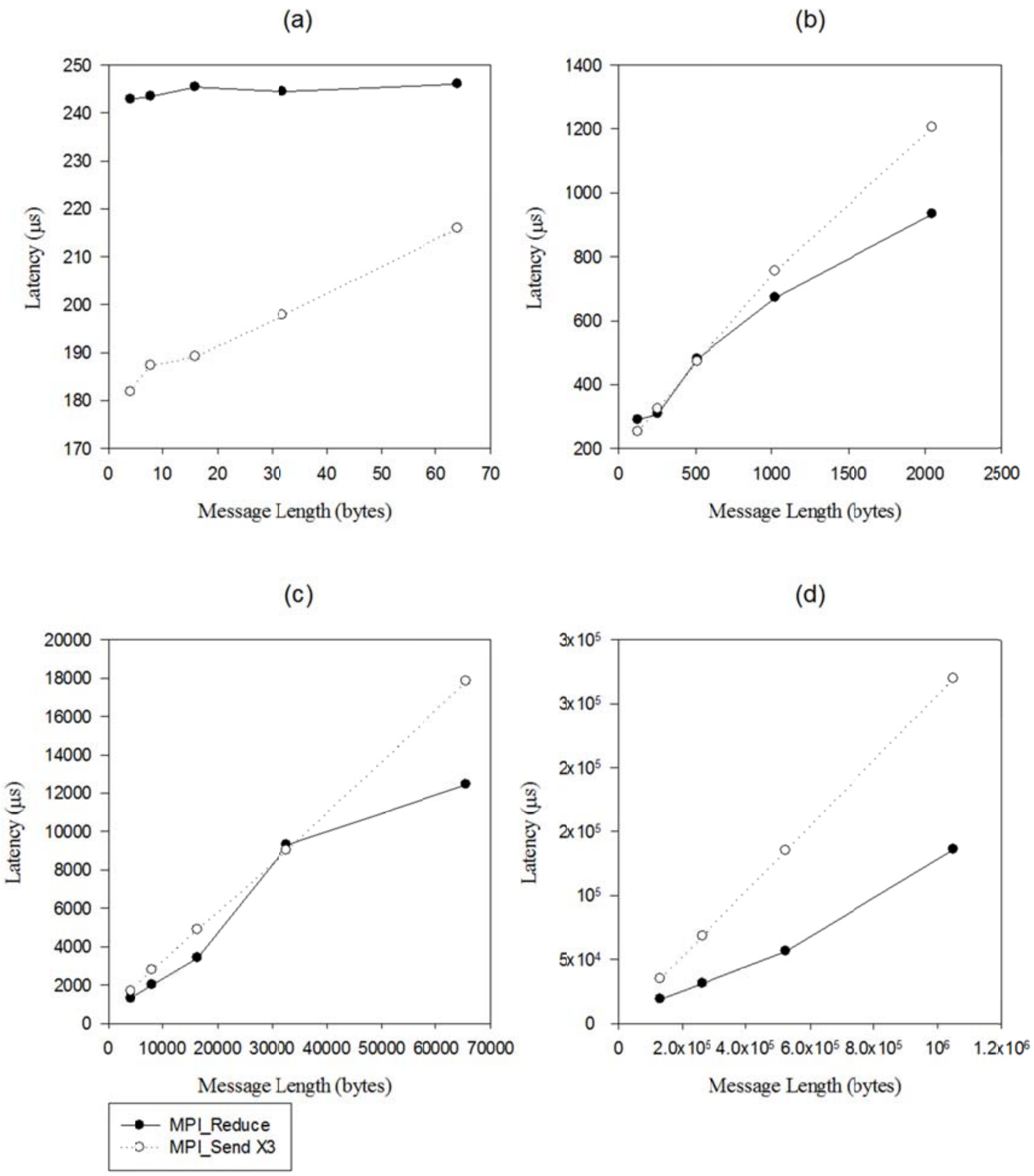

Figure A.12. MPI_Reduce versus MPI_Send on P4/100Base-T/4 nodes

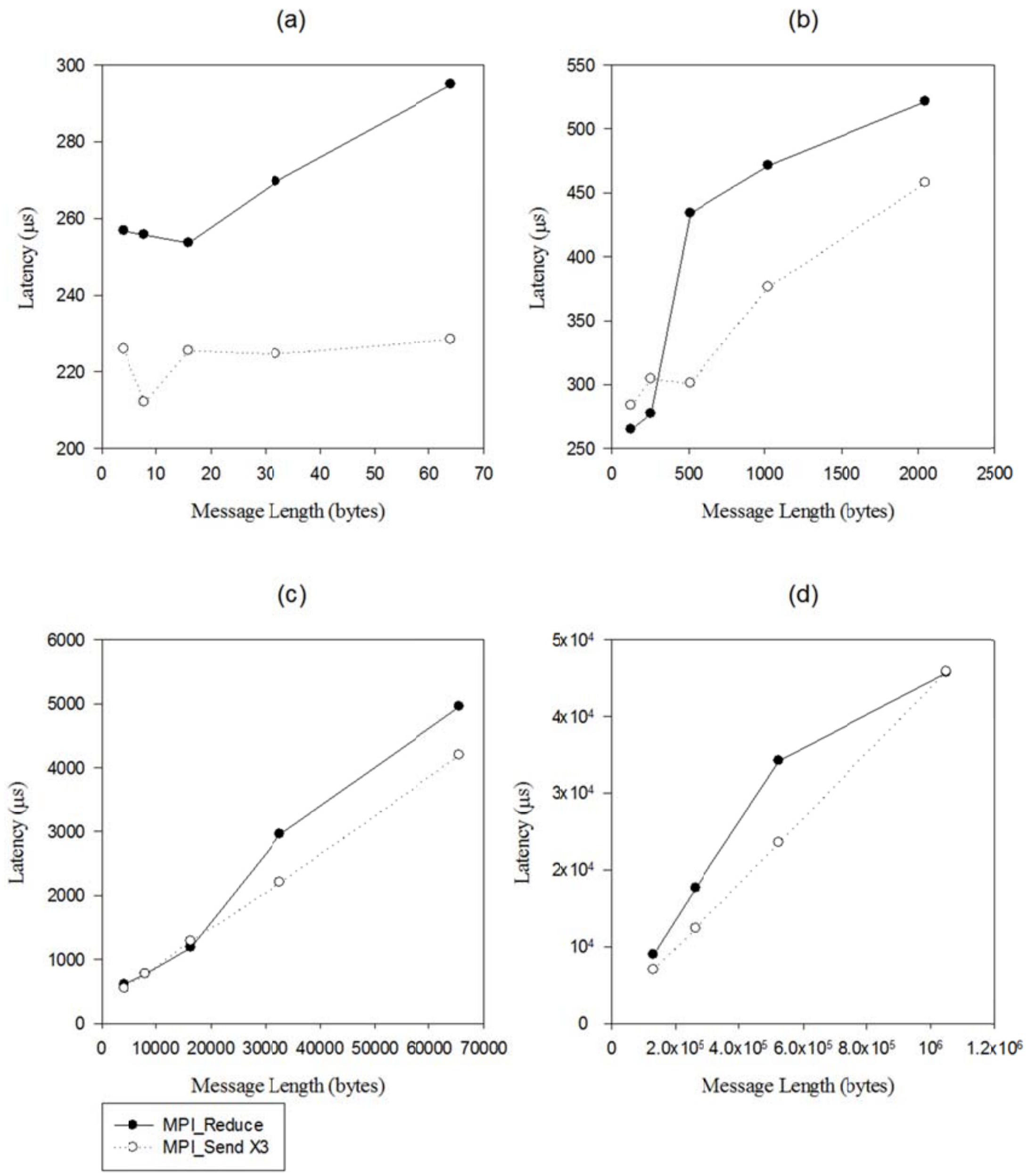

Figure A.13. MPI_Reduce versus MPI_Send on PIII/Gigabit/4 nodes

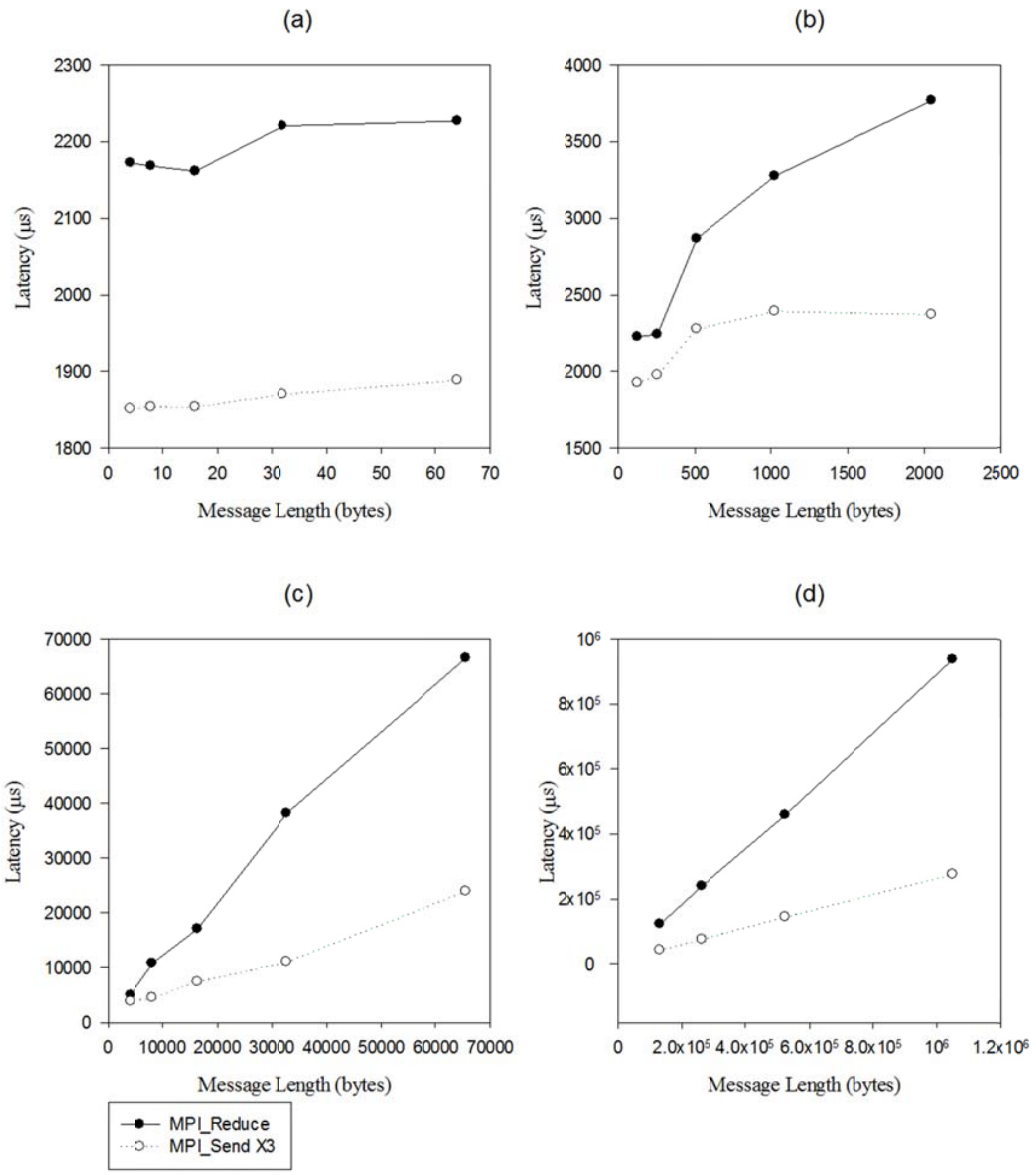

Figure A.14. MPI_Reduce versus MPI_Send on PIII/100Base-T/4 nodes

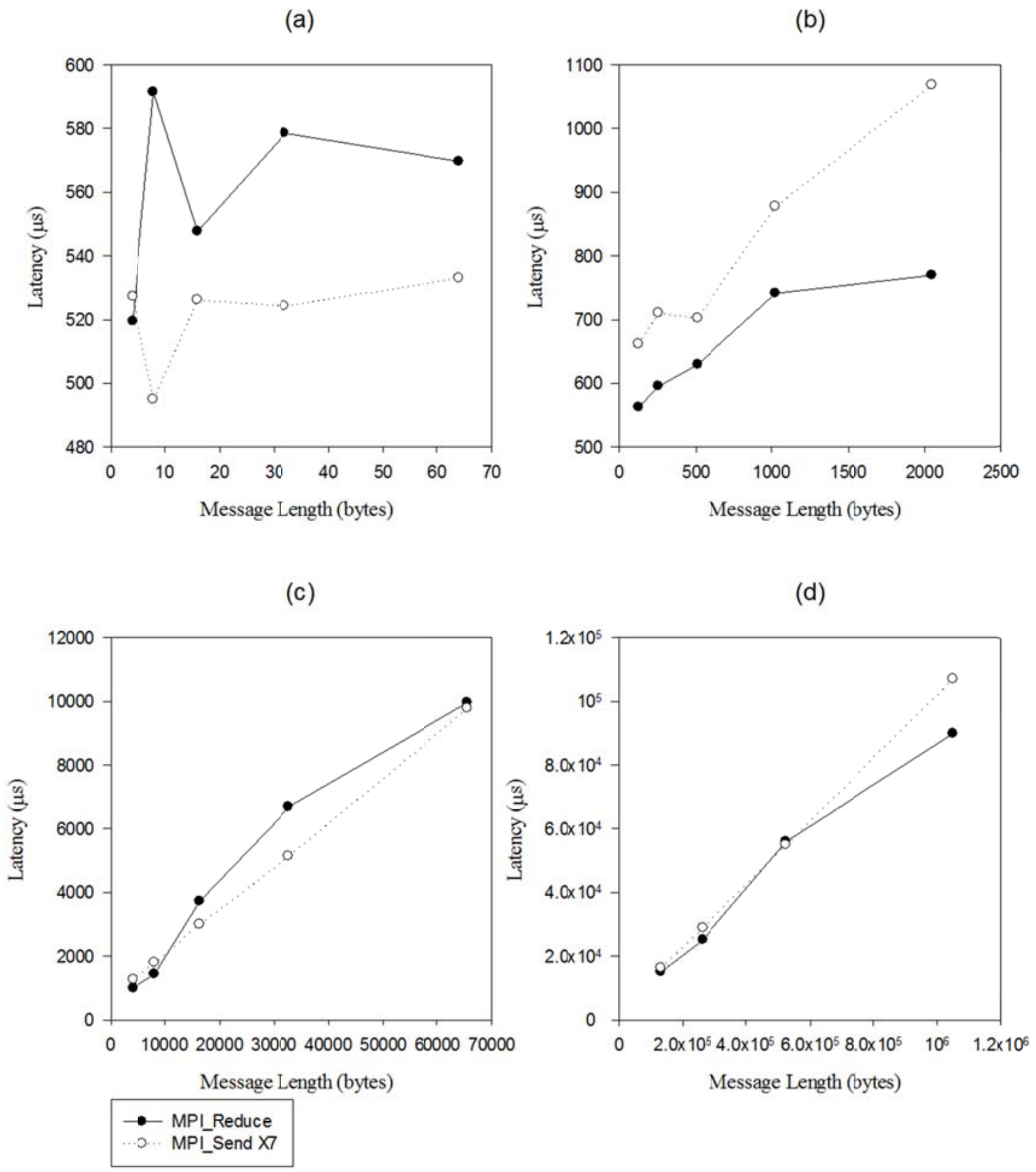

Figure A.15. MPI_Reduce versus MPI_Send on PIII/Gigabit/8 nodes

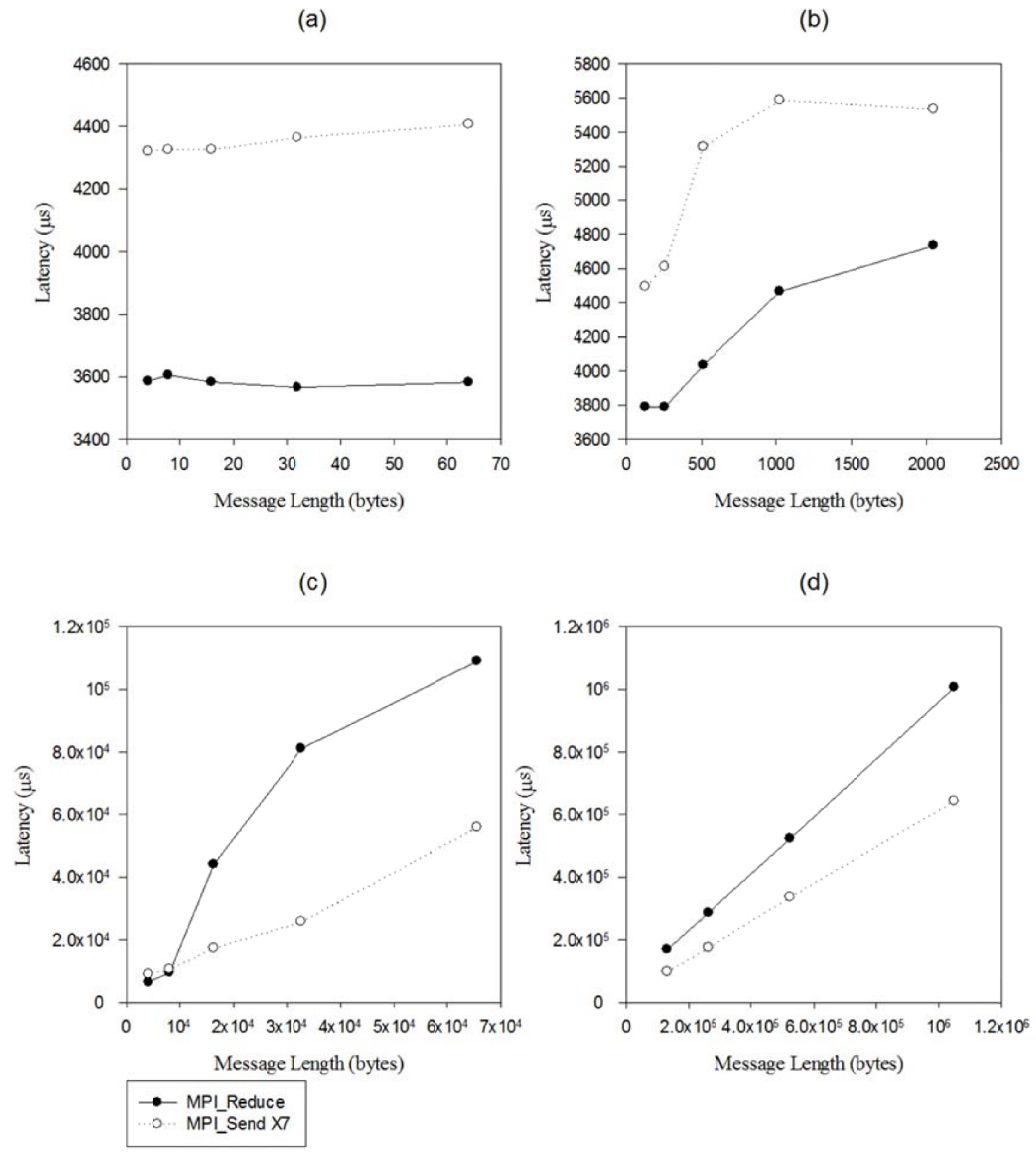

Figure A.16. MPI_Reduce versus MPI_Send on PIII/100Base-T/8 nodes

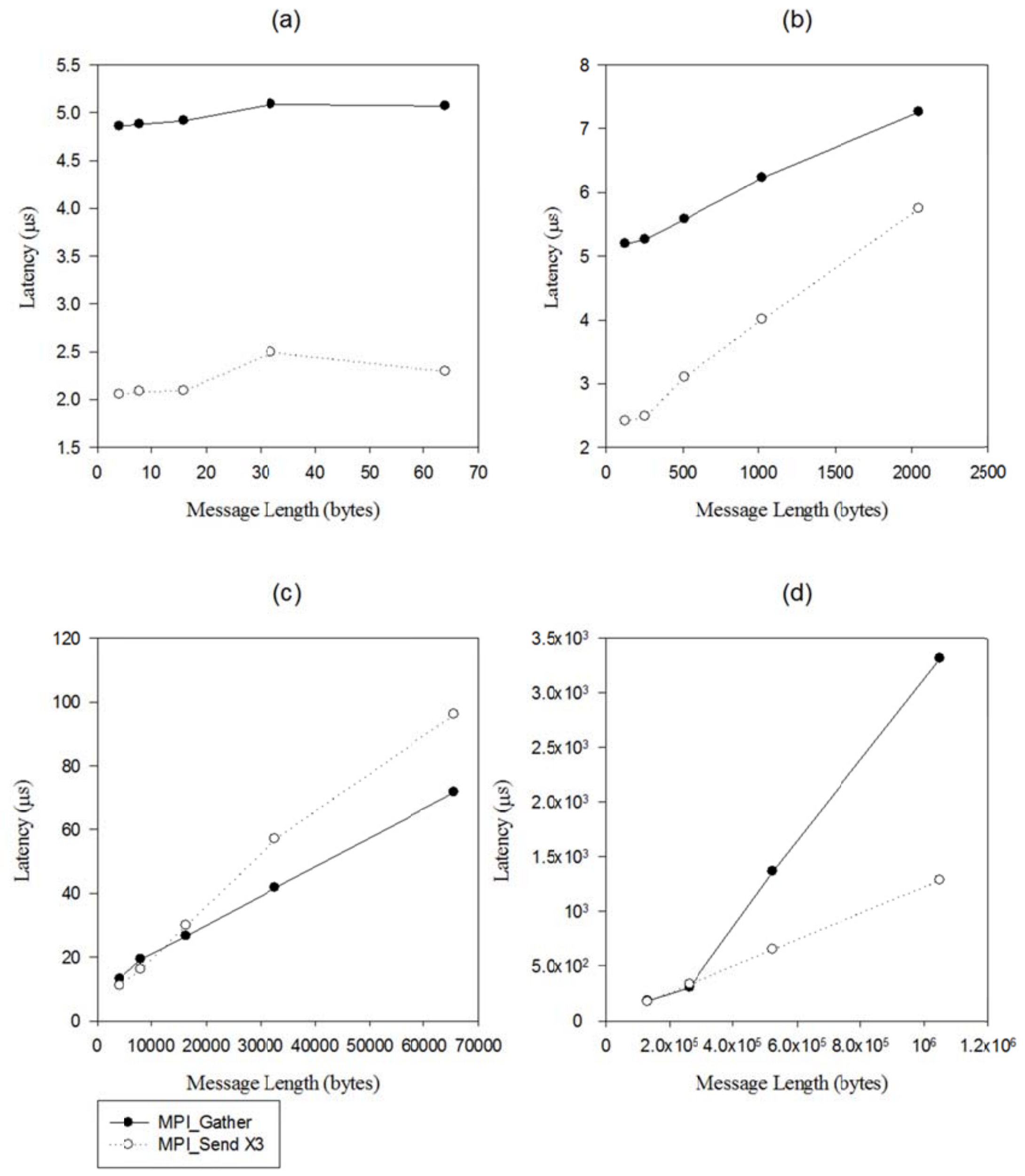

Figure A.17. MPI_Gather versus MPI_Send on Quadcore/Internal Channels/4 nodes

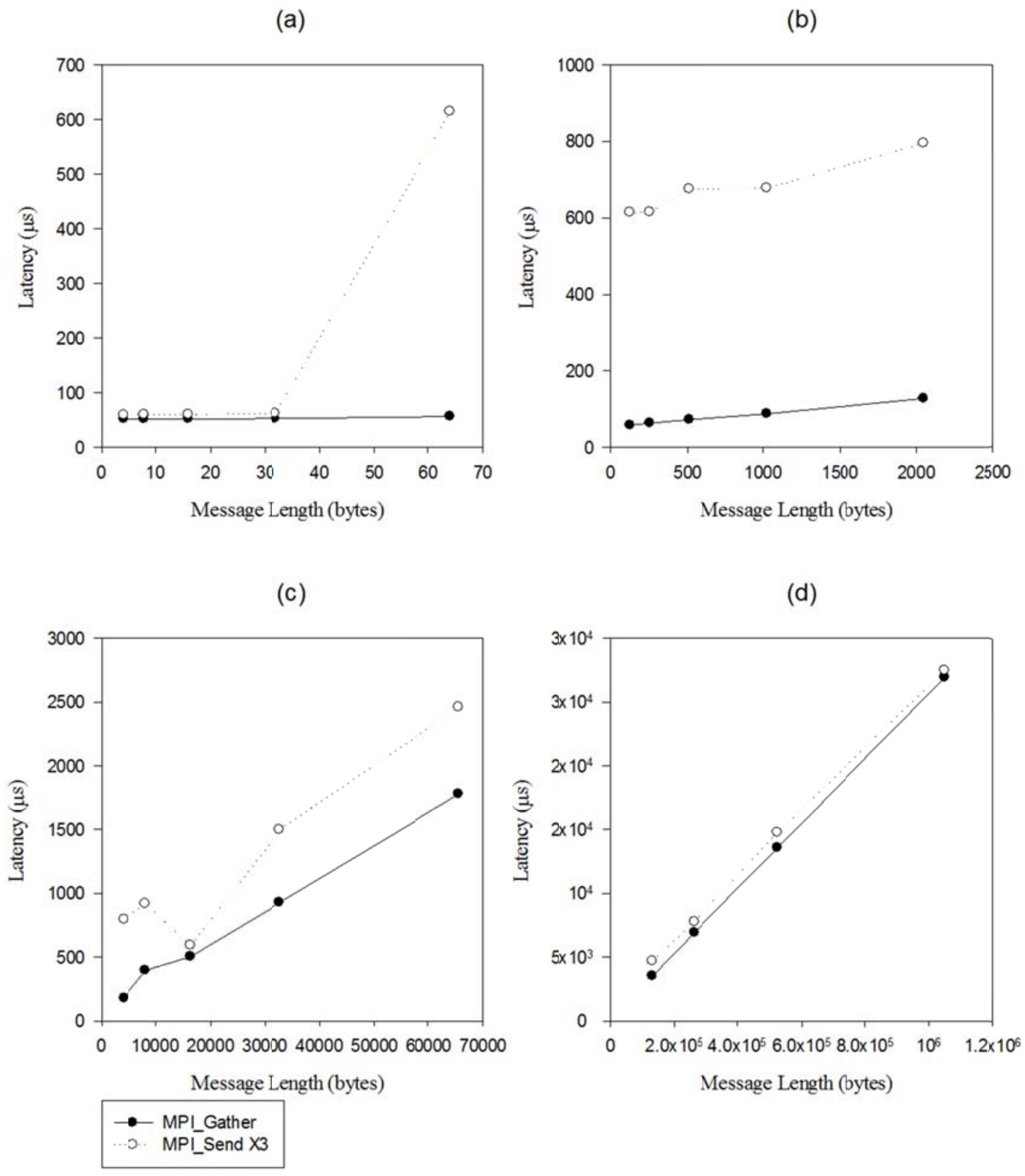

Figure A.18. MPI_Gather versus MPI_Send on Quadcore/Gigabit/4 nodes

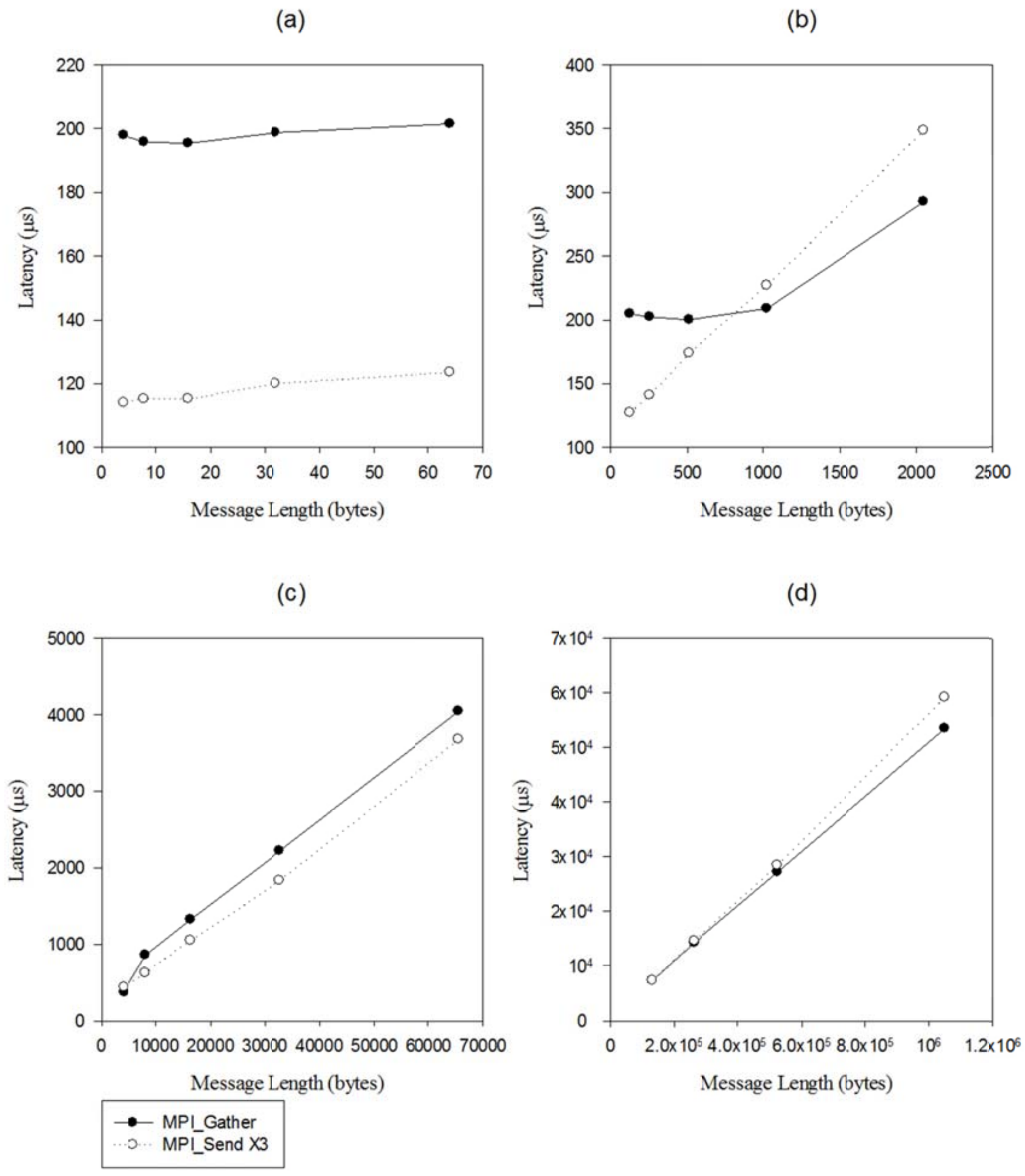

Figure A.19. MPI_Gather versus MPI_Send on P4/Gigabit/4 nodes

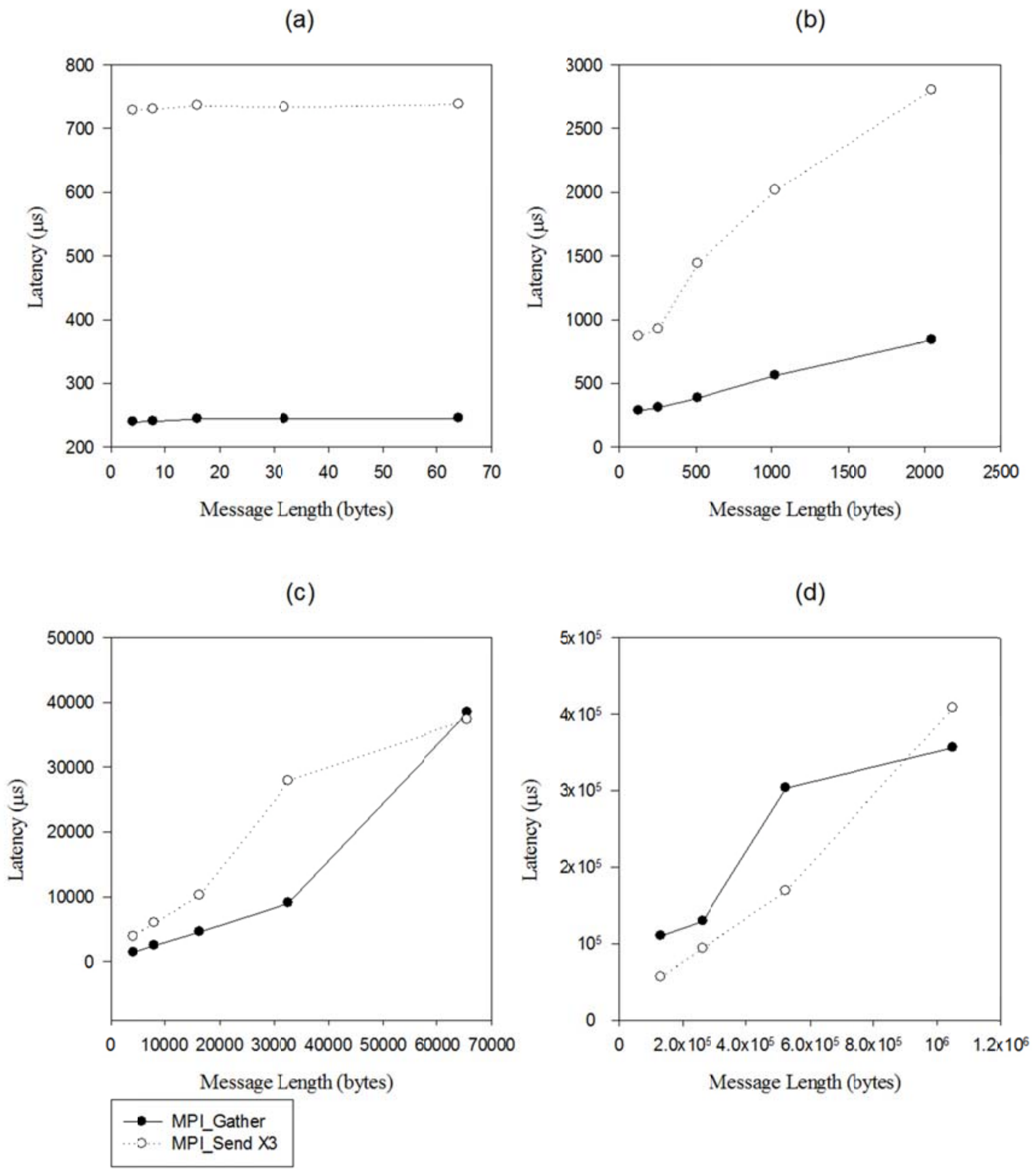

Figure A.20. MPI_Gather versus MPI_Send on P4/100Base-T/4 nodes

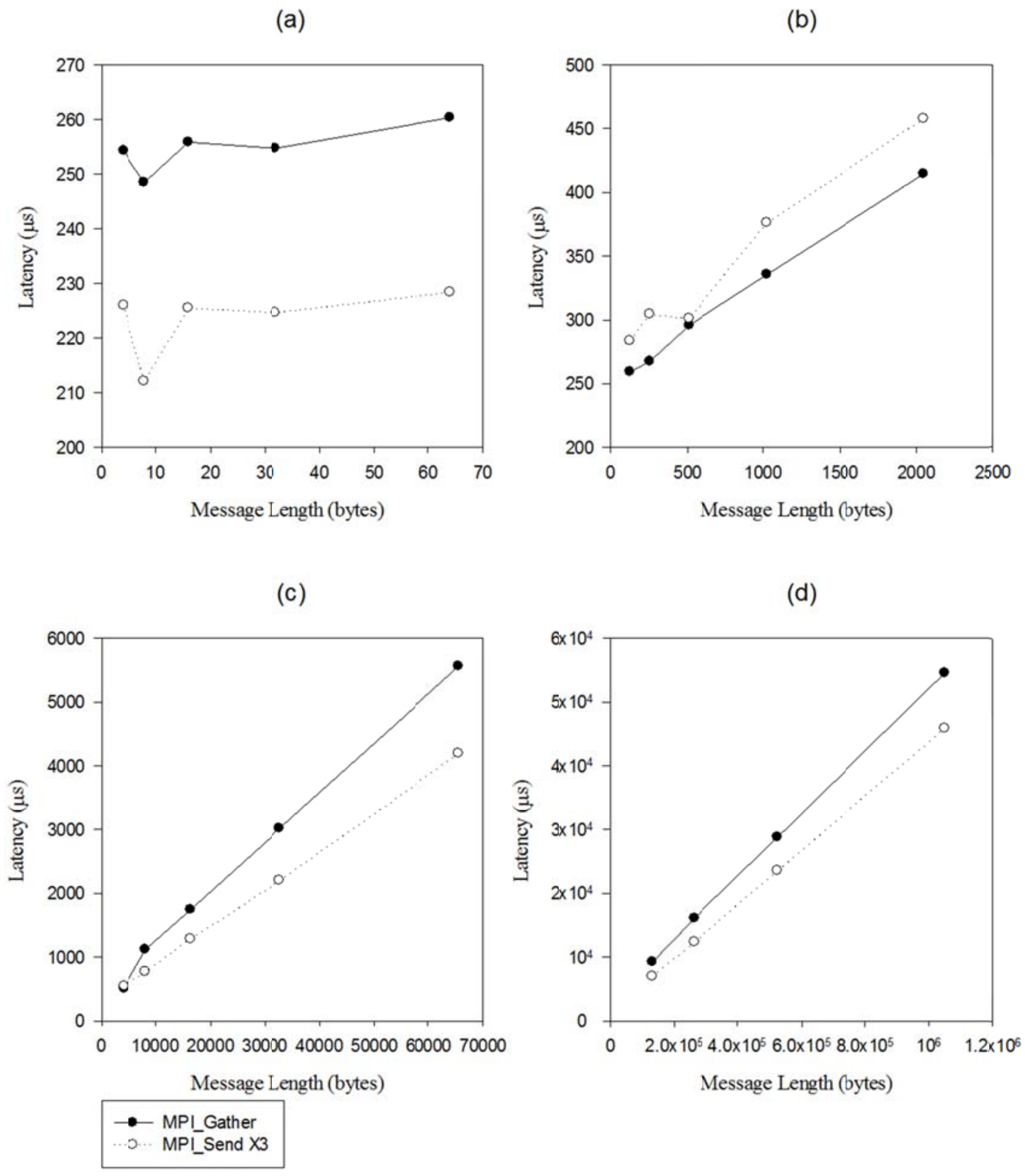

Figure A.21. MPI_Gather versus MPI_Send on PIII/Gigabit/4 nodes

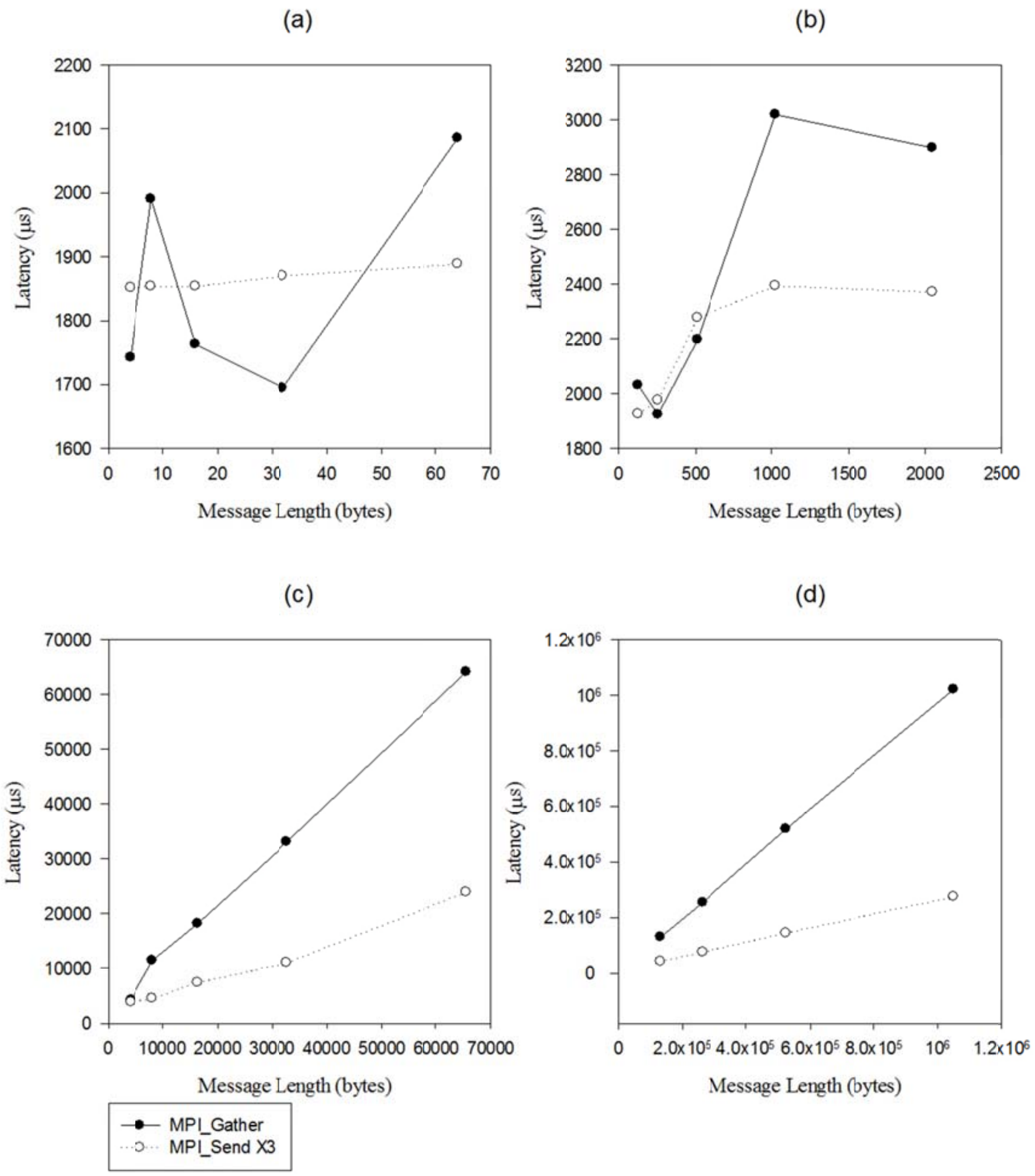

Figure A.22. MPI_Gather versus MPI_Send on PIII/100Base-T/4 nodes

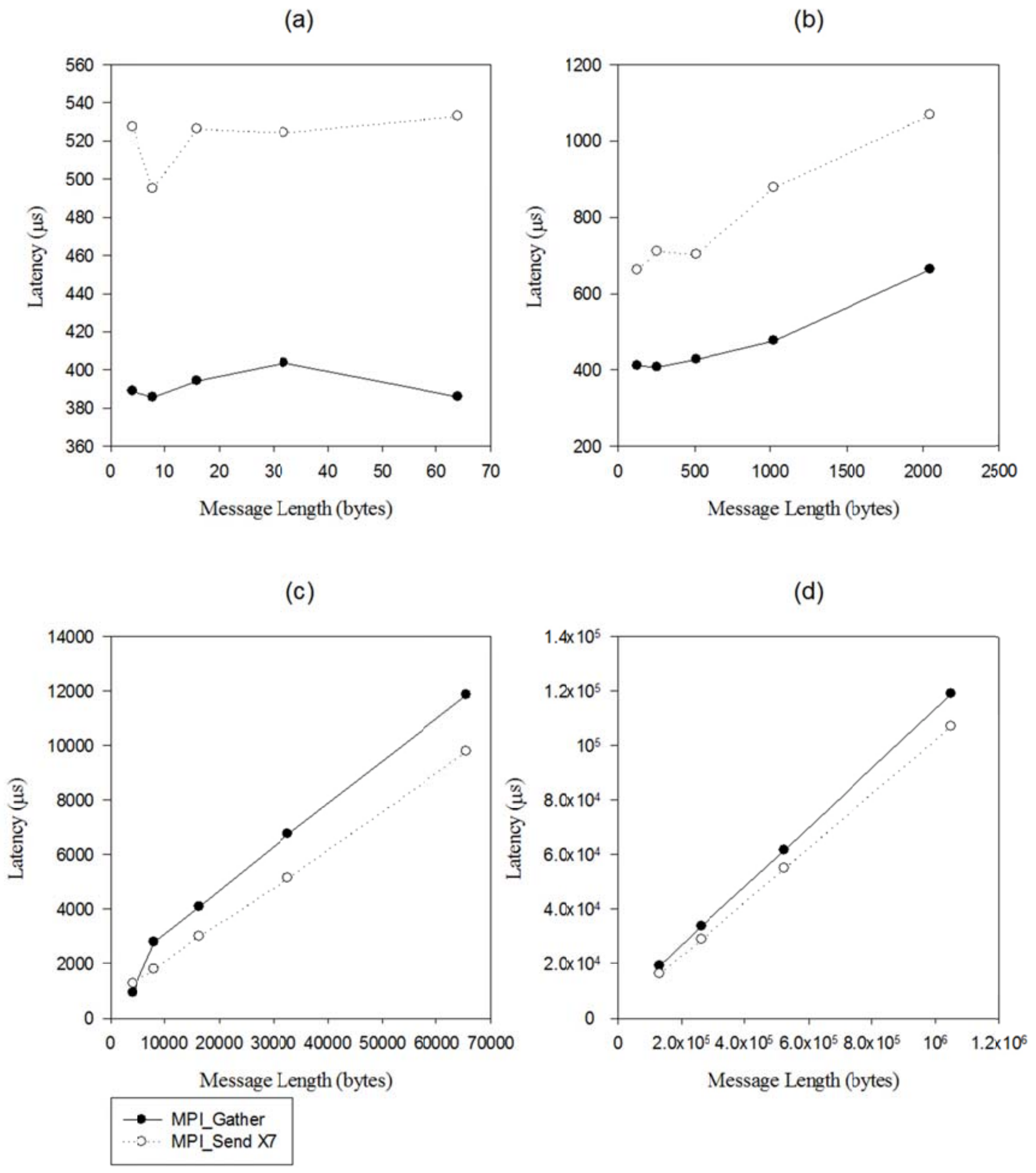

Figure A.23. MPI_Gather versus MPI_Send on PIII/Gigabit/8 nodes

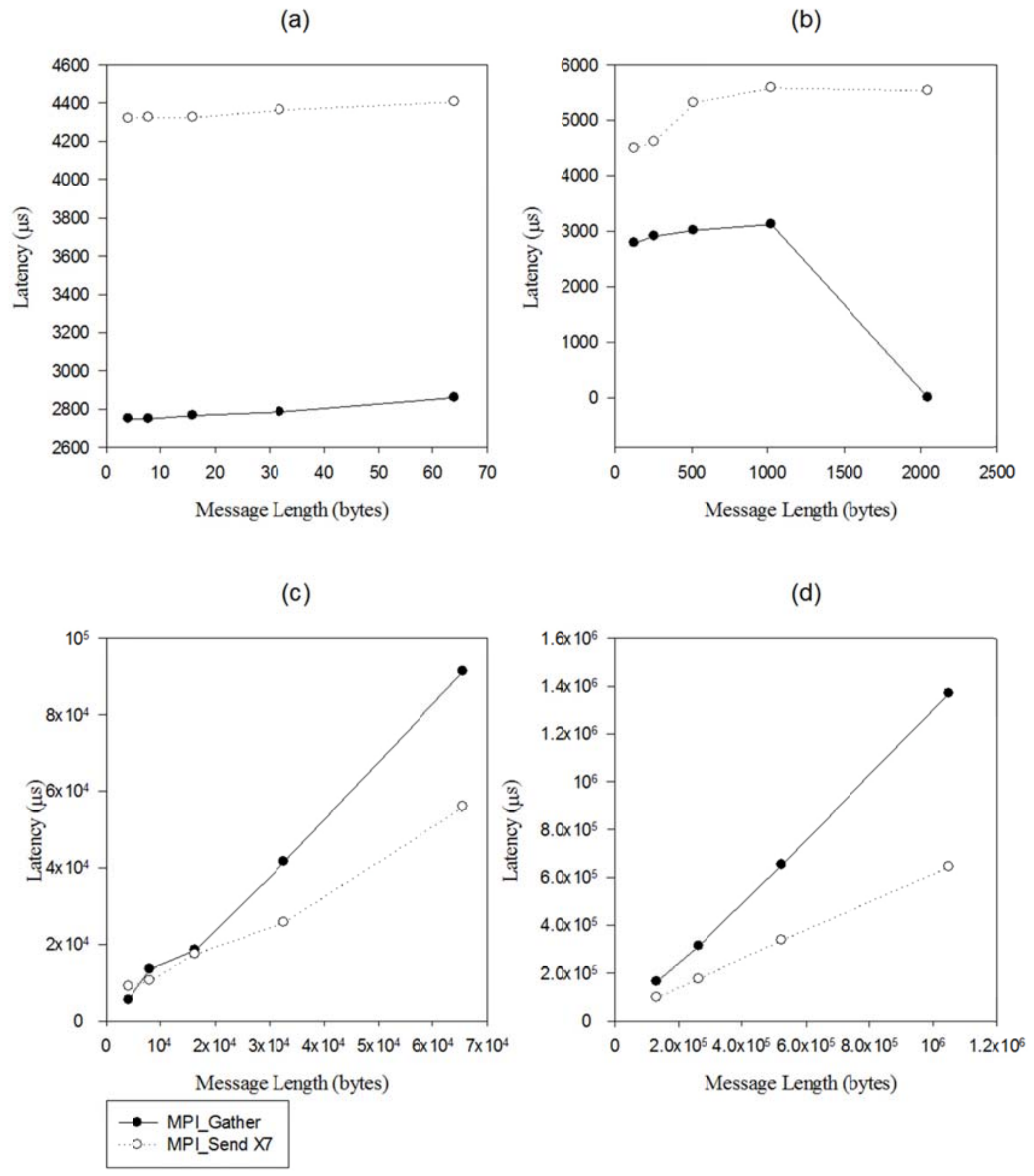

Figure A.24. MPI_Gather versus MPI_Send on PIII/100Base-T/8 nodes

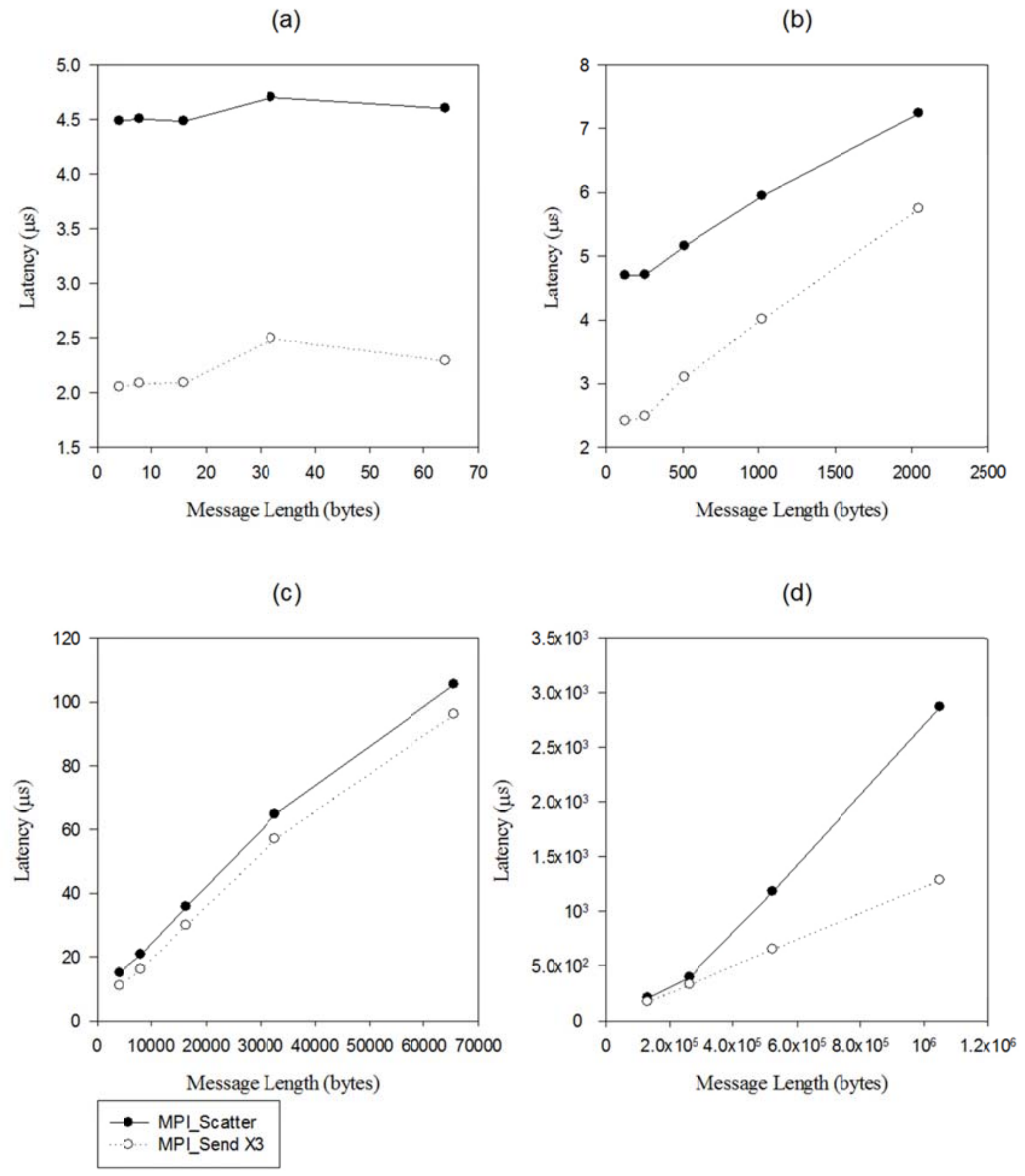

Figure A.25. MPI_Scatter versus MPI_Send on Quadcore/Internal Channels/4 nodes

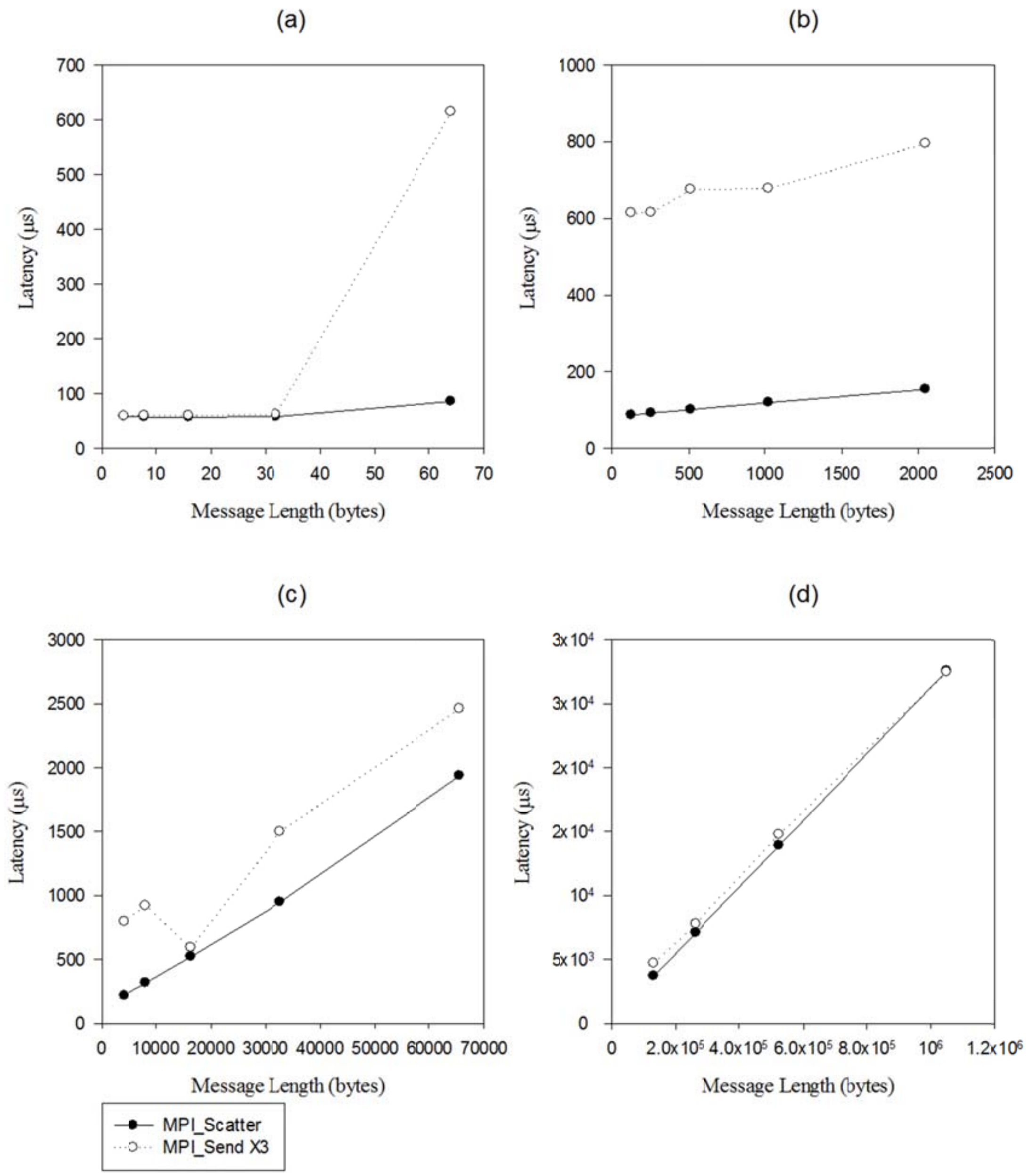

Figure A.26. MPI_Scatter versus MPI_Send on Quadcore/Gigabit/4 nodes

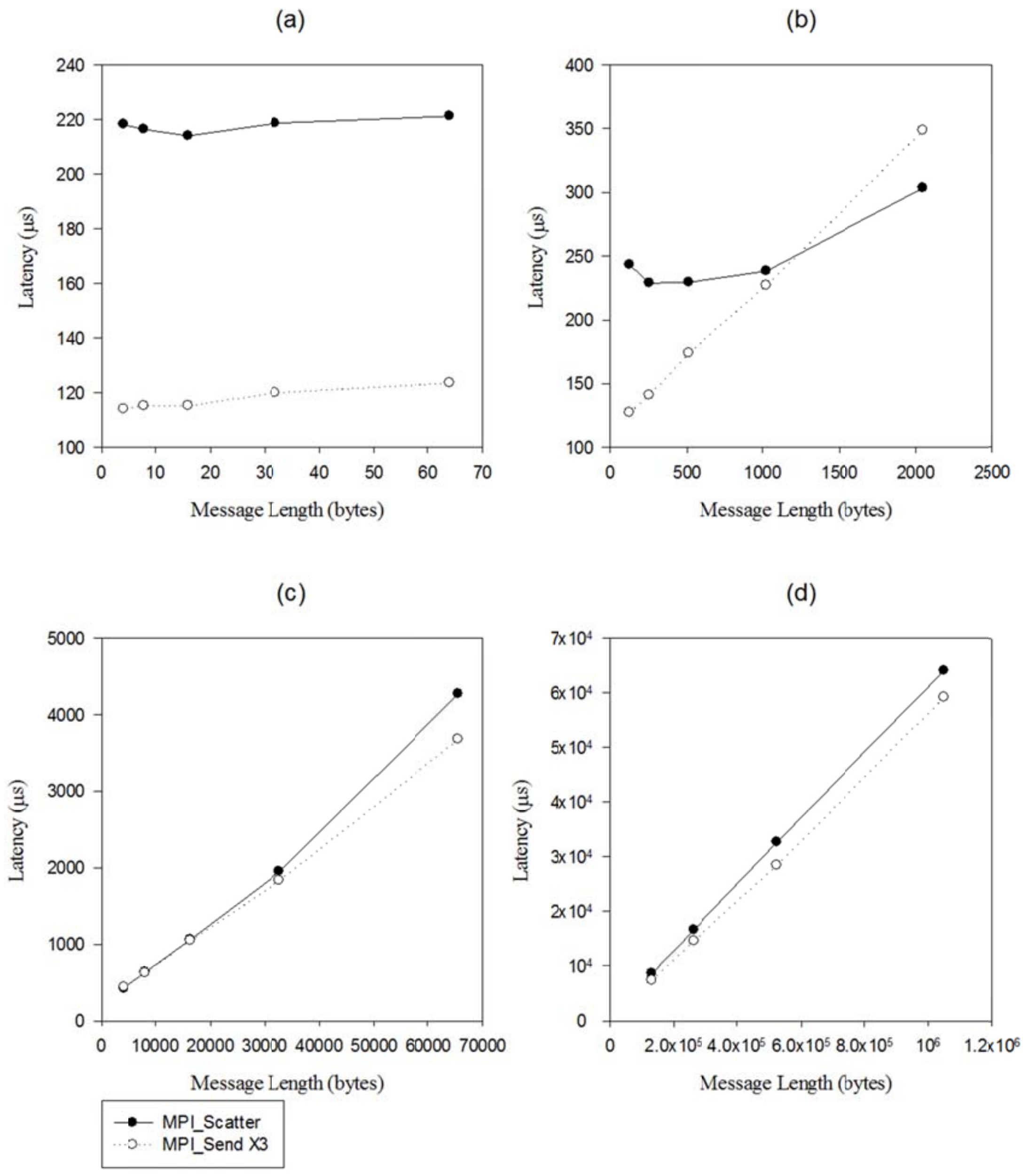

Figure A.27. MPI_Scatter versus MPI_Send on P4/Gigabit/4 nodes

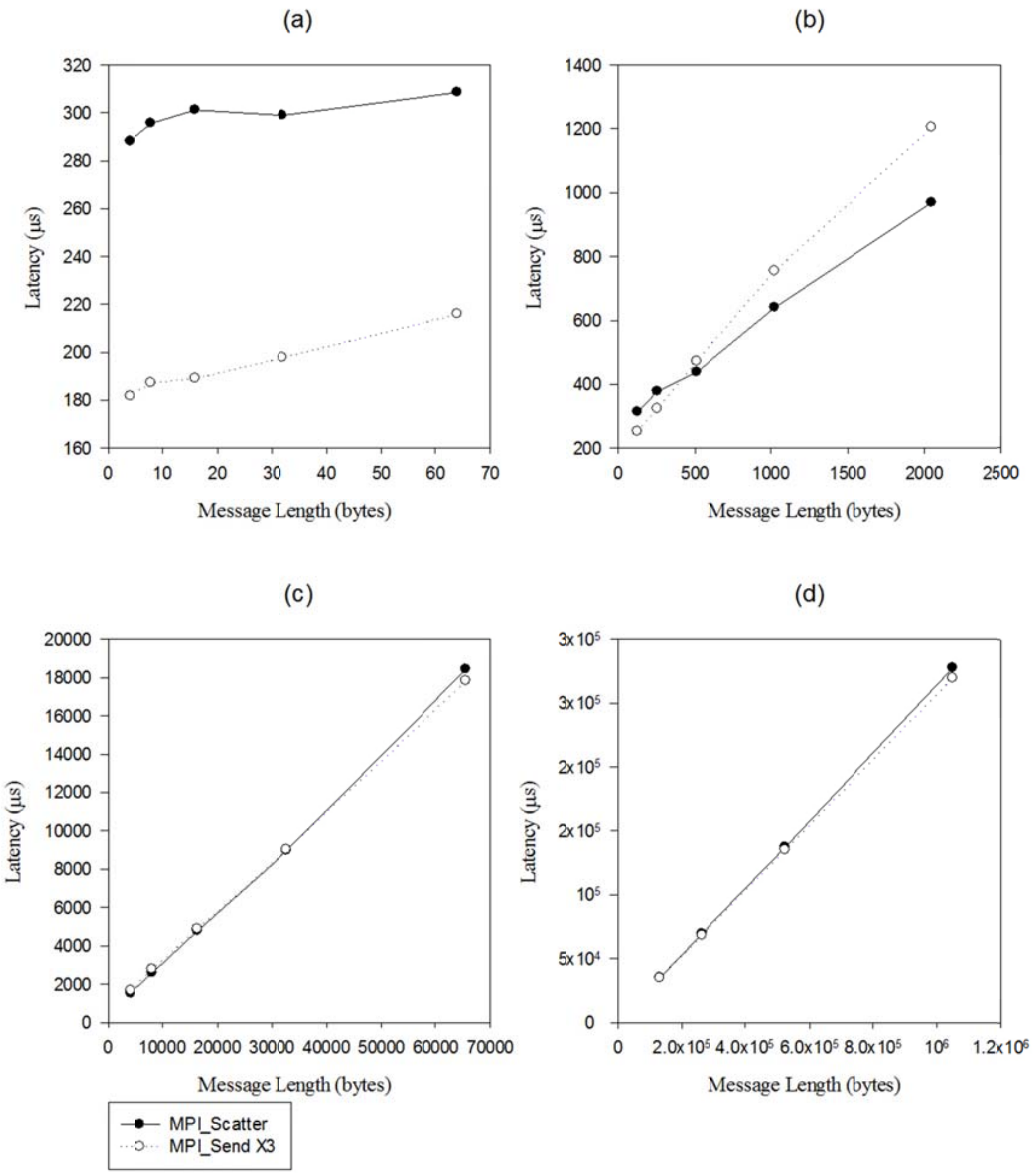

Figure A.28. MPI_ScatterBcast versus MPI_Send on P4/100Base-T/4 nodes

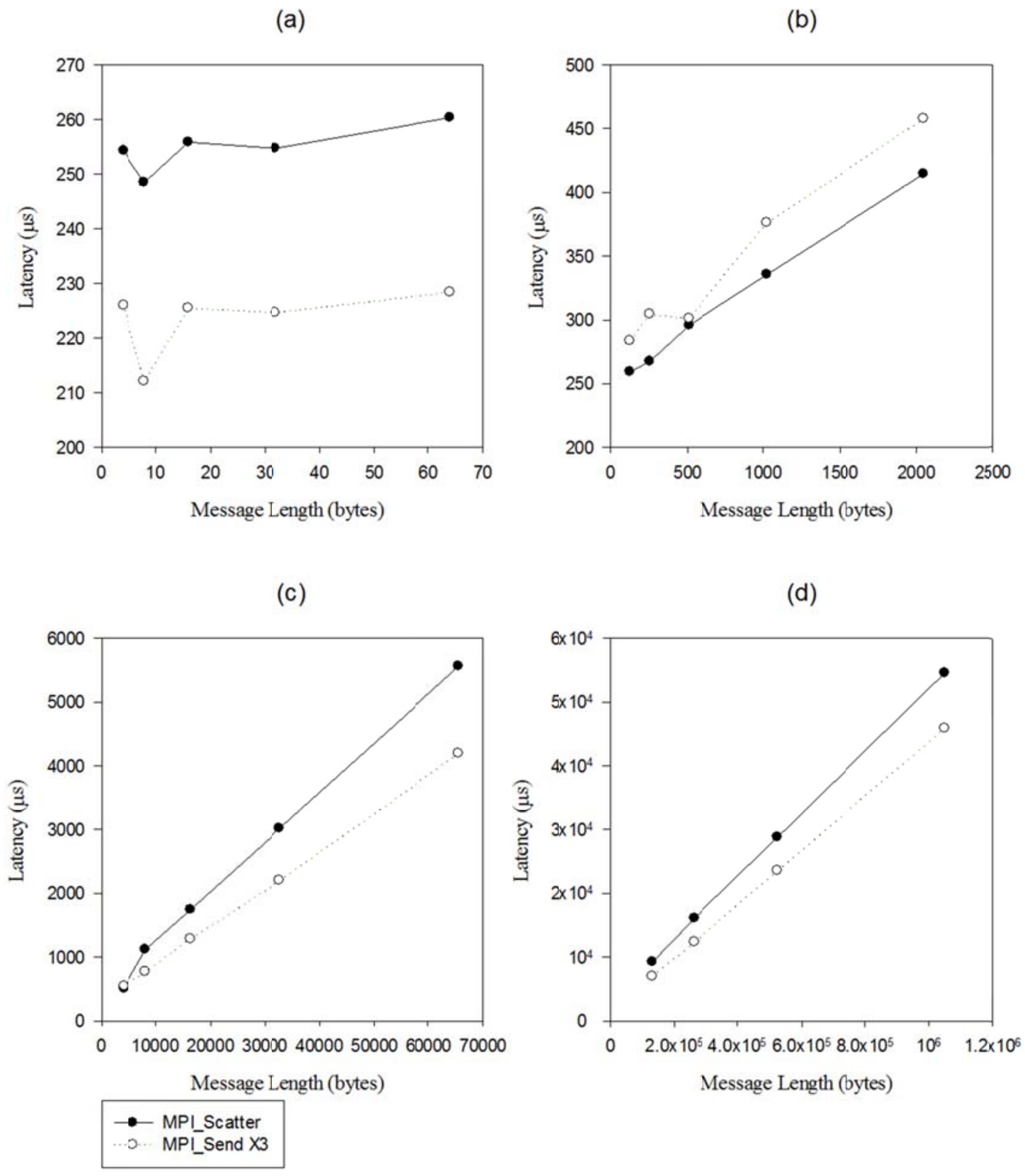

Figure A.29. MPI_Scatter versus MPI_Send on PIII/Gigabit/4 nodes

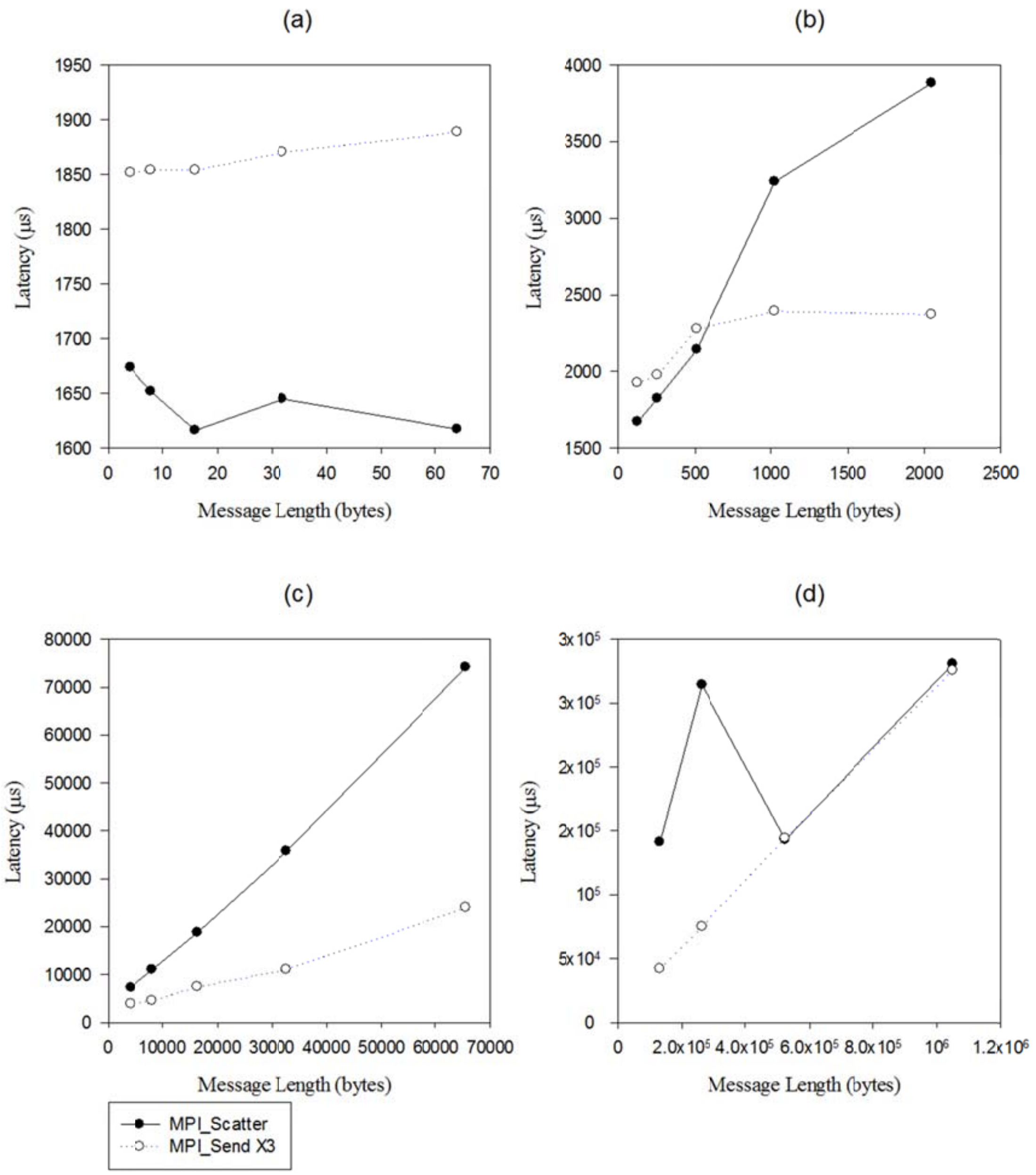

Figure A.30. MPI_Scatter versus MPI_Send on PIII/100Base-T/4 nodes

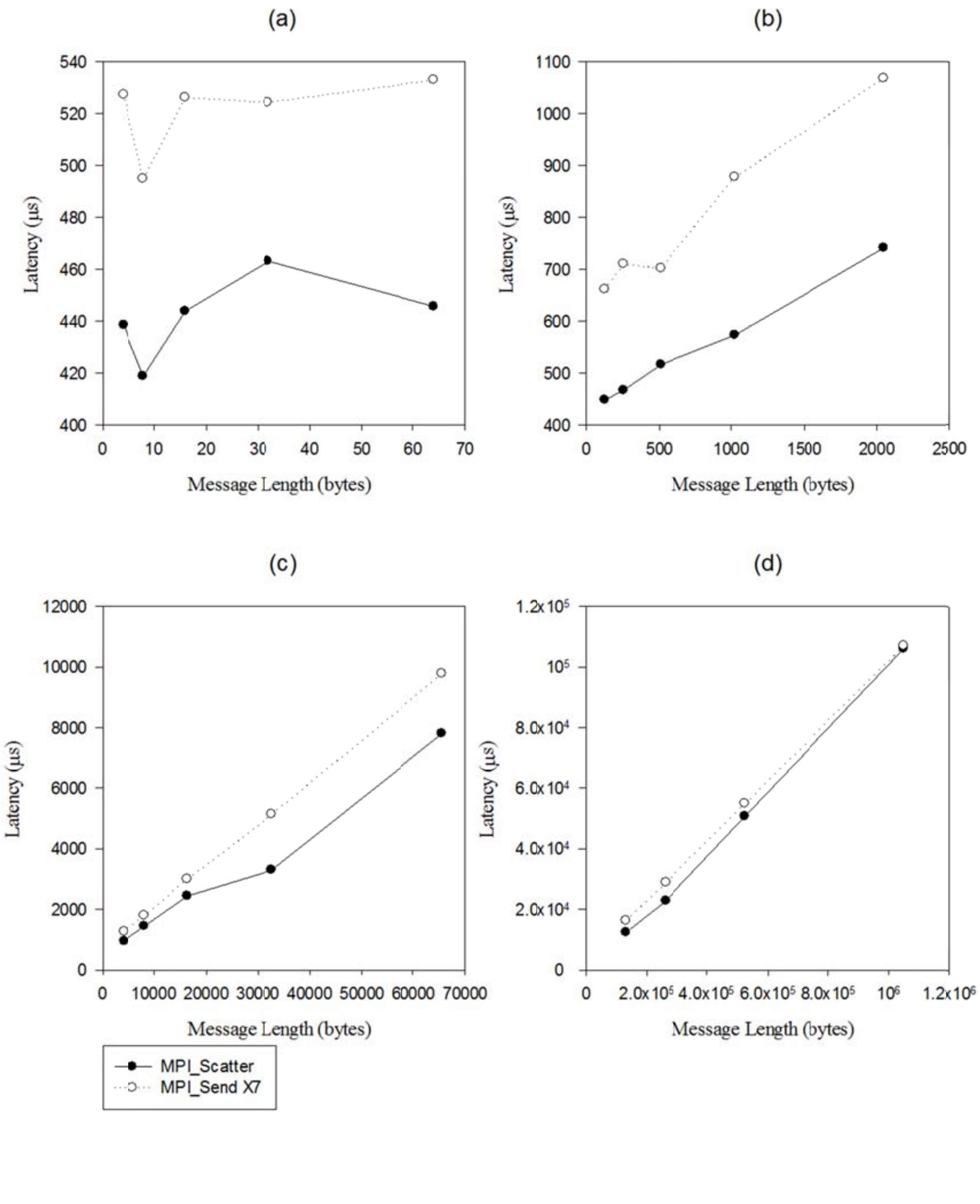

Figure A.31. MPI_Scatter versus MPI_Send on PIII/Gigabit/8 nodes

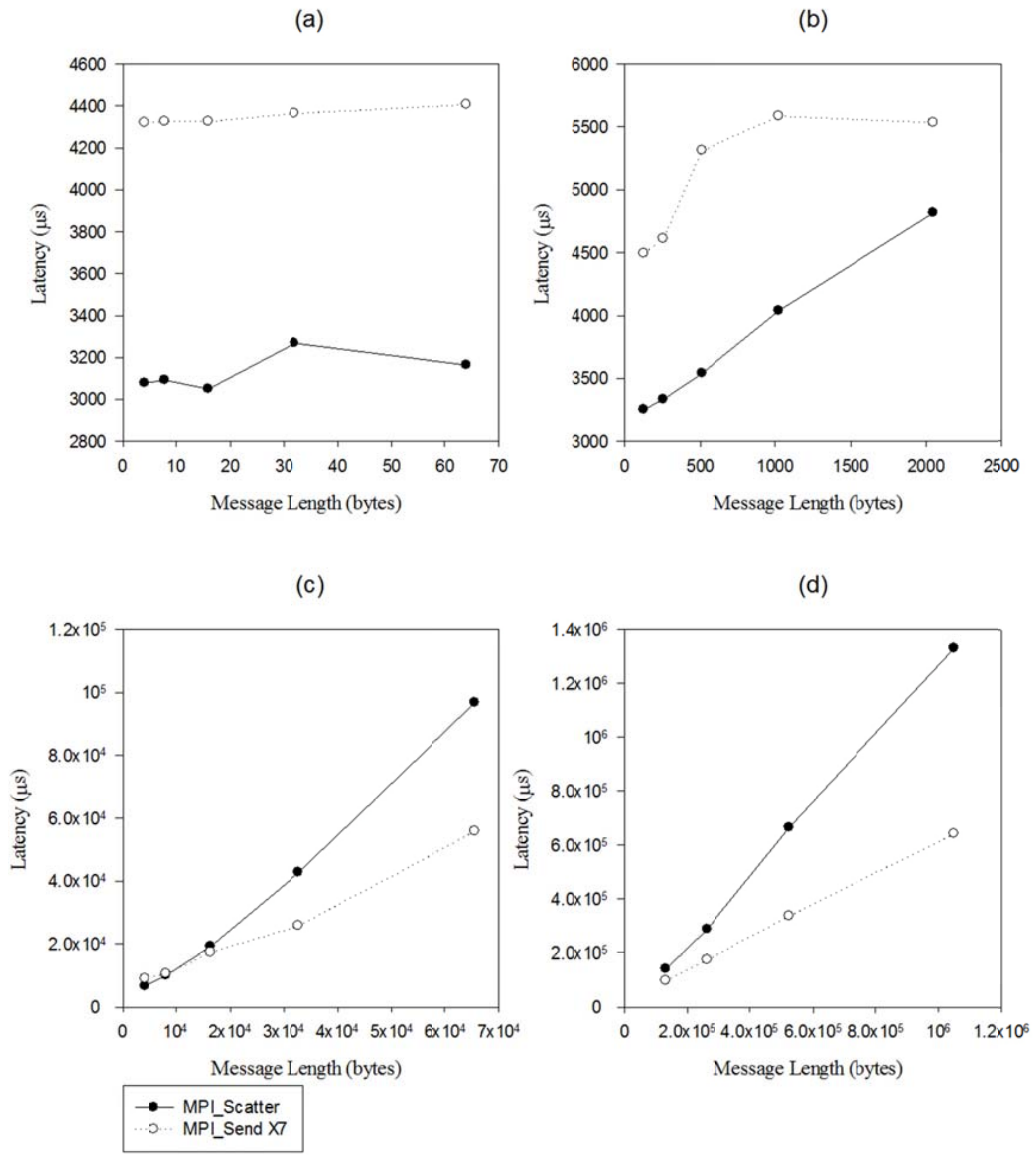

Figure A.32. MPI_Scatter versus MPI_Send on PIII/100Base-T/8 nodes

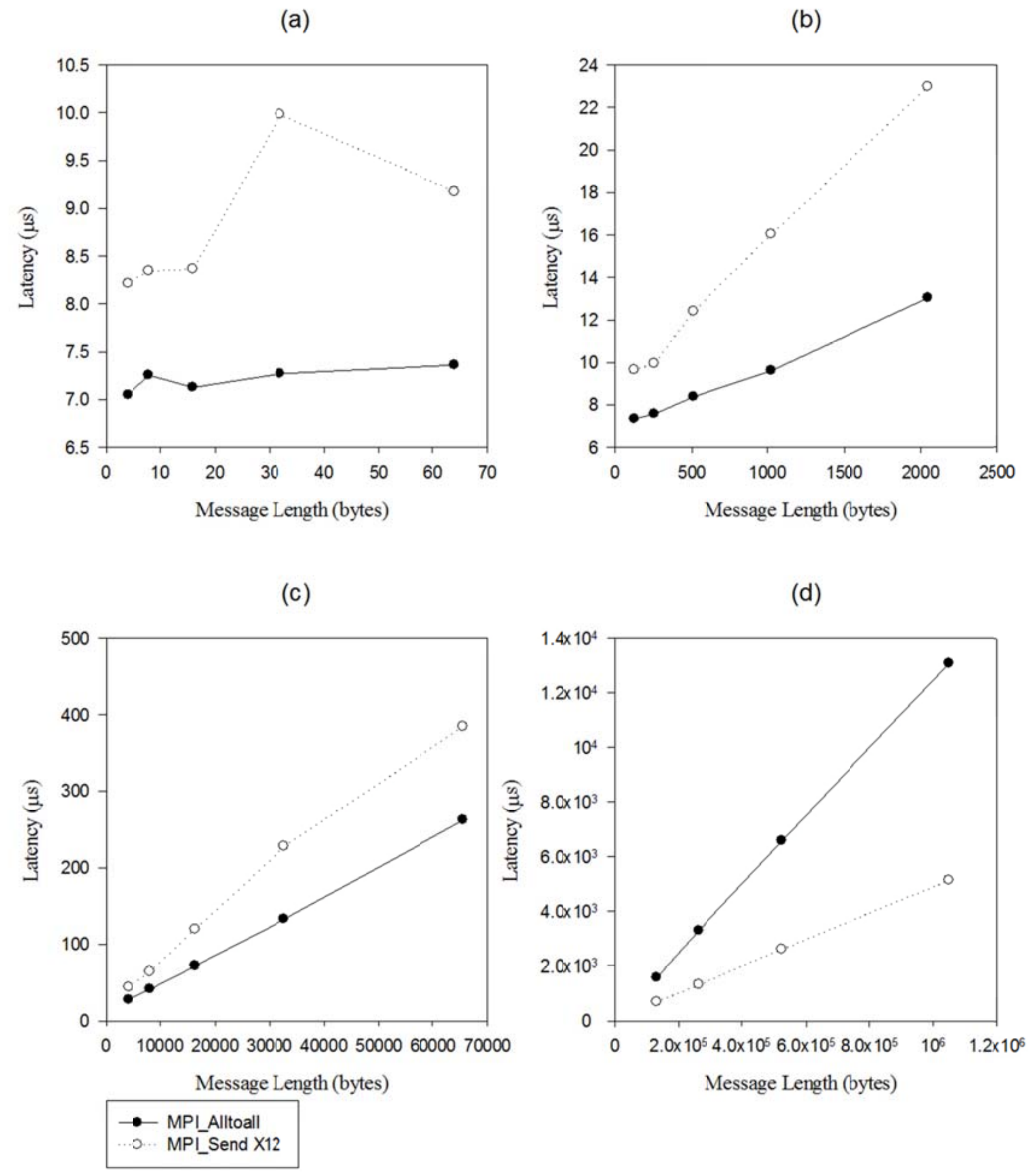

Figure A.33. MPI_Alltoall versus MPI_Send on Quadcore/Internal Channels/4 nodes

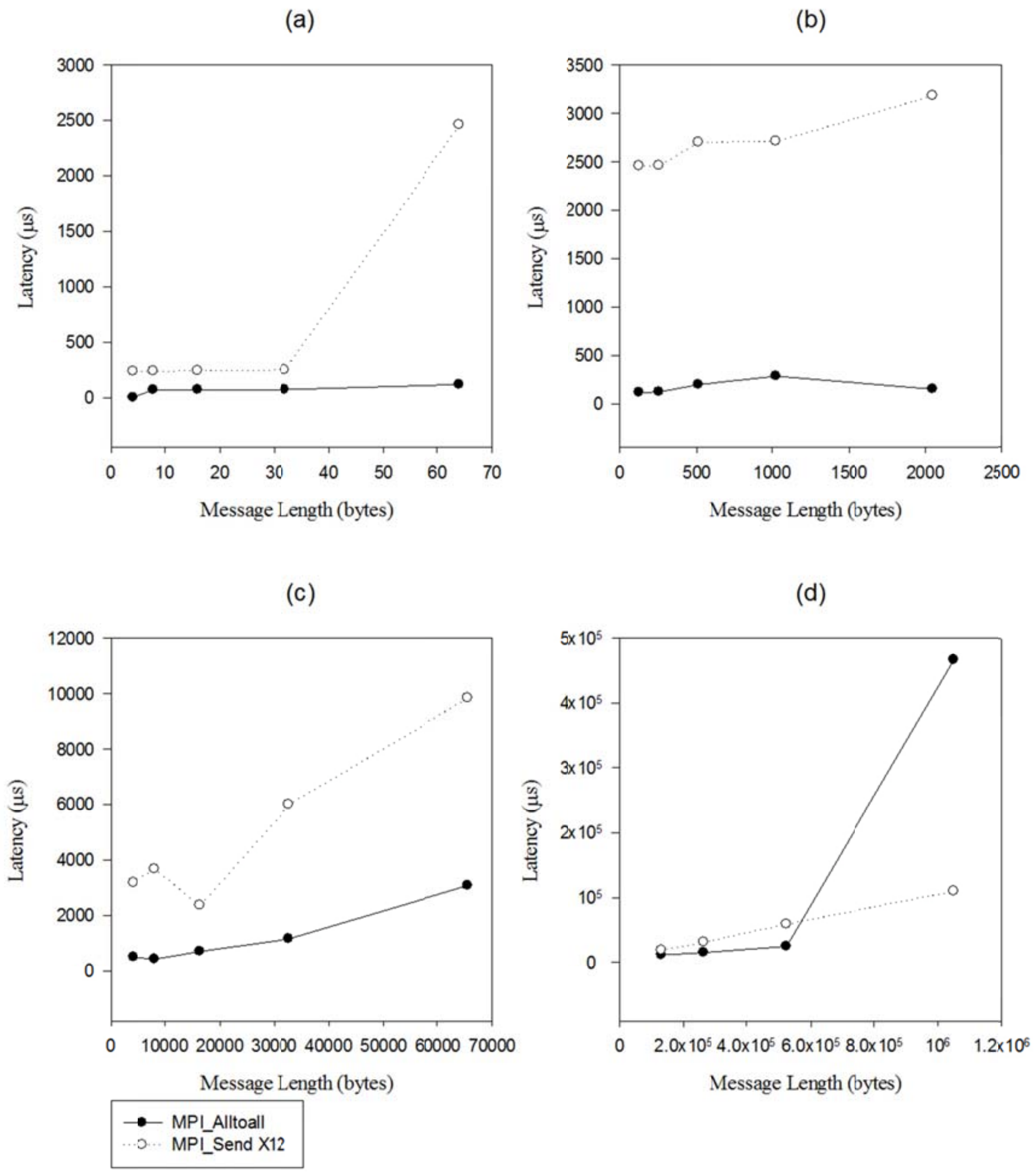

Figure A.34. MPI_Alltoall versus MPI_Send on Quadcore/Gigabit/4 nodes

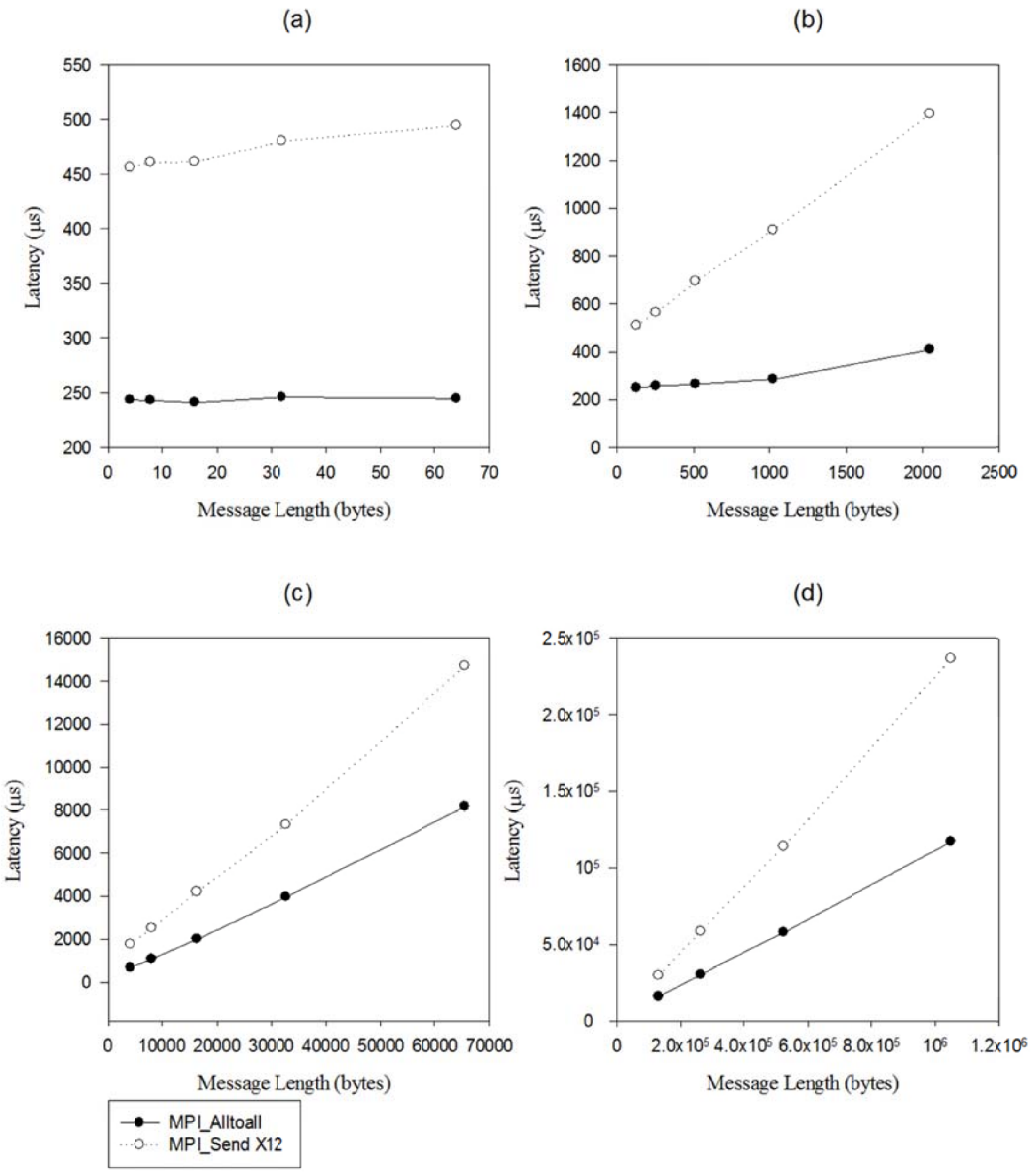

Figure A.35. MPI_AlltoallBcast versus MPI_Send on P4/Gigabit/4 nodes

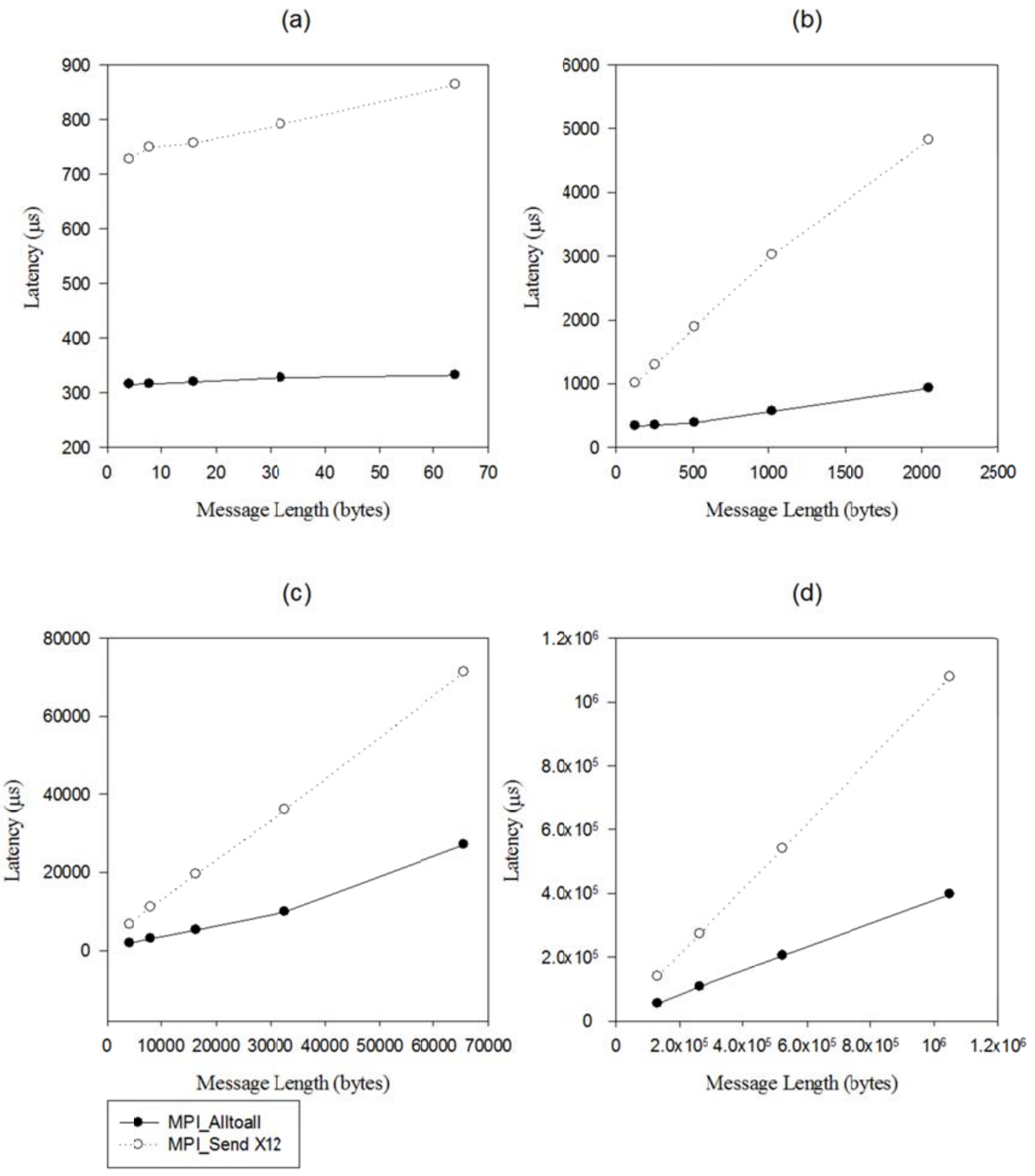

Figure A.36. MPI_Alltoall versus MPI_Send on P4/100Base-T/4 nodes

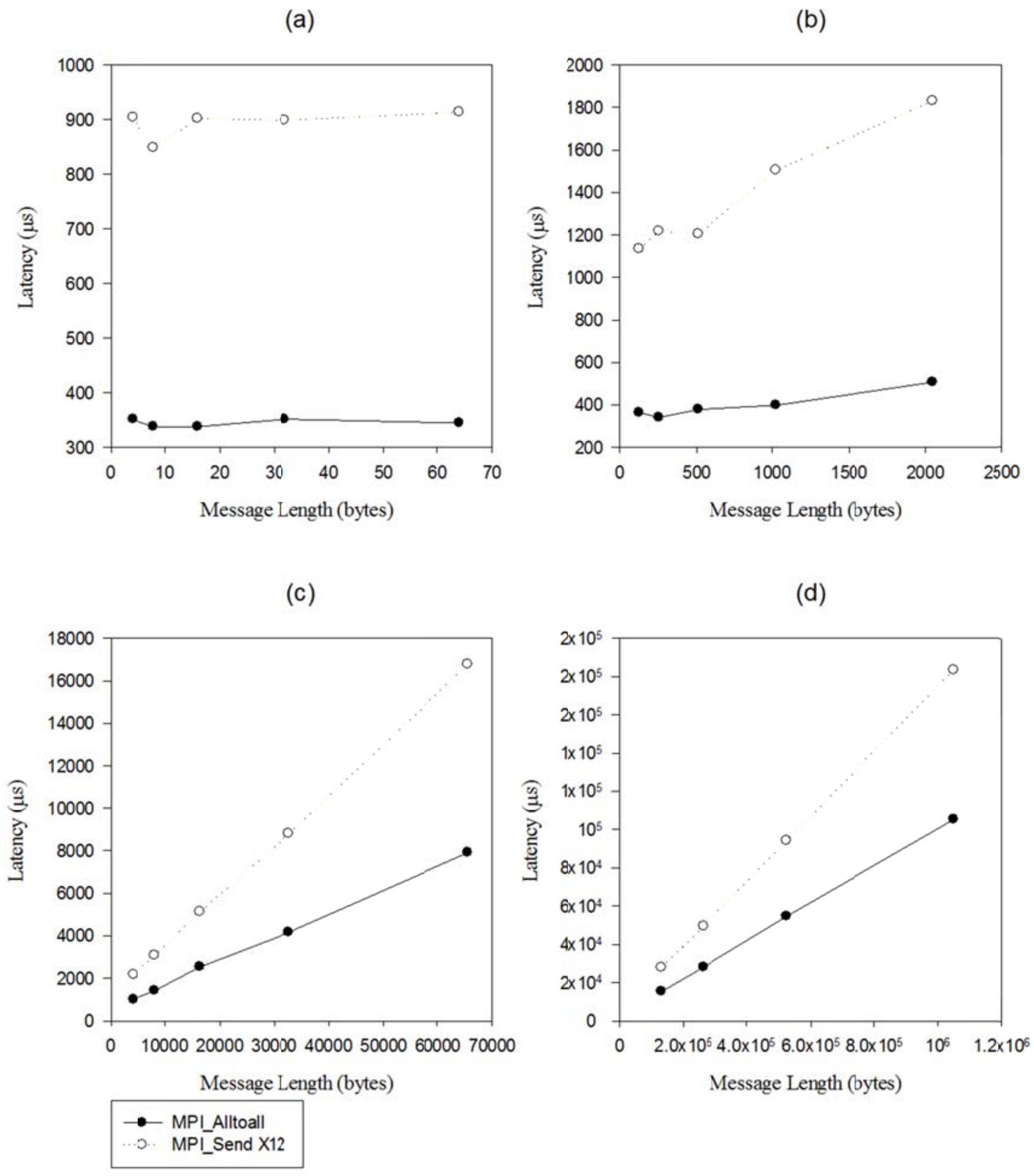

Figure A.37. MPI_Alltoall versus MPI_Send on PIII/Gigabit/4 nodes

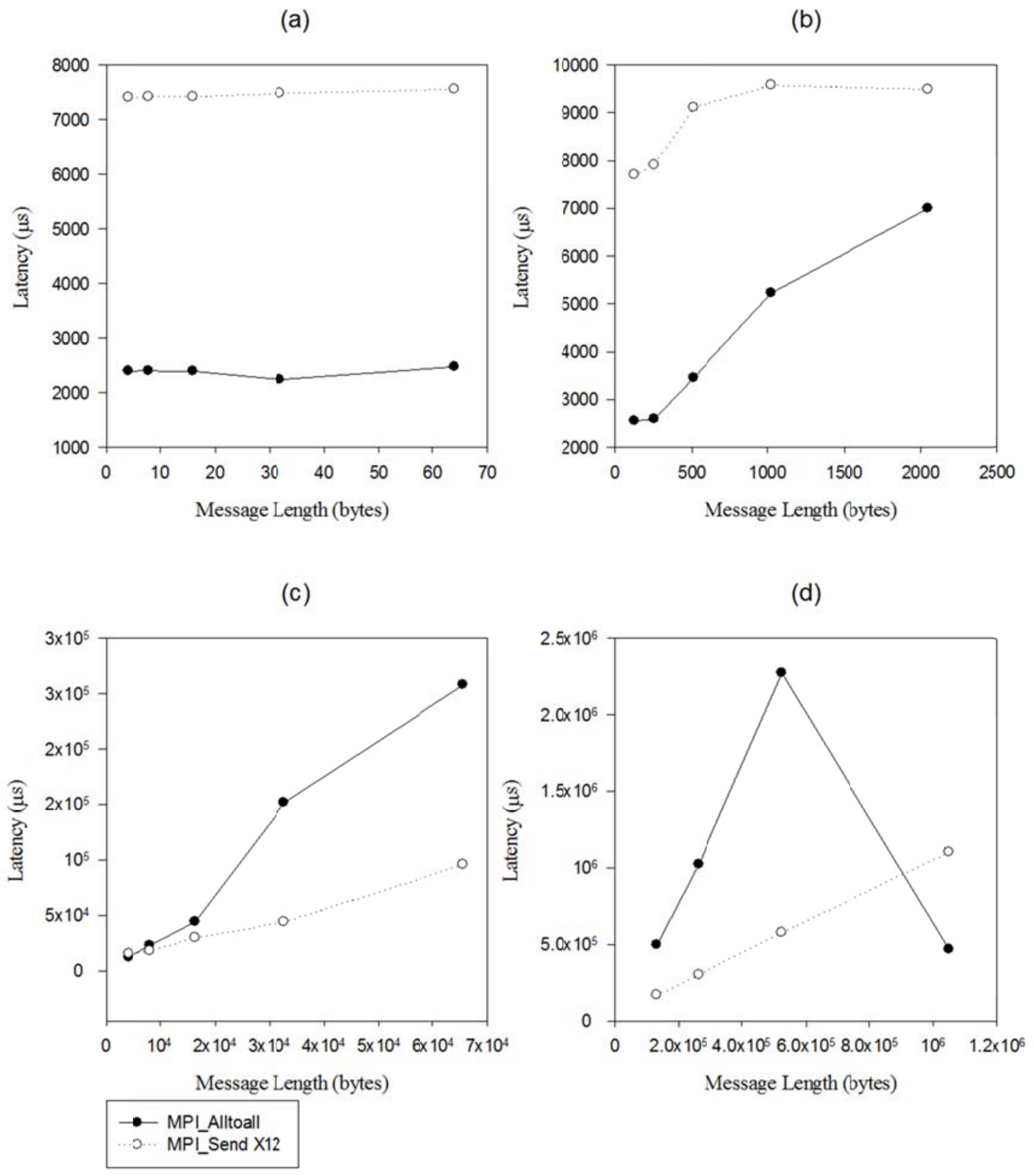

Figure A.38. MPI_Alltoall versus MPI_Send on PIII/100Base-T/4 nodes

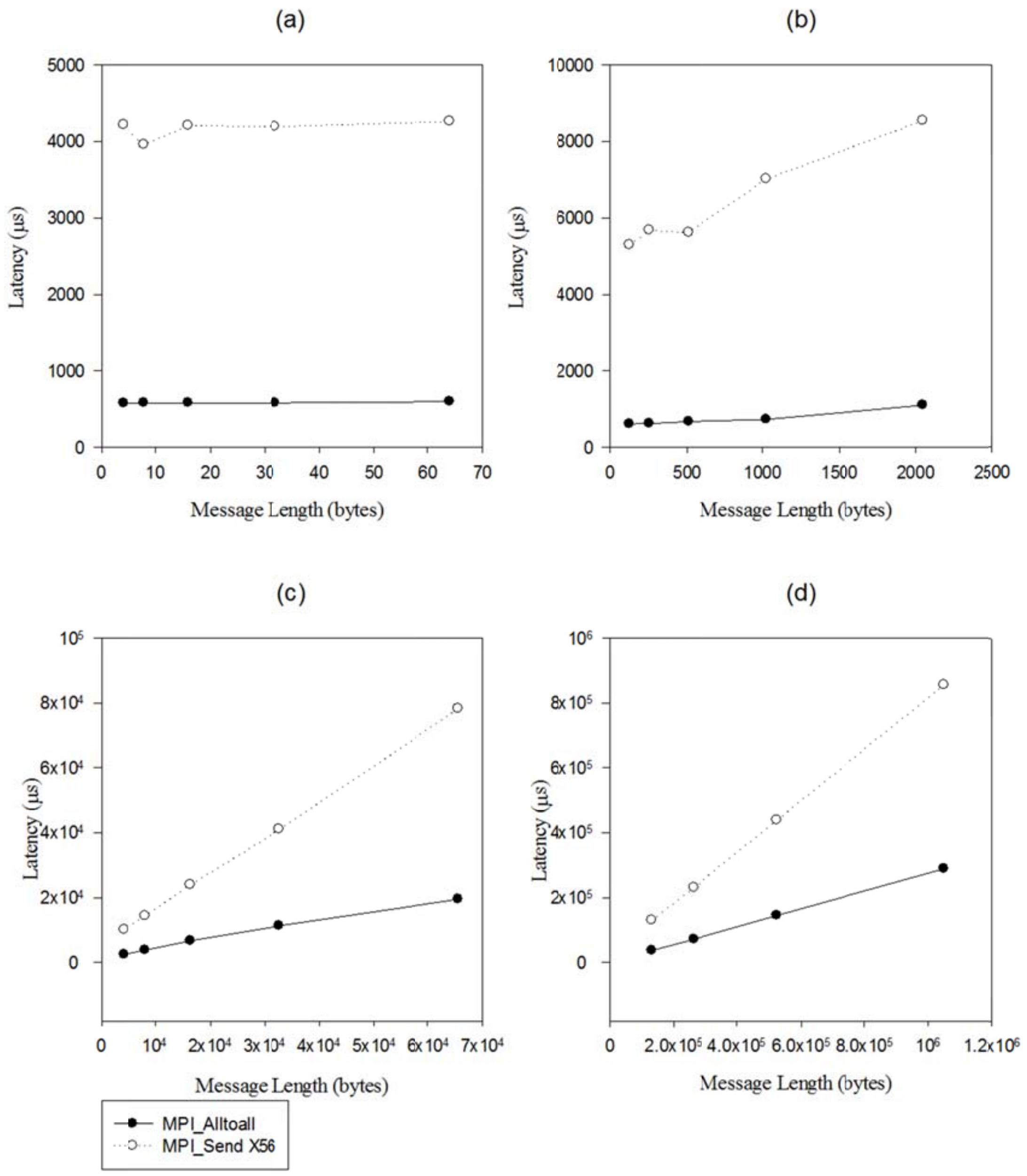

Figure A.39. MPI_Alltoall versus MPI_Send on PIII/Gigabit/8 nodes

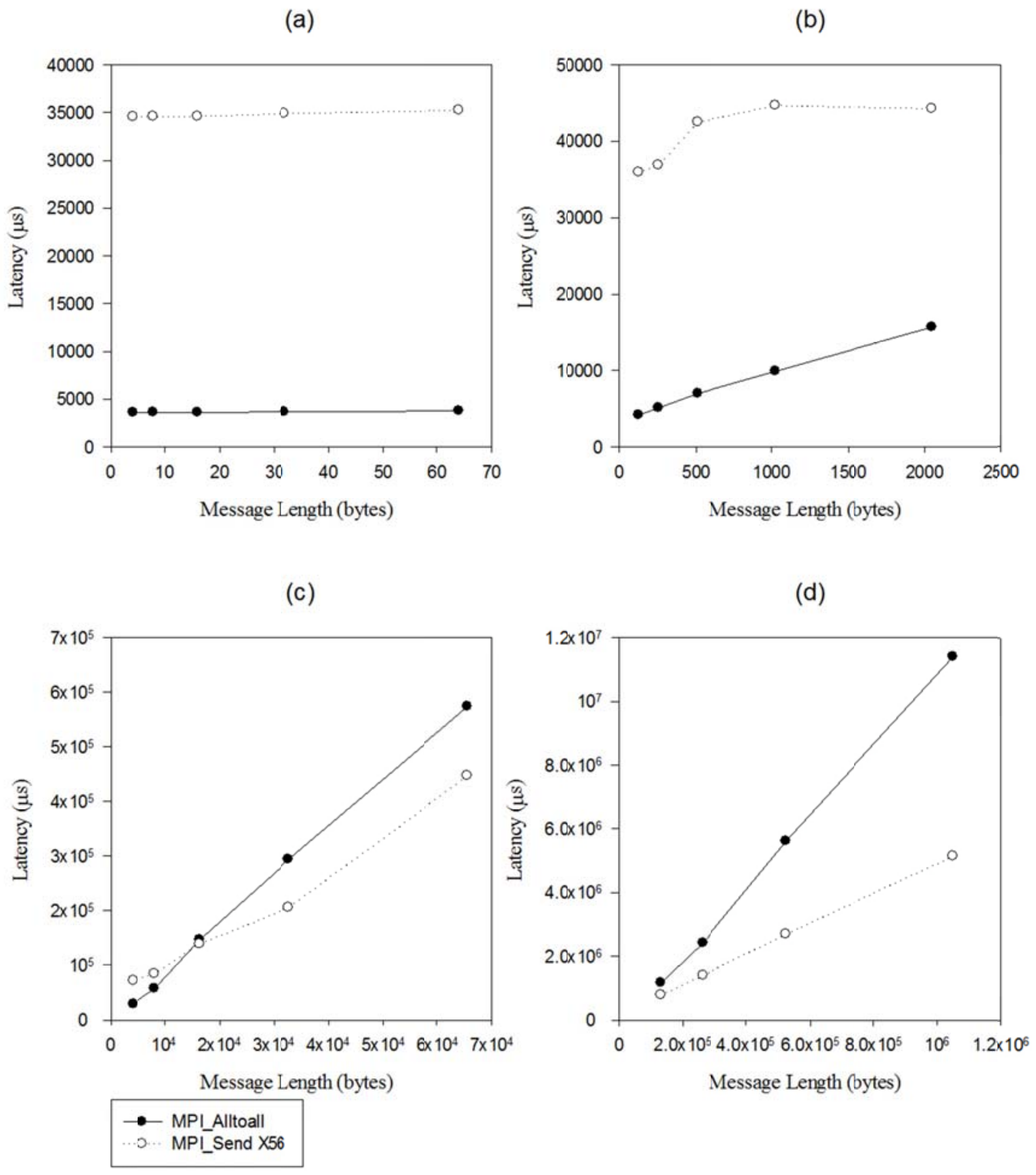

Figure A.40. MPI_Alltoall versus MPI_Send on PIII/100Base-T/8 nodes