Communication Performance Measurement and Analysis on Commodity Clusters

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF THE UNIVERSITY OF ADELAIDE
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By

Nor Asilah Wati Abdul Hamid

March 13, 2008

Table of Content

List of Figures

List of Tables

ABSTRACT

Cluster computers have become the dominant architecture in high-performance computing. Parallel programs on these computers are mostly written using the Message Passing Interface (MPI) standard, so the communication performance of the MPI library for a cluster is very important. This thesis investigates several different aspects of performance analysis for MPI libraries, on both distributed memory clusters and shared memory parallel computers.

The performance evaluation was done using MPIBench, a new MPI benchmark program that provides some useful new functionality compared to existing MPI benchmarks. Since there has been only limited previous use of MPIBench, some initial work was done on comparing MPIBench with other MPI benchmarks, and improving its functionality, reliability, portability and ease of use. This work included a detailed comparison of results from the Pallas MPI Benchmark (PMB), SKaMPI, Mpptest, MPBench and MPIBench on both distributed memory and shared memory parallel computers, which has not previously been done. This comparison showed that the results for some MPI routines were significantly different between the different benchmarks, particularly for the shared memory machine.

A comparison was done between Myrinet and Ethernet network performance on the same machine, an IBM Linux cluster with 128 dual processor nodes, using the MPICH MPI library. The analysis focused mainly on the scalability and variability of communication times for the different networks, making use of the capability of MPIBench to generate distributions of MPI communication times. The analysis provided an improved understanding of the effects of TCP retransmission timeouts on Ethernet networks.

This analysis showed anomalous results for some MPI routines. Further investigation showed that this is because MPICH uses different algorithms for small and large message sizes for some collective communication routines, and the message size where this changeover occurs is fixed, based on measurements using a cluster with a single processor per node. Experiments were done to measure the performance of the different algorithms, which demonstrated that for some MPI routines the optimal changeover points were very different between Myrinet and Ethernet networks and for 1 and 2 processors per node. Significant performance improvements can be made by allowing the changeover points to be tuned rather than fixed, particularly for commodity Ethernet networks and for clusters with more than 1 process per node.

MPIBench was also used to analyse the MPI performance and scalability of a large ccNUMA shared memory machine, an SGI Altix 3000 with 160 processors. The results were compared with a high-end cluster, an AlphaServer SC with Quadrics QsNet interconnect. For most MPI routines the Altix showed significantly better performance, particularly when non-buffered copy was used. MPIBench proved to be a very capable tool for analyzing MPI performance in a variety of different situations.

DECLARATION

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due to reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

NOR ASILAH WATI ABDUL HAMID
Phd Candidate,
Department of Computer Science
University of Adelaide
13 March 2008

LIST OF PUBLICATIONS

The following papers were written based on the work presented in this thesis.

Papers in Refereed Conference Proceedings

1. N. A. W. A Hamid and P. Coddington. "Analysis of Algorithm Selection for Optimizing Collective Communication with MPICH for Ethernet and Myrinet Networks", *Proc. of The* 8th *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'07), Adelaide, Australia, December 2007.*

2. N. A. W. A Hamid and P. Coddington. "Averages, Distribution and Scalability of MPI Communication Times for Ethernet and Myrinet Networks", *Proc. of Parallel and Distributed Computing and Network (PDCN'07), 25th IASTED International Multi-Conference, Congress Innsbruck, Austria, February 2007.*

3. N. A. W. A Hamid, P. Coddington and F. Vaughan. "Comparison of MPI Benchmark Programs on an SGI Altix ccNUMA Shared Memory Machine"**,** *Proc. of Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'06), Rhodes, Greece, April 2006.*

4. N. A. W. A Hamid, P. Coddington and F. Vaughan. "Performance Analysis of MPI Communications on the SGI Altix 3700", *Proc. of APAC'05, Gold Coast, Australia, September 2005.*

ACKNOWLEDGEMENT

The work described in this thesis was carried out at the University of Adelaide in the Department of Computer Science under the supervision of Dr. Paul Coddington.

I would like to express my deepest gratitude to Dr. Paul Coddington for his excellent guidance, enthusiastic supervision and tolerance throughout this research. His dedication and contribution gave me tremendous help in completion of this research and the publication of the papers. I also would like to thank him for his commitment to this research and for giving me the chance to present our papers at conferences around the world.

Many thanks are due to the University of Adelaide and in particular the Department of Computer Science, as well as the South Australian Partnership for Advanced Computing (SAPAC) for making supercomputer time available to me. Special thanks to Patrick Fitzhenry and Grant Ward for their professional advice and patience throughout the experimental program. I also would like to give thanks for the help from a team of programmers from the University of Adelaide and the South Australian Partnership of Advanced Computing, Alex Chichowski, Tim Seely and Paul Martinaitis.

Lastly to my family, my husband Dr. Raizal Saifulnaz Muhammad Rashid and my son Muhammad Haziq Raizal Saifulnaz and my parents for their love, encouragement and continual support throughout the years of doing this research.

CHAPTER 1

Introduction

Since the invention of computers, users have demanded more powerful and faster computers to solve a huge variety of numerical problems, particularly the numerical simulation of scientific and engineering problems. In order to cope with the demand, computers have evolved from sequential to parallel computers. Sequential means that the computer is running with a single processor, while parallel computers allow more than one processor to be used at the same time. Parallel computing refers to the concept of speeding-up the execution of a program by dividing the program into multiple fragments that can execute simultaneously, each on its own processor. A program being executed across *n* processors might execute *n* times faster than it would using a single processor. A parallel computing system generally involves networks of multiple commodity micro-processors, rather than a single, larger mainframe. These multiprocessor computers consist of a number of autonomous processors which can each execute separate programs concurrently, or a single program can use multiple processors. These autonomous processors are coupled by hardware and by software to attain the collective performance needed to master applications that cannot be handled by the use of individual uniprocessor computers.

Traditionally, the developments of parallel computing have been motivated by numerical simulations of complex systems such as weather, climate, mechanical devices, electronic circuits, manufacturing processes, and chemical reactions. However, the most significant forces driving the development of faster computers today are emerging commercial applications that require a computer to be able to process large amounts of data in sophisticated ways. These applications include online transaction processing, data mining, web servers and web search, on-demand video streaming, computer-aided diagnosis in medicine, parallel databases used for decision support, and advanced graphics and virtual reality, particularly in the entertainment industry [160, 161, 162].

1.1 Parallel computing architectures

In general, parallel computing is classified into shared memory multiprocessor (SMP), distributed memory multicomputers with compute nodes connected by a network (particularly those built from commodity components, which are often call cluster computers), or a combination of both. Due to the increase in network hardware speed and the availability of low-cost high-performance workstations, personal computers and servers, cluster computing has become increasingly popular over the past ten years. Many research institutes, universities and companies around the world have purchased or built low cost clusters, such as commodity Linux or Beowulf [146] clusters, for their parallel processing needs at a fraction of the price of mainframes or custom supercomputers. In recent years it has become more common to find cluster computers built with SMP nodes. Table 1.1 shows that cluster architecture has grown rapidly in the last ten years compared to the other architectures, and this trend has increased with the advent of multi-core CPUs, so that all new clusters now have SMP nodes. This thesis will focus mainly on cluster computers, particularly SMP clusters.

Many different interconnection technologies have been used to build clusters, including Ethernet [134], Myrinet [110], Giganet [6], QsNet [145], SCInet [150] as well as proprietary networks from different vendors such as IBM [159] and Cray [158]. When the work in this thesis was begun, the most common networks were Myrinet, Ethernet and networks based on the IBM SP Switch [157], whereas currently the most popular networks are Gigabit Ethernet, Infiniband [144], SP Switch and Myrinet, based on the statistics of interconnet networks used in machines in the list of the at the TOP500 websites [91]. Ethernet is a cheap LAN technology that can deliver 100Mbit/sec bandwidth (Fast Ethernet), 1 Gbit/sec (Gigabit Ethernet, which is currently the most commonly used) or 10 Gbit/sec, while maintaining the original Ethernet's transmission protocol, CSMA/CD. TCP/IP is the most popular communication protocol for Ethernet, although other protocols can be used, such as VIA [33]. TCP/IP is a robust protocol set developed to connect a number of different networks designed by different vendors into a network of networks. However, the reliability provided by TCP/IP has a price in communication overhead. Infiniband and Myrinet are two of the leading cluster interconnect technologies for commodity clusters, which use different communication protocols that allow lower latency, but these networks are more expensive than Gigabit Ethernet. Both provide low-latency, high-bandwidth, end-to-end communication between two processes in the cluster.

In the high performance computing area, the Message Passing Interface (MPI) [81] is the standard that is most commonly used for writing distributed memory parallel applications. To achieve optimal performance in a cluster, it is very important to implement MPI efficiently on top of the cluster interconnect. For networks such as Myrinet that are designed for high performance computing environments, their hardware and software are specially optimized to achieve better MPI Performance. Low cost commodity cluster computers typically use Ethernet for the network interconnect, TCP/IP as the protocol, and MPICH [69] as the communication library for parallel computing, which is an implementation of the MPI standard. However these Beowulf-style commodity clusters typically do not have as good performance for parallel applications that require a lot of interprocessor communication. This is because the MPICH implementation was not designed for commodity machines which use TCP/IP and Fast Ethernet, while the problems with TCP/IP are because it was designed to use in a wide area network instead of a parallel computer where low latency is important.

Architecture	**Count**		**Share %**	
	November 1997	**November 2007**	**November 1997**	**November 2007**
Constellations	10	3	2.0 %	0.6 %
MPP	226	91	45.2 %	18.2 %
Cluster	1	406	0.2 %	81.2 %
SMP	263	0	52.6 %	0

Table 1.1 : Comparison of the architecture for high performance computers in the list of Top 500 supercomputers for year 1997 and 2007 [91].

Interconnect	Nov 1999	Nov 2003	Nov 2007
Ethernet	1.8 %	22.4 %	54.0 %
Myrinet	8.6 %	38.6 %	3.6 %
Quadrics	0.2 %	5.2 %	1.8 %
Infiniband	0 %	0.6 %	24.2 %
Crossbar	23.4 %	7.4 %	1.2 %
SGI NUMAlink/flex	0 %	7.4 %	2.2 %
SP Switch	27.6 %	12.6 %	4.6 %
Cray Interconnect	11.2 %	2.0 %	2.2 %
Other	1.0 %	3.0 %	6.2 %
N/A	26.2 %	0.8 %	0 %

Table 1.2 : Percentage of the most used interconnects for supercomputers in the TOP500 list [91].

1.2 MPI communications performance

Much research work has focussed on the communications performance of cluster computers with different network technologies. Previous research has found that the common problems that frequently occurs in networks that use TCP/IP is network congestion and packet loss. This is because TCP/IP applies Retransmit Timeouts for packets that fail to deliver, so the unsent packets need to wait for a certain time to be redelivered. Since TCP/IP is designed for use in wide area networks the default resend time is much longer than is suitable for tightly coupled processors, although this can be customized. Interestingly, there were several previous research papers that found that Ethernet can perform as well as more expensive networks like Myrinet if additional software and/or particular hardware (e.g. high-end network interface cards) is used, with a low-latency custom protocol such as VIA [33, 44, 111] or Genoa Active Message Machine (GAMMA) [38, 39]. However, it would be easier and more practical if MPI performance using Ethernet with TCP/IP and MPICH could be improved without needed any additional software or hardware.

Table 1.3 compares the performance for different protocols in different types of networks. The purpose of this table is to show that the causes of high latency and low bandwidth are not just because of the networks, but also due to the choice of protocol. Based on Table 1.3, TCP/IP with Fast Ethernet has a latency of 103 microsec and GAMMA with Fast Ethernet has a latency of 12.7 microsec. These results show that using GAMMA can significantly improve the latency performance of Ethernet so that it is close to the latency in Myrinet networks, which is 10.0 microsec.

Platform	**Latency (•s)**	**Bandwidth (Byte/sec)**
GM – Myrinet	10.0	100.0
GAMMA – Gigabit Ethernet	9.6	90.0
GAMMA – Fast Ethernet	12.7	12.2
VIA – Fast Ethernet	27.0	12
TCP – Fast Ethernet	108	10.0
TCP – Gigabit Ethernet	105	62.0
TCP – Myrinet	103	42

Table 1.3 : Protocol Comparison (Ping-Pong application). Results are from [40, 115, 116, 132].

There are several different benchmark programs which can be used to measure the performance of MPI communications on parallel computers, such as SKaMPI [21], Pallas MPI Benchmark [65], MPBench [19] and Mpptest [17]. The latest benchmark software suite that has been developed for measuring MPI performance is MPIBench [1,2], which has several important improvements over the existing benchmark software and provides additional results such as distributions of communication times and communication times for different processors, that can reveal more details of communication performance.

1.3 Research Rationale

Although most of the existing MPI benchmark software is widely used, Grove [8,9] pointed out that all of these benchmarks have one or more of four main inadequacies. Firstly, they use relatively coarse grained clocks for measuring the communication time, so they will only give average results over a high number of repetitions for the communication routines being measured. Secondly, the timing for the benchmarks does not use a clock that is synchronized across all processors, which means that measurements of point-to-point communication use a ping-pong test to measure the total round-trip time for two send_recv messages, rather than a single send_recv message, and results for collective communications cannot be given for different processes, only for the slowest process. Thirdly, the collective communication measurement is taken using a ping-pong measurement between two processors, which cannot show the effects of contention and non-uniform communication times that can occur when multiple processors are used. Finally, previous benchmarks were not designed for clusters of SMPs, so users have to be careful to ensure that the communication they are measuring is done between nodes, not between processors within a node.

There has been no detailed comparison of MPIBench with existing MPI benchmarks, and no study has previously been done comparing the techniques used and the results obtained for all of the different MPI benchmarks. Thus, the current research has compared MPIBench to the other existing MPI benchmark software. Consequently, from the results of the comparisons of the results and functionality of the different MPI benchmarks, improvements and changes have been made to MPIBench.

Grove developed MPIBench [1,2] to overcome these limitations of existing MPI benchmarks. He showed that it was a very useful tool in analyzing the performance of different MPI implementations on different parallel computer architectures, and successfully used it to identify performance problems in three different MPI libraries [1]. However his research was mainly focussed on using the distributions of communication times that MPIBench provides in order to provide more accurate modeling and estimation of parallel program performance than can be obtained using average times for MPI communication routines. One of the main motivations for the research presented in this thesis is to extend Grove's initial work by making improvements to MPIBench and applying it more widely to investigate MPI performance on a variety of different parallel computer architectures, communications networks, and MPI implementations. The aim is to gain a better understanding of the capabilities and advantages of MPIBench for the analysis of MPI performance as well as potentially identifying and analyzing performance issues with different parallel computers and MPI libraries.

Grove used MPIBench to measure the performance of a few distributed memory parallel computers [8]. He found there were several problems in performance for MPICH on commodity Beowulf-type clusters using Fast Ethernet and TCP/IP, due to the protocol, network congestion and other problems related to the operating system and MPI implementation [8, 9]. However, there was no further work to investigate possible solutions to the loss of performance. Grove also measured the MPI performance of networks designed for parallel computers, using MPIBench on a Sun cluster using Myrinet and a DEC/Compaq/HP Alphaserver cluster using Quadrics QsNet, with both clusters having quad-processor SMP nodes. One particular objective of this thesis was to do a detailed analysis of the MPI performance of Ethernet in cluster computers and to try to find ideas for improving its performance. In order to achieve the objective, this thesis will use MPIBench to compare the performance between Ethernet with TCP/IP and Myrinet with GM using the same cluster. Grove found some interesting results when comparing Ethernet and Myrinet networks, particularly to do with TCP/IP and retransmit timeouts, however these measurements were done on two different machines. Using the same cluster would provide a much better comparison. Also, since Grove's work was done, new versions of MPICH have been released that provide significant improvements in the implementation of most of the collective communication routines [11].

The main purpose of this comparison is to obtain insight into the problems that occur in the Ethernet network, particularly for TCP/IP, and the MPI implementation, which in the case of MPICH is designed for parallel computers with a high-speed communications network and low latency communication protocol. The outcomes from the analysis may provide ideas on how to improve the communication performance for commodity cluster computers with Ethernet networks and TCP/IP. Therefore, the proposed research will seek an answer to what circumstances TCP/IP and MPICH cause a problem, why it happens, and how to solve the problem or to improve performance.

Previous work on improving communication performance for Beowulf clusters has focused more on designing or using new protocols such as VIA or GAMMA to replace TCP/IP on Ethernet networks and developing new implementations of MPI to use these protocols. These approaches have merit but so far they have had very little uptake. This is probably because a new protocol will require a lot of effort to develop new software to make it compatible with the commodity cluster computer, for example new drivers for the many different types of Ethernet cards and operating systems, and new MPI libraries. It would be useful if some of the problems of TCP/IP and MPICH could be fixed and the MPI performance of commodity clusters with Ethernet networks could be improved.

Another aim of this thesis is to analyze the MPI performance of a large shared memory machine and compare it to a distributed memory cluster with a fast communications network. Large shared memory machines such as the SGI Altix offer the potential for very good MPI performance compared to a cluster with a high-speed communication network, although they are significantly more expensive, However, little work has been done comparing MPI performance on large shared memory machines with distributed memory clusters. In 2004 the Australian national computing facility migrated from a large AlphaServer SC with Quadrics QsNet network, which was one of the fastest distributed memory communications networks at the time, to a large shared memory SGI Altix. This offered a good opportunity to compare MPI performance on these two machines, which was of particular interest to the users of these machines and also of more general interest in terms of comparing MPI performance between large shared memory and distributed memory machines.

1.4 Research Aims and Overview

Fundamentally the aims of the study were as follows:

1. To compare MPIBench with the other existing MPI benchmark software. The comparison will test the scalability, functionality and usability of MPIBench compared with the existing benchmark software.

2. Based on the comparison results, improvements and changes can be done to MPIBench.

3. To analyze the performance between Myrinet with GM and Ethernet with TCP/IP on a high performance cluster computer. Results obtained from the test will be analyzed and may provide ideas on how to upgrade the communication performance for Ethernet network in a commodity cluster.

4. Based on the results of the comparison of different cluster interconnects, investigate possible approaches to improving communications performance, particularly for Ethernet networks.

5. MPI performance evaluation of a large shared memory machine. SGI Altix 3000, and comparison with a high-end cluster with a fast communications network.

An overview of the work that was done in this thesis to fulfill these aims is given below.

1.4.1 Comparison of Different Benchmark Software

There are several benchmark programs that have been developed to measure the performance of MPI on parallel computers. Each of the MPI benchmark programs has its own specialty. However, there have been few comparisons done between the different benchmarks, and no detailed, comprehensive analysis and comparison of the functionality, measurement techniques and results produced by all the different benchmarks. Furthermore, the MPI benchmark programs were primarily designed for, and have mostly been used on, distributed memory machines. However it is interesting to measure MPI performance on shared memory machines such as the SGI Altix, which has become a popular system for high-performance computing. The hierarchical non-uniform memory architecture (NUMA) that is typical of large shared memory machines means that analysis of the performance of shared memory machines is likely to be more complex than distributed memory machines, which are typically clusters with fairly uniform communications architecture.

Thus, this study provides a comparison of techniques used and functionality of each benchmark, and also a comparison of the results on a distributed memory machine and a shared memory machine. All of the most commonly used MPI benchmarks will be compared in this analysis.

1.4.2 Improvements to MPIBench

One of the objectives of the comparison analysis between MPI benchmarks was to identify any weaknesses of MPIBench compared to other MPI benchmarks and to use this information to make improvements to MPIBench. MPIBench has been tested on the SGI Altix (which uses a CC-NUMA architecture) and distributed memory architecture with two different types of interconnect, Myrinet and Ethernet. Many tests were done, which has helped to identify problems in installing and running MPIBench and to make it more portable and robust. The new version of MPIBench is available online at [2].

The analysis from the MPI benchmark comparisons revealed several disadvantages in MPIBench and also in the course of doing the work presented in this thesis some additional useful tools have been added to MPIBench and a number of bugs and problems have been spotted and fixed. One of the issues that has been addressed is regarding the cache effect, whether the cache should be used or not during taking of measurements. The procedure of compiling and running the program has also been improved by adopting the approach of most of the other MPI benchmarks, by providing a default option for running the benchmark programs using defaults for configurable parameters such as the range of message sizes for each communication routine. Several new settings have also been included (with default options), including the ability to choose MPI_Wtime instead of the globally synchronized clock provided by MPIBench,

1.4.3 Performance Analysis and Investigation of Communication Performance on Different Communication Networks

Most modern parallel computers are clusters using Myrinet or Ethernet communication networks. Several studies have been published comparing the performance of these two networks for parallel computing, however these focus on average performance, and do not address the distributions of communication times, which can have long tails due to contention effects. In the case of Ethernet with TCP, retransmit timeouts (RTOs) can also occur. Slow communication events may have significant impact, particularly for applications requiring frequent synchronization, where the performance is determined by the slowest process. This study used MPIBench to analyse the distributions of communication times for standard MPI routines on Ethernet with TCP and Myrinet with GM communications networks on the same cluster, and study the scalability of the distributions as the number of communicating processes is increased, and the effect of RTOs for Ethernet with TCP. One of the goals of this work was to investigate in more detail the effect of these RTOs on Ethernet performance, and how much could be gained from reducing the effects of RTOs.

This study provides a comparison of the performance of MPI communications for Myrinet with GM and Fast Ethernet with TCP networks on the same cluster. Measurements were done for both point-to-point and collective communications for up to 200 CPUs (100 dual CPU nodes), which allows in depth analysis on the scalability of the two networks to large numbers of processors.

1.4.4 Analysis of Algorithm Selection for Optimizing Collective Communication with MPICH for Ethernet and Myrinet Networks

This study was motivated by some strange results for certain collective communication routines in the performance analysis outlined in section 1.4.3. In some cases, larger message sizes were giving smaller communication times. So, more work was done on analyzing MPICH in order to understand the results.

MPICH is one of the main implementations of the MPI standard. Recent versions of MPICH combine the best algorithms known for each MPI collective communication, and those multiple algorithms are differentiated based on message sizes. The message sizes mainly divide into two, the short-message algorithms aim to minimize latency, while the long-message algorithms aim to minimize the bandwidth. Currently, the message sizes where the algorithm changes in MPICH are the experimentally determined change-over points based on the work of Thakur et al. [11], which used an IBM SP and a

Linux cluster machine connected with Myrinet, both with one processor per node. In the paper, they acknowledged having a plan to determine automatically the algorithm change-over points based on system parameters, since the optimum change-over point probably will be different for parallel computers with different architectures, and particularly with different networks. However, the MPICH source code shows that the message sizes where the algorithm is changed are still defined as constants and hard coded.

The aim of this study is to investigate the feasibility of using MPI benchmarks to provide an automated process for selecting the optimal choice of collective communication algorithms for a particular parallel computer and communication network, and to see if this approach is worthwhile by comparing the performance of the optimized MPICH implementation with the current MPICH implementation where the algorithm selection is hard coded. So, this study measured performance over a range of message sizes for all of the different algorithms for all of the most common use collective communication routines in MPICH that use multiple algorithms.

Measurements were done on a cluster of dual processor machines using two different networks, Myrinet with GM and Ethernet with TCP. In order to compare the different algorithms for all message sizes, the MPICH code was modified so that the change-over points could be modified. For each collective communication routine, an MPI benchmark such as MPIBench can be run to measure the performance for each possible algorithm, by varying the change-over parameters to ensure that only a single algorithm is used for each benchmark run. Then the benchmark results for all the different algorithms for a particular collective communication routine can be compared and the optimal change-over points for that particular parallel computer can be determined.

1.4.5 Performance Evaluation on ccNUMA Shared Memory Machine SGI Altix 3000

The SGI Altix [70,27] is a cache coherent, non-uniform memory architecture (ccNUMA) shared memory multiprocessor system that is a popular machine for high-performance computing, with several large systems now installed, including the 10,160 processor Columbia machine at NASA. In Australia, a 1680 processor Altix (the APAC AC) has recently replaced an ageing AlphaServer SC [72] with a Quadrics network [20] (the APAC SC) as the new peak national facility of the Australian Partnership for Advanced Computing (APAC) [84], and was number 26 in the June 2005 list of the Top 500 supercomputers [91]. There are several other Altix machines at APAC partner sites, including two systems with 160 processors and another with 208 processors.

Most parallel programs used for scientific applications on high-performance computers are written using the Message Passing Interface (MPI), so the performance of MPI message passing routines on a parallel supercomputer is very important. Shared memory machines such as the Altix typically have very high-speed data transfer between processors, however this will only translate into good MPI performance if the MPI library can efficiently translate the distributed memory, message-passing model of MPI onto shared memory hardware. It is therefore of interest to measure the performance of MPI routines on a shared memory machines such as SGI Altix, and to compare it with a distributed memory supercomputer with a high-end communications network. This study will provide results for MPI performance on the SGI Altix, and comparisons with similar measurements on the AlphaServer SC with a Quadrics network.

1.5 Thesis Outline

This section will explain the organization of this thesis. Chapter 2 presents a basic introduction to parallel architectures, parallel programming and interconnects. The idea of this chapter is to briefly give the required information on parallel computing. Next, Chapter 3 discusses the MPI benchmark comparison on both distributed memory and shared memory ccNUMA architecture machines. Chapter 4 discusses the improvements to MPIBench that have been made, mainly based on the comparison results from Chapter 3. The performance comparison between Ethernet and Myrinet is described in Chapter 5. Chapter 6 investigates how MPICH performance can be improved by analysis of the threshold points changing between different algorithms for collective communication routines, with results for Myrinet and Ethernet networks. Chapter 7 presents the results of the performance evaluation for the a ccNUMA shared memory machine, the SGI Altix 3000, and comparison with a high-end distributed memory machine. Finally, Chapter 8 will conclude the thesis and discuss future work that could be done.

CHAPTER 2

Parallel Computing

Fundamentally, the concept of parallel computing is to have more than one processor in the same computer. The use of multiple processors in the same computer system introduces some additional requirements on the architecture, software and hardware of the computer. This chapter will explain relevant issues related to parallel computers, parallel programming, interprocessor communication and the evaluation of communication performance.

2.1 Parallel Computers

The taxonomy of Flynn [109] has classified computers into four categories. The classification is based on the way instructions are manipulated and the flow of the data streams. Single Instruction Single Data (SISD) is a conventional system that contains one CPU and the parallelism is incorporated at the level of the arithmetic operations in the central processing unit, for example vector pipeline machines. Single Instruction Multiple Data (SIMD) is where each instruction may operate on more than one data element simultaneously. Most SIMD machines comprise very large numbers of custom processors, which is often called massively parallel processing (MPP). The connectivity between processors depends on the actual machine but it is usually very tight so that there is rapid interchange of data between neighboring processors. The array and pipelined computers are examples of this type [109].

Multiple Instruction Single Data (MISD) theoretically has multiple instructions for single stream of data, however this type of machine has not yet has been constructed. Finally, Multiple Instruction Multiple Data (MIMD) machines can execute different instruction streams in parallel on different data. Interconnection of these machines is much looser than in SIMD architecture.

SIMD and MIMD machines are the two main classes of parallel computers. On SIMD machines, the same task, usually of small granularity, is executed simultaneously on different data, while for the MIMD machines, different tasks can be executed concurrently on different processors. The distinctive aspect of SIMD execution consists of the control unit broadcasting a single instruction to all processors, which execute the instruction in lockstep fashion on local data. The MIMD architecture consists of multiple processors that can execute independent instruction streams. Thus, MIMD computers support parallel solutions that require processors to operate in a largely autonomous manner. Modern supercomputers are virtually all MIMD architecture, so this thesis will focus on this model of parallel computing.

MIMD parallel computers can be divided into two distinct classes: shared memory machines, including symmetric multiprocessors (SMPs), and message-passing multiprocessors or multicomputers, including clusters. Shared memory machines have a set of processing elements and a pool of memory available to all processors. The processors have access to a large global random access memory of which they have the same view. Message-passing multiprocessor systems consist of a number of identical processors where each processor has its own local memory. These systems are also known as local memory systems, loosely coupled systems or distributed memory systems [160]. Each processor is provided with a small private random access memory and interconnects. The processors have no direct access to the memory of other processors, access is only via message passing between the processors.

There are also parallel computers that combine both technologies, which is becoming more common. Most of the largest and fastest computers in the world today make use of both shared and distributed memory architectures, with SMP nodes (usually with multi-core processors) connected by a message-passing communications network.

Parallel applications in turn can be classified as a fine-grained or coarse-grained depending on the frequency of communication between various processing nodes [160, 162, 163]. The fine-grained parallel applications involve relatively small amounts of computational work between communication events, while the coarse-grained involve relatively large amounts of computational work between communication events [162, 163]. More detailed discussion on parallel applications and programming is in Section 2.3, while Section 2.1 and Section 2.2 provide more detail on parallel architectures.

2.1.1 Shared Memory MIMD Systems

All modern computer systems have cache memory, high-speed memory closely attached to each processor for holding recently referenced data and code. Such cache memory is used because the speed at which a processor can make references to memory locations greatly exceeds the time that main memory requires to respond. A higher-speed, but smaller, cache memory can be matched more closely to the speed of the processor. Systems may even have more than one level of cache memory; a small first-level cache connecting the processor and a larger second-level cache between the first-level cache and the main memory. Programs consist of executable instructions (code) and associated data. It is current practice that executable instructions are not altered when the program is executed, the CPU just reads and executes the instructions. In contrast, the data may be read and altered by different processors [28].

This may cause significant complexities to the system design with a cache and affect the performance. When a processor first references a main memory location, a copy of its content is transferred to the cache memory associated with the processor. Suppose the information being brought into the cache is data. When the processor subsequently references the data, it accesses the cache for it in the first instance. If another processor references the same main memory location, a copy of data is transferred to the cache associated with that processor, thus creating more than one copy of the data. This is not a problem until a processor alters its cached copy, which writes a new data. Here the cache coherence protocol is needed to ensure that subsequently processors obtain the newly altered data when they reference the data. ***Cache coherence*** protocols use either an update policy or an invalidate policy. In the update policy, copies of data in all caches are updated at the same time one copy is altered. In the invalidate policy, when one copy of data is altered, the same data in any other cache is invalidated, this is more common technique used in modern computers. These copies are only updated when the associated processor makes reference to it [28].

Shared memory systems have multiple CPUs, all of which share the same address space. This means that information on where data is stored is of no concern to the user as there is only one memory accessed by all CPUs on an equal basis. Shared memory machines can be divided into UMA and NUMA, which are the two main classes based on memory access times. The Uniform Memory Access (UMA) is most commonly represented by Symmetric Multiprocessor (SMP) machines, where each processor has equal access and access times to memory. Sometimes it is called CC-UMA (Cache Coherent UMA).

Non-Uniform Memory Access (NUMA) is a different approach in which two or more SMPs with their own local memory are linked by an interconnect that preserves the shared memory access. However in NUMA, not all processors have equal access time to all the memory, since access to the local memory within the SMP is faster than memory access across the interconnect link between the SMPs. If cache coherency is maintained then this is called CC-NUMA (Cache Coherent NUMA).

The advantages of shared memory computers are that global address space provides a simpler programming model, and data sharing between tasks is fast. The disadvantages are that cache-coherent shared memory machines are expensive compared to distributed memory machines, and the lack of scalability between memory and CPUs means that adding more CPUs can geometrically increase traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management [28].

The main problem with shared-memory systems is that of the connection of the CPUs to each other and to the memory. As more CPUs are added, the collective bandwidth to the memory ideally should increase linearly with the number of processors, while each processor should preferably communicate directly with all others without the much slower alternative of having to use the memory in an intermediate stage. Unfortunately, full interconnection is quite costly, growing with $O(n^2)$ while increasing the number of processors with $O(n)$. So, various alternatives have been tried. Figure 2.1 shows some of the interconnection structures that have been used.

Referring to Figure 2.1, a crossbar uses n^2 connections, an Ω-network uses $n\log_2 n$ connections while, with the central bus, there is only one connection. This is reflected in the use of each connection path for the different types of interconnections: for a crossbar each data path is direct and does not have to be shared with other elements. Crossbar or multistage is a network in which all input ports are directly connected to all output ports without interference from messages from other ports. In a one-stage crossbar this has the effect that for instance all memory modules in a computer system are directly coupled to all CPUs. This is often the case in multi-CPU vector systems. In multistage crossbar networks the output ports of one crossbar module are coupled with the input ports of other crossbar modules. In this way one is able to build networks that grow with logarithmic complexity, thus reducing the cost of a large network. In case of the Ω-network there are $\log_2 n$ switching stages and as many data items may have to compete for any path. For the central databus all data has to share the same bus, so n data items may compete at any time [137, 147].

The bus connection is the least expensive solution, but it has the obvious drawback that bus contention may occur thus slowing down the communications. Various intricate strategies have been devised using caches associated with the CPUs to minimise the bus traffic. This leads however to a more complicated bus structure which raises the costs. In practice it has proved to be very hard to design buses that are fast enough, especially where the speed of the processors have been increasing very quickly and it imposes an upper bound on the number of processors thus connected that in practice appears not to exceed a number of 10-20. In 1992, a new standard (IEEE P896) for a fast bus to connect either internal system components or to external systems has been defined. This bus, called the Scalable Coherent Interface (SCI) should provide a point-to-point bandwidth of 200-1,000 Mbyte/s. It is in fact used in the HP Exemplar systems, but also within a cluster of workstations as offered by SCALI. The SCI is much more than a simple bus and it can act as the hardware network framework for distributed computing [149, 150].

A multi-stage crossbar is a network with a logarithmic complexity and it has a structure which is situated somewhere in between a bus and a crossbar with respect to potential capacity and costs. The Ω-network depicted in Figure 2.1 is an example. Commercially available machines like the IBM eServer p575 and the SGI Altix 4000 use such a network structure [137]. For a large number of processors the $n\log_2 n$ connections quickly become more attractive than the n^2 used in crossbars. Of course, the switches at the intermediate levels should be sufficiently fast to cope with the bandwidth required. Obviously, not only the *structure* but also the *width* of the links between the processors is important: a network using 16-bit parallel links will have a bandwidth which is 16 times higher than a network with the same topology implemented with serial links.

In all present-day multi-processor vector processors crossbars are used. However, when the number of processors is increased, technological problems might arise. Not only does it become harder to build a crossbar of sufficient speed for the larger numbers of processors, the processors increase in speed over time, compounding the problems of making the speed of the crossbar match that of the bandwidth required by the processors [137, 147].

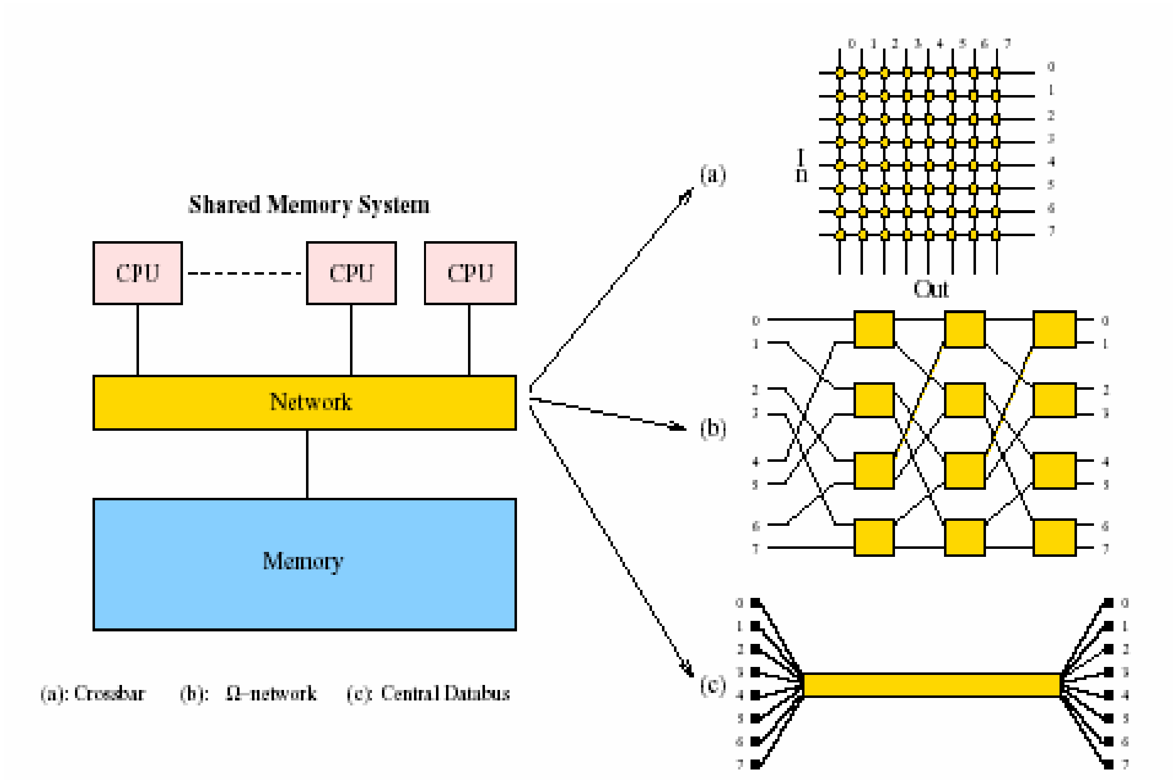

Figure 2.1 : Examples of interconnection structures used in shared-memory MIMD systems [137, 147].

2.1.2 Distributed Memory MIMD Systems

Distributed memory systems vary widely but share a common characteristic, that they require a communication network to connect the memory of different processors. Each of the processors have their own local memory and the memory addresses in one processor do not automatically map to another processor, so there is no concept of global address space across all processors. Since each processor has its own local memory, it operates independently and the changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply. When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated, which is usually done using the standard Message-Passing Interface (MPI) [28, 53, 137, 147]. Synchronization between tasks is likewise the programmer's responsibility. The interconnects used for data transfer vary widely, though it can be as simple as Ethernet.

There are several advantages to the distributed memory architecture. It is more easily scalable to large numbers of processors. If the number of processors is increased, then the size of memory increases proportionately. Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency. It is also very cost effective since it can use commodity, off-the-shelf processors and interconnects. However the disadvantages are that the programmer is responsible for many of the details associated with data communication between processors. It may be difficult to map existing data structures, based on global memory, to this memory organization.

Distributed memory computers can also be built from scratch by using mass produced PCs and workstations or servers. These commodity cluster computers are referred to by many other names, such as Beowulf clusters, COWs (clusters of workstations), and NOWs (networks of workstations). They are much cheaper than traditional MPP (massively parallel processing) supercomputers that used mostly custom components. Referring to Table 1.1, 81.2% of the supercomputers in the November 2007 list of the Top 500 supercomputers in the world are clusters, up from only 0.20% ten years ago [91].

Figure 2.2 shows some examples of common network topologies for distributed memory machines, the hypercube and fat tree topology. It shows the hypercube with 2d nodes where the number of steps to be taken by a message between any two nodes is at most d. The dimension of the network grows logarithmically with the number of nodes. It is also possible to simulate any other topology on a hypercube such as tree, ring, 2-D and 3-D. In practice, the exact topology for hypercubes is not as important anymore because all systems now employ the “wormhole routing” technique. This technique will send a header message from node i to j, resulting in a direct connection between these nodes. As soon as the connection is established, all the data are sent through this connection without disturbing the operation of the intermediate nodes. Another cost effective way to connect a large number of processors is using a fat tree topology. In theory, a simple tree structure for a network is sufficient to connect all nodes in a computer system. However, in practice the congestion occurs at the tree root, since messages have traversed at the higher levels in the tree structure before being distributed to the target nodes [137, 147]. The fat tree compensates for this limitation by providing more bandwidth in the higher levels of the tree.

Hypercube networks used to be common in MPPs, but are rarely used in clusters, which mostly use switches (or routers) to connect nodes. Ethernet, Myrinet and Infiniband all have switches that allow many nodes to be connected to the ports on each switch. The switches need to be connected to each other in some topology, which is usually some variant of a fat tree. This thesis will focus mostly on distributed memory systems using fat tree topology.

NOTE: This figure is included on page 22 of the print copy of the thesis held in the University of Adelaide Library.

Figure 2.2 : Examples of common networks for Distributed Memory machine [147].

2.1.3 Distributed Memory System with SMP Nodes

Small SMP servers with 2 or 4 processors have been used in clusters for several years now. Increasing cache sizes, memory bandwidth and bus speeds have meant that clusters with SMP nodes have become more popular over the past few years. The recent development of commodity multi-core processors from Intel and AMD has only increased this trend, and the largest and fastest supercomputers in the world today are mostly clusters with SMP nodes, that employ both shared and distributed memory architectures. The availability of quad core processors means that new clusters often have 8 and 16 CPUs per node, and the number of CPUs per node is likely to increase in future.

The common programming approach used for this type of machine is message passing. The shared memory component is usually a cache coherent SMP machine, so that the processors on a given SMP can address that machine's memory as global. The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, message passing network communications are required to move data from one SMP to another. Current trends indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future. There are some issues with message passing using this type of architecture, in particular that there may be communication bottlenecks since all CPUs on a node typically share a single network interface card (NIC). Also, commonly used MPI implementations such as MPICH implement collective communication routines in a way that does not take into account the fast internode communication (via shared memory), slower intranode communication (over the network), and NIC bottlenecks.

Even large shared memory machines such as the SGI Altix have adopted a similar architecture, with a NUMA which is basically SMP nodes connected by a very fast network, although in these machines there is additional custom hardware to enable all processors to share all memory and to ensure cache coherency. Figure 2.3 shows a cluster of four CPUs are connected by crossbar. This thesis will study the performance of a shared memory machine, the SGI Altix 3000, which has an SMP component (called a C-brick) consisting of four CPUs which are connected in a hierarchical architecture, with C-bricks grouped together with a router (called an R-Brick) and these connected in turn by a Metarouter. A further explanation of the SGI Altix 3000 is in Section 3.5.1 and in [22, 70, 124].

Figure 2.3 : Block diagram of a system with a “hybrid" network: clusters of four CPUs are connected by a crossbar [147].

2.2 Cluster Computer Interconnect

In the early days of clusters, Fast Ethernet was widely used as an interconnect since it was available as an inexpensive, off-the-shelf commodity component [102, 105, 107, 146]. Gigabit Ethernet is a version of Ethernet technology that offers one Gigabit per second (1 Gbps) raw bandwidth, which has 10 times higher bandwidth than Fast Ethernet and now available as a commodity, although the latency is similar, particularly when using TCP/IP. 10 Gbps Ethernet is becoming available and is likely to be an important future technology for cluster networks and expected to become commodity within a few years. 10 gigabit Ethernet is still an emerging technology, and it remains to be seen whether it can be used effectively with low-latency communications protocols and can scale well with lots of nodes, and how long it will take to reduce to a commodity price similar to Gigabit Ethernet.

The invention of lower latency and higher bandwidth interconnects gave more alternatives for cluster computers. Some networks developed by supercomputer vendors were proprietary and used only in the particular vendor’s computers. However, several network technologies aimed at parallel computing were developed which use a standard

PCI interface and hence could be used for any cluster computer. These include Myrinet, Giganet, SCI and Quadrics [6, 110, 145, 150]

Myrinet was developed by Myricom [71] based on communication and packet-switching technology originally designed for massive parallel processors (MPPs). It was the most popular high-end interconnect used to build clusters for several years, but has declined in popularity over the last couple years. Apart from the high bandwidth of over 1000 Mbps, the main advantage is that it is entirely operated in user space, thus avoiding operating system interference and the delays that come with it. It can also use the light-weight communications protocol called GM which has been designed for parallel computing, rather than the much more heavyweight TCP/IP. These two innovations mean that Myrinet has a low message passing latency of around 10 microseconds, ten times better than TCP/IP over Ethernet. Recently, Myricom supplies Myrinet components and software in two series: Myrinet-2000 and Myri-10G. Myrinet-2000 (which is used for this thesis) is a superior alternative to Gigabit Ethernet for clusters, whereas Myri-10G offers performance and cost advantages over 10-Gigabit Ethernet, while still supporting 10G Ethernet standards.. Myri-10G uses Myrinet Express (MX) protocols and software to provide lower latency and higher performance than 10G Ethernet, in a similar way to GM in Myrinet 2000. Myricom provides MPICH-MX, an MPI library for Myri-10G. The performance for Myrinet 2000 with one-port NIC is 10 microseconds latency and approximately 230Mbytes/s for the bandwidth. The emergence of Myri-10G with the latency of 2.1 microseconds and 1215Mbytes/s (MX or MPI unidirectional rate) significantly improves the performance of Myrinet [71, 137]. The use of Myrinet’s lightweight MX protocol over 10G Ethernet has a great potential in providing very high bandwidth and low latency at commodity prices. Refer to [166] which shows that Myri-10G (1.2Gbytes/s) produce a higher bandwidth compared to Quadrics (0.9Gbytes/s) and almost similar bandwidth with Infiniband (1.3Gbytes/s), while for latency Myri-10G (2.1μs) is lower than Quadrics (2.7μs) and Infiniband (4.0μs).

QsNet is a product of Quadrics and like Infiniband and Myrinet the network has effectively two parts: the ELAN interface cards, comparable to Infiniband Host Bus Adaptors or Myrinet's Lanai interface cards, and the Elite switch, comparable to an Infiniband switch/router or a Myrinet switch [137]. Quadrics is the most expensive compared to other interconnects, as shown in Table 1.1Quadrics also offers 10 Gbit Ethernet cards and switches under the name QSTenG. As yet Quadrics does not seem to consider developing multi-protocol products like Myricom's Myri-10G and Infiniband [137]. Recently, Quadrics has announced $300 per-port price tag for its 24-Port CX4 10Gbps Ethernet Switch and also their pricing strategy for its QsTenG-TG201 Switch, the latest member of the QsTenG family, designed for smaller networks. If compared with the Quadrics price per port from Table 1.1 it is almost 16 times more expensive compared to QsTenG. Probably in future Quadrics will abandon their proprietary networks and moving to 10 Gigabit Ethernet, since it is much cheaper and the performance is greater.

Recently, Infiniband [144] has entered the high performance computing market. Unlike Myrinet and QsNet, Infiniband is an industry standard that was developed as a generic interconnect for inter-process communication and I/O, rather than specifically designed for parallel computing. The Infiniband Architecture [144] defines a System Area Network (SAN) for interconnecting servers with remote storage, networking devices and other servers, as well as for use inside servers for interprocessor communications. The Infiniband standard, which is based on VIA, was designed to eventually replace the PCI bus, although with the popularity of the recent PCI-X standard and the development of 10 Gbps Ethernet, Infiniband has not yet become as popular (and hence as cheap) as was originally expected. However, its high performance, low latency and scalability make it very attractive as a communication layer for high performance computing, and this is the area where Infiniband has become most popular. Currently, Infiniband can be considered as the main HPC interconnect, since it is fast and relatively cheap. However, the cost of Infiniband is still over the standard of commodity price, so may not be as cheap or as fast as 10Gigabit Ethernet in future.

At the time this research work was done we did not have access to machines with these technologies (Gigabit/10 Gigabit Ethernet, Myri-10G, QSTenG and Infiniband), and some of them were not available. So, a detail analysis and comparison on the performance of MPI between new networks and cluster with multicore architecture will be part of our future work.

There has been lots of research comparing the communications performance of different networks for clusters, Labosco [6] compared the performance between Fast Ethernet, Giganet and Myrinet. Grove [9] compared the performance between two different clusters using Fast Ethernet. Chen [25] compared between Gigabit Ethernet and

Myrinet. A study conducted by groups of researchers from Ohio State University and Ohio Supercomputer Centre [130] showed that Infiniband network can provide better performance than Quadrics and Myrinet with the use of the PCI-X bus. Other work such as [91, 93, 101, 114, 129, 130] either compared clusters using several different networks or investigated network performance including some analysis with MPI or other message-passing protocol.

Interconnect	**Bandwidth (MBit/s)**	**Latency (µs)**	**Cost / port**
10 Gbps InfiniBand Switch [148]	800	7 – 10	$495
QsNet (Quadrics) [116]	360	5	$4770
Myrinet (Myricom) [116]	245	8	$2050
Gigabit Ethernet [116]	125	30 – 100	$477
Fast Ethernet [116]	12	100	$28

Table 2.1 : Comparison for bandwidth, latency and cost between different interconnect [148, 116].

2.3 Parallel Programming

Parallel programming involves decomposing an algorithm or data into parts, distributing the parts as tasks which are worked on by multiple processors simultaneously, and coordinating the work and communications of those processors. Parallel programming needs to consider the type of parallel architecture being used and the type of interprocessor communications and synchronization used. There are many methods of programming parallel computers, the most common being shared memory, message passing and data parallel. Shared memory programming relies upon multiple processes or threads sharing common memory space, whereas message passing programming is a more general paradigm in which processes have direct access to local memory and message passing is used to access the memory of other processes. These models are machine or architecture independent, any of the models can be implemented on any hardware given appropriate operating system support. Distributed memory machines such as clusters to not provide shared access to a common memory space, although shared memory programming can be enabled on these machines using a software emulation of shared memory hardware, however this approach has limited performance and scalability. An effective implementation of a parallel program will utilize the method that closely matches with the target hardware and provides the user ease in programming for the particular application.

The data parallel model is a high-level parallel programming model in which processing of all data elements can conceptually be done concurrently by multiple processes in a SIMD style. The data is distributed across physical processors and on a distributed memory machine, all message passing is done invisibly to the programmer. Programming with data parallel model is accomplished by writing a program with data parallel constructs in a data parallel language such as High Performance Fortran (HPF) [28, 37, 53] and compiling it with a data parallel compiler. The compiler converts the program into standard code and calls to a message passing library to distribute the data to all the processes and do all the required inter-processor communication.

Message passing [28, 53, 152, 154] is the parallel programming method that will be studied in this thesis. The message-passing model is defined as a set of processes using only local memory and processes communicate only by sending and receiving messages. For data transfer this will require cooperative operations to be performed by each process involved in the communication, for example a send operation must have a matching receive. Programming with message passing is done by linking with and making calls to libraries which manage the data exchange between processors.

A standard portable message-passing library definition called the Message Passing Interface (MPI) [53, 69, 81] was developed in 1994 by a group of parallel computer vendors, software developers, and computer scientists. It is available to both Fortran and C programs and also available on a wide variety of parallel machines. All parallelism is explicit, so the programmer is responsible for parallelism of the program and all inter-process communication, by calling the appropriate MPI library routines. A revised version of the MPI standard known as MPI-2 has been released [136]. There are many different implementations of the MPI standard. Some are portable to most parallel computers, for example LAM [42, 132], OpenMPI [37] and MPICH [69, 136]. Some are specific to particular parallel computers and are often provided by the vendors of those computers, although they may be based on the portable libraries. MPICH was developed jointly by Argonne National Laboratory and Mississippi State University and has been one of the most widely used MPI implementations since the MPI standard was first developed. Recently, MPICH released a new version, MPICH2, which supports MPI-2 and uses a new more extensible software architecture to more easily handle different kinds of networks and protocols.

2.4 MPI Benchmark Software

There are several existing MPI benchmark software packages that are widely used to measure the MPI message-passing performance of parallel computers. The most commonly used are Pallas MPI Benchmarks (PMB) [65] (now known as Intel MPI Benchmarks), MPBench [19], SKaMPI [21], Mpptest [17] and MPIBench [1,2], which is the most recently developed MPI benchmark. All of these benchmarks have similar basic functionality, providing average completion times for the most commonly used MPI communication routines, however the techniques used and the routines measured vary somewhat between the different benchmarks.

The main reason that MPIBench was developed is that previous MPI benchmarks have a number of limitations. Firstly, the use of relatively coarse grained clocks for timing measurements, which forces the benchmarks to average results over a high number of test repetitions. This means that the earlier benchmarks cannot generate distributions that show the variability in completion times that occur due to network contention and other effects, which may have a large effect on program performance. Secondly, the earlier benchmarks are not based on using an accurate, globally synchronized clock on each processor. This means measurements of point-to-point communication have to measure the round-trip time of a ping-pong communication rather than an individual send and receive, and measurements of collective communications can only give the average time for the slowest process, not the times for all the individual processes, which can provide a lot more insight into performance issues. Thirdly, point-to-point communication is only done between two processors, which does not give any insight into the effects of the network hierarchy and contention for large numbers of processors. Finally, none of the communication patterns used in the earlier benchmarks were design to take into account clusters of SMPs. The main issue on this matter is that if care is not taken with process placement, it can lead to the measurement of intra-node communication performance when usually the intention is to measure inter-node communication performance.

MPIBench used new measurement techniques to overcome these problems. It provides a very accurate clock that uses the 64-bit cycle count registers that are available on modern processors. Before and after each measurement phase, MPIBench does a synchronization of the clocks on each process, including interpolation to take into account the clock drift during the time period of the measurements. The use of an accurate, globally synchronized clock means that MPIBench can do accurate timings of individual messages and collective communications and provide completion times for individual processes. It can also provide distributions (histograms) of communication times for MPI routines, with the option of increasing the number of repetitions in order to generate more accurate distributions. MPIBench also allows point-to-point communications to be measured using many processors communicating simultaneously, and ensures that the communication partners are on different nodes in the case of SMP clusters.

2.5 Performance Analysis with MPIBench

Grove [8] used MPIBench to measure the performance of a number of large message-passing parallel computers. The machines involved in the tests were:

- Two different Beowulf clusters of Linux PCs, each with two Intel processors per node, connected by Fast Ethernet, with different switches and network topologies;
- A Sun Technical Compute Farm with four SPARC processors per node, connected with Myrinet;
- A Compaq AlphaServer SC with four Alpha processors per node, connected with Quadrics' QsNet.

In each case, some performance problems were identified in the MPI implementation. Grove's work demonstrated that MPIBench is a useful tool for analyzing MPI communication performance, particularly due to its ability to generate distributions of communication times, which allows more detailed investigation of the causes and effects of contention and variability in communication times.

The main problems identified in the performance analysis were in the Beowulf PC clusters and were primarily due to TCP/IP timeouts and congestion control. There were also problems with the MPI implementation and network congestion for the PC clusters, and to a lesser extent for the other clusters. The measured distributions of communication times showed some unusual results, including some distributions with long tails and outliers with very long communication times. These can result in large delays in message-passing parallel computing, especially when it involves a large number of processors.

In contrast, for Myrinet and QsNet only a small numbers of outlier were observed. This is because Myrinet and QsNet hardware and protocols are highly-tuned to provide good message-passing performance. However there were some problems identified in the implementation of some of the routines, and some bottlenecks due to shared access to the NIC for all the processors on the node. Operating system interrupts were another possible cause for the observed variation of communication times.

One of the problems identified was with MPI_Alltoall, which gave very poor performance for large message sizes and large numbers of processors on the Beowulf clusters. The cause of the problem was identified as an unnecessary bottleneck in the implementation that caused particular problems when using Ethernet networks, due to packet collisions and subsequent resends. A proposed fix to the problem was incorporated into later versions of MPICH.

2.6 Variation of Communication Performance

There has been some research investigating the variation and degradation of message-passing communication times in clusters. Mraz [117] observed performance variation in point-to-point communication in the IBM SP1 and determined that the variation was caused by several factors such as daemons and interruptions from other system activities. He noted that since these operating system interrupts were not synchronized across all the nodes in the parallel computer, their effect on a parallel program would increase with the number of nodes. He proposed multiple techniques to reduce the variance but these required control of interrupts at different levels, process execution priorities and time synchronization during run time. Schaubschlager [118] also recorded a large number of slow message-passing times on nCube-2 and Origin 2000 hardware. This slow message-passing is worse when it involves heavy network load which leads to contention effects. More recent work by Petrini *et al.* [15] succeeded in improving the performance of an AlphaServer system known as ASCI Q, which was the second fastest supercomputer in the world at the time. Their research found that ASCI Q did not perform as expected due to interference caused by several types of daemons run by the operating system and the cluster management and queueing system. Their solution was to confine daemons to the cluster manager, and remove the cluster manager and RMS cluster monitor from each cluster's compute pool. In conclusion, they found that the main cause of the discrepancy is because of the combination of tasks for system activities and applications in a same processor, separation of these activities is a good solution to eliminate the problems.

Based on the research cited above, there are several reasons for loss of performance in message-passing multicomputers, for example, interruptions from the operating system, interruptions at process level, the scheduling technique and the MPI implementation. Based on Grove's analysis [8,9], the main reasons for problems in clusters using Ethernet is because of the use of TCP/IP, which provides a general purpose communication protocol with time outs and congestion control mechanisms that are not tuned for message-passing in parallel programs. Cozzini [119] showed that MPICH using the GM protocol on Myrinet gives much better performance than Fast Ethernet, but using TCP/IP over Myrinet gives a similarly high latency to Ethernet. The communication problems that Grove found in PC clusters using Ethernet might be reduced with the use of new, low-latency communication protocols or modifying the TCP/IP protocol, and improved MPI implementations that are better optimized for commodity Ethernet networks.

2.7 Improving the Communication Performance of Cluster Computers

Cluster computers have revolutionized supercomputing. Connecting inexpensive, commodity servers with fast, off-the-shelf networks is much cheaper than the custom-built MPPs that dominated supercomputing in the 1980s and 90s. However as was seen in the previous sections, inexpensive commodity Ethernet networks do not provide good communications performance, while the high-performance networks that are designed for parallel computing are relatively expensive. The challenge for cluster computing is to develop an inexpensive commodity network that provides low latency and high bandwidth communication for parallel computing. There have been several different approaches to this problem.

Active Messages [45] was one of the earlier efforts to improve the performance for communication in multiprocessor systems. Active Messages is an asynchronous communication mechanism intended to expose the full hardware flexibility and performance of modern interconnection networks. Active messages were aimed at reducing the communication overhead and allowing communication to overlap computation. The advantage of Active Messages over other communication paradigms is that it eliminates the need of intermediate copies of messages along the communication path, thus speeding up communications. Many researchers have applied the Active Message approach and idea to develop a new protocol or hardware (compatible with Active Messages) that can improve the performance of communication for multiprocessor, for example Fast Messages, U-Net, VIA and GAMMA.

The U-Net architecture [43] aims to remove the operating system kernel from the critical path of sending and receiving messages. These activities will eliminate the system call overhead, and more importantly, offers the opportunity to streamline the buffer management, which can now be performed at user-level. Eliminating the kernel from the send and receive path required some form of a message multiplexing and demultiplexing device. However, U-net took a radical approach by removing all protocol-based communication abstractions from the Operating System Kernel, and due to this U-Net has several weaknesses. This is supported by Chiola and Ciaccio [40], who compared the performance of U-Net to their new approach, which is GAMMA, an extension layer in the communication layer for Linux. They found that the level of a virtualization is very low and also the usability of U-Net for parallel programming is quite poor.

Chiola and Ciaccio [38,39,115] have developed Genoa Active Message Machine (GAMMA), which is an efficient communication layer for Linux PC clusters using Ethernet. It is based on Active Ports, a communication mechanism derived from Active Messages [39]. GAMMA Active Ports deliver excellent communication performance at user level, thus enabling cost-effective cluster computing on Ethernet. It was initially developed for Fast Ethernet, but has since been upgraded to support Gigabit Ethernet, and a new driver developed for a Gigabit Ethernet adapter [115]. Moreover, they have applied the Abstract Device Interface (ADI) [38] of MPICH, so that all MPI calls are implemented in terms of ADI functions. Therefore, porting the ADI layer to GAMMA means running the whole of MPICH on top GAMMA. GAMMA provides excellent latency that is comparable to Myrinet, the latency and bandwidth for GAMMA with Fast Ethernet network is 12.7 us and 12.2 Mbytes.

VIA or Virtual Interface Architecture [33, 111] was proposed as a standard communication infrastructure for System Area Networks (SANs) that provides protected, zero-copy user-space inter-process communication. VIA defines a mechanism that bypasses the intervention of the operating system layers and avoid excess data copying during sending and receiving of packets, which solves the problems of many multiple memory copies and use of the operating system for receiving and transmission of packets. It also reduces latency and lowers the impact on bandwidth. Since the introduction of VIA, several software and hardware implementations of VIA have become available, for example Giganet VIA and Servernet VIA. MVICH [42,44,113] is an MPICH-based implementation of MPI for VIA. However, studies of MVICH [42,44] found several weaknesses, and it has never been widely used. In fact VIA has not become a widely-used industry standard for SANs as expected, and has been superceded by Infiniband [122,130], which is based on VIA, and by the popularity of the PCI-X bus [76].

BIP [36] is the abbreviation for Basic Interface for Parallelism, an interface for network communication targeted towards message-passing parallel computing. The idea in BIP was to provide protocols with low level functionalities. The functionalities of BIP is similar to MPI. The basic idea of BIP is to build a library interface accessible from applications that will implement a high-speed protocol with the fewest accesses to the system kernel. However, to implement BIP for UNIX needs an IP-BIP driver. BIP also has a problem with flow control, since it relies on the hardware flow-control [36].

Based on this literature review, previous research in this area has been more focused on designing new hardware, protocols and software to overcome the limitations of commodity Ethernet networks and TCP/IP for parallel computing. However, these approaches require significant investment of effort, such as designing new protocols, software to support them, new driver software for the wide variety of Ethernet hardware and also new MPI libraries. For examples, GAMMA has released a new driver to support Gigabit Ethernet, but it only works for a particular Ethernet card [115]. The use of VIA required a new MPI implementation (MVICH), and in order to optimize performance, special hardware for VIA was designed, such as Servenet VIA and Giganet VIA. It would perhaps be easier and more cost effective to modify TCP/IP and MPICH to improve their performance for parallel computing on Ethernet clusters.

Some of the features of TCP/IP are unsuitable for new applications such as mobile computing, storage area networks and parallel computing. There has been much recent interest in modifying TCP/IP to make it more flexible, so that different algorithms for handling congestion and packet loss can be applied in different situations, and more dynamically responsive to changes in network conditions. For example, Pope *et al.* [14] have introduced a modified stack model, the “Embedded Inverted Stack” (EIS), which is an instantiation of the generic Compliant System Architecture [41]. Their aim is to provide flexible TCP/IP and they proposed the argument for separation of policy and mechanism, and examined what policies are suitable for TCP/IP stacks, which depends on the types of communication. Currently, their focus is in mobile networking and they have identified the policies and mechanisms needed to support a high level of adaptivity for mobile network devices and applications, however they have also explored the suitability of this model to parallel computing. Future work in this area may have significant implications for cluster computing.

Chapter 3

Comparison of MPI Benchmark Programs on Shared Memory and Distributed Memory Machines

3.1 Introduction

Several benchmark programs have been developed to measure the performance of MPI on parallel computers, such as SKaMPI [21], Pallas MPI Benchmark [65], MPBench [19], Mpptest [17] and MPIBench [1, 2]. Each of the MPI benchmark programs has its own speciality, since the development of new software is usually because of some limitation or inadequacy of the existing software. However, there have been few comparisons done between the different benchmarks, and no detailed, comprehensive analysis and comparison of the functionality, measurement techniques and results produced by all the different benchmarks.

Furthermore, the MPI benchmark programs were primarily designed for, and have mostly been used on, distributed memory machines. However it is interesting to measure MPI performance on shared memory machines such as the SGI Altix, which has become a popular system for high-performance computing. The SGI Altix [70,27] is a cache coherent, non-uniform memory architecture (ccNUMA) shared memory multiprocessor system that is a popular machine for high-performance computing, with several large systems now installed, including the 10,160 processor Columbia machine at NASA. In Australia, a 1680 processor Altix (the APAC AC) has recently replaced an ageing AlphaServer SC with a Quadrics network (the APAC SC) as the new peak national facility of the Australian Partnership for Advanced Computing (APAC) [84], and was number 26 in the June 2005 list of the Top 500 supercomputers [91]. The hierarchical non-uniform memory architecture (NUMA) that is typical of large shared memory machines means that analysis of the performance of shared memory machines is likely to be more complex than distributed memory machines, which are typically clusters with fairly uniform communications architecture.

Thus, this chapter will discuss a comparison of techniques used and functionality of each benchmark, and also a comparison of the results on a distributed memory machine and shared memory machine. All of the MPI benchmarks listed above will be compared in this analysis. It is expected that the results from difference benchmarks should be similar, however this analysis found substantial differences in the results for certain MPI communications, particularly for shared memory machines.

3.2 Related Work

There has been surprisingly little work on comparing the results produced by different MPI benchmark programs. The papers describing the different MPI benchmark programs [1,17,19,21,65] typically provide a discussion of the differences in some of the measurement techniques used by the different benchmarks, but give little or no results comparing measurements from the different benchmark programs on different machines.

Mierendorff et al. [23] compare the results of PMB, SKaMPI, MPBench and Mpptest on an SGI Origin 2000, but only for point-to-point communication and only for 4 CPUs. However they provide useful insights into communication performance issues related to cache effects on ccNUMA architectures.

Unlike the previous related work, the work presented in this chapter provides a comparison of the functionality and the measurement techniques for all the main MPI benchmarks, as well as a detailed, comprehensive comparison of the results produced by the benchmarks. Results are presented for both a shared memory machine, the SGI Altix 3000 [70], and a distributed memory machine, the IBM eServer 1350 Linux cluster [35], for more MPI operations and more processes (up to 128), and a more detailed analysis of the differences in the results between different benchmarks is presented.

3.3 MPI Benchmark Measurement Technique

There are several different MPI benchmark programs that are in common use. They typically measure the average times to complete a selection of MPI routines for different data sizes on a specified number of processors using the following basic approach:

```
loop over different MPI routines
    loop over different message sizes
        get start time
            loop over number of repetitions
                if this is a collective communication routine, do a barrier synchronization
                call the MPI routine
            end loop over repetitions
        get finish time
        average time = (finish time - start time) / number of repetitions
    end loop over message sizes
end loop over MPI routines
```

Figure 3.1: MPI benchmark measurement technique pseudocode

Most benchmarks use the standard MPI timer MPI_Wtime, and get accurate results by making lots of repetitions of the measurements. Most benchmarks have a fixed number of message sizes (at least by default), but some also provide adaptive message length refinement in order to focus on message sizes where the communication time is changing rapidly. Some benchmarks also consider error control mechanisms to handle potentially large variations in communication times that may be caused by external influences, for example operating system interrupts, or other programs that are also using the communications network.

Most benchmark programs measure the time for collective communications on the root process. However, since the root process finishes first for many collective operations, this can bias the results. This is usually avoided by adding a barrier synchronization before each collective communication call. This will add some additional time to the result, but it will be negligible unless the message size is very small.

Most of the benchmarks use *ping-pong* to measure point-to-point communication times, where a process will send a message to another process (the ping) and then receive a message back from the same process (the pong). In this case, only local clock times are needed, instead of a globally synchronized clock. The benchmark program usually divides the result for the ping-pong by two and reports that as the time for a single point-to-point communication.

An important point that can significantly affect the results is whether the message to be sent is in cache memory. Most benchmarks provide an option for specifying whether or not the data to be sent is in cache [2,17,19,21]. The default setting for most benchmarks is that the data is in cache, and they do some preliminary repetitions of the

MPI routine, which are not measured, in order to warm up the cache. In some cases the benchmarks also provide an option to ensure that the message data for each iteration is accessed from main memory instead of cache.

3.3.1 Mpptest

The fundamental design philosophy of Mpptest [17, 63] is that the results of performance benchmarks should be reproducible. To reduce biases due to external influences, Mpptest spreads the test for each message length over the full time of the benchmark run, and measures the minimum average time over a number of repetitions in order to reduce variations. The structure of the measurement process is:

```
loop over number of repetitions
    loop over different message sizes
        get start time
            loop over a small number of iterations
                call the MPI routine
            end loop over iterations
        get finish time
        average time = (finish time - start time) / number of iterations
        if this is the fastest average time yet, accept it
    end loop over message sizes
end loop over number of repetitions
```

Figure 3.2 : Mpptest pseudocode

Mpptest uses MPI_Wtime as a timer. The output is data files which provide message size, average communication time and bandwidth for all MPI communications. It also provides several error control options such as adaptive message length refinement in order to focus on message sizes where the communication time is changing rapidly. Mpptest provides a basic selection of MPI communication routines, which are MPI_Send, MPI_Recv, MPI_Bcast and MPI_Scatter. Although the selection is limited, it has many options to fit a variety of circumstances. Mpptest divides the options into several groups such as those in the following list (note that the options that are listed for each point are only an example - for a complete list of options refer to [17,63]) :

- *Protocol* provides synchronous and asynchronous options;
- *Message Data* provides option to clear the cache and vector data option;
- *Message Pattern* includes roundtrip and head-to-head messages;
- *Message Test Type* considers overlap communication and computation;
- *Message Sizes* provides options to produce data by logscale or dynamic selection of message sizes;
- *Detailed Control of Test* provides options for maximum number of seconds for all tests or the number of times a test is run.
- *Collective Test* provides options for collective communication such as broadcast, scatter.
- *Collective Test Control* which provides specific range of processors to run the collective test, for example from nodes *n* to *m*;
- *Output* which provides choices for output file name and several other document settings to generate the output;
- *Pattern (Neighbour) Choices* which provides choices to measure distance of processors for different topology.

For point-to-point communication, Mpptest does a ping pong and divides the results by two. To counter the problem for MPI collective communication for which the root node finishes first, Mpptest changes the root node at each repetition. By default Mpptest warms-up the cache but it also provides the option to not do this. However, not warming up the cache just means that the cache is cleared for every new message size, but it does not clear the cache for each iteration in the same message size, and this is the only approach that would make a difference in the results.

3.3.2 Pallas MPI Benchmark

Pallas GMBH has recently been taken over by Intel, so the Pallas MPI Benchmark (PMB) is now officially known as Intel MPI Benchmark (IMB) [65], however this thesis will refer it as PMB. PMB is a thorough, well documented and easy to use benchmark program that is commonly used. It provides a wide selection of common MPI routines, and even more are provided in PMB Part-2 [65]. PMB Part-1 measures MPI_Send, MPI_Sendrecv, MPI_Bcast, MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Reduce, MPI_Reduce_Scatter, MPI_Allreduce and MPI_Barrier. For MPI_Bcast and MPI_Reduce, instead of using barrier synchronization to avoid biases due to pipelining effects, PMB changes the root node for each repetition. PMB also provides a multi version for all of the MPI routines that it measures, which will group the process numbers that are specified by user. Based on the documentation, PMB warms up the cache in order to hide initialization overheads of message passing systems. However, for MPI_Bcast and MPI_Reduce the message data will not be in cache since the root node is changed at every iteration.

PMB presents the results in data files that include average communication time and computed bandwidth (data size / communication time) for point-to-point communication in data sizes that are a power of two, and only average communication time for collective communication. PMB uses MPI_Wtime as a timer. PMB synchronizes all the processes using MPI_Barrier before collective communication benchmarks are started and averages the results over the numbers of repetitions. PMB does not provide any specific technique for error control, basically it uses the repetitions to obtain more accurate results. By default only 2 processes are involved for point-to-point measurements and the ping-pong results are divided into two and reported as the time for a single communication.

3.3.3 MPBench

MPBench [19, 64] follows the basic MPI benchmark approach, except that it uses the UNIX timer *gettimeofday()* [80] rather than MPI_Wtime. It measures the performance of the most common MPI routines, which are point-to-point for Bandwidth, Bidirectional Bandwidth, Roundtrip and Application latency measurement, and for collective communications are Broadcast, Reduce, Allreduce, and Alltoall. MPBench synchronizes the processes before the benchmarking is started using MPI_Barrier and then the receiving processes will send a signal to the root after the message is received. MPBench records the measurement time until it receives the completion feedback from each of the processes. MPBench produces data files which provide computed bandwidth for most of the measurements except for application latency and roundtrip measurement, for which the average communication time is provided.

MPBench does not provide any special error control besides allowing the user to change the number of repetitions for every message size. MPBench reports the times for for MPI_Send and MPI_Recv as half the roundtrip time. For the problem of MPI collective communications for which the root node finishes first, MPBench uses MPI_Barrier to synchronize the processes and does not change the root node. By default MPBench warms-up the cache and provides the option for message data not to be in cache, using a technique similar to Mpptest.

3.3.4 SKaMPI

SKaMPI [21, 24, 62, 103] probably provides the most functionality of all the MPI benchmarks, with a large number of user-definable parameters and MPI routines that can be measured. SKaMPI has divided the measurements into five categories: Point-to-Point, Master-Worker, Barrier Measured Collective, Synchronous Measured Collective and Simple. The Point-to-Point category includes all types of Point-to-Point communication such as MPI_Send, MPI_Recv, MPI_Isend, MPI_Bsend and MPI_Sendrecv. The purpose of the Master-Worker category is to test the network throughput and its handling of simultaneous communication, using MPI routines such as MPI_Waitsome, MPI_Waitany and MPI_Any_Source as well as asynchronous send and receive routines. The Barrier Measured Collective category is an older version that uses the standard approach of a barrier synchronization before each collective communication. The new version is the Synchronous Measured Collective approach, which uses a globally synchronized clock to specify the time that each processor should call the collective communication routine, and uses the time taken by the slowest process as the time for each repetition. This is expected to give more accurate results for collective communications, since it eliminates the need for an additional barrier operation, and possible pipelining effects due to processes completing the barrier operation at different times, however it takes about twice as long to run and the effects are only noticeable for small message sizes [24]. Both the new and old versions measure essentially all the MPI collective routines. Finally, the

Simple category covers MPI routines that involve only one process and without any communication, such as MPI_Wtime and MPI_Comm_rank.

SKaMPI has more sophisticated error controls than the other MPI benchmarks. SKaMPI aims to control all the systematic and statistical errors, and has identified that systematic errors occur due to the measurement overhead such as the calling time for MPI_Wtime, while the statistical error is cause by the finite clock resolution, execution time fluctuation and outliers. SKaMPI handles problems cause by external delays such as operating system interrupts by providing the option to ignore the 25% lowest and highest results to get the average. It also allows the user to specify a maximum statistical error (the default is 0.03%), and the measurements are repeated until the statistical error drops below this value, or the number of repetitions reaches a specified maximum value. SKaMPI also allows adaptive refinement of message sizes.

This study uses the synchronous measured collective pattern for the testing because this pattern is the default setting for SKaMPI and also based on their paper [24] this pattern is more accurate and reliable. Below is the pseudocode describing the Synchronous Measured Collective approach in SKaMPI.

```
/ server code /
clock synchronization
repeat
        start synchronous with other nodes
        start_time = MPI_Wtime()
          routine_to_measure()
        end_time = MPI_Wtime()
        finalize_server_routine()
        wait till end of time slot
        collect results from each process, maximum is the result for single measurement
until result exact enough
send stop signal
/client code /
clock synchronization
repeat
        start synchronous with other nodes
        start_time = MPI_Wtime()
          client _routine()  /* counterpart of routine_to_measure */
        end_time = MPI_Wtime()
        finalize_server_routine()
        wait till end of time slot
        send result to server
until stop signal received
```

Figure 3.3 : SKaMPI pseudocode

For point-to-point measurement SKaMPI presents the measurement for MPI_Send and MPI_Recv as a roundtrip time and does not divide the result by two to get the point-to-point communication time. Unlike the other MPI benchmarks, by default SKaMPI ensures that the messages are not in cache by using randomized numbers for the message data in every iteration. SKaMPI provides a detailed configuration file to change this default as well as many options, and to enable the user to choose which MPI routines to measure.

3.3.5 MPIBench

MPIBench [1, 2, 8] is the most recently developed MPI benchmark. The main feature of MPIBench is that it uses a very accurate, globally synchronized clock that is based on CPU cycle counters. This allows accurate measurement of individual MPI communications. MPIBench is therefore able to provide distributions (histograms) of communication times, rather than just average values, which can provide additional insight into communications performance.

```
loop over different MPI routines
    run global clock synchronization process
    loop over different message sizes
        loop over number of repetitions
            if this is a collective communication routine, do a barrier synchronization
            save a timestamp for the start time (done by each process)
            call the MPI routine
            save a timestamp for the finish time (done by each process)
        end loop over repetitions
    end loop over message sizes
    run global clock synchronization process
    fix the timestamps by correcting for clock skew based on the synchronization process
    compute communication times for each repetition and each process
    compute average time by averaging over all repetitions and
        1) all processes for point-to-point communication
        2) the slowest process for each repetition for collective communications
    generate histograms of completion times
end loop over MPI routines
```

Figure 3.4: MPIBench Pseudocode

Rather than using a simple 2 processor ping-pong for point-to-point communications, MPIBench measures results for N processors communicating concurrently, and can therefore take into account effects of network contention. For point-to-point communications it can measure the time for a single communication, not just the average over a measurement of multiple ping-pongs. For collective communications, it can measure the different completion times for each process.

MPIBench measures the most common MPI communications: MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv, MPI_Sendrecv, MPI_Bcast, MPI_Barrier, MPI_Scatter, MPI_Gather, MPI_Allgather and MPI_Alltoall. In addition, MPIBench defines ***each*** and ***total*** keywords to identify message sizes for collective communications, which specify the amount of message data sent by each processor, or the total amount of message data sent by all processes, respectively.

Originally MPIBench assumed the message data was in cache and warmed up the cache before each measurement, however a newer version has been developed that provides the option of using data that is not in cache, by using a very large array to store the message data and giving a pointer to a different part of the array for each iteration. MPIBench can optionally handle outliers by discounting measurements that are larger than a specified factor above the average value.

Because MPIBench uses a globally synchronized clock, it is possible to apply the process synchronization required for measuring collective communications times by using a synchronized start, where each processor starts each collective routine at a prescribed time, and the time reported for each repetition is the time taken by the slowest process to complete the communication. This is the same as the Synchronous Measured Collective approach used by SKaMPI. However by default, MPIBench uses a barrier operation to synchronize the start of all collective communications, with the option of using a synchronized start for some routines.

There are 5 different data files that are generated by MPIBench, which have filenames ending with .summary, .subsamples, .histograms, .outliers and .gnu, which respectively contained the minimum and average values at each message size, a per-process subsampling of completion times at each message size, a histogram of completion times at each message size, outliers recorded at each message size and gnuplot instructions.

3.4 MPI Benchmark Functionality and Ease of Use

The different MPI benchmarks all provide different functionality, so in order to standardize the comparison of functionality, only some items will be discussed, which are the compile and run procedure, the MPI routines that are measured, the presentation of results and documentation. Some other functions that are available only in specific benchmarks will also be highlighted.

3.4.1 Compiling and Running the Benchmarks

The procedure and settings for compiling and running the benchmark programs can vary for different kinds of machine, operating system and MPI implementation. The purpose of this comparison is to compare a general task that is commonly required for the compile and run procedure, such as auto-generate for compiling or scripts for running a benchmark. Mostly, the benchmark software will provide an auto-generate function to compile the program, which is usually by providing a makefile. All the benchmark software provides this function except for SKaMPI [62]. SKaMPI only has one source file, the user only needs one compiler call to compile the program.

There are several aspects of running the program that will be discussed, such as the use of a configuration file, the parameters involved and user definable functions. Among the MPI benchmark software, PMB and SKaMPI use a configuration file to make the benchmark run more structured and easily definable. The use of a configuration file in SKaMPI makes it the simplest to run, this is because it does not have any additional parameter besides one compiled source file to put with the basic MPI instruction to run the benchmark software. In SKaMPI, the configuration file includes various user definable settings such as the interval and range for message size, number of repetitions, and the error controls. In the configuration file all the settings are grouped by the MPI routine and each MPI routine has its own reference number which refers to their category, either point-to-point, master-worker, simple, barrier collective or synchronous collective. Although the configuration file in SKaMPI makes it the simplest to run, however under certain circumstances, such as if the user would like to run only a certain type of MPI routine at a particular time, the user needs to make a change in the configuration file to select the required MPI routine to run the benchmark. But, if using the default settings, the task to run the benchmark program is really easy.

PMB [65] also uses a configuration file, but MPI routine selection has to be identified in the run command, which is more straightforward than SKaMPI approach. The configuration file that PMB uses only contains all the details that seldom need to change, for example number of repetitions and the range of message sizes. However, if the configuration file is changed then PMB needs to be recompiled. In PMB, the interval of message sizes is fixed, which is to use a log scale. In our experience PMB is the easiest benchmark to use.

MPIBench [2], Mpptest [63] and MPBench [64] use a similar approach by passing the parameters as arguments to the program, such as number of processors and repetition, selection of MPI routine and the interval of message sizes. There are advantages and disadvantages of passing the parameters as arguments to the program. The disadvantages are the user has to write their own scripts to run the program. The advantages are there is no re-compilation after changing the number of repetitions or range of message size and there is no tedious task to reset the configuration file every time to change the routine to measure.

In addition, MPIBench and Mpptest provide a simple function for specifying the interval of message sizes by allowing the user to define the minimum, maximum and the increment of the message sizes. The message sizes will start with the minimum value and increase using the given increment until it reaches the maximum value. Notably, the new version of MPIBench has made taking measurements for larger message sizes simpler by adding the capability to use message sizes that are a power of 2, from advice based on the analysis done in this work. Additionally, Mpptest and SKaMPI provide the capability to adaptively choose the message sizes in order to isolate sudden changes in performance. They also allow for measurements of cache effects and computation and communication overlap.

3.4.2 Measured communication routines

This section analyses the selection of MPI communications routines that are measured in each of the benchmarks. Generally, all of the benchmarks will group the tests into point-to-point and collective communications. SKaMPI has the largest selection of MPI routines. It has divided the routines into five categories, which are point-to-point, master-worker, barrier collective, synchronous collective and simple. Basically, the point-to-point and master-worker categories involve all types of Point-to-Point communication, which has been explained in section 3.3.4, while the barrier and synchronous categories are for collective routines, which are measured using different techniques. The benchmark with the least selection for MPI routines is MPBench, which provides only a common MPI routine for each type of communication, such as the Roundtrip, MPI_Send, MPI_Bcast, MPI_Alltoall and MPI_Allreduce. Mpptest also measures the performance of only a few of the basic MPI routines, but Mpptest measures the routines in a variety of situations. As an example, for the point-to-point (ping-pong) test Mpptest can measure performance with many participating processes, which can expose contention and scalability problems.

3.4.3 Presentation of output

Generally, most of the benchmark programs will generate a set of output data files that will include a user-specified range of parameters, such as the type of MPI routine, message size, the average communication time and calculated bandwidth. In addition, some of the benchmarks provide additional output such as gnuplot files for plotting the results and functions to auto-generate postscript files of results to ease the task of data processing for users.

MPIBench and Mpptest produce gnuplot files to enable easy plotting of results. SKaMPI provides a script to auto-generate a postscript file, which will read the output file and generates a postscript file that contains a graphical representation of the results. Similarly, MPBench automatically generates a postscript file containing graphical output for every measurement, but only if using the default selection of measurements. PMB only provides a set of data files with the average communication time and calculated bandwidth for the output.

MPIBench has an extra capability from the other benchmark software, which is that it can log the results of all measurements for all processes, or a subset of the measurements (e.g. every 10th iteration). It also records the distribution of communication times for generating a histogram and also a list of outlier events. The advantage of the distribution data is that researchers can analyze more detail about the behaviour of the MPI routine.

3.4.4 Documentation

This study found that the Pallas MPI Benchmarks (PMB) [65] and SKaMPI [62] have the best user manual documentation among all of the benchmark software. They provide a complete documentation that describes in detail the purpose for each of the functions in their software and the procedure to compile and run the program. The other benchmark software, which are MPIBench [2], Mpptest [63] and MPBench [64], provide brief documentation, which basically explains the procedure to compile and run the program and also main functions in the software.

3.5 Machines Used

3.5.1 ccNUMA Shared Memory Machine

The SGI Altix 3000 [70,124] series has a cache coherent non-uniform memory architecture (ccNUMA). It is based upon the hierarchical composition of two basic building blocks, or *bricks*: computational nodes (C-bricks) and routers (R-bricks). The C-brick units contain two computational nodes, each consisting of two Itanium-2 processors connected to a custom network and memory controller ASIC (known as the SHUB *(Scalable HUB)*) (refer to Figure 3.5). The two processors share a 6.4 Gbytes/s bus to a SHUB. The two SHUBs in each C-brick are linked by a further 6.4 Gbytes/s link. Each SHUB is provided with one SGI NUMAlink channel to the outside, with a bandwidth of 3.2 Gbytes/s (1.6 Gbytes/s each direction) for NUMAlink3. These external links provide the cache coherent interconnection between C-Bricks. It is possible to directly connect a pair of C-Bricks, however for large machines a set of routers (the R-Bricks) are employed to expand the network in a scalable manner. Each R-Brick contains a router chip, which provides eight connections. Each connection is again 3.2 Gbytes/s (1.6 Gbytes/s each direction). The R-Bricks are configured so that four ports connect to C-Bricks, and the other four interconnect with other R-Bricks to form a fat tree network. Pairs of R-Bricks are connected by two links, and in large machines the remaining two links connect to the next higher layer of the tree, to routers (called meta-routers) that use each of their eight links to provide connectivity to the lower levels. Figure 3.6 depicts a 128 node Altix.

The benchmark results reported in this thesis were carried out on Aquila, an SGI Altix 3000 managed by the South Australian Partnership for Advanced Computing (SA-PAC) [131]. SGI Altix has 160 1.3 GHz Itanium 2 processors with a total of 160 Gbytes of memory, and a NUMAlink3 network. At the time of the benchmarks, it was running SGI Linux ProPac3. Intel compilers were used to compile the MPI benchmark programs, and the SGI MPI libraries were used. On shared memory machines, the operating system can switch processes between processors to try to improve overall system utilization. However this can adversely affect parallel programs, since after process migration, data will no longer be available in local cache. The performance of MPI programs on the Altix can be improved significantly by binding each process to a particular processor. So, for this analysis the experiment has done benchmark measurements using the MPI_DSM_CPULIST environment variable, which assigns MPI processes in order to the specified list of CPUs. The Altix documentation suggests that applications should avoid using processor 0, particularly for parallel jobs, since it is used to run system processes. Therefore this analysis only used processors 32 to 159 for the measurements. It started with processor number 32 in order to maintain the hierarchical pattern of 32 processor groups shown in Figure 3.6.

By default, the SGI MPI implementation buffers messages, but uses single copy (i.e. no buffering) for large message sizes in most collective communication routines and in MPI_Sendrecv, which significantly improves performance [70, 124]. The message size where the communication changes over to single copy is not specified in the documentation but our measurements indicate it is around 2 Kbytes. By default, single copy is not used for MPI_Send, however it is possible to force it to use single copy by setting the environment variable MPI_BUFFER_MAX *n*, where *n* is the maximum message size where buffering will be used, so messages larger than *n* will be communicated using single copy. The choice of buffering or single copy can give a big difference in the performance of MPI_Send for large message sizes, and hence the bandwidth reported by an MPI benchmark program.

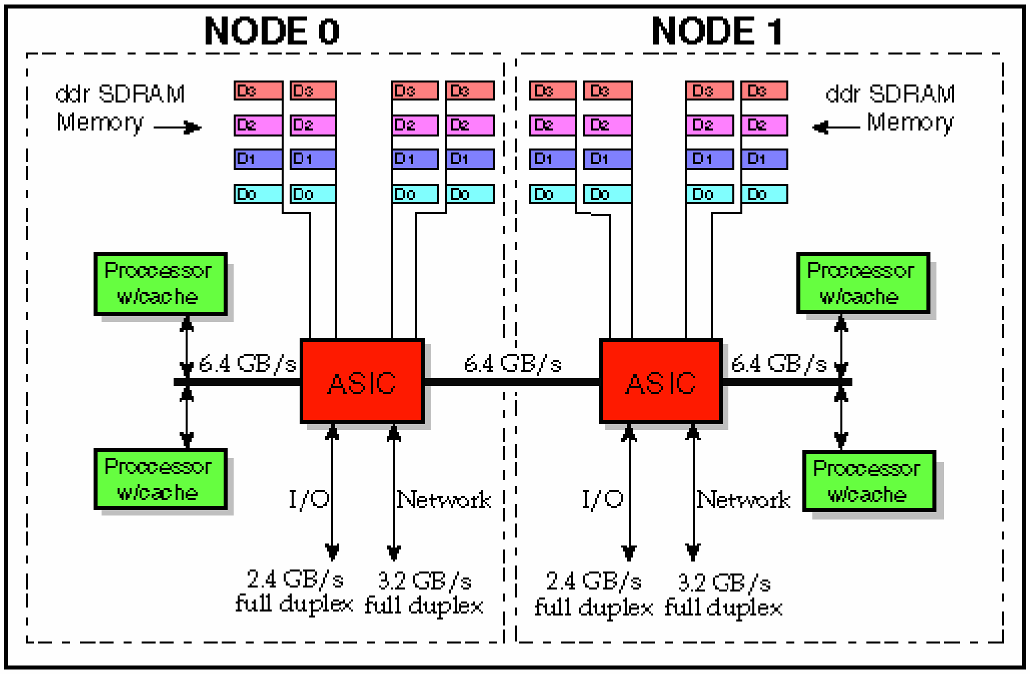

Figure 3.5 : An Altix C-brick with 2 nodes, 2 NUMAlink-3 and 2 XIO channels [124].

Brick Type	**Purpose**
C-Brick	Computational module housing CPUs and memory.
M-Brick	Memory expansion module.
R-Brick	NUMAflex router interconnect module.
D-Brick	Disk expansion module.
IX-Brick	Base system I/O module.
PX-Brick	PCI-X expansion module.

Table 3.1 : The SGI Altix brick type.

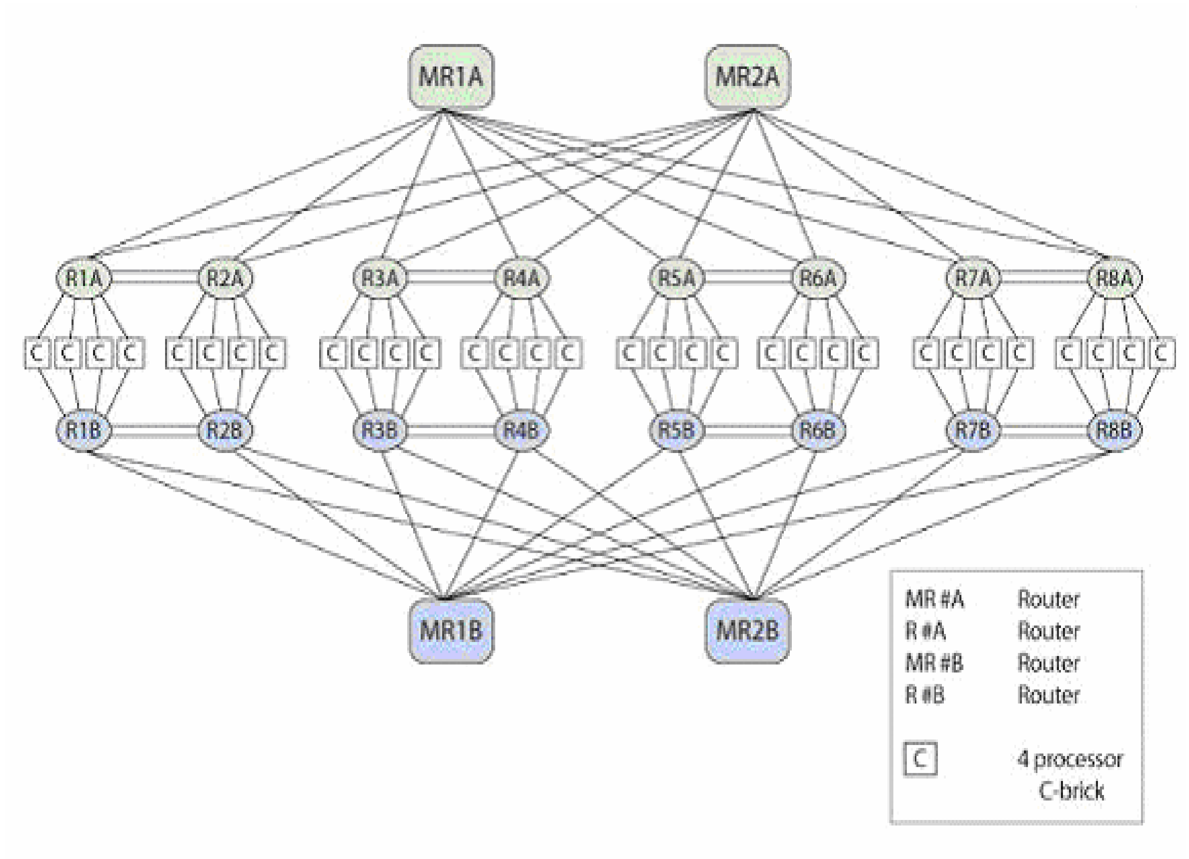

Figure 3.6 : SGI Altix 3000 communications architecture for 128 processors [124].

3.5.2. Distributed Memory Machine

The measurements reported in this thesis were done on Hydra, which is an IBM eServer 1350 Linux cluster [35] which is managed by SAPAC [131]. The cluster has 128 compute nodes connected by a Myrinet 2000 [71,110] network as well as a 100 Mbit/s Fast Ethernet network [134]. Each of the nodes are IBM xSeries 335 servers with dual 2.4 GHz Intel Xeon processors and 2 GBytes of RAM, so the machine has a total of 256 CPUs.

The Myrinet configuration has 8 nodes connected to each switch, and the switches connected together in a fat tree topology. The Ethernet configuration is that each rack of the cluster has a Fast Ethernet switch (100 Mbit/s full duplex) connecting all the nodes in the rack. The nodes in each rack are 1-38, 39-76, 77-114 and 115-126. Each of these switches has a Gigabit Ethernet (full duplex) uplink to a Cisco Gigabit switch. The cluster nodes were running Redhat Enterprise Linux version 3.2.3-47 with kernel 2.4.21-27.ELsmp of the Myrinet drivers, MPICH-GM version 1.2.6..14a was used with the Myrinet and MPICH version 1.2.6-gnu for the Ethernet network. All compilations were performed with gcc v3.2.3.

All measurements were run with dedicated access to the cluster, so there were no other processes affecting the results. At the time the measurements were taken, not all of the nodes were usable, so the benchmark only took measurements for up to 100 nodes (200 CPUs). The measurement used 2 CPUs per node, with 1000 repetitions of each of the MPI operations except for MPI_Alltoall (the slowest operation) where only 100 repetitions were used.

3.6 Point-to-Point Communication

All of the MPI benchmark applications provide measurement for basic point-to-point communication using MPI_Send/MPI_Recv. The main difference between the MPI benchmark applications is the communication pattern. Figure 3.7 to Figure 3.9 illustrate the communication patterns of the different benchmarks for 8 processors. Figure 3.7 shows the point-to-point communications pattern for PMB and Mpptest, which involve processors 0 and 1 only. In order to measure communications times between processors that are not on the same node of a cluster of SMP nodes, the locations of the processors would have to be specified when calling mpirun, otherwise the benchmark would measure the performance of the shared memory system on the node rather than the performance of the communications network connecting the nodes. Figure 3.8 is for SKaMPI and MPBench, which use the first and last processor. In fact the approach used by SKaMPI is more complicated, in that it does short tests on all the processors to find which processor has the slowest communication with processor 0, and then does its timings using that processor. However for the communication networks on both of the machines used in this work, this would be equivalent to choosing the last processor.

MPIBench measures not just the time for a ping-pong communication between two processors, but can also take into account the effects of contention when all processors simultaneously take part in point-to-point communication. The default communication pattern used by MPIBench is shown in Figure 3.9. MPIBench sets up pairs of communicating processors, with processor p communicating with processor (p + n/2) mod n when a total of n processors are used. Half of the processors send while the other half receive, and then vice versa. The send/receive pairs are chosen to ensure that for a cluster of SMPs or a hierarchical communications network (such as on the SGI Altix) the performance of the full communication hierarchy can be measured, not just local communications within an SMP node (or a brick on the SGI Altix). MPIBench also allows the user to specify another communication pattern by specifying a list of communication partners.

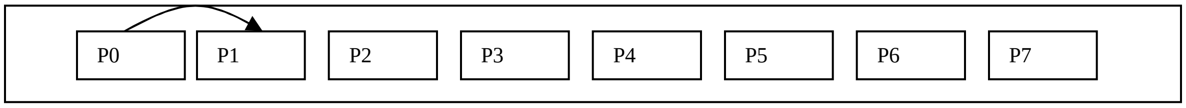

Figure 3.7 : PMB and Mpptest Point-to-Point pattern

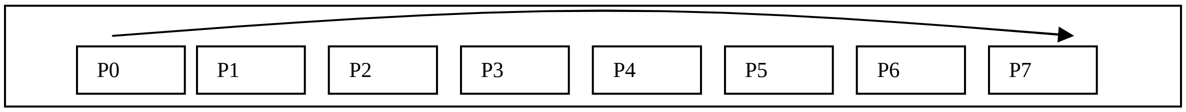

Figure 3.8 : SKaMPI and MPBench Point-to-Point pattern

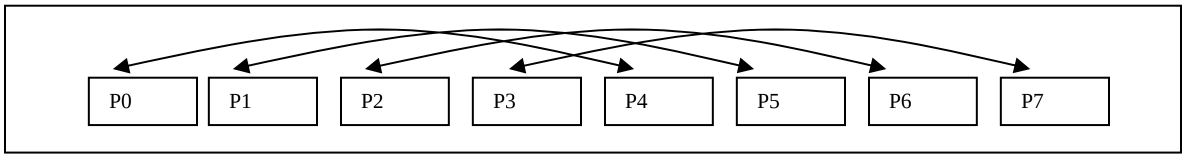

Figure 3.9 : MPIBench Point-to-Point pattern

3.6.1 MPI_Send/MPI_Recv

The difference in communication patterns between the different benchmarks leads to different results, as shown in Figure 3.10 for the default settings of the SGI MPI implementation for the SGI Altix (i.e. buffered copy for MPI_Send). MPIBench has the highest results due to the contention effects from all 8 processors, while MPBench and

SKaMPI obtain the second highest results since they are measuring the communication times between two C-Bricks. The lowest results are obtained by Mpptest and PMB, since they just measure intranode communication within a C-Brick. By carefully selecting the processors that are used (e.g. P0 and P7), it is possible to force each of the benchmarks to measure the same thing, i.e. point-to-point communication between two processors across any level of the communication hierarchy, and the results for different benchmarks agree fairly closely, within a few percent, as shown in Figure 3.11. This is similar to benchmarking clusters of SMP nodes, where care must be taken (particularly for PMB and Mpptest) in choosing the processors to ensure measurement of internode rather than intranode communication. On the SGI Altix it is possible to significantly improve the benchmark results for MPI_Send by enabling the option of single copy (i.e. non-buffered) sends in the SGI MPI implementation, as shown in Figure 3.10 and Figure 3.12. This is done by setting the environment variable MPI_BUFFER_MAX to be the maximum message size (in bytes) for which buffered copy send will be used, so single copy send is used for any message larger than this specified size. As shown in Figure 3.12, it is best to set this value to be very small, although there is no effect below about 128 bytes. Note that the improvement from using single copy can be large, up to a factor of 10, however it is much less than this for very large message sizes.

In measuring the results using single copy MPI_Send, we were surprised to find that while most of the benchmarks gave the expected improvement in performance, the results for SKaMPI and MPIBench were the same as for the default MPI setting that uses buffered copy. After much experimentation and comparison of the code for the different benchmarks, we concluded that this problem is because both SKaMPI and MPIBench use the same array to hold send and receive message data. When we changed the MPIBench code to declare different arrays for send and receive data, the results showed the expected improvement, as shown in Figure 3.10. We did not change the SKaMPI program, so we do not present SKaMPI results for the single copy option.

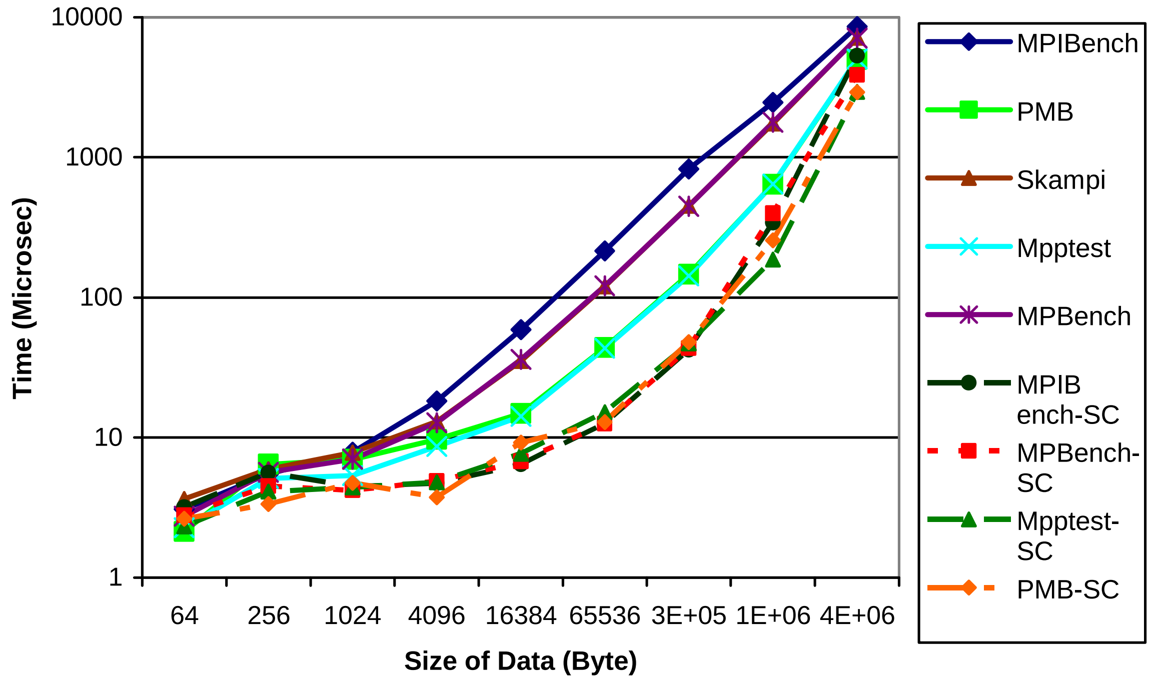

Figure 3.10 : Comparison of results from different MPI benchmarks for Point-to-Point (send/receive) communications using 8 processors between default settings and Single Copy (indicated by SC) on SGI Altix.

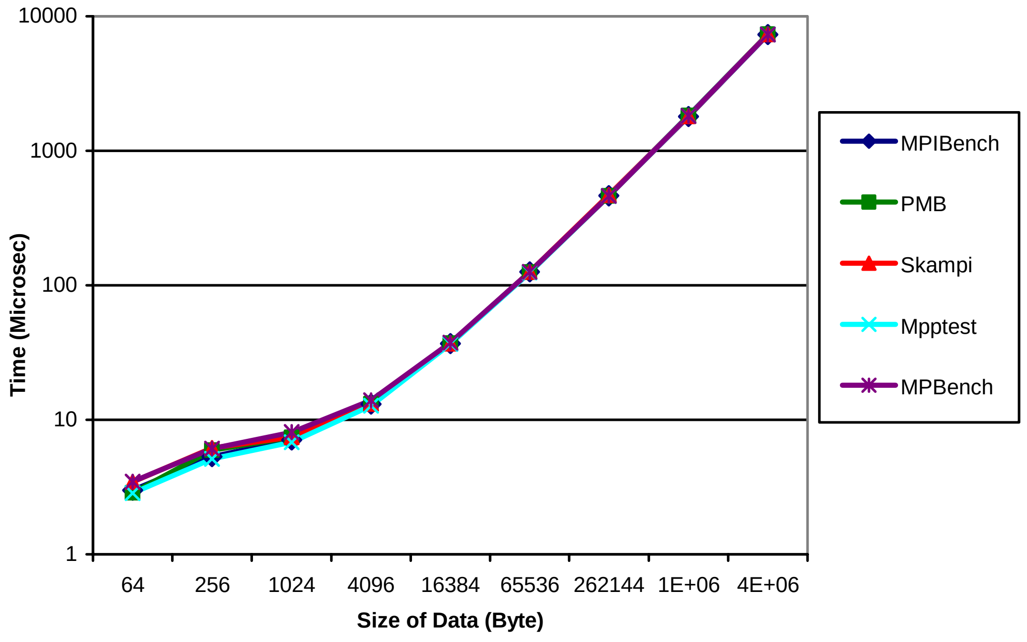

Figure 3.11 : Comparison of results from different MPI benchmarks for Point-to-Point (send/receive) communications using the same process placement, with a single process on each of 2 different C-Bricks connected by a router, on the SGI Altix.

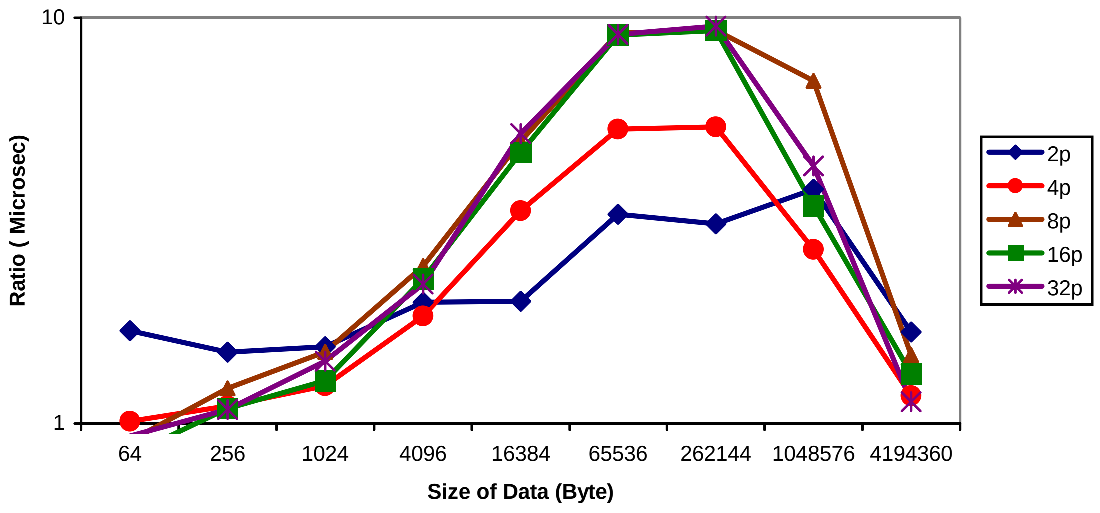

Figure 3.12 : Ratio of Send/Recv time using buffered compared (default) to non-buffered communication for PMB from 2 to 32 Processors .

Figure 3.13 shows the results comparison from different MPI benchmarks for Point-to-Point (send/receive) communications using 8 processors on the IBM Linux cluster. Similarly with the results from the SGI Altix, the lowest times are obtained by PMB and Mpptest, while the highest time is obtained by MPIBench. The results for PMB and Mpptest are lower because they measure intranode communication (within the same node). The gap between the results for the different benchmarks appears to be getting closer for large message size due to the effect of the log plot. The difference in results between MPIBench (where all processes are communicating) with SKaMPI and MPBench (where only 2 processes are communicating) is much smaller than for the SGI Altix. The differences start to increase after 65 KByte due to more contention occurring for larger message sizes, which is seen in the MPIBench results but not in the other benchmarks. However, in the SGI Altix the results start to differ from as small as 1 KByte. Although there are still differences among the MPI benchmarks, the differences are not as big as for the SGI Altix, except for Mpptest and PMB since they measure shared memory communication within a node. As with SGI Altix, by selecting specific processors across different nodes (e.g. P0 and P7), it is possible to force each of the benchmarks to measure the same thing, and then the results for different benchmarks agree fairly closely, within a few percent, as shown in Figure 3.14.

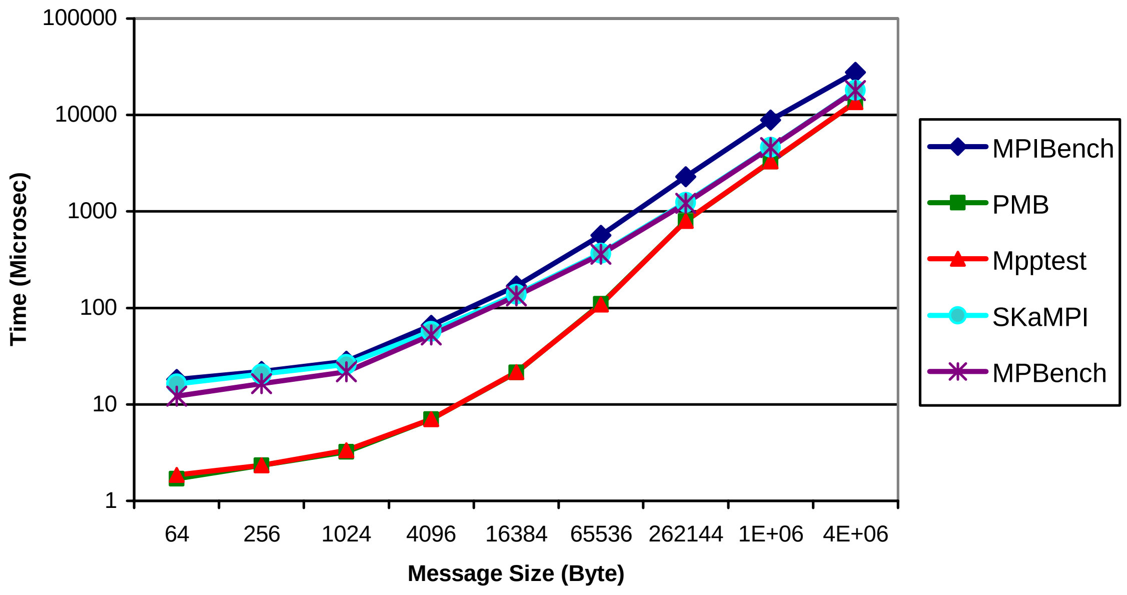

Figure 3.13 : Comparison of results from different MPI benchmarks for Point-to-Point (send/receive) communications using 8 processors on IBM Linux Cluster.

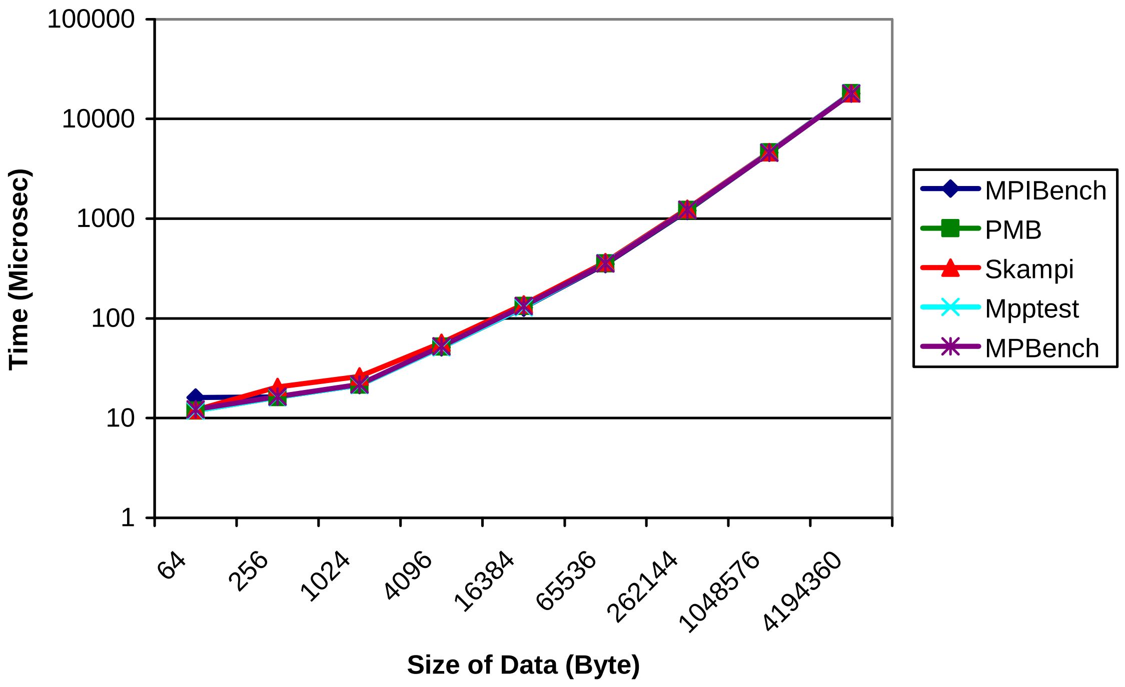

Figure 3.14 : Comparison of results from different MPI benchmarks for Point-to-Point (send/receive) communications using the same process placement, with a single process on each of 2 different nodes on IBM Linux Cluster.

3.6.2 Bandwidth for MPI_Send/MPI_Recv

Bandwidth results are calculated by default for all message sizes for Mpptest and PMB and in this section the CPUs have been chosen to make the measurement the same as with MPBench and SKaMPI. MPBench provides bandwidth results instead of communication time for all the collective communications but not for point-to-point communication, while SKaMPI and MPIBench do not provide any bandwidth results. However, SKaMPI does provide bandwidth results, calculated only for the smallest and the highest message sizes, in their auto-generated postscript files of results.

The bandwidth results in Table 3.2 and Table 3.3 are calculated based on the measurement results for 1 MByte and 4 MByte message sizes for the SGI Altix and the IBM Linux cluster, respectively. As shown in Table 3.2, at 1 MByte for 2 processors MPIBench and SKaMPI obtained significantly higher bandwidth than the other benchmarks, while the results are similar for 4 MByte messages. For two processors, all of the benchmarks show a difference of almost a factor of 2 between the results for the two different message sizes, so the Altix does not conform to the usual expectation that larger message sizes should give similar or larger bandwidth measurements. The results between both message sizes started to get closer for MPIBench at 4 processors, and for the other benchmarks at more than 4 processors. The bandwidth decreases drastically as the message sizes and number of processors increase. Referring to [27], the bandwidth reported by SKaMPI is higher than for the largest times plotted for each node. So, there is a possibility that the reported bandwidth in Table 3.2 may be smaller than the peak bandwidth.

Table 3.3 shows the bandwidth results for the IBM Linux cluster. Mpptest obtains the highest bandwidth for both message sizes. There is little difference between the results for both message sizes for all MPI benchmarks. The bandwidth results for 4 up to 32 CPUs is the same for all benchmarks except MPIBench, which shows the lowest performance due to more contention effects from having all the processes doing point-to-point communications concurrently. All the other benchmarks are just doing a ping-pong between two nodes, which are at different distances in the network in each case. The fact that the results are the same in each case indicates the low overhead of Myrinet's hierarchical fat tree network.

No. of CPU	**MPIBench**		**MPBench**		**SKaMPI**		**PMB**		**Mpptest**	
	1MB	**4MB**	**1MB**	**4MB**	**1MB**	**4MB**	**1MB**	**4MB**	**1MB**	**4MB**
2	1775	851	1320	831	1756	832	1308	806	1314	843
4	606	671	987	925	1056	963	1012	945	1090	987
8	405	464	562	562	570	560	563	560	593	588
16	396	462	564	562	573	559	563	560	592	587
32	260	256	552	549	560	548	555	548	579	574

Table 3.2 : Bandwidth results in MBytes/sec for various numbers of processors using default settings on SGI Altix.

No. of CPU	**MPIBench**		**MPBench**		**SKaMPI**		**PMB**		**Mpptest**	
	1MB	**4MB**	**1MB**	**4MB**	**1MB**	**4MB**	**1MB**	**4MB**	**1MB**	**4MB**
2	306	294	306	293	308	295	305	294	322	309
4	112	140	218	222	218	223	218	222	228	233
8	112	144	218	222	218	222	218	222	228	232
16	113	142	218	222	217	222	218	222	228	233
32	97	100	218	222	218	223	218	222	228	233

Table 3.3 : Bandwidth results in MBytes/sec for various numbers of processors on IBM Linux Cluster.

Figure **3.15** and Figure 3.16 show PMB and MPIBench bandwidth results for 2 up to 32 CPUs for the SGI Altix, respectively. Note that the results for PMB are taken by selecting the CPUs to make the measurements similar with MPBench and SKaMPI (as shown in

Figure 3.8). PMB bandwidth shows the results for 2 and 4 CPUs is different with 8, 16 and 32 CPUs. The differences illustrate that the bandwidth for intra C-Brick is higher compared to inter C-Brick, while MPIBench shows the bandwidth decreases as more CPUs are used. This shows that the architecture of SGI Altix creates contention between the bricks.

Figure 3.17 and Figure 3.18 show the same plots for the IBM Linux Cluster. Interestingly, the performance for inter-node is similar between different numbers of CPUs, while MPIBench shows that the bandwidth between different number of CPUs for inter-node communication has very little difference, except for 32 CPUs for message sizes more than 1 MByte.

It is interesting that the differences between PMB and MPIBench are bigger on the SGI Altix than the IBM Linux cluster. It shows that the performance of the Myrinet network in the Linux cluster scales better with more communicating processes, while the performance of the ccNUMA SGI Altix is noticeably reduced, although the overall performance of the Altix is much better than the Linux cluster.

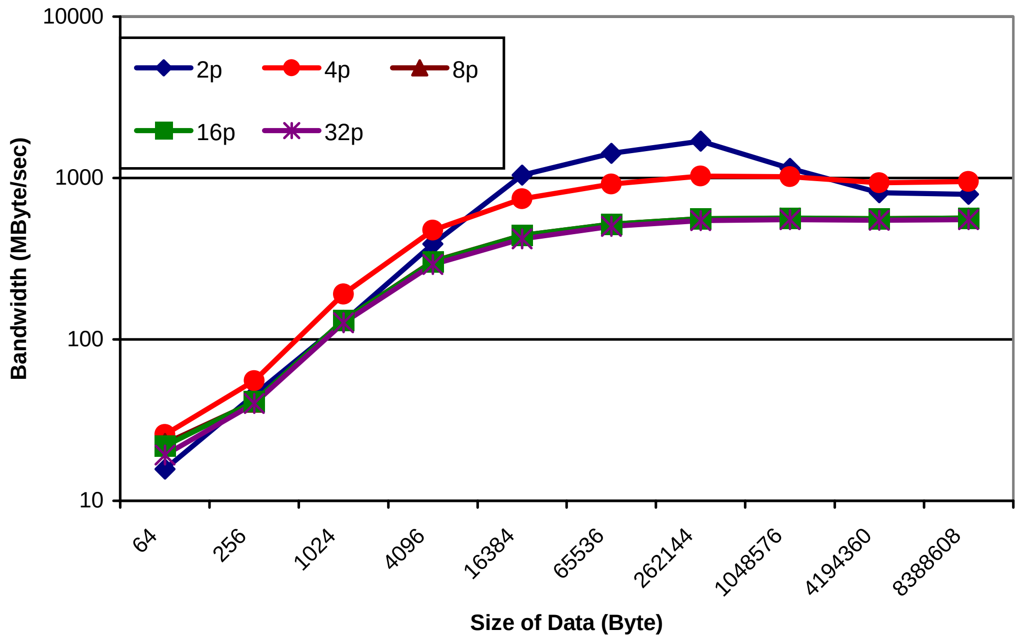

Figure 3.15 : PMB Bandwidth Results for 2 until 32 Processors for Default Settings on SGI Altix.

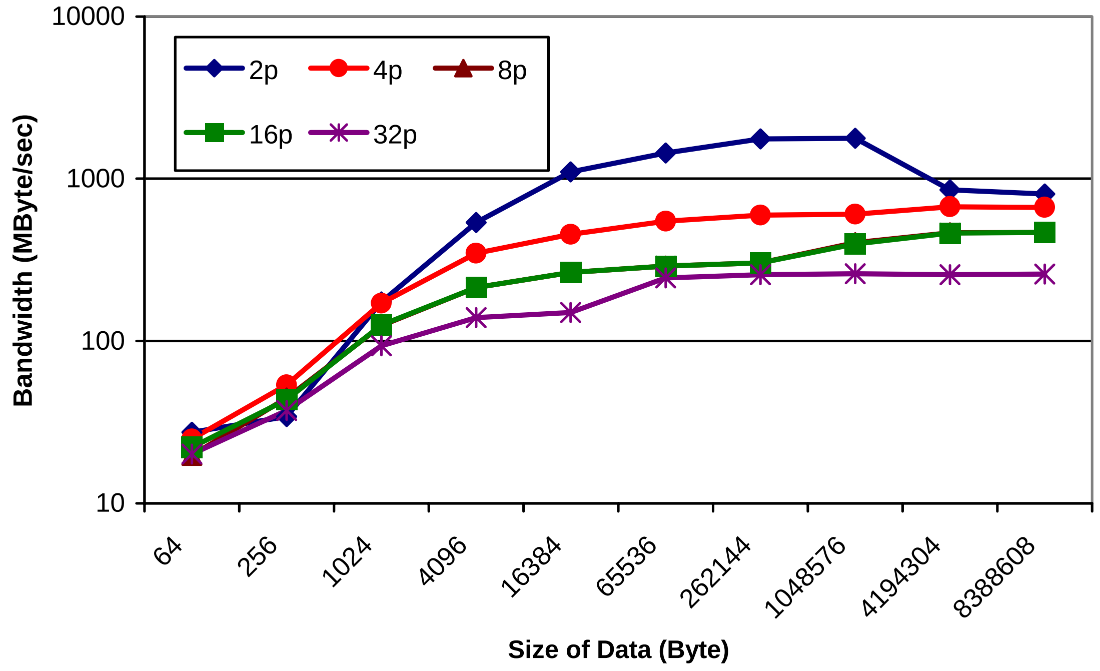

Figure 3.16 : MPIBench Bandwidth Results for 2 until 32 Processors for Default Settings on SGI Altix.

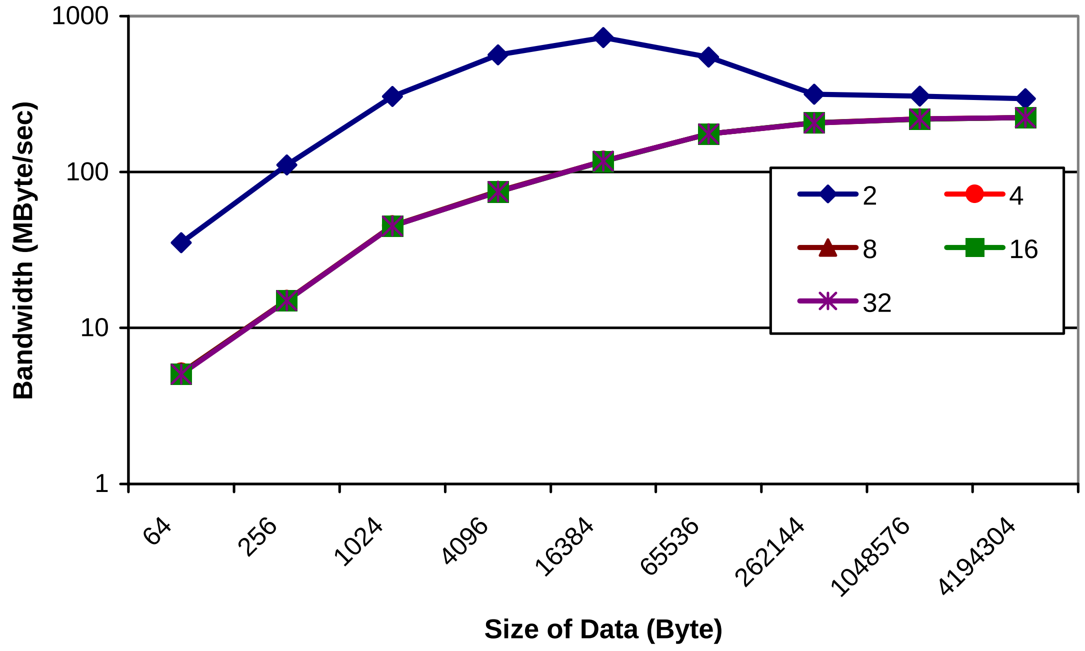

Figure 3.17 : PMB Bandwidth Results for 2 until 32 Processors for Default Settings on IBM Linux Cluster.

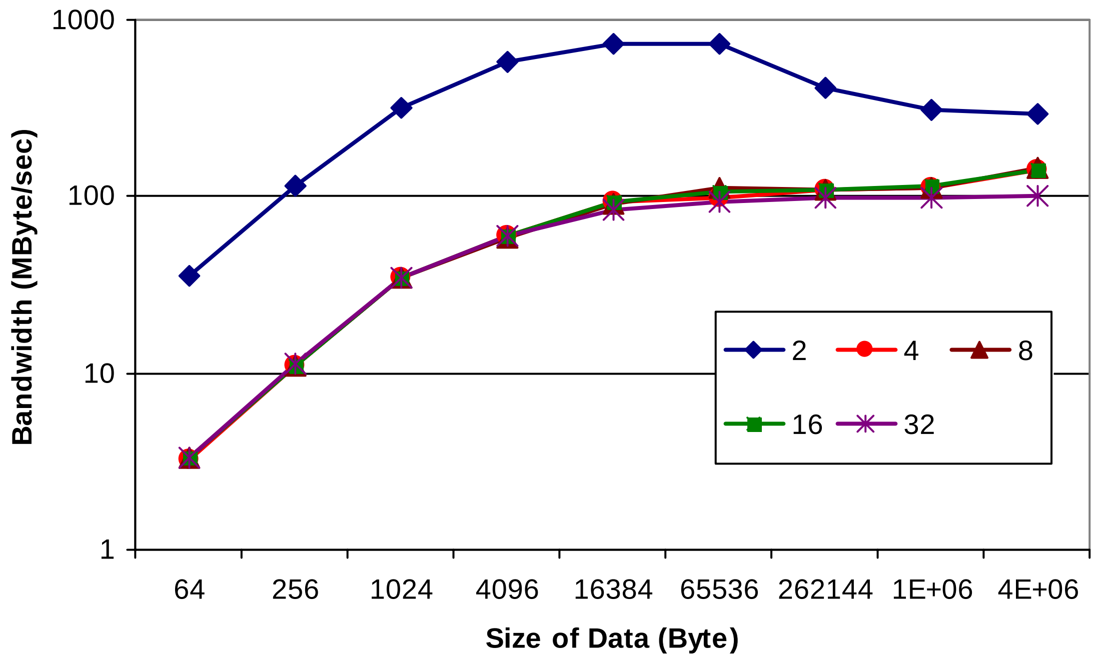

Figure 3.18 : MPIBench Bandwidth Results for 2 until 32 Processors on IBM Linux Cluster.

3.7 MPI_Sendrecv

Only MPIBench, PMB and SKaMPI provide measurements for MPI_Sendrecv. MPIBench uses the same communication pattern as for MPI_Send/MPI_Recv, however each processor does a combined MPI_Sendrecv to its communication partner, rather than alternating sends and receives. SKaMPI and PMB use a different technique, where each process sends to the right and receives from the left neighbour in a chain of N processors. Most communication networks are capable of providing the same bandwidth if messages are sent simultaneously in both directions. MPI_Sendrecv provides a good way of testing that the MPI implementation can indeed provide this bidirectional bandwidth. The MPIBench approach means that if this is the case, then the results for MPI_Sendrecv and MPI_Send/MPI_Recv should be similar.

The results for MPI_Sendrecv in Figure 3.19 show that this is the case for the SGI Altix, e.g. the result for 256 Kbyte message size for 8 processors is similar to the results for 8 processors in MPI_Send/MPI_Recv with Single Copy option (see Figure 3.10 and

Table 3.4). Noticeably, in Figure 3.19 shows that SKaMPI and PMB have a similar result, however the results are higher than MPIBench. As mentioned earlier SKaMPI and PMB are using the same ring pattern technique. The newest version of MPIBench also has a function to measure MPI_Sendrecv using the ring pattern, the results in Figure 3.19 shows that MPIBench with the ring pattern obtains similar results with SKaMPI and PMB. Note that, the results for MPIBench default sendrecv is lower than the results for PMB, SKaMPI and MPIBench ring pattern, particularly after 2KByte. It is unclear why this is the case, or why the MPIBench results for MPI_Sendrecv using the ring pattern are so much slower than for using the default MPIBench communication pattern for MPI_Sendrecv. More precise analysis should be done to understand the problem of the ring pattern with Single Copy options, which will be included for the future work.

However, the bidirectional bandwidth does not seem to be working for the Myrinet network on the IBM Linux cluster, since the results for MPI_Sendrecv are a factor of two higher than for MPI_Send/Recv (see Figure 3.13). Figure 3.20 shows the comparison results between the benchmarks on IBM Linux cluster. It shows that MPIBench obtains the highest time compared to the PMB and SKaMPI, while on the SGI Altix MPIBench obtains the lowest time. As with the SGI Altix, PMB and SKaMPI have very similar results, which due to their similar technique for MPI_Sendrecv. Figure 3.20 shows that when using MPIBench ring pattern measurement, the results are similar to the PMB and SKaMPI.

It can therefore be concluded that the differences in results between MPIBench and the other benchmarks are because of the differences in the communication partners used for MPI_Sendrecv, since using the same partners gives results that are within a few percent of the other benchmarks (see Figure 3.19 and Figure 3.20).

Message Sizes (Byte)	MPI_Send/MPI_Recv (Microsec)	MPI_Sendrecv (Microsec)
0	2.88	2.19
4	2.97	2.53
16	3.17	2.49
64	3.43	3.18
256	8.61	5.63
1024	7.59	4.48
4096	6.92	4.72
16384	8.45	6.44
65536	14.61	12.56
262144	44.64	42.22

Table 3.4 : Comparison for average communication time (microsec) between MPI_Send/MPI_Recv with MPI_Sendrecv for MPIBench on SGI Altix.

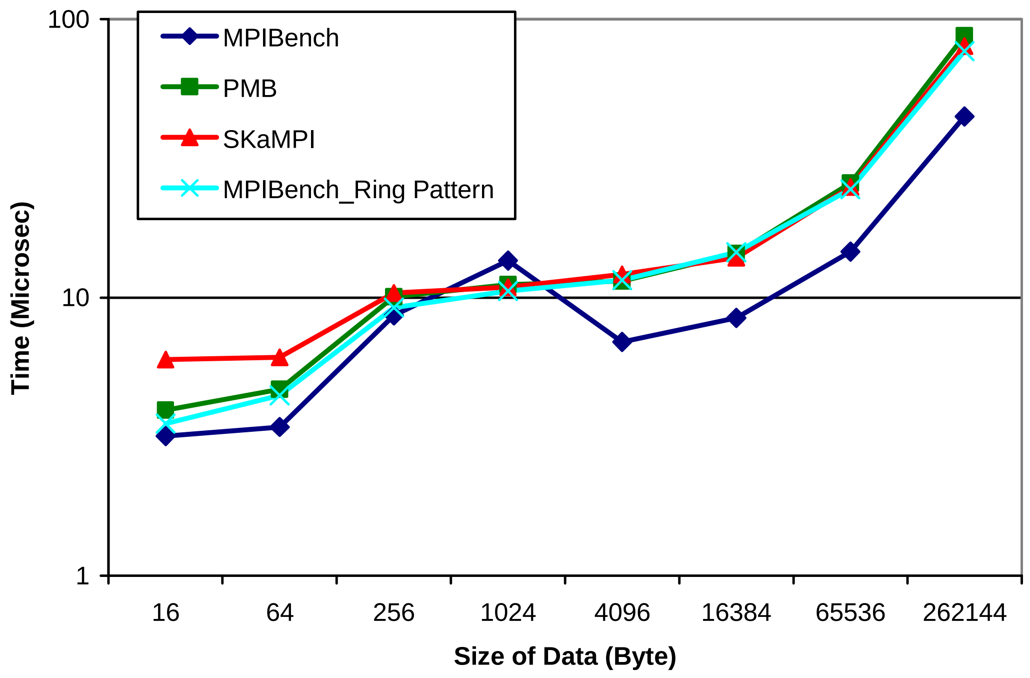

Figure 3.19 : Comparison between MPI benchmarks for MPI_Sendrecv and MPIBench ring pattern on 8 processors on SGI Altix.

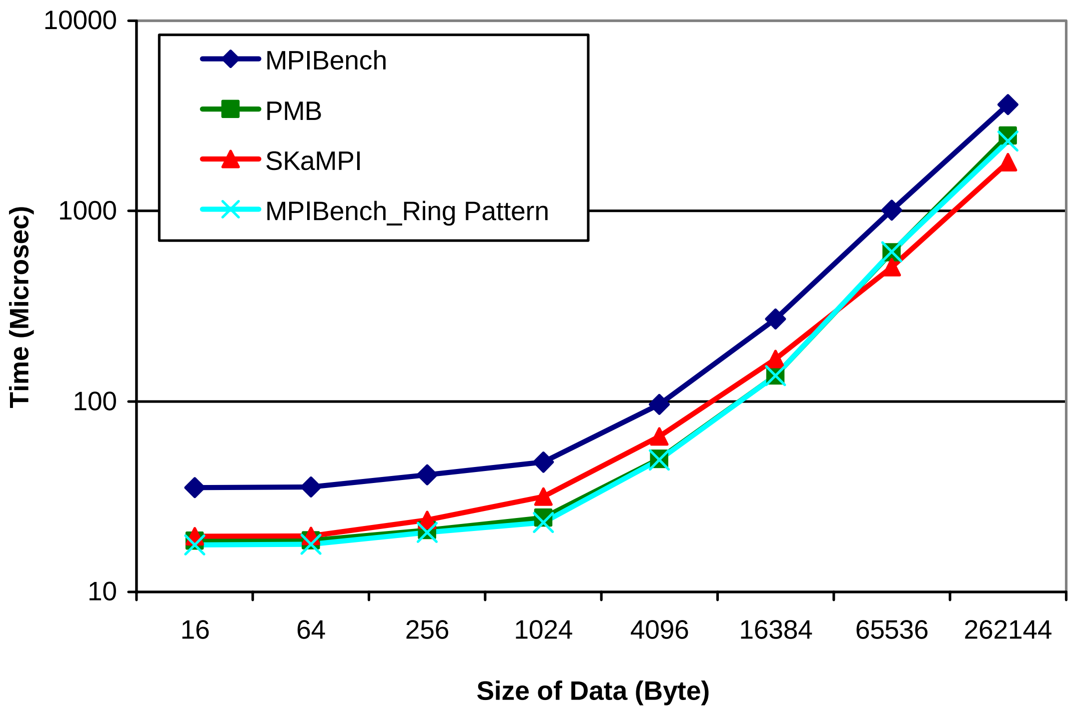

Figure 3.20 : Comparison between MPI benchmarks for MPI_Sendrecv with MPIBench ring pattern on 8 processors on IBM Linux cluster.

3.8 Barrier

Figure 3.21 shows the MPI_Barrier results on the SGI Altix for SKaMPI, MPIBench and PMB, which are the only MPI benchmarks that measure barrier. The results show that SKaMPI is a bit higher compared to MPIBench and PMB. This is probably due to the global clock synchronization that is set by default for their measurement. The developers of SKaMPI argue that this is a more accurate result since it avoids "pipelining" effects where some processes (e.g. the root) finish the barrier earlier and can start the next barrier operation before other processes have exited the barrier [24].

Figure **3.22** shows the same plot for the IBM Linux cluster. The results are almost the same for all benchmarks with only a slight difference between 16 to 64 processes.

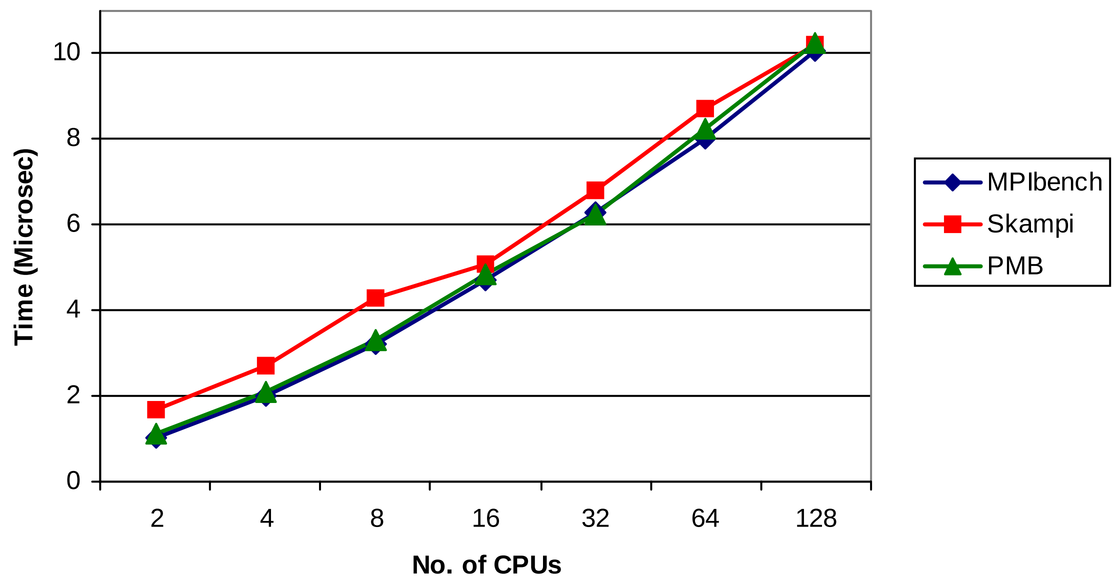

Figure 3.21 : Comparison between MPI benchmarks for MPI_Barrier for 2 to 128 processors on the SGI Altix.

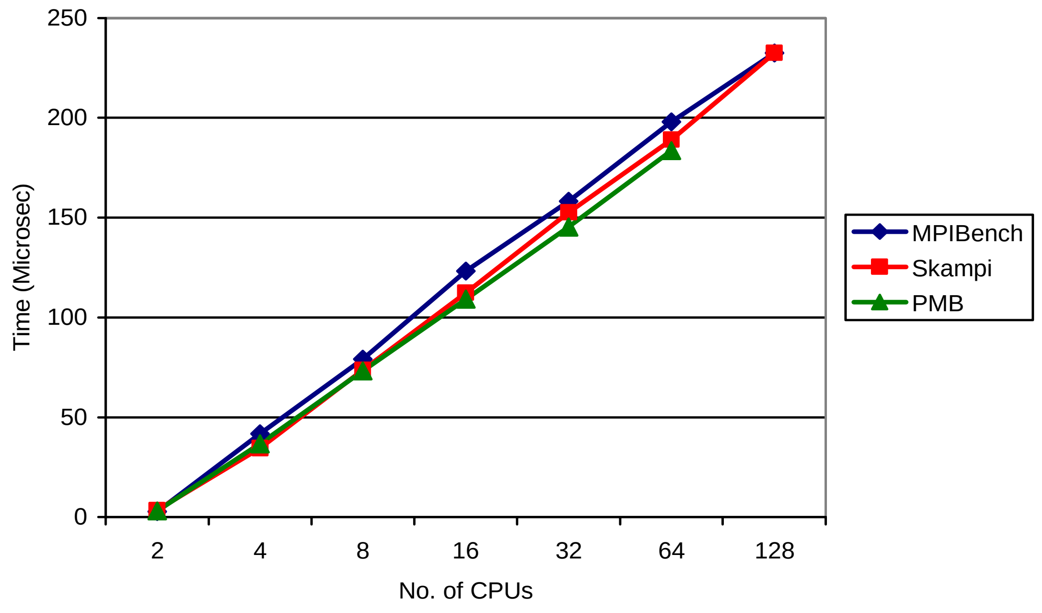

Figure 3.22 : Comparison between MPI benchmarks for MPI_Barrier for 2 to 128 processors on the IBM Linux cluster.

3.9 Broadcast

All of the MPI benchmark applications measure MPI_Bcast. There are some differences in the measurement technique between the benchmark applications. The main difference is that by default SKaMPI makes the assumption that data should not held in cache memory, so it ensures data to be broadcast is not in cache before each measurement repetition. MPIBench, on the other hand, always sends the same data for each repetition, and does some preliminary “warm-up” repetitions (that are not measured) to ensure that the data is in cache before measurements are taken. The other benchmarks allow the user to choose whether or not data to be broadcast is in cache, although the default is that data is in cache memory. In a real application, data to be broadcast may or may not be in the cache, so there is really no “right” choice for whether or not an MPI benchmark should place the data in the cache.

Another difference is how the broadcasts are synchronized. Most MPI benchmarks measure collective communication time on the root node. However for some collective operations, such as broadcast, the root node is the first to finish, and this may lead to biased results due to pipelining effects. Most benchmarks get around this problem by inserting a barrier operation (MPI_Barrier) after each repetition of the collective communication operation. This provides an additional overhead which will affect the average time, although only for very small message sizes, since broadcast of a large message takes much longer than a barrier operation. Mpptest and PMB adopt a different approach to avoid this problem – they assign a different root processor for each repetition.

Figure 3.23 shows the average times reported by the different MPI benchmarks to complete an MPI_Bcast operation on the SGI Altix. Clearly there are significant differences in the measured results due to the differences in measurement technique. Mpptest and PMB give the highest results, presumably due to the overhead of changing the root node at each iteration. Because of the cache coherency protocol on the shared memory Altix, moving the root to a different processor has a significant overhead, which is reflected in the results. We are not sure why Mpptest is so much higher than PMB. The only difference between the two approaches seems to be that PMB uses different arrays for the broadcast data on the root node and the other processors. SKaMPI has the next highest result, since it uses data that is not in cache, while MPIBench and MPBench obtained the same results with the same measurement techniques. On a distributed memory cluster the effects of changing the root and having messages in cache has little affect on the results, as shown in Figure 3.25.

To check that these differences in the benchmark measurement techniques were causing the difference in broadcast times, we enabled the option to warm up the cache in SKaMPI, and for Mpptest and PMB we commented out the code to move the root process at each repetition, and then reran the benchmarks. The results after modifying the programs were very similar, mostly within about 10% percent, as shown in Figure 3.24.

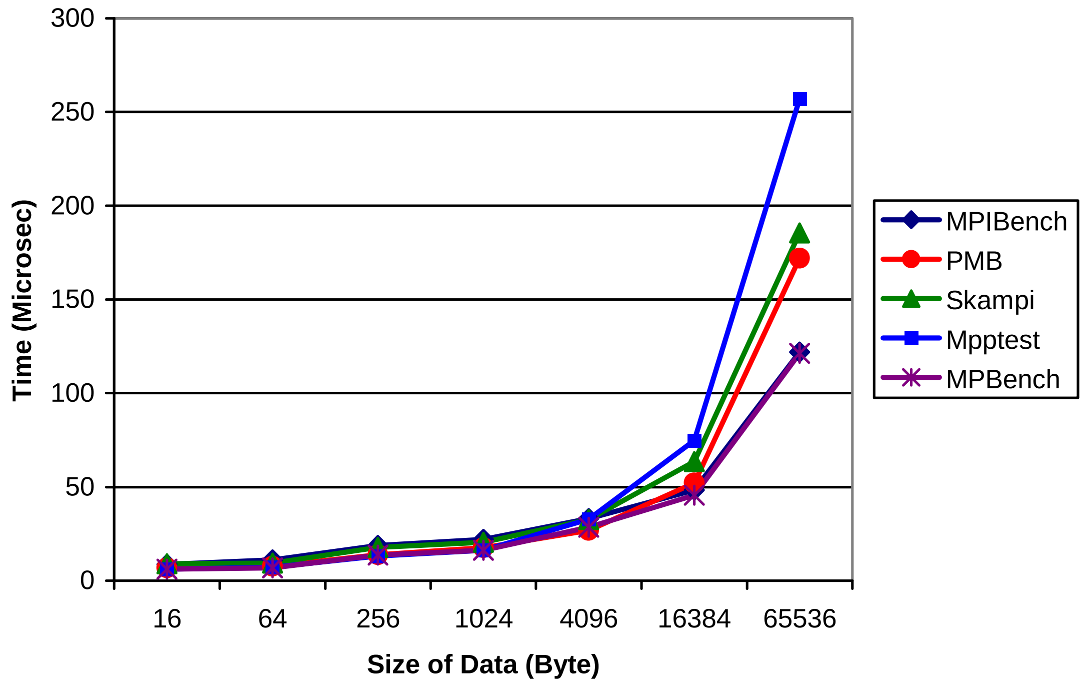

Figure 3.23 : Comparison between MPI benchmarks for MPI_Bcast on 8 processors before tuning the code on SGI Altix.

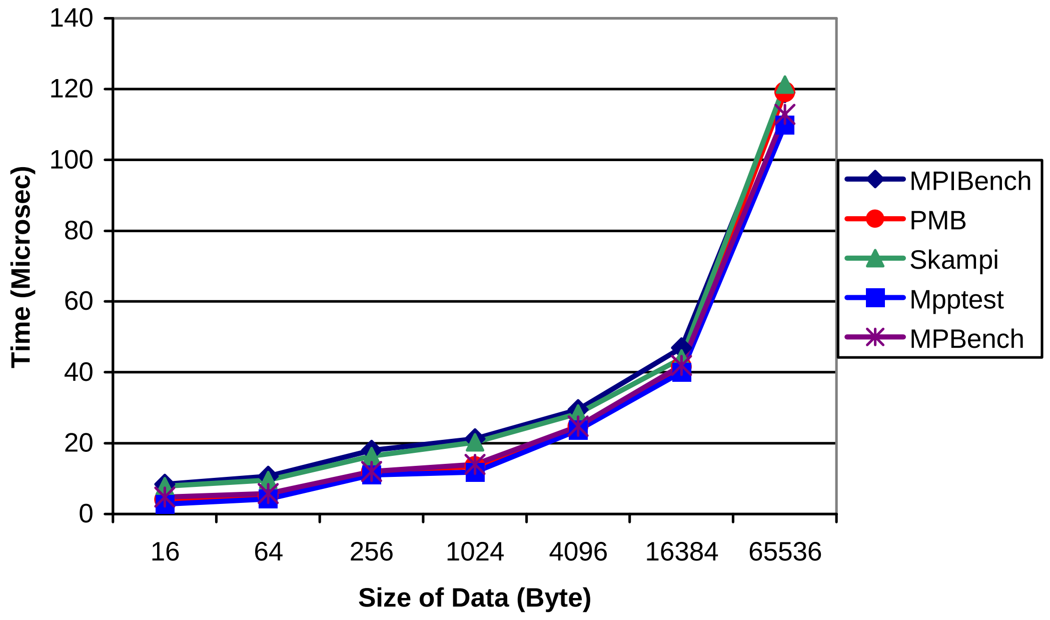

Figure 3.24 : Comparison between MPI benchmarks for MPI_Bcast on 8 processors after tuning the code on SGI Altix.

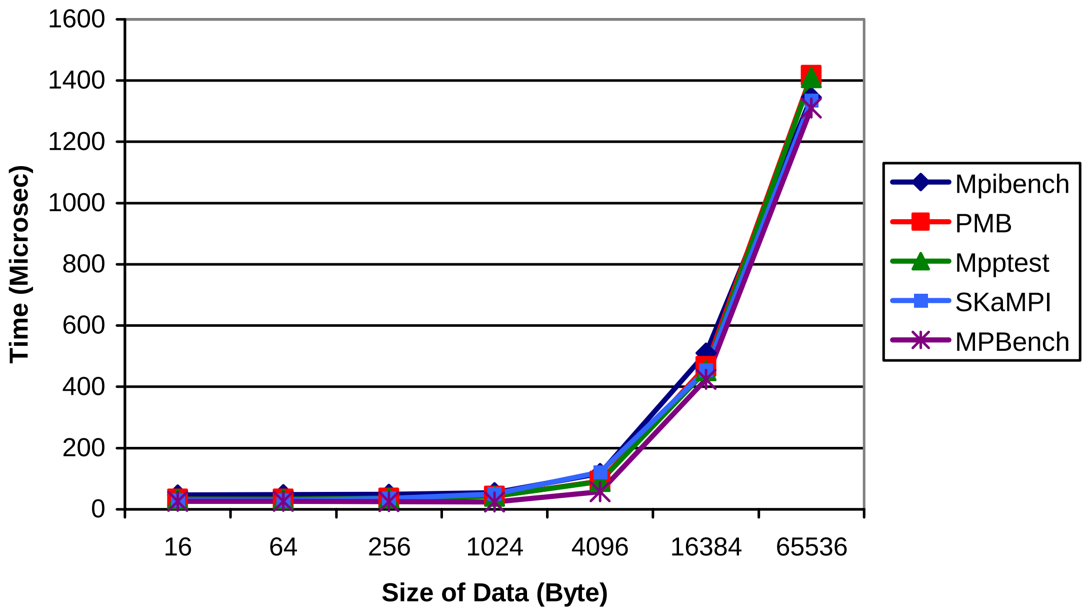

Figure 3.25 : Comparison between MPI benchmarks for MPI_Bcast on 8 processors on IBM Linux Cluster.

Both SKaMPI and MPIBench use a barrier operation to synchronize the start of all collective communications. However they also have the option of avoiding the overhead of the barrier operation by using a synchronized start, where each processor starts each broadcast at a prescribed time, and the time reported for each repetition is the time taken by the slowest process. Clearly this requires a globally synchronized clock, which is provided by MPIBench and SKaMPI. Since they both use a globally synchronized clock, they are able to generate average times for each process in a collective communication, which can be significantly different for different processes.

Figure 3.26 shows a figure from the SKaMPI report for an MPI_Bcast operation on 8 processors for SGI Altix (using cache warmup to enable a direct comparison to MPIBench results), which shows the average completion time for each processor. The SKaMPI report also states that the average time for the MPI_Bcast is about 9500 µs, which is very different to the largest times for each node shown in Figure 3.26. We are not sure why this is the case. Figure 3.27 show the distribution results for MPI_Bcast on 8 processors for the same data size on SGI Altix using MPIBench. This figure shows the combined results for all 8 processors, although recently MPIBench has been modified to allow distributions to be generated individually for each processor, so we are able to check that the overall distribution shown in Figure 3.27 shows peaks that are consistent with the binary tree broadcast algorithm, with the first peak corresponding to completion times for processors 0 and 1, the second peak is for processors 2 and 3 and final peak is for 4-7. As with SKaMPI, MPIBench gives an average time for broadcast of 9500 microsec, but unlike SKaMPI, this agrees with the value for the slowest process in the distribution of times in Figure 3.27.

Figure 3.28 and Figure 3.29 shows the same plot for the IBM Linux Cluster. However, the distribution for MPIBench does not show the effect of the binary tree broadcast algorithm. This is because the IBM Linux Cluster used the latest version of MPICH 1.2.6, which has a new algorithm for broadcast. The new algorithm uses a combination of scatter and allgather for 8 or more processors and long message sizes (greater than 512 Kbytes) [11]. For SKaMPI the average time is reported to be approximately 49000µs, but similarly with Altix, this is significantly different to the largest times for each node. We do not know the reason for this discrepancy, but it implies that the node times reported by SKaMPI may not be very accurate. As with the SGI Altix, MPIBench agrees with the SKaMPI average time of 49000 μs, but the slowest node time agrees with the average broadcast time, as it should. Figure 3.30 shows the minimum, average and maximum time for the IBM Linux Cluster on the same plot. This figure shows that the first two small peaks in the distribution plot in Figure 3.29 correspond to the minimum completion time for all processors. Then, the first high peak is for processors 0, 3, 4 and 7, the second high peak is for processor 1, 2, 5 and 6. The small distribution after the average completion time corresponds to the maximum time obtained by all of the processors, where a small number of repetitions are very slow.

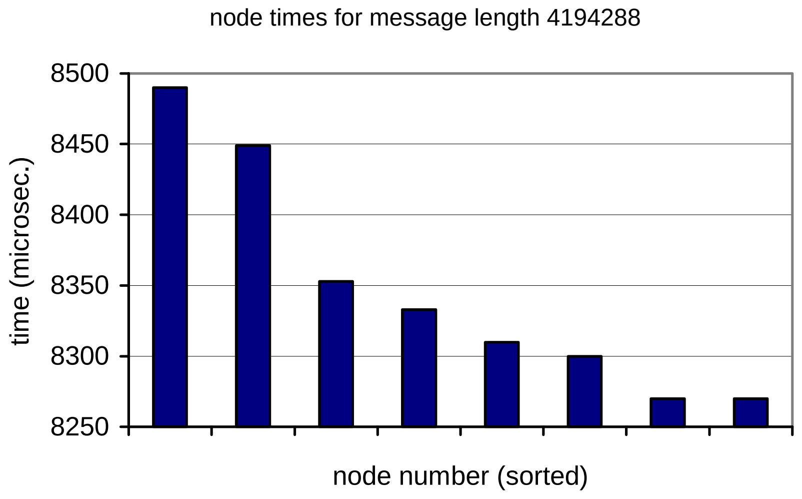

Figure 3.26 : Node time produced by SKaMPI for MPI_Bcast at 4MBytes for 8 CPUs on SGI Altix.

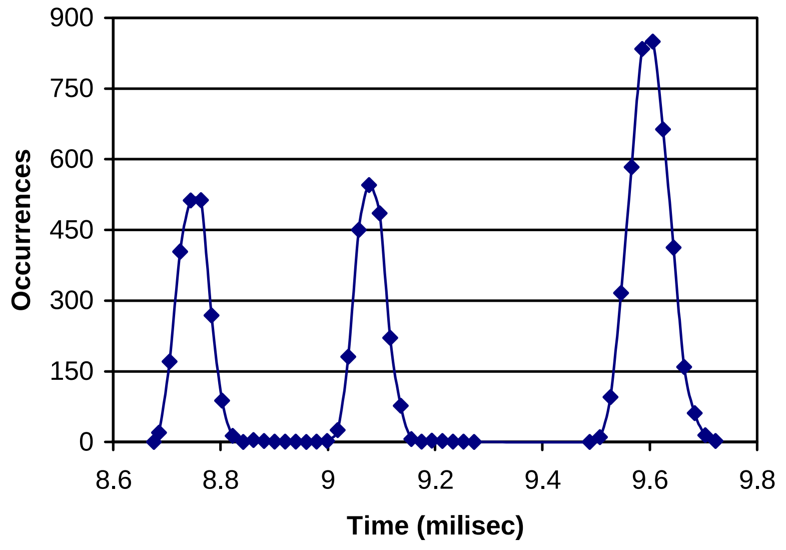

Figure 3.27 : Distribution result produced by MPIBench for MPI_Bcast at 4MBytes for 8 CPUs on SGI Altix.

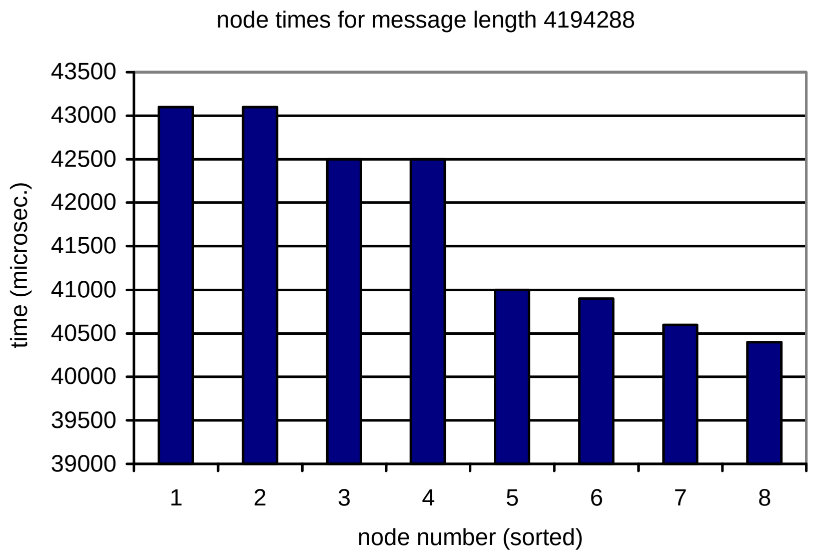

Figure 3.28 : Node time produced by SKaMPI for MPI_Bcast at 4MBytes for 8 CPUs on IBM Linux Cluster.

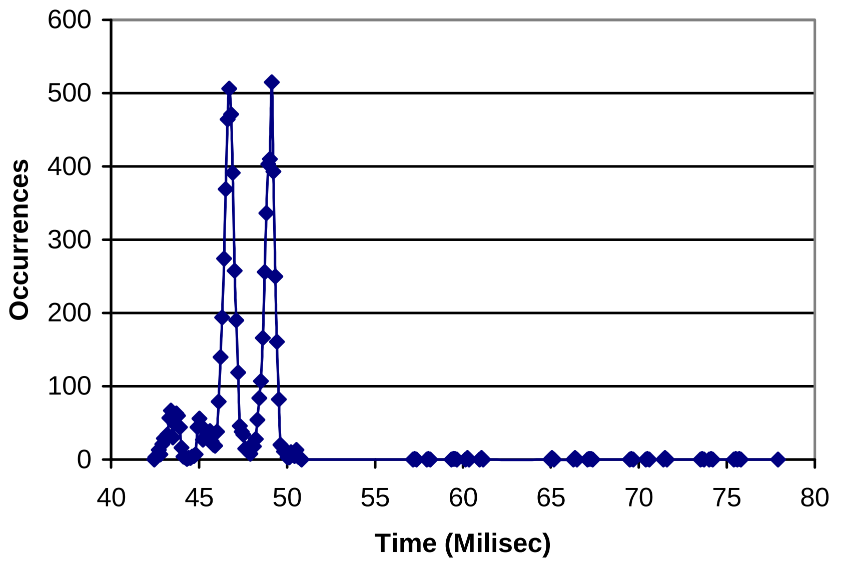

Figure 3.29 : Distribution result produced by MPIBench for MPI_Bcast at 4MBytes for 8 CPUs on IBM Linux Cluster.

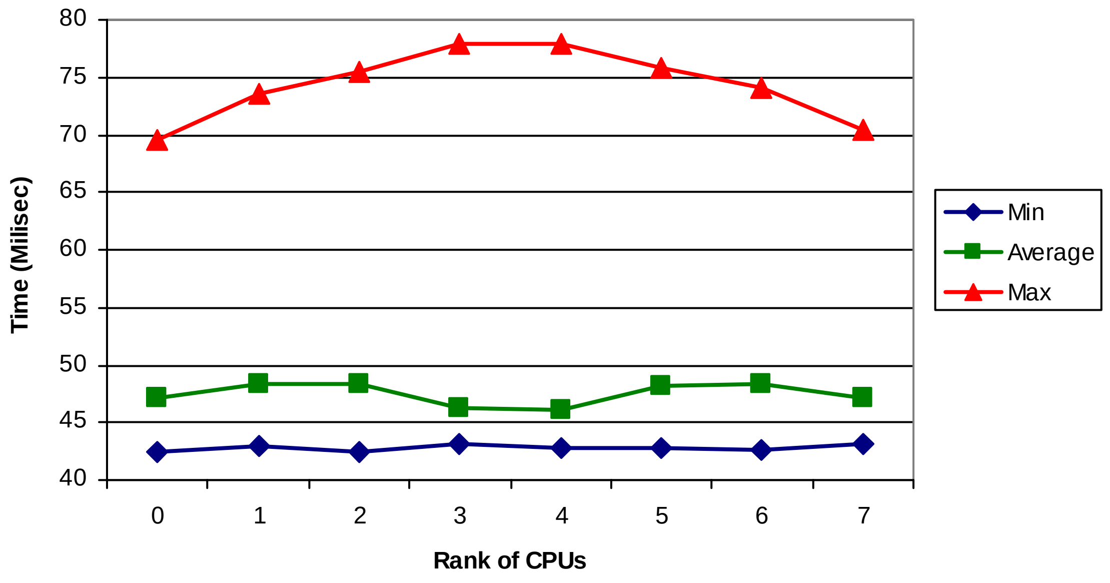

Figure 3.30 : Minimum, Average and Maximum time from MPIBench for MPI_Bcast at 4MBytes for 8 cpus on IBM Linux Cluster.

3.10 Scatter and Gather

Only MPIBench and SKaMPI provide measurements for MPI_Scatter and MPI_Gather, and both benchmarks apply the same measurement technique. Scatter and gather are typically used to distribute data at the root process (e.g. a large array) evenly among the processors for parallel computation, and then recombine the data from each processor back into a single large data set on the root process. Figure 3.31 shows the comparison between MPIBench and SKaMPI for MPI_Scatter for 32 processors on Altix. The result shows that MPIBench and SKaMPI agree with each other. The results also show an unexpected hump at a data sizes between 128 bytes and 2 KBytes per process, so that the time for scattering larger data sizes than this is actually lower. This is presumably due to the use of buffering for asynchronous sends for messages of these sizes. Note that overall, the time for an MPI_Scatter operation grows remarkably slowly with data size. Figure 3.32 shows the same plot for the IBM Linux Cluster, and as with the SGI Altix, MPIBench and SKaMPI agree with each other. On the cluster, the time for the scatter operation grows in proportion with the data size.

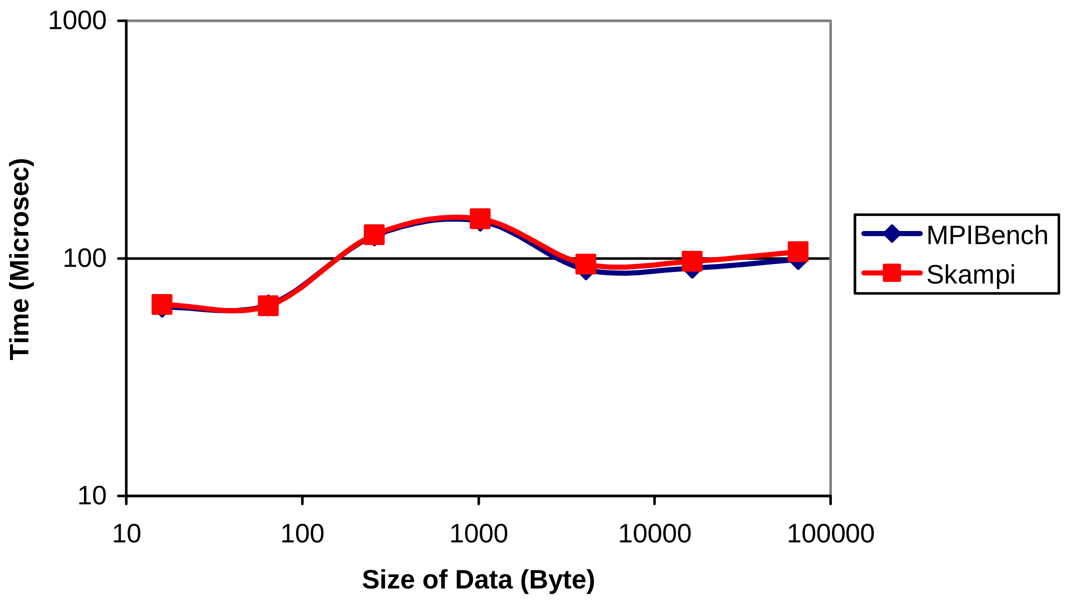

Figure 3.31 : Comparison between MPI benchmarks for MPI_Scatter for 32 processors on SGI Altix.

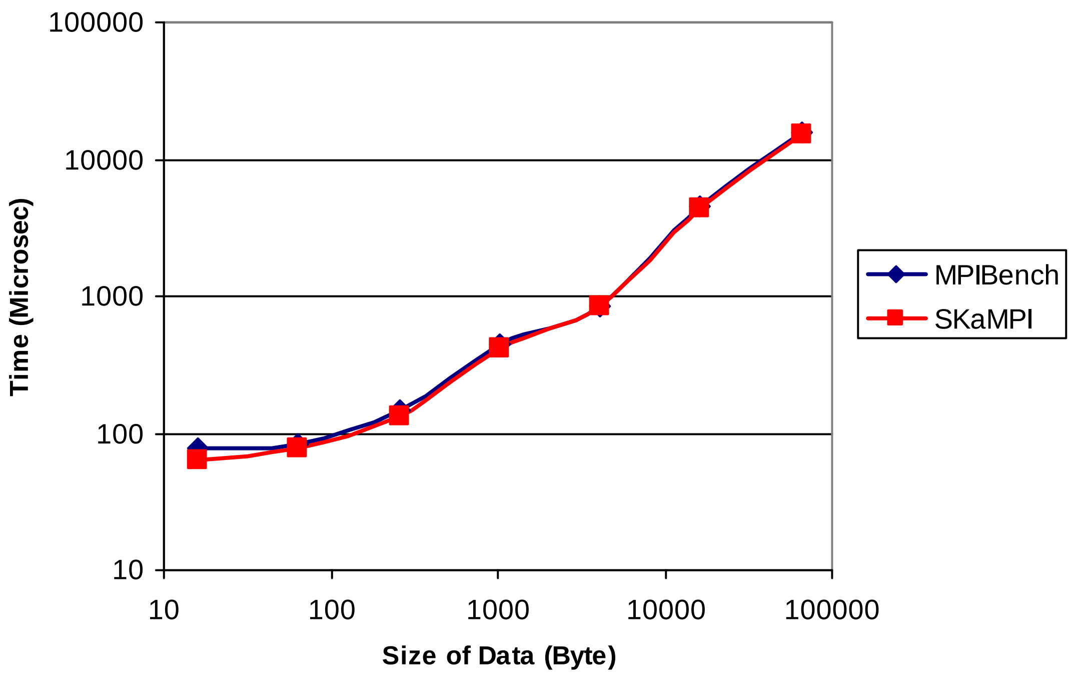

Figure 3.32 : Comparison between MPI benchmarks for MPI_Scatter for 32 processors on IBM Linux Cluster.

The performance of MPI_Gather is mainly determined by how much data is received by the root process, which is the bottleneck in this operation. Hence the time taken is expected to be roughly proportional to the total data size for a fixed number of processors, with the time being slower for larger numbers of processors due to serialization and contention effects.

Figure 3.33 and Figure 3.34 shows comparison results between MPIBench and SKaMPI for 32 processors on the SGI Altix and the cluster. Similarly with MPI_Scatter, MPIBench and SKaMPI agreed with each other. The communication time grows in proportion with the message size for both machines.

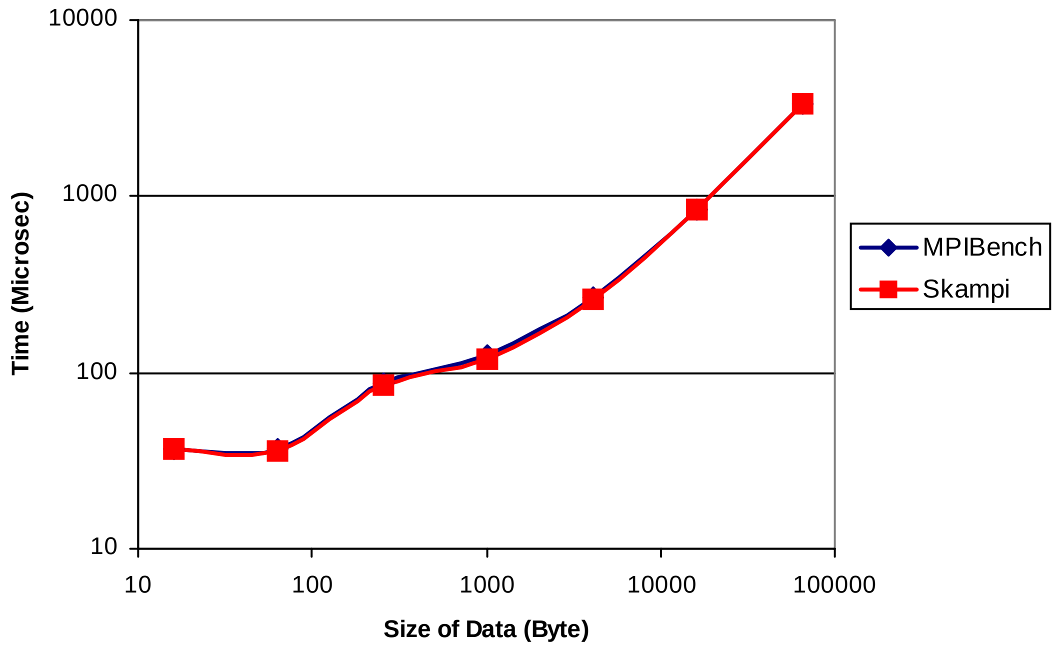

Figure 3.33 : Comparison between MPI benchmarks for MPI_Gather for 32 processors on SGI Altix

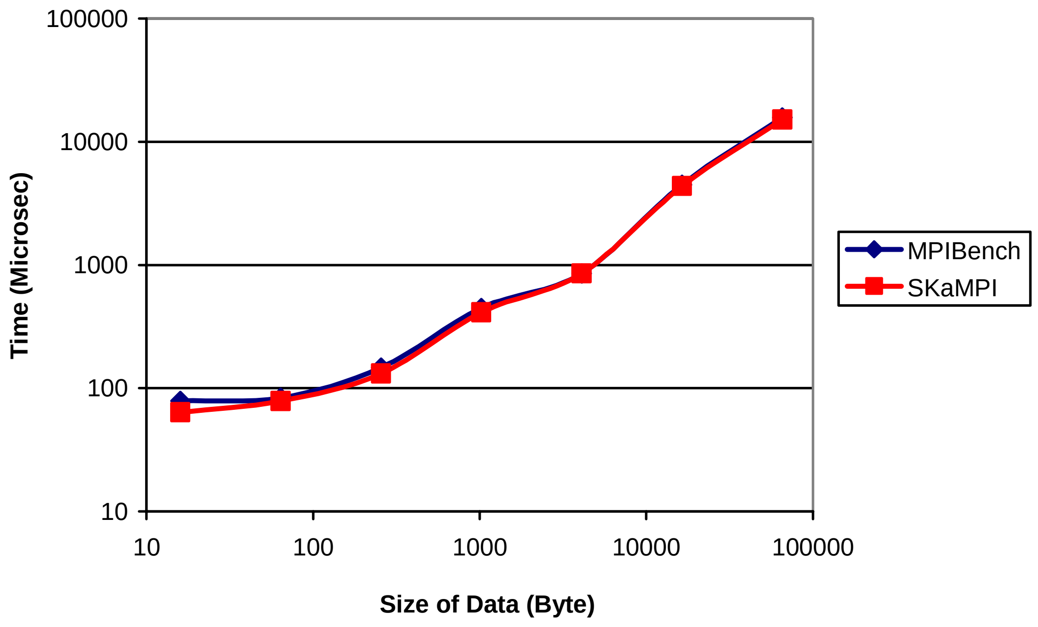

Figure 3.34 : Comparison between MPI benchmarks for MPI_Gather for 32 processors on IBM Linux Cluster.

3.11 Alltoall

The final collective communication operation that was measured is MPI_Alltoall, where each process sends its data to every other process. MPI_Alltoall is measured by MPIBench, PMB and SKaMPI. Figure 3.35 shows that the results on SGI Altix for 32 processors are similar to scatter but with a sharper increase for larger data sizes. Figure 3.36 shows the same plot for the IBM Linux Cluster, where the times increase with data size. Again similarly with MPI_Scatter and MPI_Gather, all of the benchmarks agree with each other within a few percent.

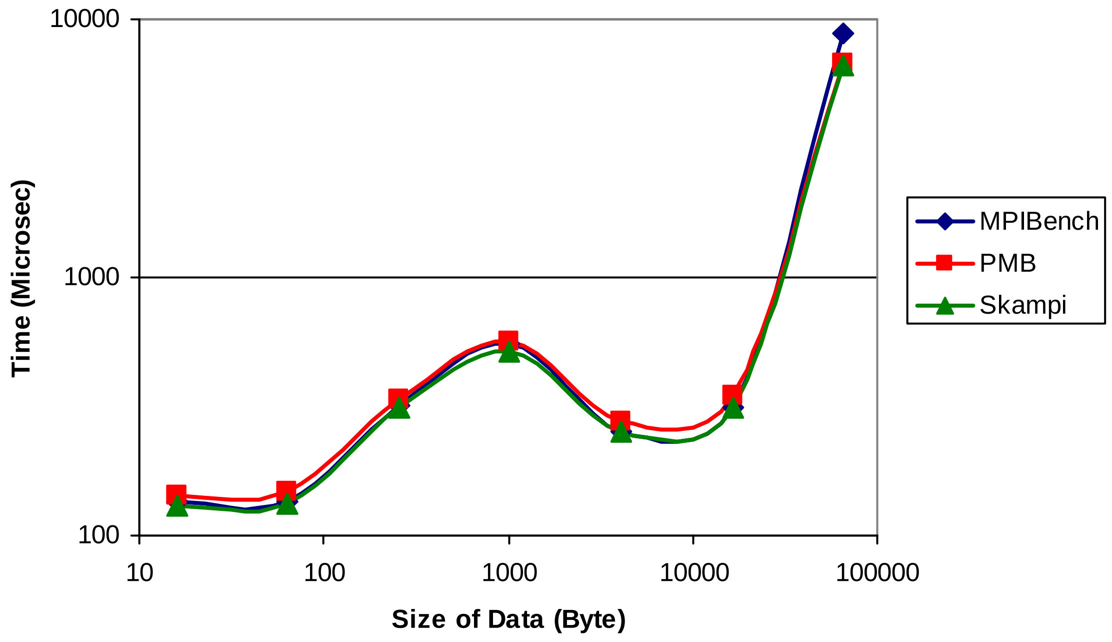

Figure 3.35 : Comparison between MPI benchmarks for MPI_Alltoall on 32 processors on SGI Altix.

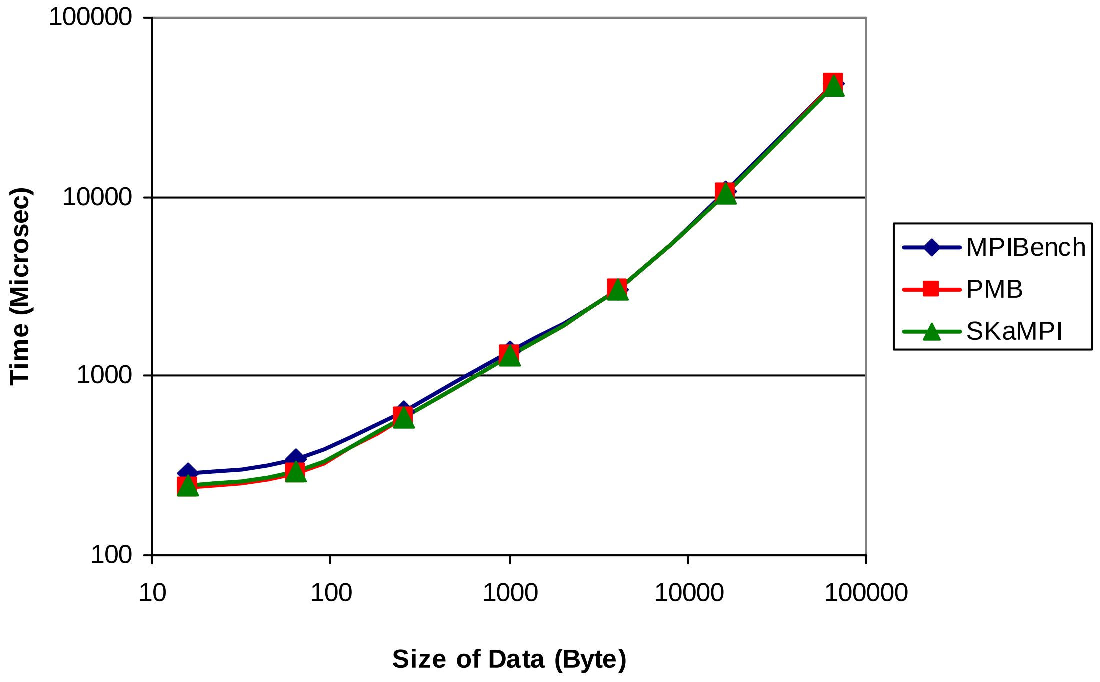

Figure 3.36 : Comparison between MPI benchmarks for MPI_Alltoall on 32 processors on IBM Linux Cluster.

3.12 Other Collective Communication

Another collective communication that is measured by PMB, SKaMPI and MPBench is MPI_Reduce. MPI_Reduce does a reduction operation such as summation of data distributed over processes and brings the results to the root process. SKaMPI and MPBench use MPI_SUM as the parameter to MPI_Reduce, and therefore do a global sum. PMB uses a null operation and therefore only measures the communication involved in the reduction operation, and hence gives very different results to the other two benchmarks.

3.13 Discussion

This analysis shows that different MPI benchmarks can give significantly different results for certain MPI routines particularly on the SGI Altix. This is primarily due to the Altix having a hierarchical ccNUMA architecture, which can enhance the variations due to different measurement techniques employed by the different benchmarks. Particularly for point-to-point communications, the variations are due to the different communications patterns used by the different benchmarks, differences in how averages are computed, errors are handles and how bandwidth is reported. There are also significant effects due to implementation details of SGI MPI on the Altix, which affects whether single copy of buffered copy is used, which has a major impact on communications speed. There are also significant differences in measurements of some collective communications routines, particularly broadcast, due to differences in use of cache and in synchronizing the calls to the routines on each processor.

MPI benchmarks were designed primarily for use on distributed memory machines, and the results show that some of the different design decisions made for the different benchmarks can significantly affect the results for ccNUMA shared memory machines. Users of MPI benchmarks on shared memory machines should therefore be careful in the interpretation of the benchmark results, and developers of MPI benchmarks may need to make some minor modifications to their codes to provide more accurate results for ccNUMA machines.

CHAPTER 4

Improvements for MPIBench

4.1 Introduction

One of the objectives for the comparison analysis between MPI benchmarks that has been done in Chapter 3 was to identify any weaknesses of MPIBench compared to other MPI benchmarks and to use this information to make improvements to MPIBench. The improvements have been implemented by a team of programmers from the University of Adelaide and the South Australian Partnership of Advanced Computing (SAPAC). The team members are Nor Asilah Wati Abdul Hamid (the author of this thesis), Alex Chichowski, Tim Seely and Paul Martinaitis. Nor Asilah Wati Abdul Hamid focused on the specification and testing of the additional functionality and identifying problems and bugs, and also implemented the user-specified point-to-point pattern and ring pattern. MPIBench has been tested on the SGI Altix (which uses a CC-NUMA architecture) and distributed memory architecture with two different types of interconnect, Myrinet and Ethernet. Many tests were done, which has helped MPIBench to be more portable and robust. The new version of MPIBench is available online at [2].

This chapter will discuss the improvements that have been done to MPIBench, based on the results of the MPI benchmark comparison in Chapter 3, The analysis from the MPI benchmark comparisons revealed several disadvantages in MPIBench and also in the course of doing the work presented in this thesis some additional useful tools have been added to MPIBench and a number of bugs and problems have been spotted and fixed. One of the disadvantages that has been solved is regarding the cache effect, whether the cache should be used or not during taking of measurements. The procedure of compiling and running the program also has been improved by adopting the approach of most of the other MPI benchmarks, by providing a default option for running the benchmark programs using defaults for configurable parameters such as the range of message sizes for each communication routine. Several new settings have also been included (with default options), including the ability to choose MPI_Wtime instead of the globally synchronized clock provided by MPIBench,

Besides improvements to address the weaknesses of MPIBench, several additional tools have been added: user-specified point-to-point communication pattern; ring pattern for point-to-point; improved measurement for collective communication; analysis of results over arbitrary sets of processes; more options to improve the ease of compiling and running the benchmarks, plotting the message sizes, and producing the output; and more information for the documentation. Additionally, this chapter also provides results from testing the new ring communication pattern for point-to-point on an SGI Altix 3000 and an IBM eServer 1350 Linux cluster, and testing of the clock synchronization mechanism used by MPIBench by comparing it with an accurate and globally synchronized implementation of MPI_Wtime. The following sub-sections will explain the details for each improvement that has been done in MPIBench.

4.2 Cache Effects

Cache effects are more important on modern CPUs, particularly machines with cache coherent shared memory architecture, for example SGI Altix (refer section 3.6.1). Typically MPI benchmark programs were developed based on older generation distributed memory machines, for which cache effects were not as important. However, in recent years more high performance computers have been developed using the architecture of cache coherent shared memory multiprocessor system. The analysis in Chapter 3 shows that if the message is accessed from the cache, the communication time will be lower than the communication time without using the cache, where the data must be accessed from memory.

There was discussion related to the cache effect from Mierendorff et al. [23] and they provide useful insights into communication performance issues related to cache effects on ccNUMA architectures. Noticeably, SKaMPI [62], Mpptest [63] and the new version of MPBench [64] addresses the cache effect issue and they all have options either to avoid using the cache or warm-up the cache before measurements are taken. However, only SKaMPI's technique affected the results, this is because it clears the cache for every single repetition, while Mpptest and MPBench only clear the cache when the message size is changing.

In a real applications, data that needs to be passed between processors may or may not be in the cache, so there is really no “right” choice for whether or not an MPI benchmark should place the data in the cache. It is useful to be able to measure results for data in and out of cache. Note that earlier versions of MPIBench only measured with data in cache. The above discussion motivated a change in MPIBench, to add the same option as SKaMPI but with a different technique for ensuring the message data is not in the cache. In SKaMPI the cache is avoided by generating new message data for each repetition using random numbers, while in the new version of MPIBench, the use of the cache is avoided by placing the message data in an array that is much larger than the cache size, to guarantee that it will be stored in memory. For each repetition, a pointer to a different part of the array is used for the message data, to avoid the data remaining in cache. Further discussion and examples on the cache effect for each MPI benchmark have been explained in Chapter 3.

4.3 Testing the MPIBench Globally Synchronized Clock

MPIBench provides an accurate and globally synchronized clock. This is enabled by the existence of 64-bit CPU cycle count registers in modern processors that are incremented on every clock cycle. These can provide greater local timing precision than has previously been possible. MPIBench also implements a global clock synchronization algorithm based on message passing [8]. In order to facilitate the use of MPIBench on parallel computers with CPUs where a 64-bit cycle count is unavailable, the use of the standard MPI timer, MPI_Wtime, is also provided. However, MPI_Wtime only can be used on machines where MPI_Wtime is globally synchronized, which can be checked using the standard MPI parameter MPI_WTIME_IS_GLOBAL, and where the granularity of MPI_Wtime is acceptably fine, which can be checked using the MPI parameter MPI_WTICK .

The implementation of MPI_Wtime in the SGI MPI library on the SGI Altix 3000 qualifies or meets the above requirements to be used for MPIBench. This enables us to do a more precise validation of the globally synchronized clock implemented within MPIBench than has previously been possible, by checking the consistency between the MPI benchmark results from using the default MPIBench clock and the results from using SGI MPI_Wtime.

The tests were done on Aquila, the SGI Altix 3000 described in Section 3.5.1. MPIBench results were obtained using the globally synchronized implementation of MPI_Wtime provided by SGI MPI (Figure 4.1) and by using the global clock synchronization techniques used by MPIBench (Figure 4.2). Note that the averages for the results obtained using these two different methods are consistent. In each case, the distributions have very similar overall shapes, the peaks occur at similar times, and the average values agree very closely. Figure 4.1 and Figure 4.2 show point-to-point communication times for 2 processors for a small message size (128 bytes), using SGI MPI_Wtime and the MPIBench clock, respectively. Figure 4.3 and Figure 4.4 show the same pattern for a large message size (256 Kbytes). There are four obvious peaks shown in Figure 4.1, followed by a long tail. In Figure 4.2 there are more peaks and finer details in the distribution. Figure 4.3 and Figure 4.4 both show a single wide peak, but Figure 4.4 again shows a finer distribution followed by the long tail.

Based on the above comparisons, the globally synchronized clock provided by MPIBench gives results that agree with the MPI_Wtime provided by the SGI MPI library, however the MPIBench clock shows finer details in the distribution results, which indicates that the MPIBench clock has higher precision than SGI MPI_Wtime. This confirms that MPIBench provides a very accurate and globally synchronized clock for its measurements of MPI performance.

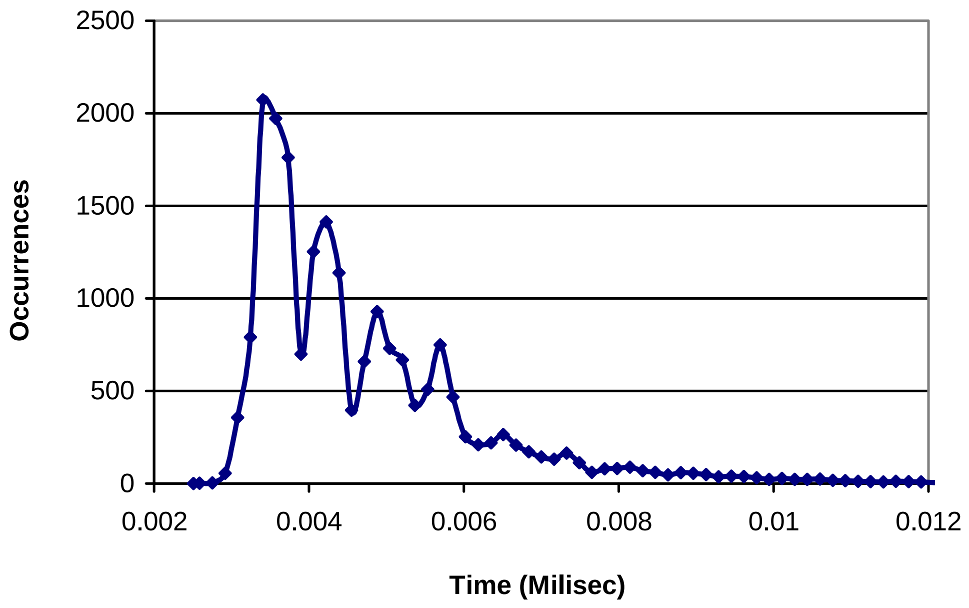

Figure 4.1: Point-to-Point with 2 processors using MPI_Wtime at 128 Bytes.

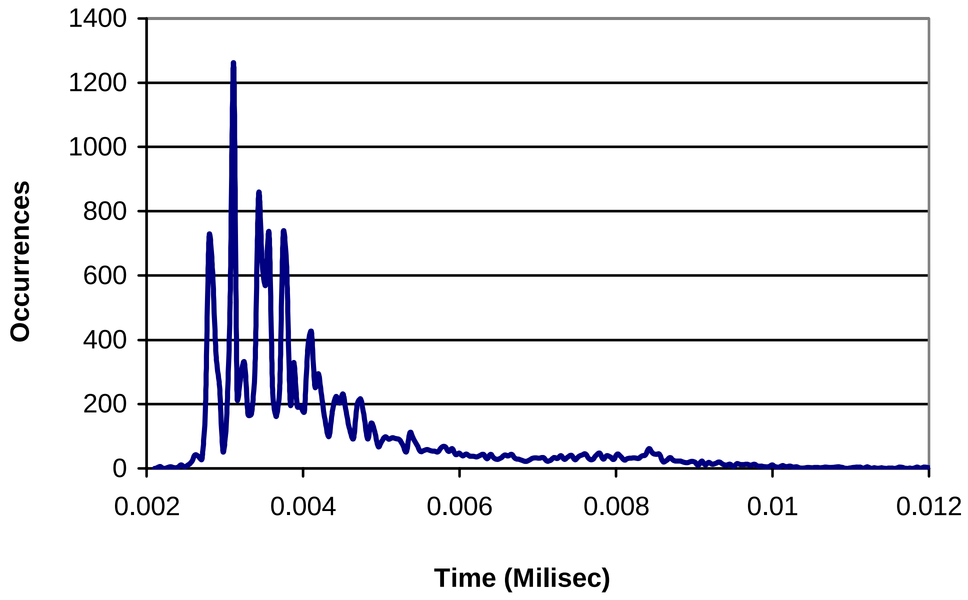

Figure 4.2 : Point-to-Point with 2 processors using MPIBench approach for global clock synchronization at 128 Bytes.

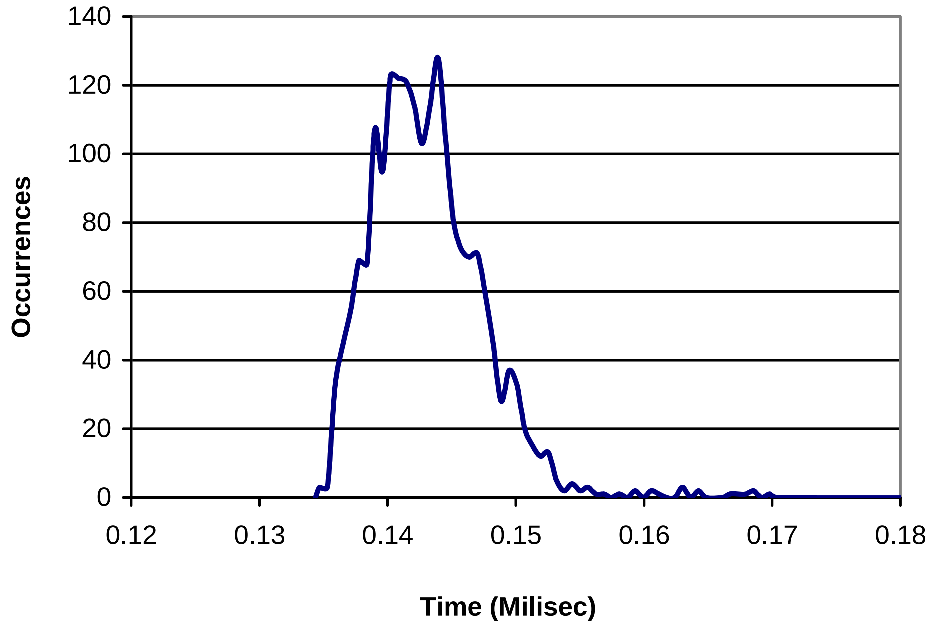

Figure 4.3 : Point-to-Point with 2 processors using MPI_Wtime at 256 Kbytes.

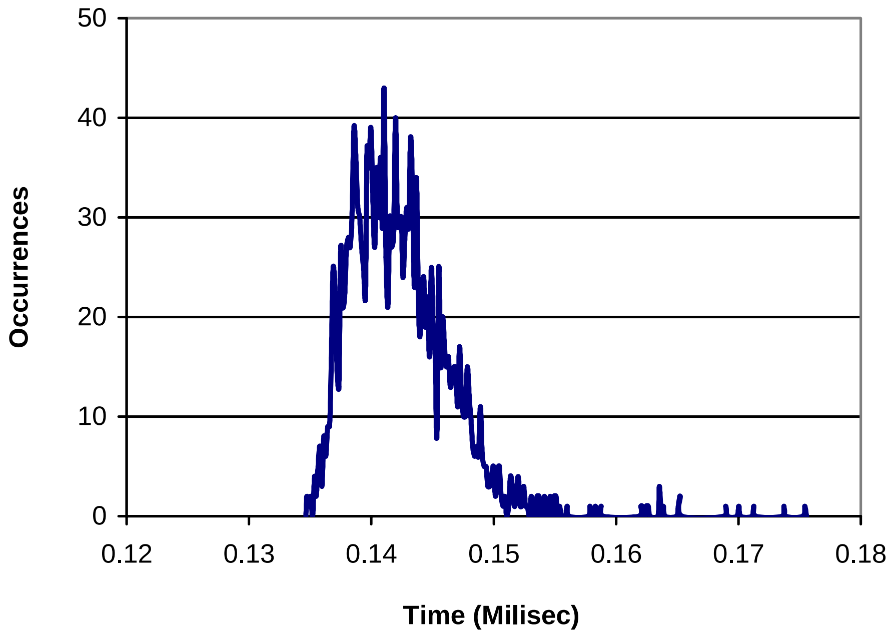

Figure 4.4 : Point-to-Point with 2 processors using MPIBench approach for global clock synchronization at 256 KBytes.

4.4 Improved Measurement for Collective Communication

Previously, MPIBench measured the average collective communication time by taking the average time from all the distribution results for all processors. However, in collective communication the time should be taken from the slowest processor to complete. The new version of MPIBench has been changed to calculate the average time by taking the maximum time from all processes for each repetition and averaging these results. SKaMPI also measures collective communication times on each process and uses the same method to calculate the average time [24]. Note that measurements done using earlier versions of MPIBench by Grove [8] required a separate analysis to get the correct results for the average collective communication time.

Figure 4.5 shows the distribution results for MPI_Bcast for 128 CPUs on Hydra using Ethernet for 64 KByte message size. If the average time is taken based on the new measurement it will be approximately 289 ms. However, if using the previous method, the time will be approximately 100 ms. Another example is shown in Figure 4.6, for MPI_Alltoall at 64 CPUs for 4 KByte on the same machine. The current reported average time is 718 ms, however if the average of all the times for all processors is used, it will be approximately 300 ms.

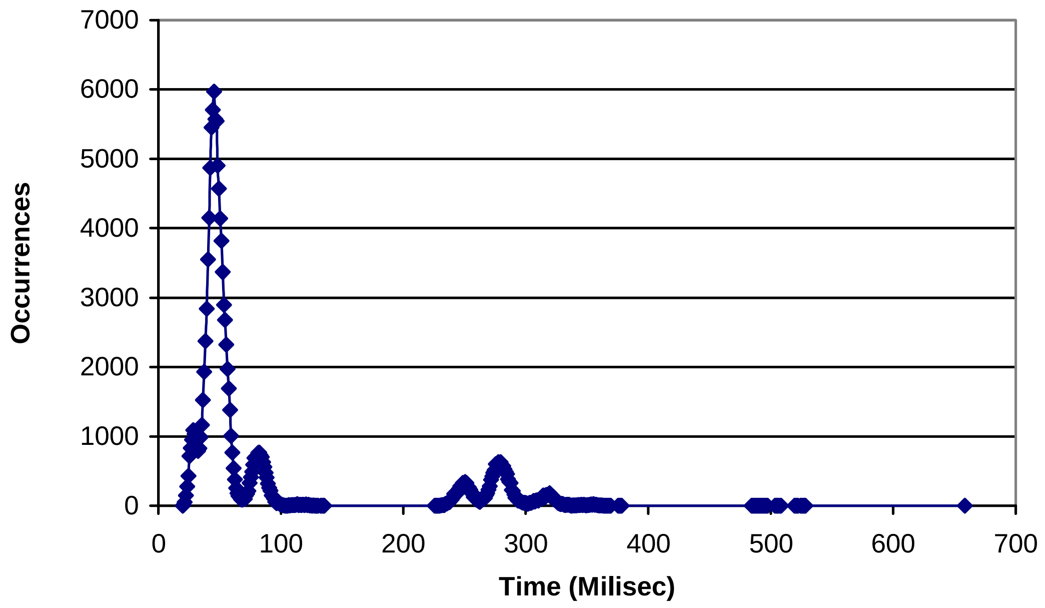

Figure 4.5: MPI Bcast on Ethernet for 128 CPUs at 64KByte.

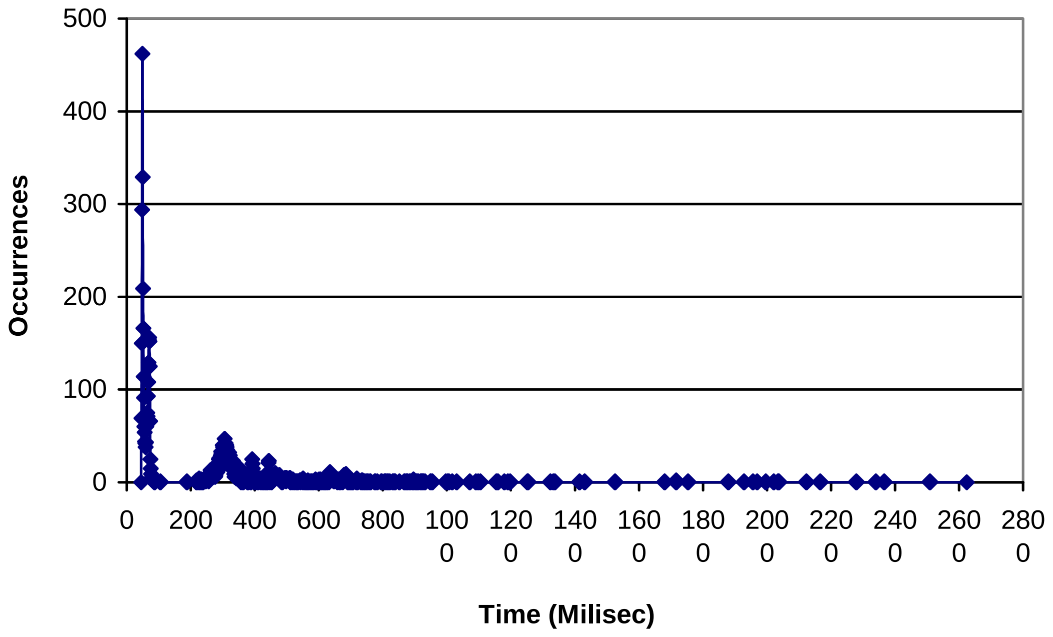

Figure 4.6 : MPI Alltoall on Ethernet for 64 CPUs at 4KByte

4.5 User-specified Communication Pattern for Point-to-Point Communications

The communication pattern used to measure point-to-point communication times in MPIBench is to set up pairs of communicating processors, with processor p communicating with processor (p + n/2) mod n when a total of n processors are used, as shown in Figure 3.9. Half of the processors send while the other half receive, and then vice versa. Thus, MPIBench measures not just the time for a ping-pong communication between two processors, but can also take into account the effects of contention when all processors simultaneously take part in point-to-point communication. PMB and Mpptest only involve processors 0 and 1. SKaMPI and MPBench use only the first and last processor. A further explanation and figures on this discussion are provided in section 3.7

In order to create more flexibility for the point-to-point communication pattern a new capability has been added to MPIBench which allows users to test any point-to-point communication pattern easily by listing pairs of communicating processes by their process number in an input file. The sequence of process numbers inside the input file should be as in the following example for 8 processes, which gives the communication pattern shown in Figure 4.7. Each communication pair should be on a different row of the input file and the program also checks that all communication pairs are correctly specified, i.e. each process number is listed once.

0	2
1	3
4	6
5	7

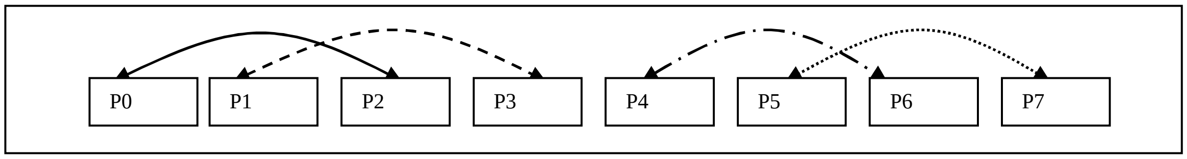

Figure 4.7 : MPIBench User-specified Point-to-Point Communication Pattern

4.6 Ring Pattern for Point-to-Point Communication

Ring communication involves each process sending to the right and receiving from the left neighbour in the process chain. This is a commonly used communication pattern for which MPI_Sendrecv is typically used. PMB and SKaMPI provide a measurement for ring communication using MPI_Sendrecv. However, MPIBench uses a different pattern to benchmark MPI_Sendrecv communication, which is the same pattern as MPI_Send/MPI_Recv. So, in order to facilitate users in measuring the performance for ring topology and to provide more options for MPIBench, the ring pattern has been added using MPI_Sendrecv point-to-point communication. The following analysis will discuss the performance measurement from SGI Altix 3000 and IBM eServer 1350 Linux cluster on the ring pattern. Note that, for sanity checking the results of both machines we have compared these with PMB and SKaMPI and agree reasonably well. Figure 4.8 shows the illustration of ring pattern on a cluster machine for 4 CPUs and 2 CPUs per node.

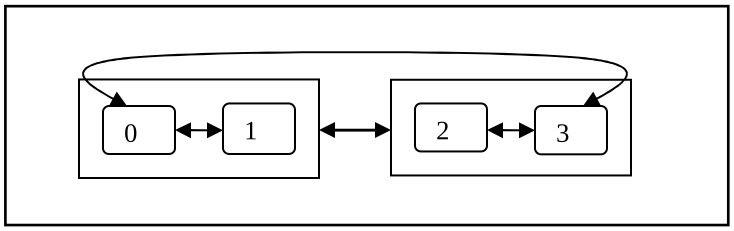

Figure 4.8 : Ring Pattern for 4 CPU and 2 CPU per node.

Figure 4.9 shows the average time on SGI Altix 3000 using ring pattern for up to 64 CPUs. The results for different numbers of processors clearly illustrate the non-uniform memory architecture of the Altix. For 4 processors the time is for internode communication within a C-Brick, which is approximately 0.009 ms for a 1 KByte message. The results for 8 processors and 16 processors are about the same, around 0.011 ms, since both communicate between C-Bricks and in the same R-Brick. Communication between 32 processors is done directly between R-Bricks, and takes around 0.015 ms. Results for 64 processors involve communication between R-Bricks through a meta-router, which is only marginally slower than direct communication between R-Bricks, however the results for MPI_Sendrecv are significantly slower, taking approximately 0.039 ms for a 1 KByte message. It is surprising that the results for 64 CPUs are almost constant with message size up to 128 Kbytes. At 256 KByte the results merge together for the different numbers of CPUs, taking around 0.08 ms, and these results are similar to sendrecv results using SKaMPI and PMB.

Figure 4.10 shows the average time for Myrinet on Hydra using ring pattern up to 64 CPUs. The results for different numbers of CPUs are closer to each other than the SGI Altix, probably because of the more uniform Myrinet network architecture. The results for different numbers of CPUs start to differ after 64 Kbyte, where the difference is approximately up to 10% and the difference is growing as the message size increases, up to 20% at 4 MBytes. The difference is higher particularly between small and large number of CPUs, for example between 4 and 64 CPUs. The increasing difference is suspected to be due to more contention occurring in Myrinet switches for larger data transfer, and perhaps contention on the network interface on each node.

Figure 4.11 and Figure 4.12 show the distributions of communication times for 4 CPUs and 256KByte messages on SGI Altix and Myrinet on Hydra, respectively. Noticeably, both figures show 2 main peaks, presumably representing times for intranode and internode communication. Note that the peaks for Myrinet on Hydra show slight double peaks, perhaps because the full bidirectional bandwidth is not obtained for MPI_Sendrecv for Myrinet with GM, so the send and recv are serialized. Grove [8] found the same problem with Myrinet on a cluster of Sun E420R SMP servers. He postulated that the reason for the limitation of Myrinet with GM layer is that the bidirectional message-passing is serialized in the GM layer implementations.

In Figure 4.11 the average time reported for SGI Altix at 256KByte is 0.08ms. The first peak, which is for process 1 and process 2, is at 0.05ms, while the second peak at 0.13ms is for process 0 and process 3. The result for send/recv for 2 CPUs at the same message size is approximately 0.14 ms, while the result using Single Copy option is 0.04 ms. So it may be that the first peak in Figure 4.11 is representing unbuffered communication for 2 CPUs, while the time for the second peak is for buffered communication. The average time reported by MPIBench is 0.08 ms, which is the average of the time for the two peaks. SKaMPI and PMB give approximately the same average time.

The average time for Myrinet on the Linux cluster for 4 CPUs at 256 KByte is 2.15ms. This result is similar to the results from SKaMPI and PMB. The result is midway between the two main peaks in Figure 4.12, where the first peak is at 0.5 ms, while the second peak is at 3.5 – 4 ms. The gap between the two peaks is larger for Myrinet on the Linux cluster compared to the SGI Altix. This illustrates the difference in the architecture between cluster and shared memory machines, since the cluster has a much bigger difference in internode and intranode communications than the shared memory machine. The same pattern as in SGI Altix obtained here, which is the first double peak is for process 0 and followed by process 3, while the second double peak is represented by process 1 and followed by process 2.

Some of the results for MPI_Sendrecv are difficult to understand, and more time than was available for this work would be required to be certain of the explanation for these results. Future work beyond the scope of this thesis could include a more detailed analysis of the MPIBench results for MPI_Sendrecv for ring communication.

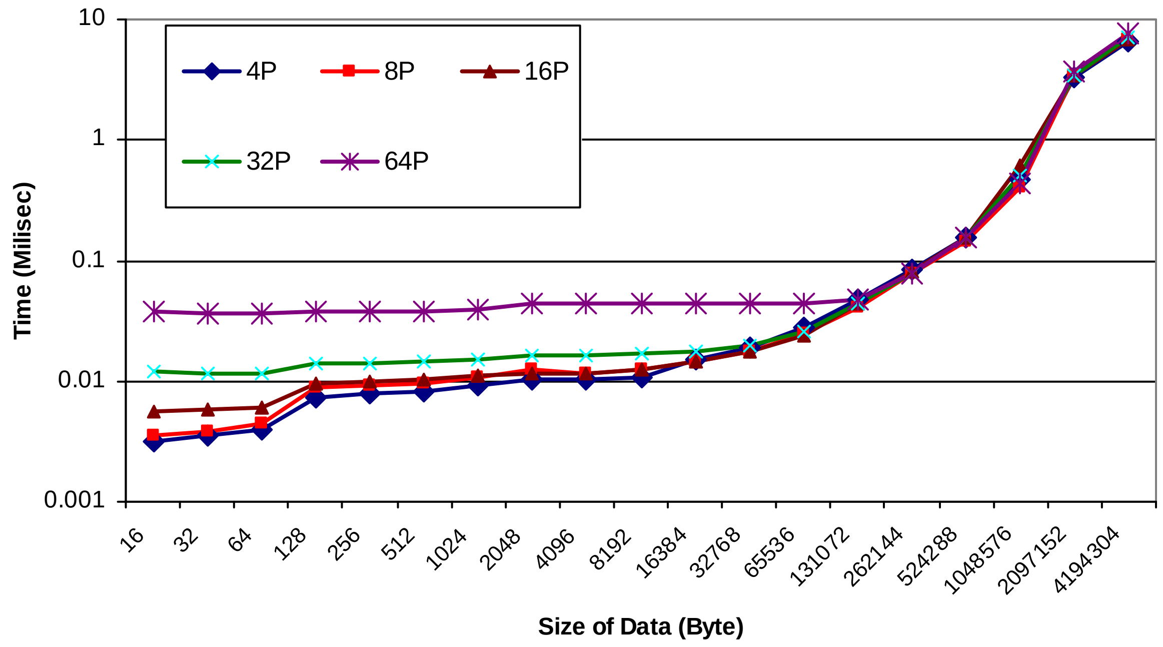

Figure 4.9 : Average times for MPI_Sendrecv with Ring pattern from 4 to 64 CPUs on SGI Altix.

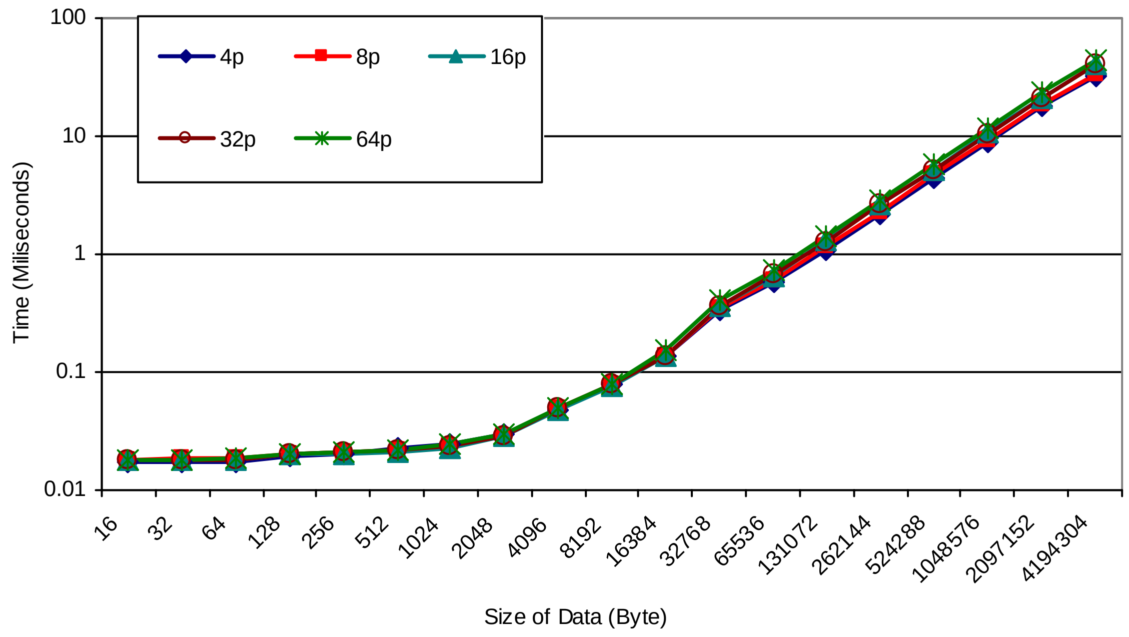

Figure 4.10 : Average times for MPI_Sendrecv with Ring pattern from 4 to 64 CPUs on IBM Linux Cluster.

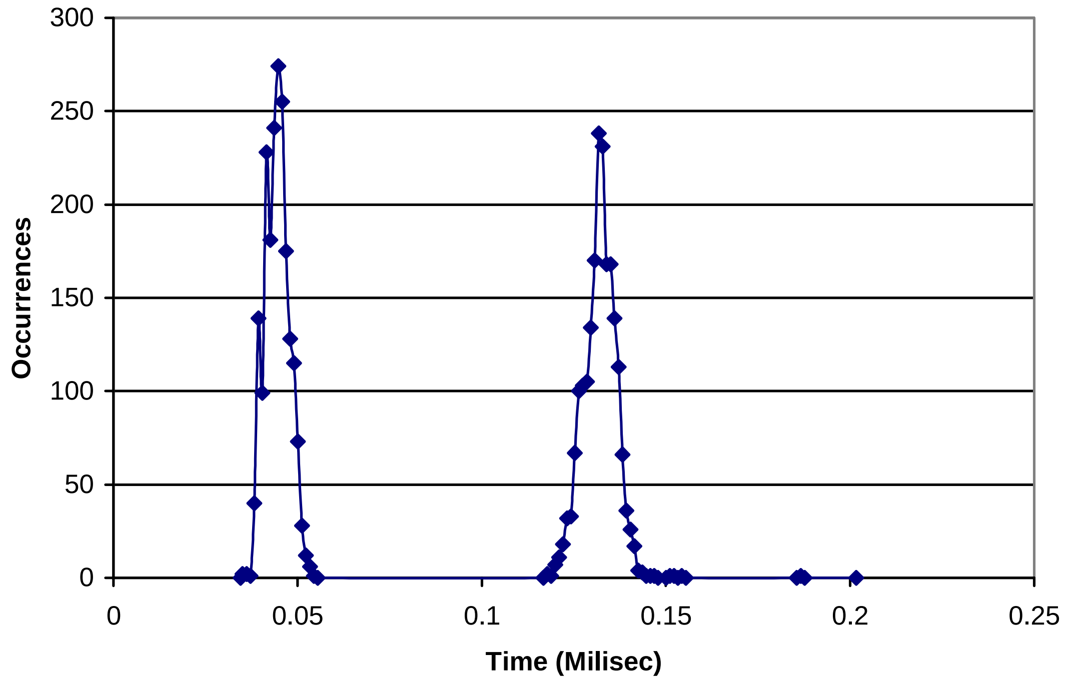

Figure 4.11 : Distribution for 4 CPUs on SGI Altix at 256KByte.

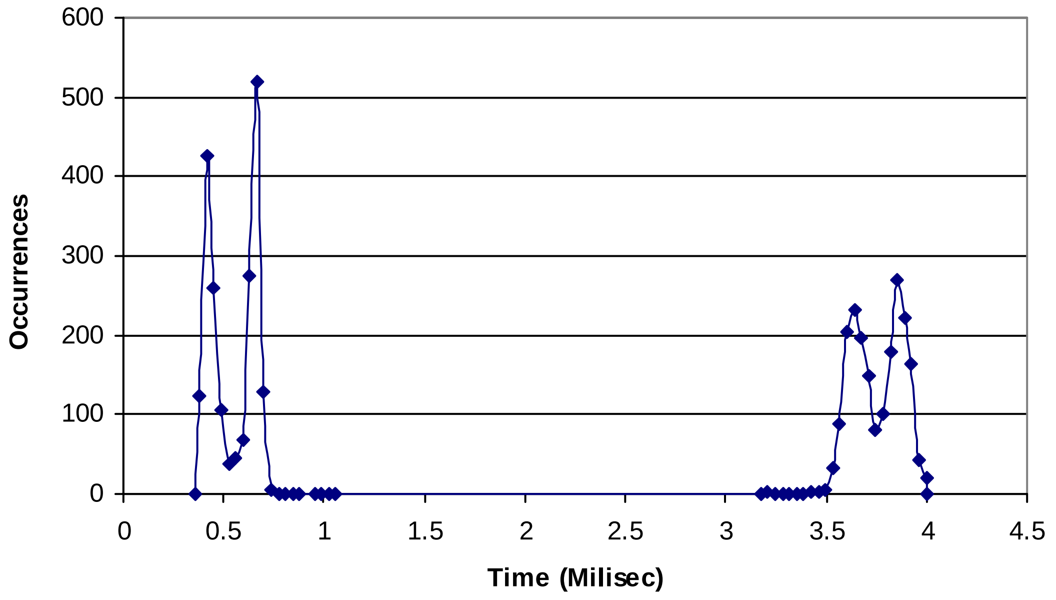

Figure 4.12 : Distribution for 4 CPUs for Myrinet on Hydra at 256KByte.

4.7 Programming Errors Fixed

A few problems in the original MPIBench code were observed during the measurements taken for collective communication on IBM eServer 1350 Linux cluster. Some of these had a big effect on the results. In certain cases, the previous MPIBench code took some shortcuts such as hardcoding certain values, for example values used to specify outliers, that should have been configurable or handled more flexibly. This caused errors including NaNs (Not a Number) for some results, particularly for collective communication. Thus, the solution to these problems was to make the code more general or with less restrictions.

Secondly, there were bus errors that would sometimes occur for MPI_Scatter. Our analysis found that the bus error was due to a memory allocation error and this has now been fixed. Furthermore, there was a problem found during the measurement for MPI_Send/Recv with the SGI Altix. It is not strictly a bug, but it caused problems for measurements using SGI MPI (refer to section 3.7.1). The problem is that while most of the benchmarks gave the expected improvement in performance by using the Single Copy option (non-buffered communication), the results for SKaMPI and MPIBench were the same as for the default MPI setting that uses buffered copy. The problem was due to both SKaMPI and MPIBench using the same array to hold send and receive message data. The MPIBench code was changed to use different arrays, which fixed the problem.

4.8 Analysis of results over arbitrary set of processes

The previous version of MPIBench was only capable of producing combined results from all the processes. The outputs are the summary of the average time for each message size, the histograms (distribution data), the gnuplot script (for plotting results) and the raw data (which is called the subsample file).

In some cases, particularly for analyzing the performance of collective communications, it would be more useful, and more in depth analysis can be done, if separate results from each of the processes could be revealed. Hence, the new version provides options whereby the users can choose which process (or processes) they would like to produce the results. With this option more detailed analyses have been done that have provided us with useful information about performance issues for both point-to-point and collective communications. Some of these findings have been reveal in the previous and in the next chapters. Examples of these results for point-to-point are in section 5.4.1, while results in section 5.6 are for collective communication.

4.9 Added Options to Ease of Use

There were several additional changes to improve the ease of use of MPIBench. Part of the changes is the auto configuration, which eases the compilation task. The auto configuration included handling the two main platform specific settings that affect portability of MPIBench, which are the mechanism for binding a process to a CPU (which is specific to the particular operating system), and for using a clock cycle counter for the MPIBench timer (which is specific to the particular CPU architecture). There were a few more options that have been considered in the auto configuration which can be referred to in [2].

Another option is the choice of message sizes for the measurements. Previously the message size is manually selected by choosing the minimum size, an increment value, and the maximum size. However, this selection will often provide too long a list of message sizes. The newer version provides more options, including the option of increasing the message size exponentially, e.g. by factors of 2. This method is often used by other well-known MPI benchmarks [17, 19, 21, 65]. MPIBench also sets default message sizes in a script to run MPIBench, so that the user can run MPIBench without having to specify the message sizes by themselves.

Besides all the above, the newer version also provides more information (and updated information) for the documentation. This is also important since it helps users to understand how the benchmark works and be aware of all the available options.

4.10 Future Work in MPIBench

MPIBench is a new MPI benchmark software that provides a more useful tool in helping researchers to analyse in detail the message-passing communication behavior of an MPI implementation on a particular parallel computer. The work described in this chapter has improved the functionality, ease of use, robustness and portability of MPIBench. Furthermore, Section 2.4 explained that MPIBench was designed to handle clusters of SMP nodes better than other MPI benchmarks, while Chapter 3 and Chapter 7 show that it works well for SMP machines. Since multi-core processors are similar to SMP nodes, MPIBench should be able to handle these new processor architectures with no change needed.

Based on our use of MPIBench and analysis of other MPI benchmarks, here are a few more suggestions for improving MPIBench that have not yet been implemented.

1. The adaptive message refinement tools that focus on message sizes where the communication time is changing rapidly, as in Mpptest and SKaMPI.
2. More MPI communication routines should be added to give more choices to the user, particularly for the collective communication.
3. Making available a variety of common communication patterns would also be useful.
4. Providing estimates of errors in the average results.
5. User-specified calculation of average time for the ring pattern, calculated based on the average time of the overall processes or the average time of the slowest processes.

CHAPTER 5

Averages, Distributions and Scalability of MPI Communication Times for Ethernet and Myrinet Networks

5.1 Introduction

Most modern parallel computers are clusters using Myrinet or Ethernet communication networks. Several studies have been published comparing the performance of these two networks for parallel computing, however these focus on average performance, and do not address the distributions of communication times, which can have long tails due to contention effects. In the case of Ethernet with TCP, retransmit timeouts (RTOs) can also occur. Slow communication events may have significant impact, particularly for applications requiring frequent synchronization, where the performance is determined by the slowest process. This chapter will analyse the distributions of communication times for standard MPI routines on Ethernet with TCP and Myrinet with GM communications networks on the same cluster, and study the scalability of the distributions as the number of communicating processes is increased, and the effect of RTOs for Ethernet with TCP.

In the past few years, commodity clusters have become the dominant architecture for high-performance computing. Currently most clusters are connected by an inexpensive commodity Ethernet network (usually 100 Mbit/s Fast Ethernet or 1 Gbit/s Ethernet, although 10 Gbit/s Ethernet is on the horizon) or by a more expensive network with higher bandwidth and lower latency (When this work began, Myrinet was the most common fast interconnect, although recently Infiniband has overtaken it). Most parallel programs that run on clusters use the Message Passing Interface (MPI) for communicating data between nodes of the clusters. It is therefore of great interest to compare the performance of MPI communication routines between different cluster communication networks, and in particular the two most common such networks, Ethernet and Myrinet.

It is well known that Myrinet with GM has significant advantages over Fast Ethernet with TCP, having much higher bandwidth (1.2 Gbit/s compared to 100 Mbit/s) and much lower latency (around 10 microseconds compared to around 100 microseconds for Fast Ethernet). Gigabit Ethernet has similar bandwidth to Myrinet, although the latency is little better than Fast Ethernet, since much of the latency is due to software overhead from the use of TCP. Some work has been done on improving Ethernet with TCP performance by developing alternative lightweight communication protocols, or by customizing the configuration of Ethernet drivers or default TCP settings (which are tuned for use over wide-area networks rather than clusters), however most Ethernet clusters use standard Ethernet and TCP. One of the problems with using TCP for communication on a cluster is that the system must wait for a specified time, known as the Retransmit Timeout (RTO), before deciding that a packet has been dropped, and retransmitting it. By default this time is very large relative to interprocessor communication times on a cluster, since is tuned for communication over a wide-area network between machines on the Internet. One of the goals of this work was to investigate in more detail the effect of these RTOs on Ethernet performance, and how much could be gained from reducing the effects of RTOs.

This chapter provides a comparison of the performance of MPI communications for Myrinet with GM and Fast Ethernet with TCP networks on the same cluster. The analysis will involve the results for both point-to-point and collective communications for up to 200 CPUs (100 dual CPU nodes), which allows in depth analysis on the scalability of the two networks to large numbers of processors. Unlike performance comparisons using traditional MPI benchmarks, which are only able to measure average communication times, the benchmark software used for this analysis is MPIBench [1,2], a recently-developed MPI benchmark that allows measurement of distributions of communication times. The distinguishing feature of this MPI benchmark is that it uses a very accurate, globally synchronized clock that is based on CPU cycle counters. This allows accurate measurement of individual MPI communications. MPIBench is therefore able to provide distributions of communication times, rather than just the average values. Also, rather than using a simple two-processor ping-pong for point-to-point communications, MPIBench measures results for N processors communicating concurrently, and can therefore measure the effects of network contention. For collective communications, it can measure the different completion times for each process. This provides greater insight into the effects of contention on network performance, the variation of communication times, and particularly the occurrence and impact of retransmit time-outs (RTOs) in Ethernet networks. For applications requiring frequent synchronization, the effects of slow communication times (i.e. the tail of the distribution, and in particular Ethernet RTOs) could have significant effects, particularly for large numbers of processors.

New clusters have Gigabit Ethernet rather than Fast Ethernet networks. Unfortunately we did not have dedicated access to a cluster with Myrinet and Gigabit Ethernet for this work, however many of the main issues addressed in this chapter, such as comparison of distributions of communication times and how RTOs affect Ethernet performance, would apply just as well to Gigabit Ethernet networks, although the values of communication times would be different. Gigabit Ethernet also supports Jumbo Frames [164, 165], which are Ethernet frames with more than the standard 1,500 bytes of payload (MTU). Most of the Gigabit Ethernet switches and interface cards support Jumbo Frames. However, all Fast Ethernet switches and interface cards support only standard-sized frames [164, 165]. The use of Jumbo Frames in Ethernet may improve MPI performance on Ethernet and reduce the effects of RTOs, however we were unable to test this because Fast Ethernet does not support Jumbo Frame. In future, we plan to include this test as part of our analysis of the MPI performance of Gigabit Ethernet networks.

5.2 Related Work

Many researchers have used standard MPI benchmarks to measure and compare the performance of Ethernet and Myrinet networks for MPI communications, although the number of published papers describing and analysing such results is fairly small. Almost all of these publications (e.g. [4,5,6,7]) measure only the average times for point-to-point (ping-pong) communications between two nodes, and do not analyse contention effects (due to multiple processes communicating concurrently) or the performance of collective communications. Grove et al. [3] have studied the effects of TCP Retransmit Timeouts (RTO) on MPI communications over Ethernet networks, however this paper is mostly focused on comparing the performance of two different cluster network topologies and only presents results for MPI_Alltoall for collective communications [3]. Many papers (e.g. [3,4,5,6]) also compare network performance using applications benchmarks such as the NAS Parallel Benchmarks, although most results are for small numbers of processors, typically 16 CPUs or less, so scalability issues are not really addressed. Some papers have analysed the effects of tuning Ethernet drivers or TCP configuration to improve MPI performance on Ethernet networks [3,4]. The work presented in this chapter compares the performance of point-to-point and collective MPI communications for up to 200 CPUs (100 dual processor nodes), using MPIBench to measure the distributions of communication times, which gives more insight into network performance, particularly contention problems and RTOs.

Grove [8] has used MPIBench to compare the MPI performance (including distributions of communication times) of Ethernet and Myrinet networks, but these were not direct comparisons since the Ethernet results were for a dual Pentium III cluster running Linux, whereas the Myrinet results were for a cluster of 4-way Sun E420R servers with SPARC 2 CPUs running Solaris. This analysis will compare Ethernet with TCP versus Myrinet with GM performance on the same Linux PC cluster. The results from Grove's comparisons, and similar work by Grove et al. [9] comparing the performance of different Ethernet network topologies in commodity clusters, showed that there were significant problems with the performance of collective communications in MPICH version 1.2.0 on Fast Ethernet networks, primarily due to the effect of TCP Retransmit Timeouts when the network becomes saturated. However, later versions of MPICH feature much improved algorithms for collective communication routines [11], which should give much better performance on Ethernet networks and perhaps reduce the number of RTOs. In this chapter the results are from the latest version of MPICH.

5.3 Methodology

The measurements reported in this chapter were done on an IBM eServer 1350 Linux cluster with 128 compute nodes connected by a Myrinet 2000 network as well as a 100 Mbit/s Fast Ethernet network. More details on this system were given in section 3.5.2. The cluster nodes were running Redhat Enterprise Linux version 3.2.3-47 with kernel 2.4.21-27.ELsmp of the Myrinet drivers, MPICH-GM version 1.2.6..14a was used with the Myrinet and MPICH version 1.2.6 for the Ethernet network. All compilations were performed with gcc v3.2.3.

Measurements of MPI communication times were obtained using MPIBench [1,2,8]. All measurements were run with dedicated access to the cluster, so there were no other processes affecting the results. At the time the measurements were taken, not all of the nodes were usable, so the analyses only took measurements for up to 100 nodes (200 CPUs). The measurement used 2 CPUs per node, with 1000 repetitions of each of the MPI operations except for MPI_Alltoall (the slowest operation) where 100 repetitions were used. Note that the number of repetitions affects the total number of occurrences on the figures showing the distributions of communication times, in particular there will be smaller numbers for MPI_Alltoall than for other MPI communications. MPIBench was run so that the message data was in cache memory.

5.4 Point-to-Point Communication

MPIBench measures not just the time for a ping-pong communication between two processors, but can also take into account the effects of contention when all processors simultaneously take part in point-to-point communication. As mentioned in section 3.7, MPIBench sets up pairs of communicating processors, with processor p communicating with processor (p + n/2) mod n when a total of n processors are used. Half of the processors send while the other half receive, and then vice versa. The send/receive pairs are chosen to ensure that for a cluster of SMPs or a hierarchical communications network, the performance of the full communication hierarchy can be measured.

5.4.1 Send/Receive

Figure 5.1 shows the average completion times for MPI_Send/MPI_Recv between Myrinet and Ethernet. The results for Fast Ethernet are about 10 times higher than Myrinet. For small message sizes this is due to the higher latency of Ethernet and the software overhead of TCP compared to the GM protocol used by Myrinet. For higher message sizes the difference is primarily due to the difference in bandwidth for each network, i.e.

100 Mbits/s for Fast Ethernet and 1.2 Gbit/s for Myrinet 2000. For a Gigabit Ethernet network, the results for larger messages would be much closer.

Note that the Myrinet performance for different numbers of CPUs is very similar, which illustrates the scalability of the Myrinet fat tree architecture, where the switch latencies are low and the available bandwidth stays constant throughout the switch hierarchy. However, for Ethernet there is a jump between 64 and 128 CPUs, which is due to the communication no longer being between processors connected by a single switch. Once this occurs, the Gbit Ethernet connection between switches becomes a bottleneck.

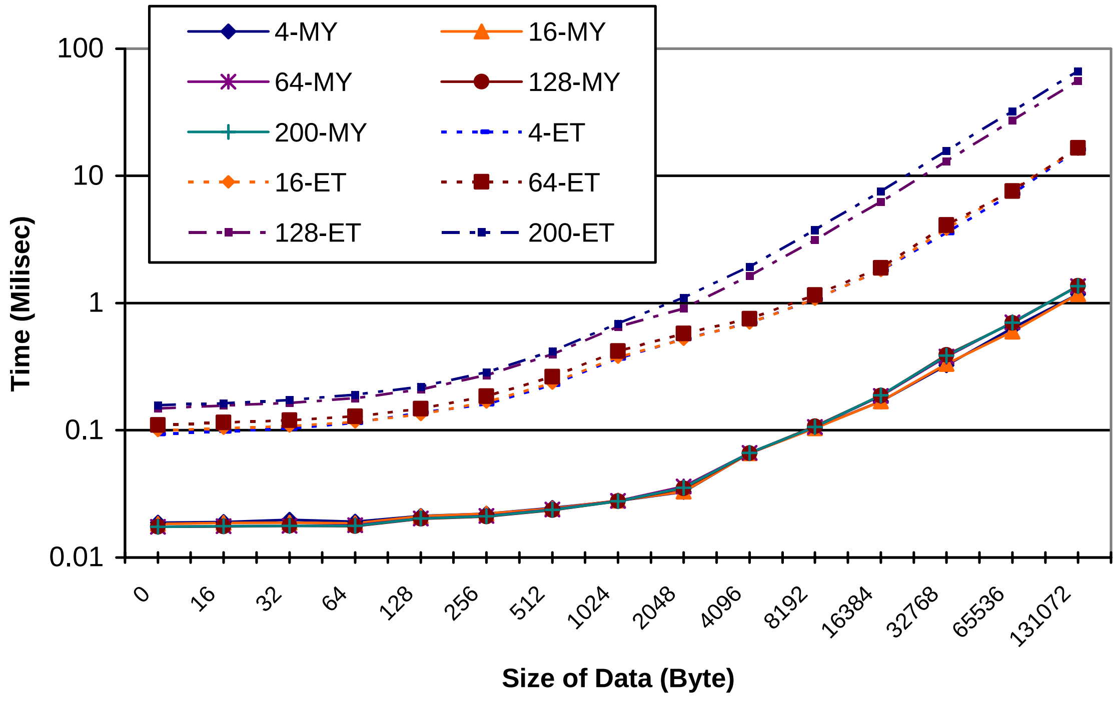

Figure 5.1 : Average Time for MPI_Send/MPI_Recv on Myrinet (MY) and Ethernet (ET).

Figure 5.2 shows the distribution of point-to-point times for 16 KByte messages on 128 CPUs using Myrinet. It shows the average is around 0.15ms and the long tail goes until approximately 1ms. Figure 5.3 shows the distribution of point-to-point communication times for 64 Kbyte messages on 128 CPUs using Myrinet. It shows that for large message sizes and numbers of CPUs, there is a wide range of completion times, due to contention effects and possibly other effects such as operating system interrupts. The minimum time is a little under 0.5 ms and the average is around 0.6 ms, and although most results are between 0.4 and 0.8 ms, there is a long tail to the distribution and some communications take several times longer than the average value. Other work [1,8,9] has shown that the distributions of point-to-point communication times across a variety of communications networks (including Ethernet and Myrinet) approximate a log-normal distribution.

Figure 5.4 shows the distribution of point-to-point times for 16 KByte messages on 128 CPUs for Fast Ethernet network. The average is approximately 8ms. Interestingly, there is a small distribution after 200 ms, which is due to the TCP Retransmit-Timeout (RTO). Then, there is also a single distribution after 600 ms, which due to a communication pair that is occasionally very slow. Figure 5.5 shows the distribution of point-to-point communication times for 64 Kbyte messages on 128 CPUs for the Fast Ethernet network. There is a main peak around the average time of 25msec, and the log-normal tail that goes out to about 5 times the average time. Then there is a gap which repeats the same pattern as in Figure 5.4, with a small peak at 225 ms followed by a reasonably long tail, then more results starting around 425 ms and 625 ms. This is due to the effect of the TCP Retransmit-Timeout (RTO), which the TCP specifications [3] say should be given by:

$$\text{RTO} = \text{SRTT} + 4 * \text{RTTVAR}$$

RTO is the Retransmit-Timeout, SRTT is the Smoothed Round-Trip Time and RTTVAR is the Round-Trip Time Variation. Both SRTT and RTTVAR are sampled and measured in the TCP stack. For MPI communications on clusters RTTVAR is quite small (unlike the variation in times for TCP packets over the Internet), however in the Linux TCP implementation the minimum time for 4 * RTTVAR is set to 200 msec. This makes the RTO approximately RTT + 200 msec for Linux. Therefore the peak at 225 msec in Figure 5.4 corresponds to the RTO, being the average communication time (SRTT = 25 msec) plus the 200 msec minimum value for 4 * RTTVAR set by the Linux kernel. The results starting around 425 msec and 625 msec are presumably caused by communications that suffer 2 or 3 RTOs before finally being completed. Therefore the default 200 msec timeout value set in Linux means that some point-to-point communications take a very long time (over 20 times the average value), although in most cases the impact on the average communication time will be fairly small, since only a small percentage of communications suffer a timeout. However for some synchronous applications where progress is determined by the completion time of the slowest process, this may have a significant effect.

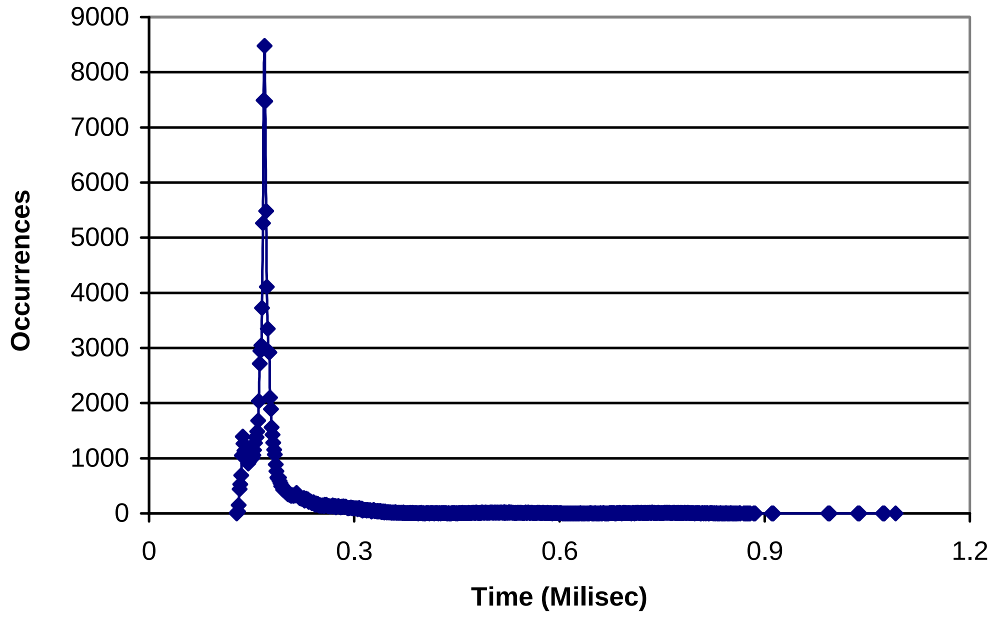

Figure 5.2 : Distribution of MPI_Send/Recv for Myrinet at 128 CPUs for 16 KByte.

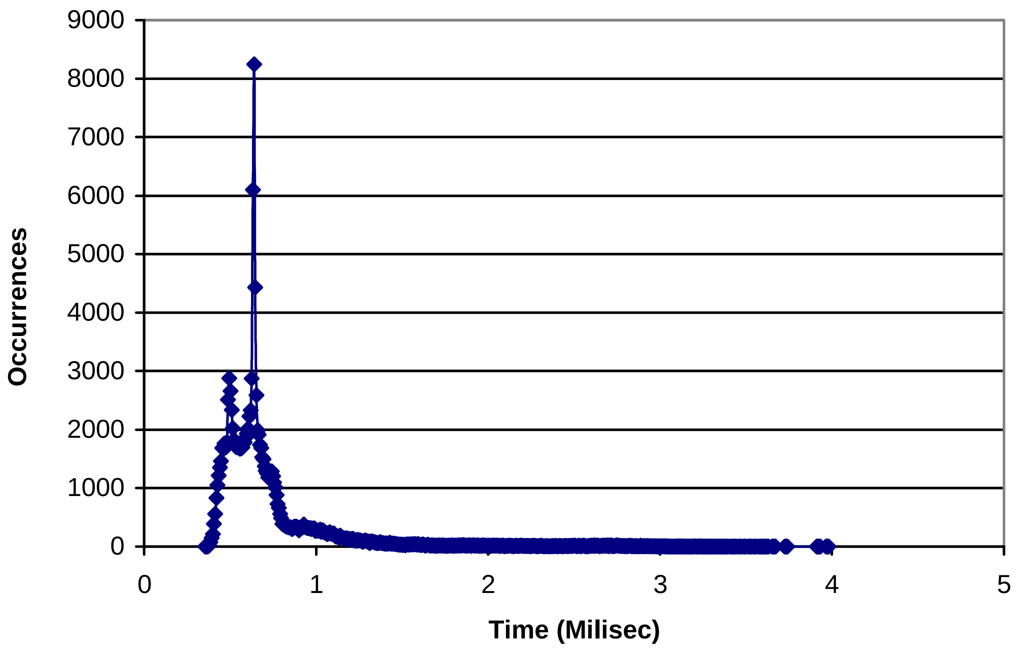

Figure 5.3: Distribution of MPI_Send/Recv times for Myrinet at 128 CPUs for 64 KByte.

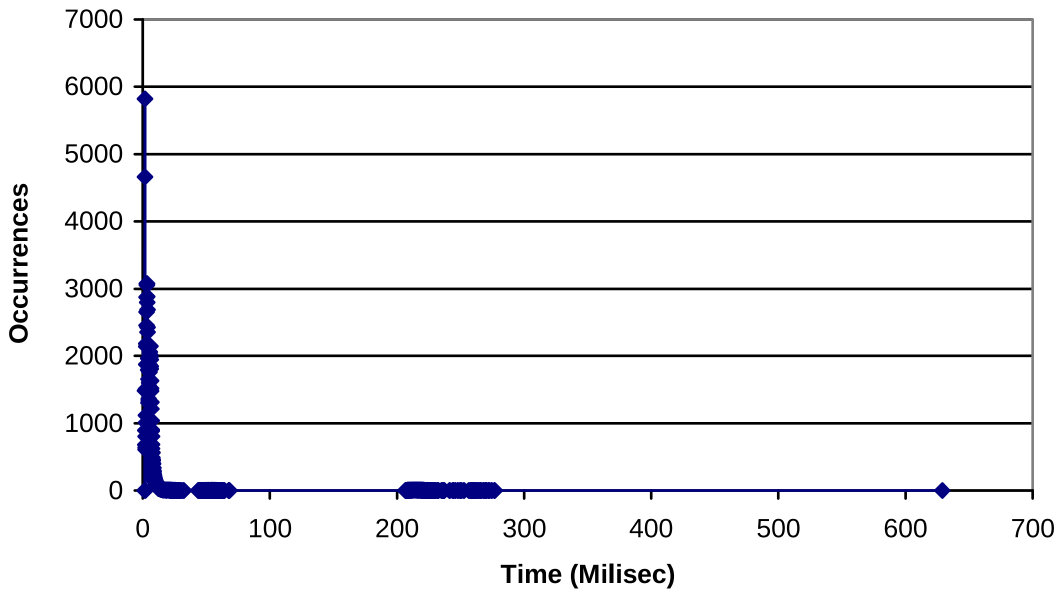

Figure 5.4 : Distribution of MPI_Send/Recv times for Ethernet at 128 CPUs for 16 KByte.

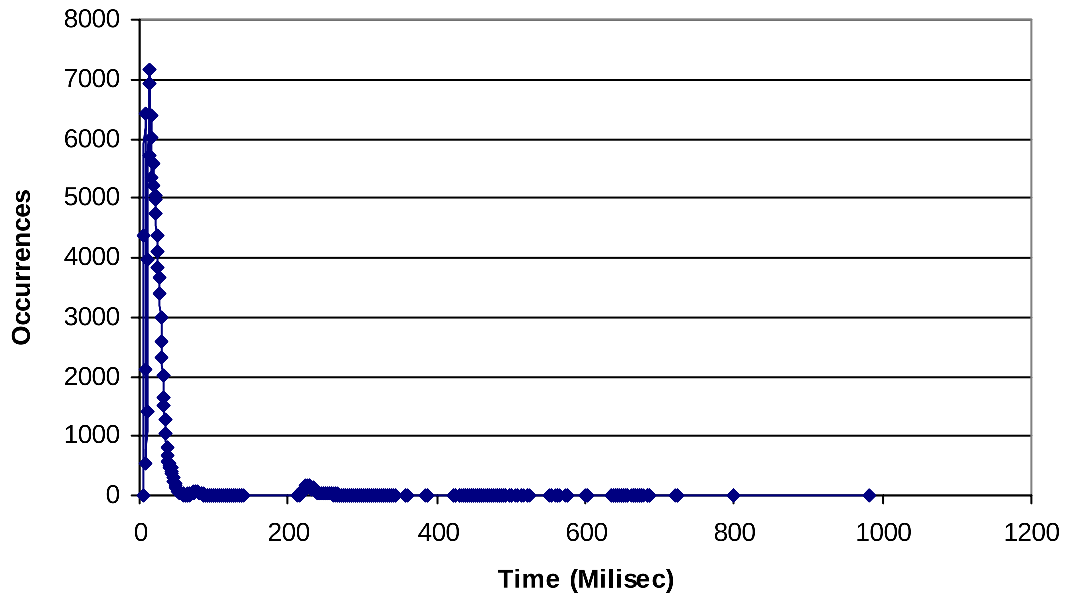

Figure 5.5 : Distribution of MPI_Send/Recv for Ethernet at 128 CPUs for 64 KByte.

In order to analyze the effect of RTO with no communication on the central switch, a check of the performance for less than 32 nodes (64 CPUs) is needed. The results showed that there were no occurrences of RTO for 64 CPUs and below for point-to-point communications. This shows that the bottleneck and contention problem at the central switch, and the 1 Gbit uplinks to the central switch, would be one of the main causes of RTO.

For any Ethernet network, the amount of RTOs will depend a lot on the network architecture. Grove [8, 9] did a comparison between two different cluster computers (Perseus and Bunyip), which both used 100 Mbit/s Ethernet but had very different switching and network topology. Perseus contained 116 dual processor nodes and was connected with a conventional stacked switch architecture by using a proprietary high speed link of 2.1 GB/s per switch, while Bunyip consisted of 96 dual processor nodes connected using Hewlett Packard ProCurve 400M Fast Ethernet switches configured in a novel tetrahedral interconnection architecture [9]. The switch used on Perseus showed a significant decrease in performance as the amount of traffic in the network was increased. In contrast, Bunyip's switches performed well with only a marginal increase in completion times, even with a significant number of communicating processes. Bunyip had a better architecture with less bottlenecks and therefore had less RTOs than Perseus, and consequently better MPI communications performance.

The rest of this sub-section discusses the scalability issues for the distributions of Point-to-Point communications as the number of processes is increased for Ethernet and Myrinet. As the message size and the number of processors is increased, the distributions of communication times broaden and the tail of the distribution gets longer, giving an increased proportion of communications that take much longer than the average time. The following analysis aims to provide a more quantitative analysis of this phenomena for the two different networks.

Two different approaches were taken in analyzing the tails of the distributions. The first was to measure the percentage of occurrences that were more than a factor of *n* times the minimum communication time, for *n*=2,3,4,… This analysis will give a comparison of the breadth of the distribution and the length of its tail, both of which give an indication of contention effects.

The second approach is to compute the percentage of measured communication times that are more than *n* standard deviations from the average time, i.e. $t_{mean}+(n * Std. Dev.)$ where n=1,2,3,4,… The rationale behind this analysis is to illustrate the skewedness of the distributions, and show long tails that might indicate deviation from the expected log-normal distribution.

The analyses were done for 8, 32 and 128 CPUs and at 16 KByte, 64 KByte and 256KByte. Figure 5.6 shows an example of the point of Min, Mean and Standard Deviation for 32 CPUs at 16 KByte for Myrinet.

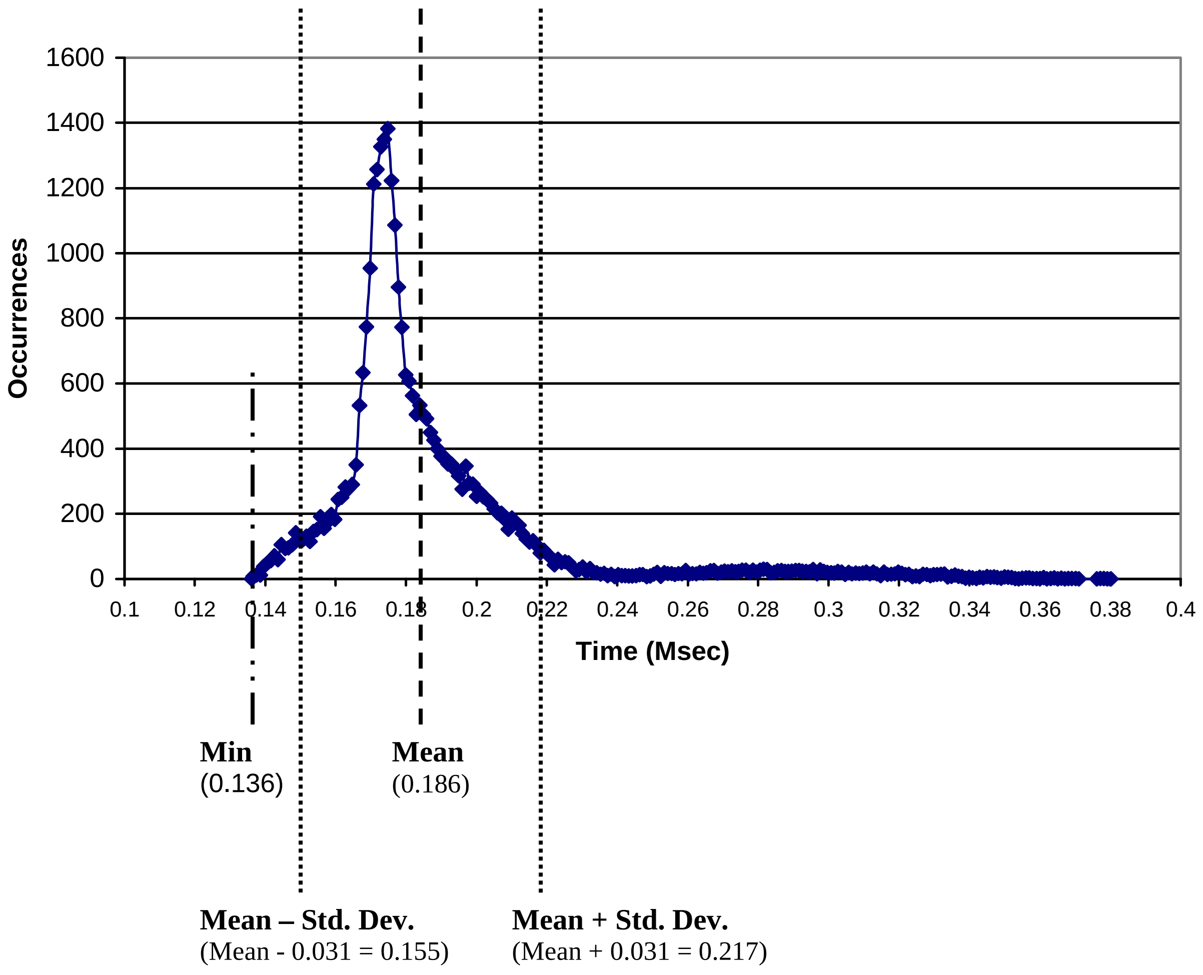

Figure 5.6: Examples of the calculation of Min, Mean and Std. Dev. for 32 CPUs at 16 KByte for Myrinet

Table 5.1 and Table 5.2 show the percentage of times that are greater than n * Min and smaller than (n+1) * Min, for both Myrinet and Ethernet. Interestingly, for 32 CPUs for all message sizes Ethernet shows a smaller percentage of times *>2 * min* compared to Myrinet. This is possibly due to the contention that occurs during communication between switches, note that for Myrinet there are only 8 nodes (16 CPUs) per switch compared to 32 nodes (64 CPUs) per switch for Ethernet. However, for 128 CPUs the results for Myrinet remain about the same, whereas for Ethernet percentage of times that are *>2 * min* increases dramatically. Ethernet performs badly because of the communication between switches for 128 CPUs. As mention in the above section, the occurrences of

RTOs and more contention occur for Ethernet when there is communication between switches.

Table 5.3 and Table 5.4 show the percentage of communication times that are between than the mean plus *n* times the standard deviation and the mean plus (n+1) times the standard deviation, for *n* = 1,2,3,4 on Myrinet and Ethernet. The results are surprisingly similar, probably because the Ethernet results have RTOs that increase the standard deviation. Myrinet has a slightly higher percentage of values that are close to the average.

Based on the scalability analyses and discussion in this section, it can be concluded that Ethernet performance is reasonably good for communication between processes on a single switch, however it experiences a significant amount of contention and RTOs for communication between switches. Myrinet performs well and scales well even for communication between switches, but still experiences some variation in communication times, presumably due to contention, particularly for larger number of CPUs.

	8 CPUs			**32 CPUs**			**128 CPUs**		
	16 KB	**64 KB**	**256 KB**	**16 KB**	**64 KB**	**256 KB**	**16 KB**	**64 KB**	**256 KB**
Time < 2x Min	100	100	99.99	95.93	75.57	71.76	92.51	72.53	54.96
2 x Min > Time < 3 x Min	0	0	0.01	4.07	20.09	19.59	5.56	19.68	30.13
3 x Min > Time < 4 x Min	0	0	0	0	1.68	5.05	0.58	4.84	7.42
4 x Min > Time < 5 x Min	0	0	0	0	2.51	0.06	0.57	1.23	3.66
5 x Min > Time < 6 x Min	0	0	0	0	0.14	2.60	0.53	0.65	1.81
Time > 6 x Min	0	0	0	0	0.01	0.94	0.25	1.07	2.03

Table 5.1 : Percentage of times that are greater than n times and smaller than n+1 times the minimum values for Myrinet.

	8 CPUs			**32 CPUs**			**128 CPUs**		
	16 KB	**64 KB**	**256KB**	**16 KB**	**64 KB**	**256KB**	**16 KB**	**64 KB**	**256KB**
Time < 2x Min	100	99.96	99.61	99.97	99.71	99.43	26.90	14.22	10.22
2 x Min > Time < 3 x Min	0	0	0.39	0.03	0.28	0.57	30.23	28.94	13.02
3 x Min > Time < 4 x Min	0	0	0	0	0	0	26.34	23.17	22.37
4 x Min > Time < 5 x Min	0	0	0	0	0.01	0	9.92	16.26	23.50
5 x Min > Time < 6 x Min	0	0.01	0	0	0	0	2.92	8.17	15.94
Time > 6 x Min	0	0.03	0	0	0	0	3.68	9.24	14.95

Table 5.2 : Percentage of times that are greater than n times and smaller than n+1 times the minimum values for Ethernet.

	8 CPUs			**32 CPUs**			**128 CPUs**		
	16 KB	**64 KB**	**256 KB**	**16 KB**	**64 KB**	**256 KB**	**16 KB**	**64 KB**	**256 KB**
Time < mean +(1xStd)	84.60	83.65	97.49	90.91	90.91	90.72	90.66	88.92	88.81
mean +(1xStd) < Time < mean +(2xStd)	14.56	16.34	2.40	3.73	3.73	5.68	5.66	7.39	7.54
mean +(2xStd) < Time < mean +(3xStd)	0.59	0.01	0.08	2.42	2.42	0.14	1.78	2.05	2.43
mean +(3xStd) < Time < mean +(4xStd)	0.19	0	0.03	2.45	2.45	3.45	0.42	0.97	1.08
Time > mean +(4xStd)	0.06	0	0	0.49	0.49	0.01	1.49	0.67	0.14

Table 5.3 : Myrinet, percentage for average plus standard deviation for n = 1,2,3,4.

	8 CPUs			**32 CPUs**			**128 CPUs**		
	16 KB	**64 KB**	**256 KB**	**16 KB**	**64 KB**	**256 KB**	**16 KB**	**64 KB**	**256 KB**
Time < mean +(1xStd)	87.34	71.20	77.28	89.95	83.58	80.09	92.66	90.39	86.48
mean +(1xStd) < Time < mean +(2xStd)	11.40	28.77	22.59	0.97	10.63	13.13	6.11	8.53	13.52
mean +(2xStd) < Time < mean +(3xStd)	1.19	0	0.04	5.87	3.52	6.69	0.97	0.96	0
mean +(3xStd) < Time < mean +(4xStd)	0	0	0.06	3.01	2.26	0.05	0.12	0.12	0
Time > mean +(4xStd)	0.08	0.04	0.03	0.21	0.01	0.05	0.14	0	0

Table 5.4 : Ethernet, percentage for average plus standard deviation for n = 1,2,3,4.

5.4.2 Combined Send and Receive

Most communication networks are capable of providing the same bandwidth if messages are sent simultaneously in both directions on the same communications link. MPI_Sendrecv provides a good way of testing that the MPI implementation can indeed provide this bidirectional bandwidth. MPIBench uses the same communication partners for MPI_Sendrecv and MPI_Send/MPI_Recv, so if the MPI implementation can make full use of duplex communication links, the results for these two measurements should be similar. However all of the results for MPI_Sendrecv, for both Ethernet and Myrinet, give results that are approximately a factor of 2 larger than the MPI_Send/MPI_Recv results shown in Table 5.5, indicating that the duplex capability of these networks is not being utilized. This may be due to limitations of the nodes, perhaps caused by bottlenecks in the network interface cards or memory accesses.

Grove [8] found the same problem with Myrinet on a cluster of Sun E420R SMP servers and with QsNet on the AlphaServer SC. He postulated that the reason for the limitation of Myrinet with GM layer is that the bidirectional message-passing is serialized in the GM layer implementations. For QsNet, Petrini et al. [13] suggest that PCI bottlenecks and DMA contention between system memory and the network interface are the cause of the unexpectedly poor performance.

The distributions for MPI_Sendrecv on Myrinet and Ethernet are very similar to MPI_Send/MPI_Recv. For Ethernet, RTOs occur for more than 64 CPUs, where inter-switch communication is required, and this affects the average communication time. There are no RTOs for less than 64 CPUs. So, the comparison of Myrinet and Ethernet for MPI_Sendrecv is basically the same as for MPI_Send/MPI_Recv. Figure 5.7 shows the average time on Ethernet and Myrinet for MPI_Sendrecv. Note that the pattern of the figure is the same as MPI_Send/MPI_Recv (Figure 5.1) except that the values are increased by approximately a factor of two.

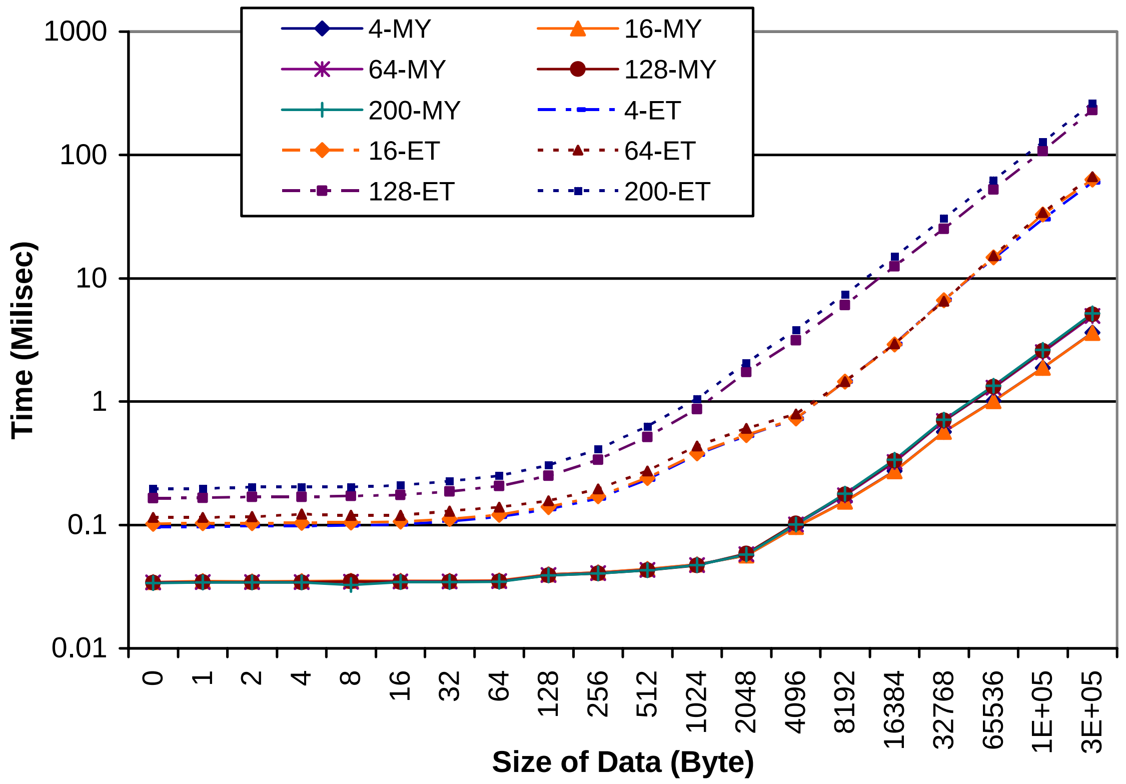

Figure 5.7 : Average Time for MPI_Sendrecv on Myrinet (MY) and Ethernet (ET).

	Ethernet		**Myrinet**	
No. of CPU	**MPI Send / MPI_Recv**	**MPI_Sendrecv**	**MPI Send / MPI_Recv**	**MPI_Sendrecv**
4	30.19	59.65	2.29	3.62
16	33.46	62.99	2.27	3.61
64	33.46	66.61	2.90	4.96
128	119.85	230.31	2.74	5.09
200	134.64	261.86	2.74	5.21

Table 5.5 : Comparison for MPI_Send/MPI_Recv and MPI_Sendrecv Between Myrinet and Ethernet for 256 KByte messages.

5.5 Barrier

Figure 5.8 shows the comparison of times for MPI_Barrier. As expected, the time grows approximately logarithmically with the number of processors, although Ethernet is approximately 4-5 times slower than Myrinet. The reason for the big jump in the Ethernet result for 200 CPUs is probably due to a different algorithm being used in MPICH 1.2.6 code when the number of CPUs is not a power of two [69], although this is not noticeable for smaller number of CPUs, for example 40 or 48 CPUs. Figure 5.9 shows the distribution of the communication times for Ethernet for 64, 128 and 200 CPUs. The peak for 200 CPUs corresponds to the average value, so it shows that the jump in the average value is not due to some anomalous large values or RTOs that are dragging up the average, which can cause results such as this, as will be seen in the next section. Figure 5.10 shows that MPI_Barrier on Myrinet performs as expected, with the completion time gradually increasing as more processes participate.

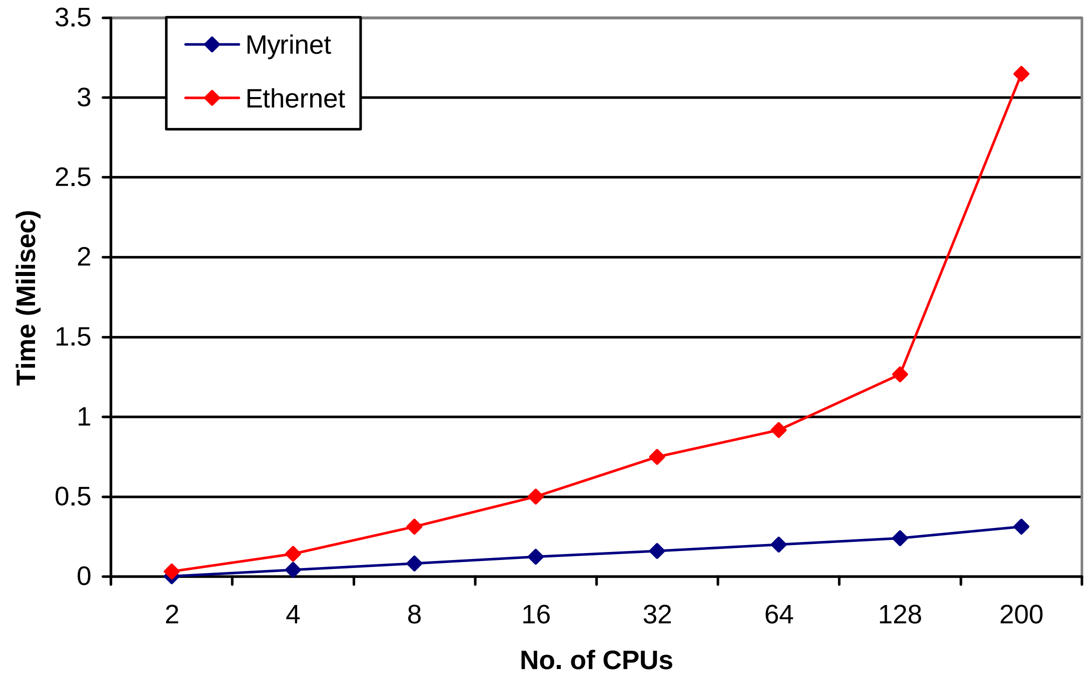

Figure 5.8 : Average time for MPI_Barrier Myrinet and Ethernet.

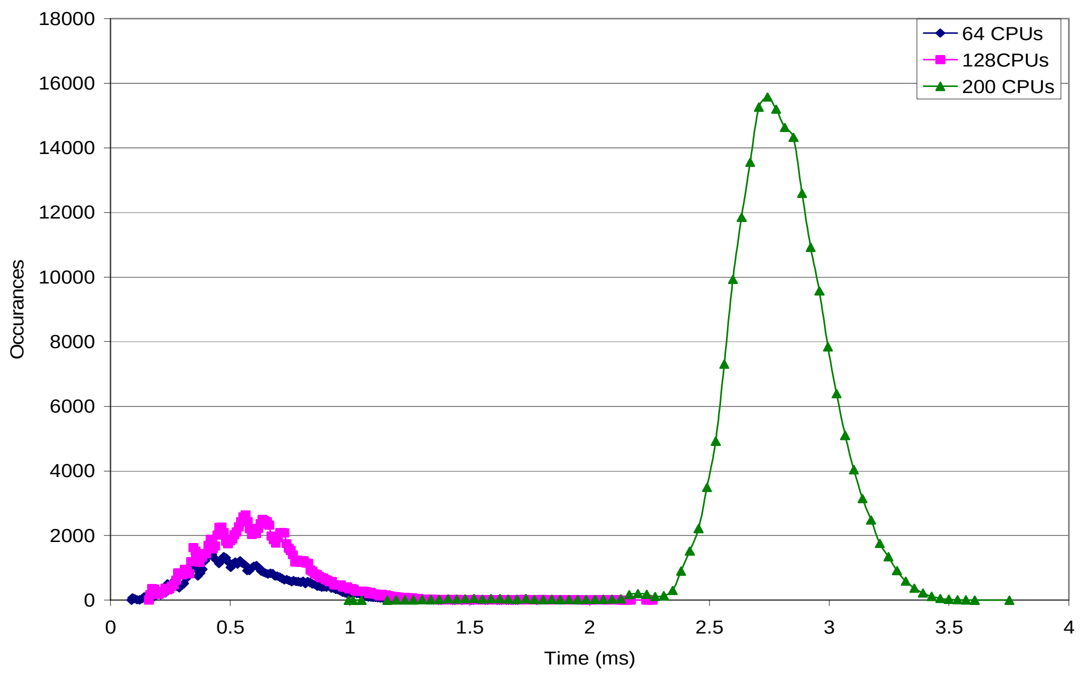

Figure 5.9 : Distribution of MPI_Barrier times for Ethernet at 64,128 and 200 CPUs.

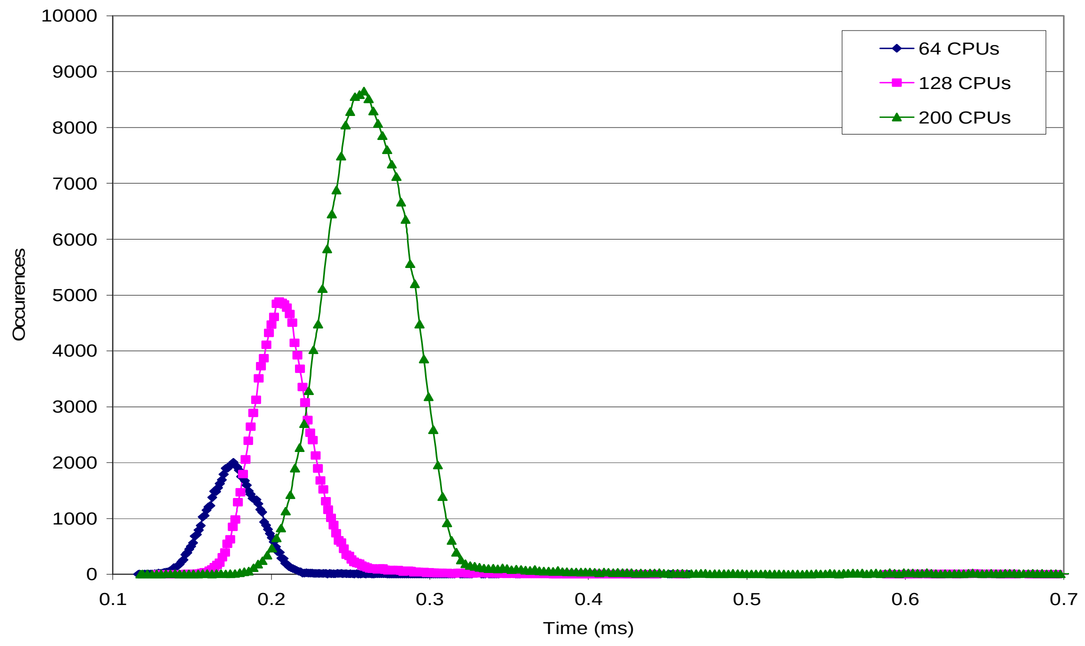

Figure 5.10 : Distribution of MPI_Barrier times for Myrinet at 64,128 and 200 CPUs.

5.6 Broadcast

MPICH 1.2.6 uses a new broadcast algorithm [11,69]. For small message size(<12KByte) and for less than 8 CPUs the binomial tree algorithm is used, while for medium (12KByte < medium < 512KByte) and long message size (>512KByte) it uses the scatter followed by allgather algorithm. Furthermore, the allgather algorithm for medium message size and for power of two number of processes uses the recursive doubling algorithm, while for medium message size and for non power of two number of processes and also for long message size, the ring algorithm is used.

Figure 5.11 and Figure 5.12 show the average completion time for MPI_Bcast for Myrinet and Ethernet. The average times for both networks increases gradually as the message size and number of processes is increased. For 200 CPUs both networks show the same pattern, with a jump at 1 KByte and again at 16 KByte. The precise reason for the jump at 1 KByte is still unknown, but a similar jump for 48 and 80 CPUs is also observed. So, it is suspected that the cause of the jump is related to the fact that the number of CPUs is not power of two. The jump at 16 KByte is clearly due to where a different set of algorithms is used [11,69]. For Ethernet there is also a large jump for 128 CPUs at 16 KByte. The reason for the large increase for Ethernet is clear when the distributions of communication times is analysed - it is caused by retransmit timeouts.

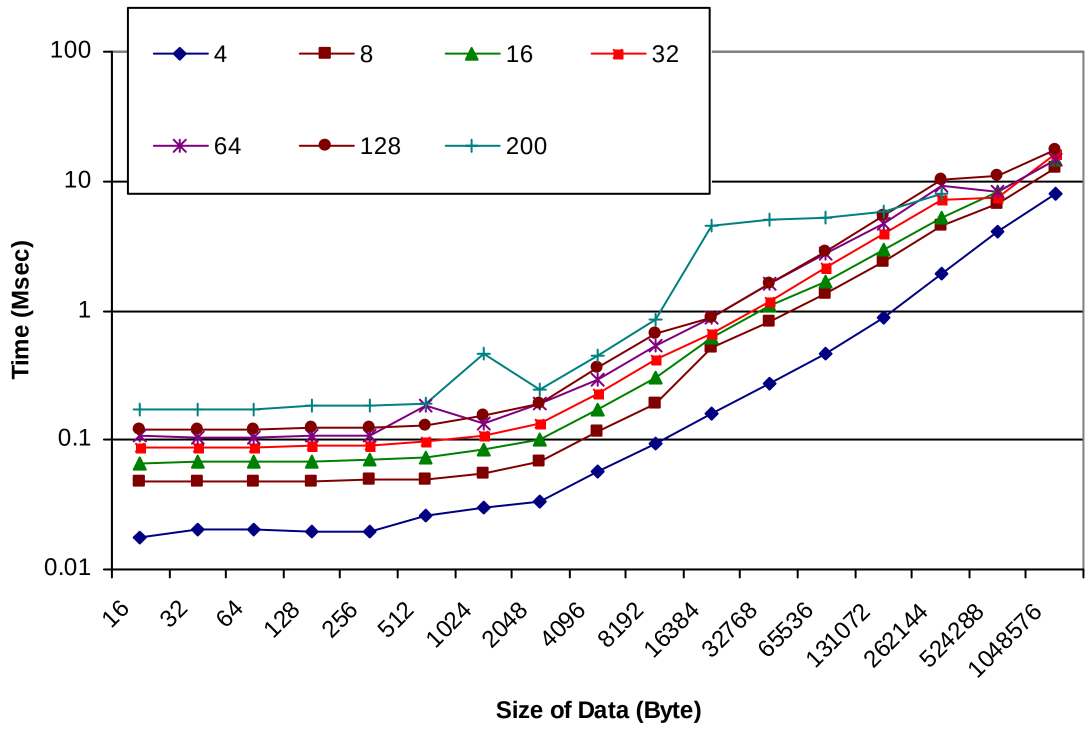

Figure 5.11 : Average time for MPI_Bcast on Myrinet

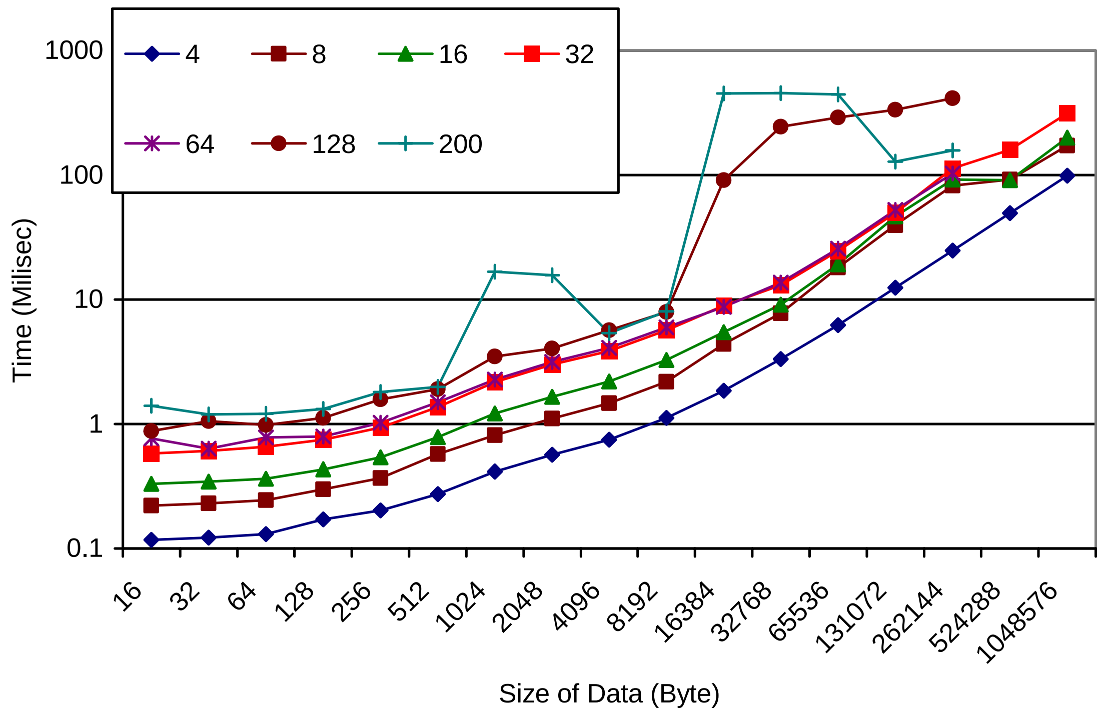

Figure 5.12 : Average time for MPI_Bcast on Ethernet

Figure 5.13 shows the distribution of times for 128 CPUs at 256 KByte for Myrinet. The main peak shows a fairly narrow log normal distribution, with a much smaller average time than for Ethernet. However there are a small number of measurements, corresponding to just a few iterations that are much slower than the rest, which create a long tail after the main peak.

Figure 5.14 shows that for 128 CPUs for Ethernet, there are no retransmit timeouts for 8 KByte messages, although there are a few larger times in the distribution, that were due to one of the 1000 repetitions taking significantly longer. A small number of RTOs start to occur for 16 Kbytes, as seen in Figure 5.15, and a few iterations have two RTOs, giving a time over 400ms. The number of RTOs increases significantly for larger message sizes, as seen in Figure 5.16 to Figure 5.18.

Table 5.6 shows the percentage of measured communication times that have RTOs, which grows to be a substantial fraction of the total number of measurements. The table also shows an estimate of the time that the broadcast would have taken if there were no RTOs, which can be up to half the actual measured time. Note that the times shown in Figure 5.14 to Figure 5.18 are times for all repetitions and all processes. The time for a collective communication is measured as the time for the slowest process, so only one of the processes has to suffer an RTO for it to affect the average broadcast time.

The results presented in Table 5.6 and Table 5.7 are calculated based on the following technique. The range of measurements with no occurrences of RTO is taken by the minimum communication time plus 200ms (t_m + 200ms). The reason 200ms is chosen for the above estimation is because that is the minimum time sets for *4 * RTTVAR* in the Linux TCP implementation [3], as explained in Section 5.4.1. So, the percentage of times without RTOs is calculated from all occurrences of communication times between t_m and t_m + 200ms. Another 200ms will be added (t_m + 200ms + 200ms = t_m + 400ms) to make it as the maximum point for the first RTO. Then the percentage of measurements with one RTO (1xRTO) will be calculated from t_m + 200ms until t_m + 400ms. Next, the percentage with two RTOs (2xRTO) will be calculated from t_m + 400ms until t_m + 600ms. This is continued until all measured times were accounted for. Finally, the estimation of the average communication time without RTOs is calculated by taking the average from the 10% of maximum time data without the occurrences of RTO. This is because for collective communication the time is taken from the slowest processors to complete.

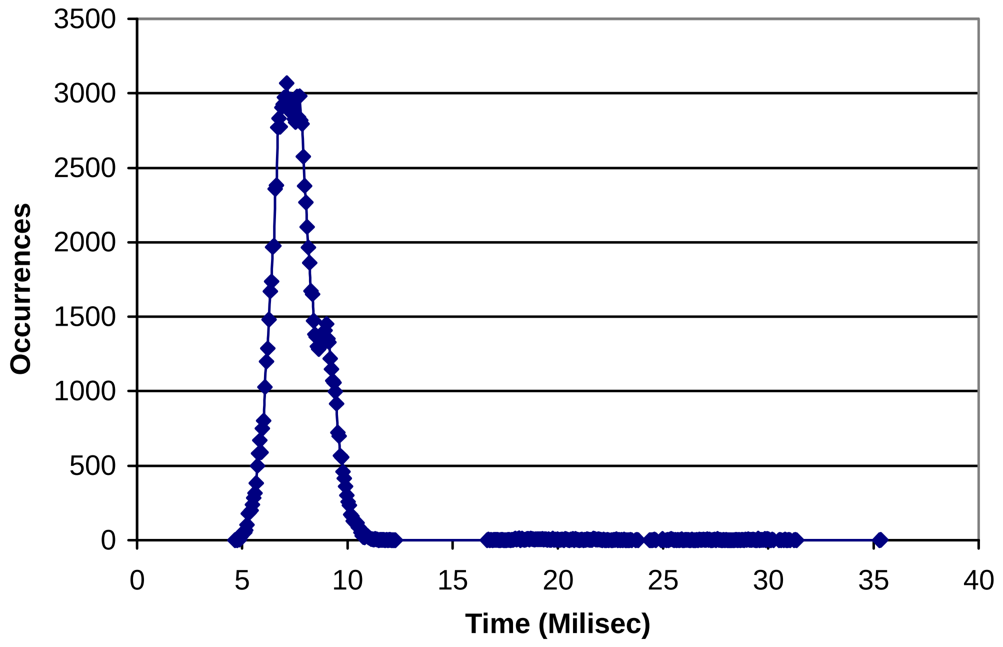

Figure 5.13 : Myrinet at 128 CPUs for 256 KByte.

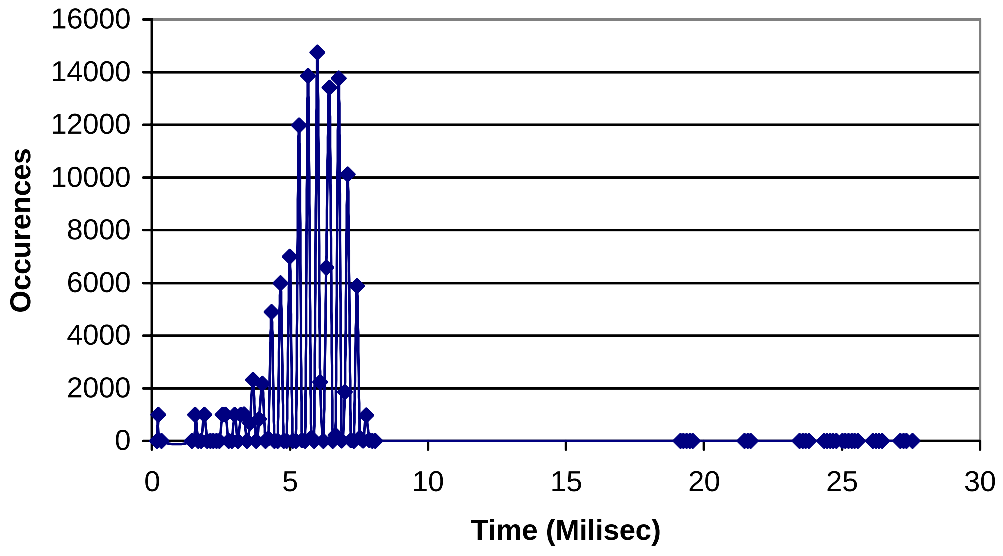

Figure 5.14 : Ethernet at 128 CPUs for 8 KByte.

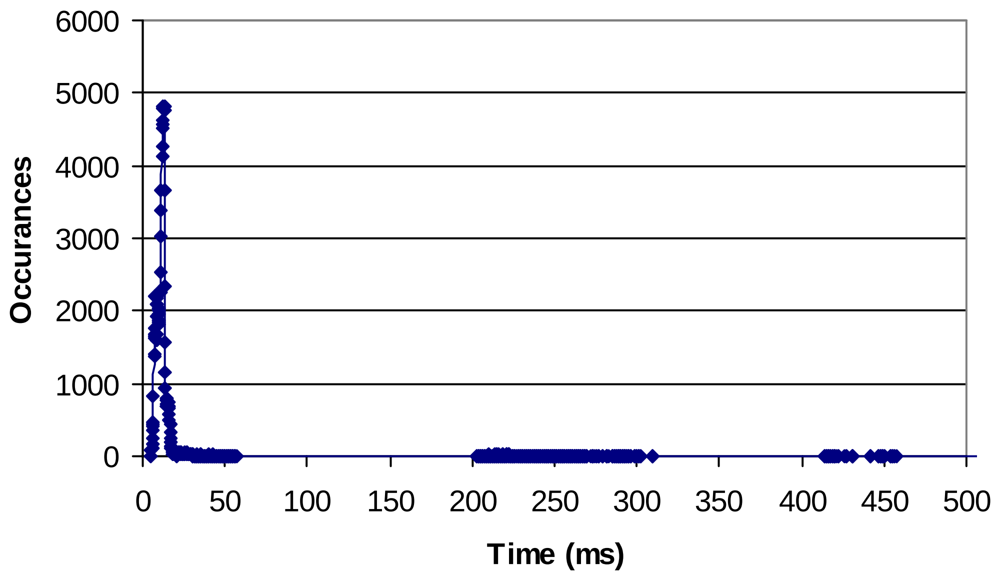

Figure 5.15 : Ethernet at 128 CPUs for 16 KByte.

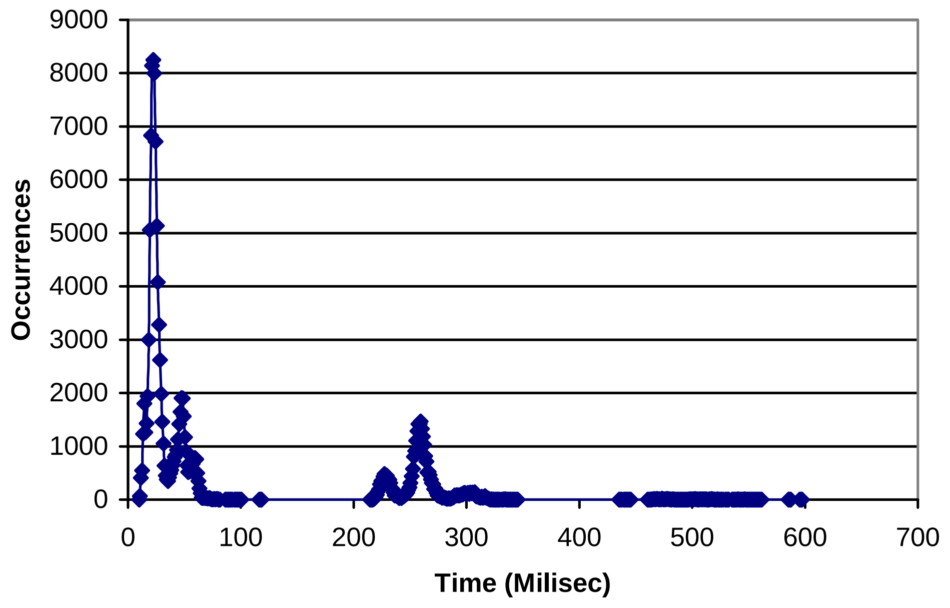

Figure 5.16 : Ethernet at 128 CPUs for 32 KByte.

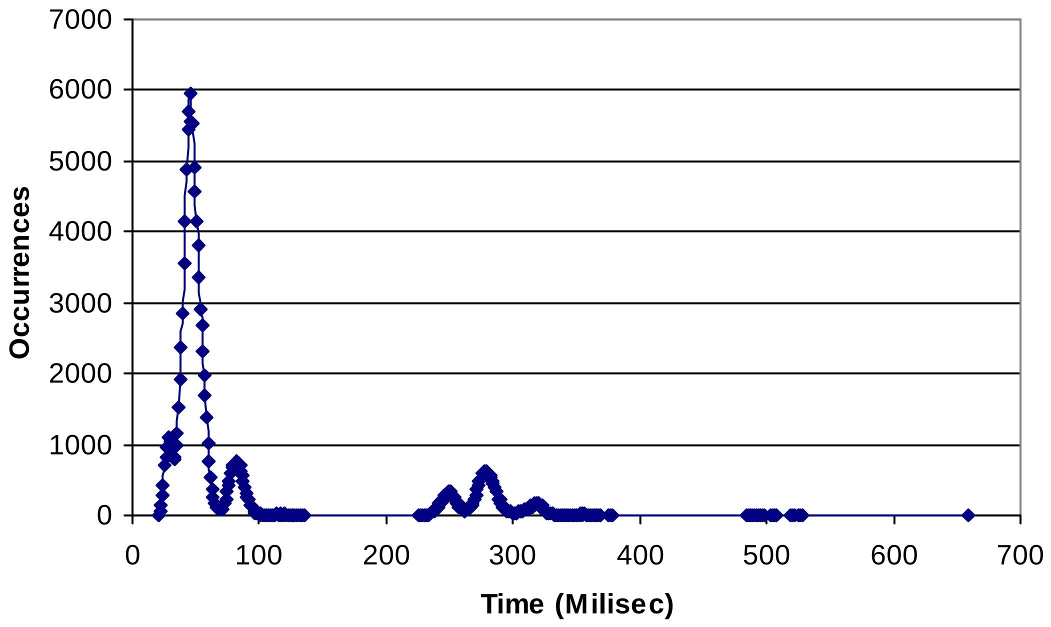

Figure 5.17 : Ethernet at 128 CPUs for 64 KByte.

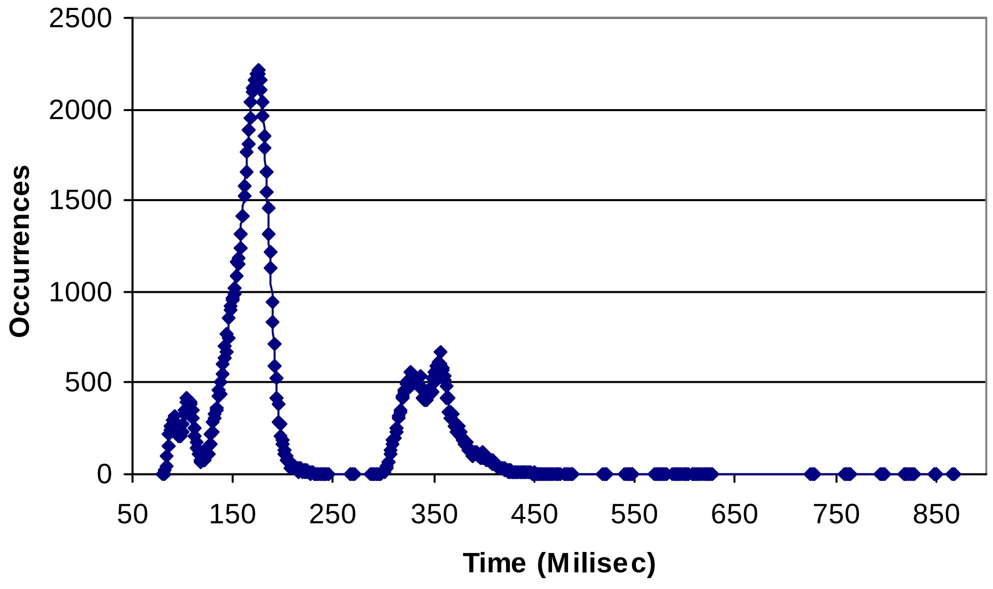

Figure 5.18 : Ethernet at 128 CPUs for 256 KByte.

	No RTO	**1 x RTO**	**2 x RTO**	**3 x RTO**	**Average Time (msec)**	**Estimated Average Time Without RTO (msec)**
8 KByte	100	0	0	0	7.92	7.92
16 KByte	98.9	0.99	0.01	0	90.81	49.33
32 KByte	78.4	21.3	0.29	0	243.7	69.39
65 KByte	85.2	14.7	0.01	0	289.70	94.43
256KByte	73.0	26.9	0.04	0	412.38	202.99

Table 5.6 : Percentage of RTO occurrences for Broadcast for Ethernet on 128 CPUs and estimated average time without RTOs.

In order to check whether RTOs occur with no communication on the central switch, the performance for CPU less than 64 has been analysed. The results for 64 CPUs showed that there were no RTOs, but surprisingly for 32 CPUs the results in Figure 5.19 showed a small number of RTOs. However, these RTOs were anomalous, since they only occurred for one of the 1000 repetitions of the broadcast, and did not occur again when the benchmark runs were repeated two more times. These kinds of outliers occur occasionally, possibly due to a problem in the switch, or an operating system interrupt or some other problem on one of the nodes of the cluster. Similar outliers occur occasionally even for Myrinet, as shown in Figure 5.21.

Although these outliers only occur for a single test, it is instructive to explore them in more detail, by investigating the minimum, average and maximum times for each process. This also illustrates the capability and usefulness of MPIBench in analyzing the behavior of message passing communication in more detail. Figure 5.20 shows that all of the processes were affected by the RTO that occurred on one of the iterations. However the maximum time was obtained by CPU rank 0, 8, 16 and 24. This sequence indicates the pattern of the binomial tree algorithm, where a delay at one of the processes would affect the results of all others further down the tree. An analysis of the outliers measured for Myrinet shows a similar pattern, with Figure 5.22 showing the maximum time is obtained by processes 3, 11, 19 and 27.

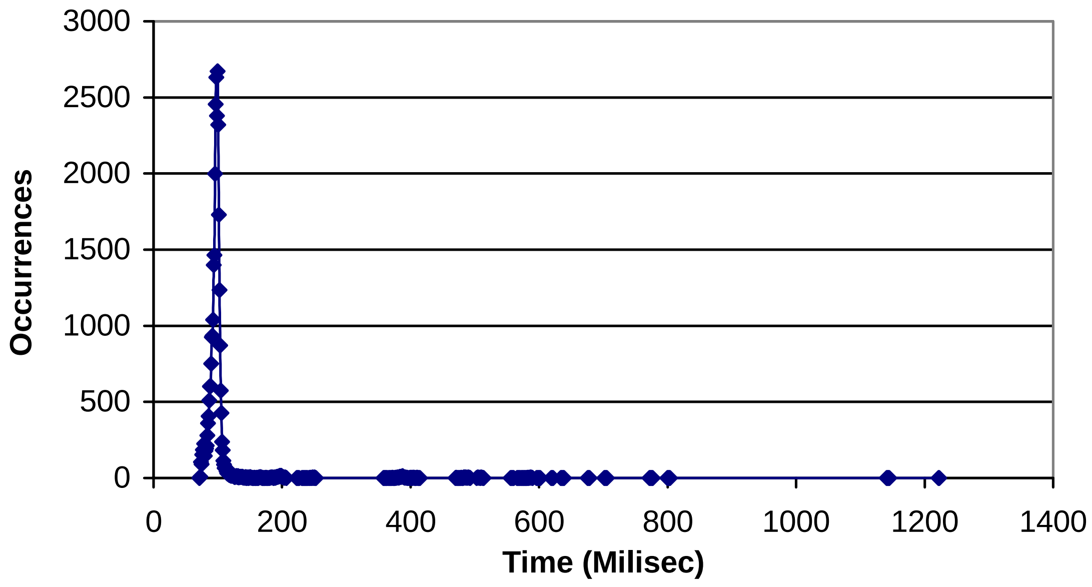

Figure 5.19 : Ethernet at 32 CPUs for 256 KByte.

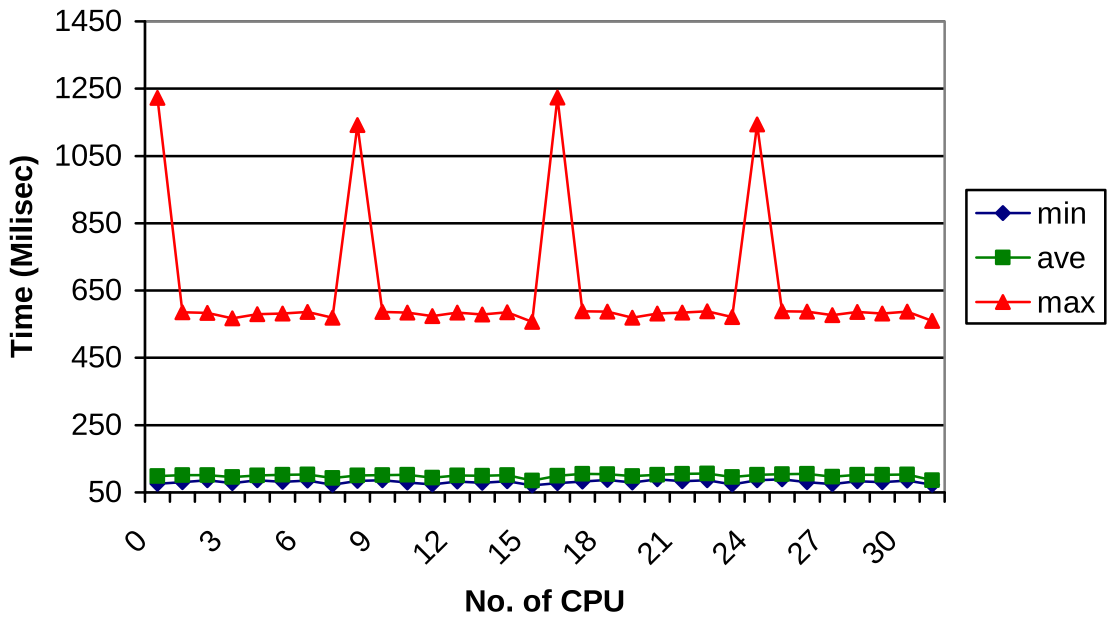

Figure 5.20 : Minimum and Average Time for each CPU on Ethernet for 32 CPUs at 256 KByte

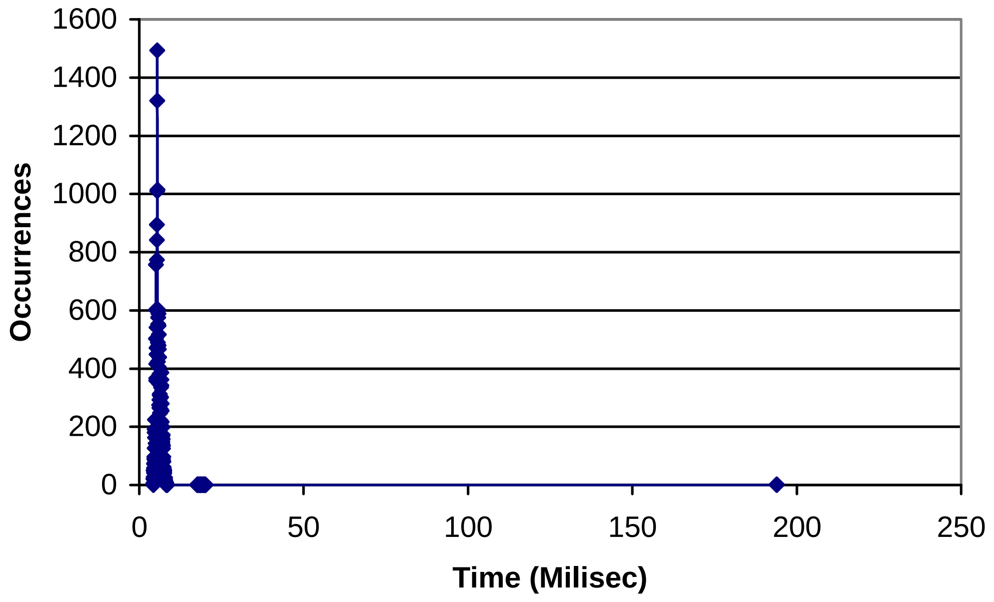

Figure 5.21 : Myrinet at 32 CPUs for 256 KByte.

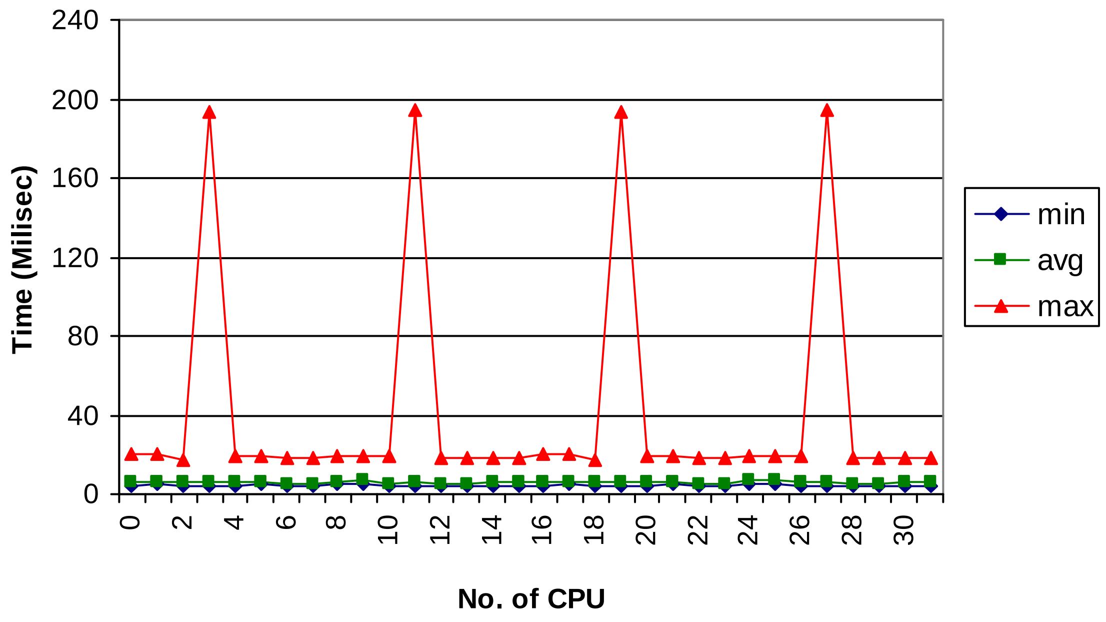

Figure 5.22 : Minimum and Average Time for each CPU on Myrinet for 32 CPUs at 256 KByte.

5.7 Scatter and Gather

Scatter and gather are typically used to distribute data at the root process (e.g. a large array) evenly among the processors for parallel computation, and then recombine the data from each processor back into a single large data set on the root process. The performance of MPI_Scatter is dependent on how fast the root process can send all the data, since it is a bottleneck. However the root process can use asynchronous sends, which means that the overall performance of the scatter operation is also dependent on the overall communications performance of the system and the effects of contention. The algorithm used by MPICH 1.2.6 in all data sizes for Scatter and Gather is the binomial tree algorithm [11,12,69].

5.7.1 Scatter

Figure 5.23 and Figure 5.24 shows the average completion time for MPI_Scatter on Myrinet and Ethernet for 4 to 200 CPUs. The average times for both networks increase slowly as the message size and number of processes are increased. The larger message sizes the time increases approximately linearly with the message size. There is a small jump at 32 Byte, 512 Byte and 2048 Byte for 200 CPUs on Myrinet and this jump is not experienced by Ethernet. It is also noticeable that the results for Ethernet are about 10 times higher than Myrinet.

Figure 5.25 and Figure 5.27 shows the distributions at 64 Kbyte for Myrinet and Ethernet, respectively. The completion times for Myrinet are mostly between 45 and 50 ms, with a long tail going out to 70 ms, then a big gap to a small number of results around 90 ms, which are from just a single iteration. Figure 5.26 shows that the results at 90 ms were from half of the processes, from process 0 to 63, while the another half of the processes had maximum times between 50 to 70 ms. On Ethernet there are several peaks, for node 0 the average completion times is at 710 ms, nodes 63 the times is 730 ms and for nodes 127 the completions times is 752 ms as shown in Figure 5.28. Figure 5.26 and Figure 5.28 shows the main difference between the Myrinet and Ethernet results, that the average time for each process is almost the same for Myrinet but for Ethernet the average time is increasing gradually. The constant performance of Myrinet is presumably due to the scalability of the Myrinet fat tree architecture, where the switch latencies are low and the available bandwidth stays constant throughout the switch hierarchy. It is surprising that the Myrinet distribution has a long tail. This appears to be due to a very small number of repetitions that take much longer than the rest.

Although the average time for scatter is higher than broadcast for the same number of processes and message size, there is no occurrence of RTOs for scatter. This is because for scatter, the size of the data received by each process is the message size divided by the number of processes, so the total amount of data passing through the network and the central switch will be lower compared to broadcast.

Notice that in Figure 5.26, Figure 5.28 and Figure 5.29, process 0 finishes first for Scatter on Myrinet and Ethernet. This is different from the SGI MPI results on SGI Altix 3700, given in Section 7.9, where process 0 finishes last [22]. This indicates that for SGI MPI, process 0 will wait for the acknowledgement from all CPUs, while in MPICH 1.2.6, it only waits for the acknowledgements from the processes that it sent the message to.

Figure 5.29 shows the minimum and average time for each of 16 processes for 64 KByte message sizes. The purpose of this figure is to show the effect of binomial tree algorithm more clearly. In binomial tree algorithm, process 0 will send data to processes

1, 2, 4 and 8, process 1 to processes 3, 5 and 9, process 2 to processes 6 and 10, process 3 to processes 7 and 11, process 4 to process 12 and from process 7 and higher, each process will distribute to one process only. Since process 1 is placed in the same node with process 0, it will finish after process 0. However, process 2 finishes a bit later than process 4 and 8 since it has to distribute data to 2 other processes, while processes 4 and 8 only have to distribute to 1 process. Referring to the figure, the completion of each process is synchronized with the sequence of the binomial tree algorithm and affected by the position of the process, either in the same node or otherwise.

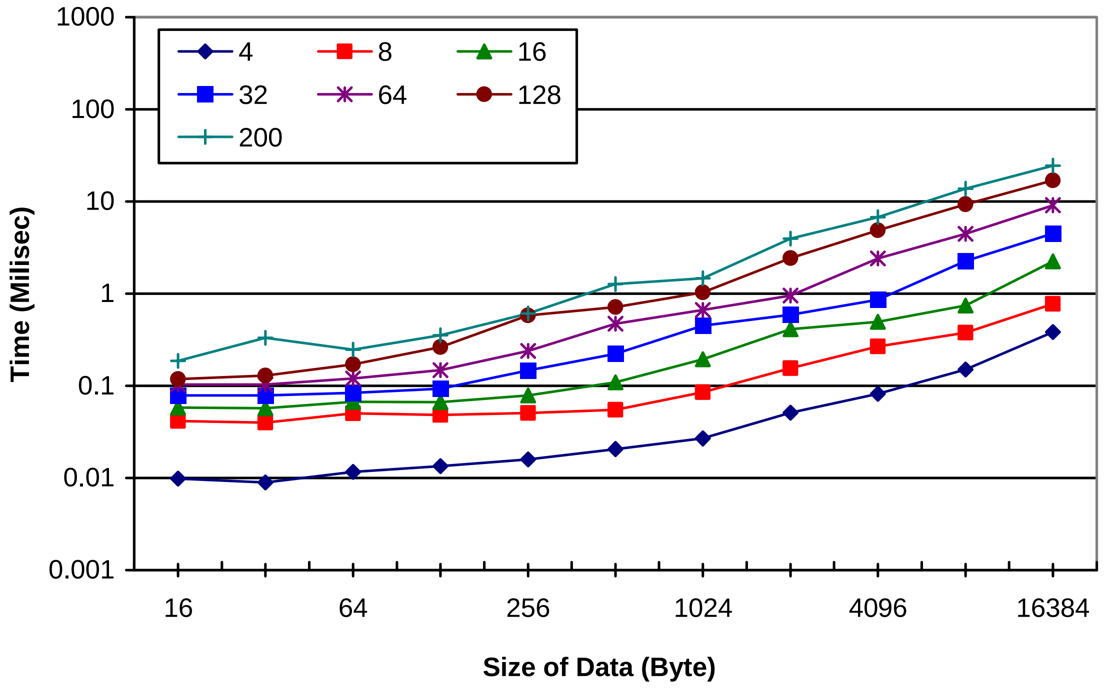

Figure 5.23 : Average time for MPI_Scatter on Myrinet.

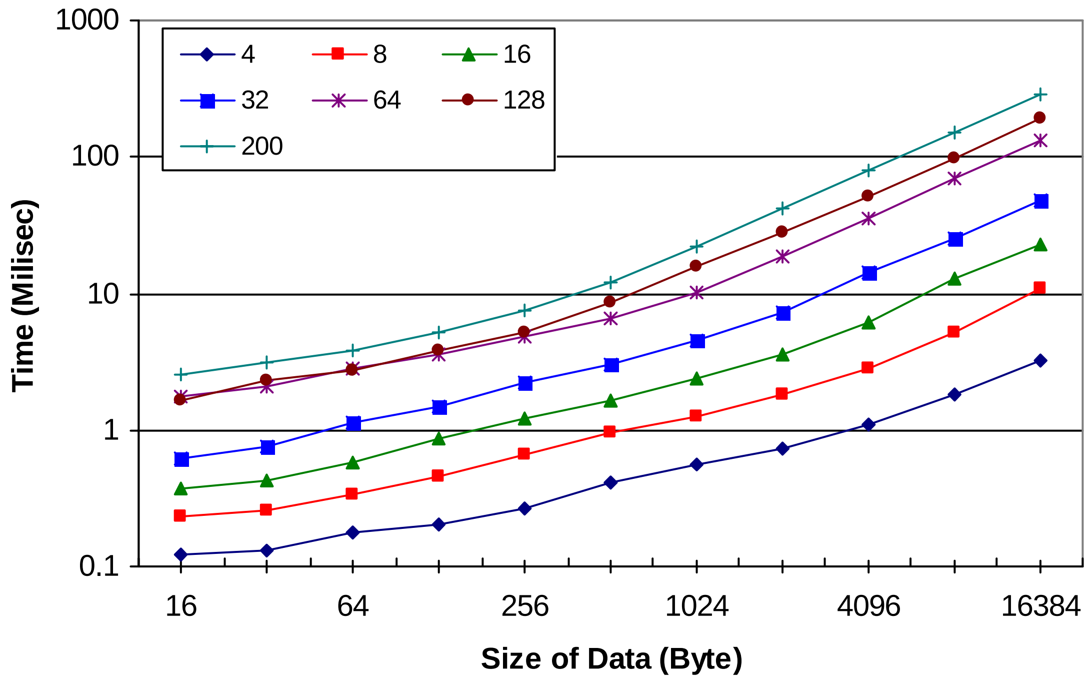

Figure 5.24 : Average time for MPI_Scatter on Ethernet.

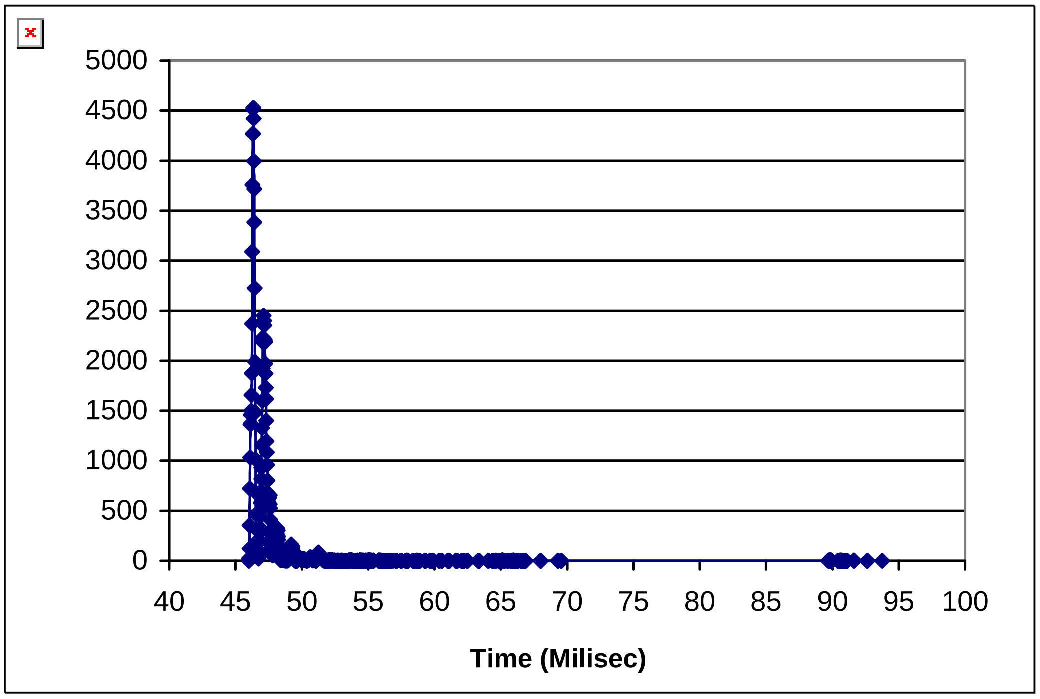

Figure 5.25 : Myrinet at 128 CPUs for 64 KByte.

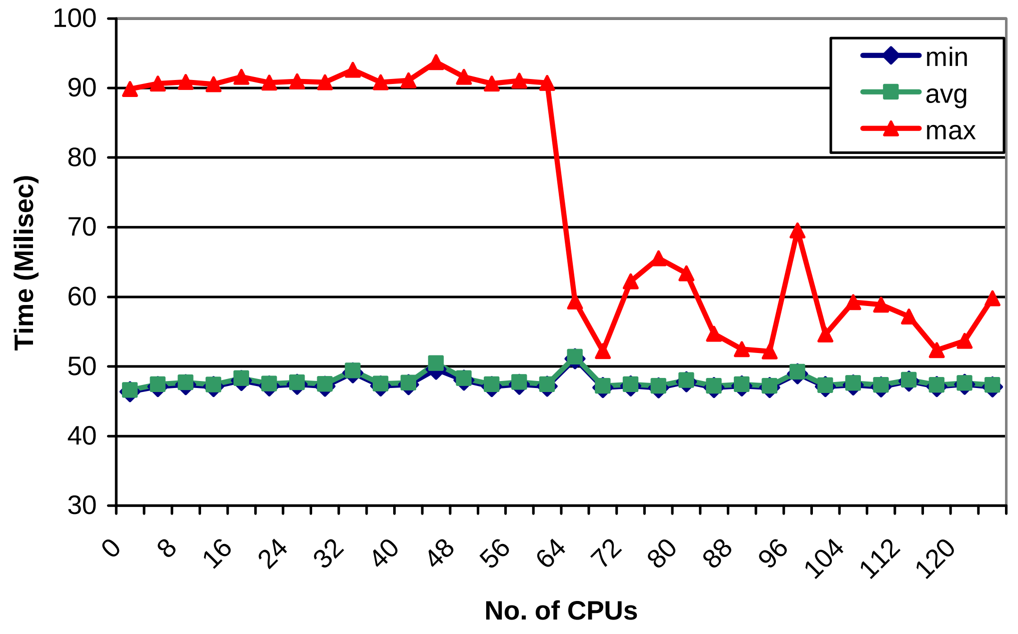

Figure 5.26 : Minimum, Maximum and Average Time for each CPU on Myrinet for 128 CPUs at 64 KByte

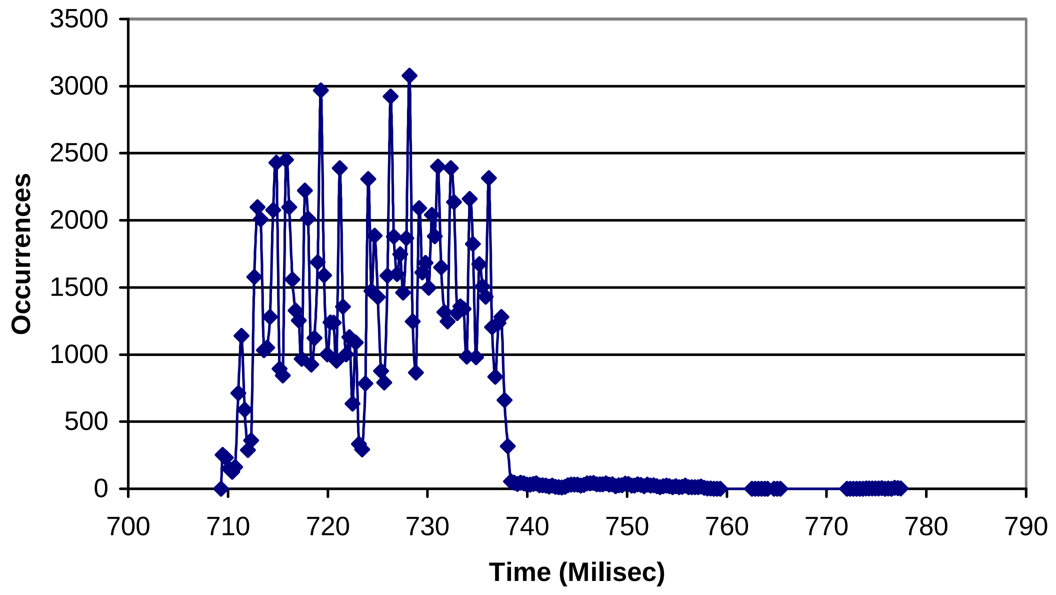

Figure 5.27 : Ethernet at 128 CPUs for 64 KByte.

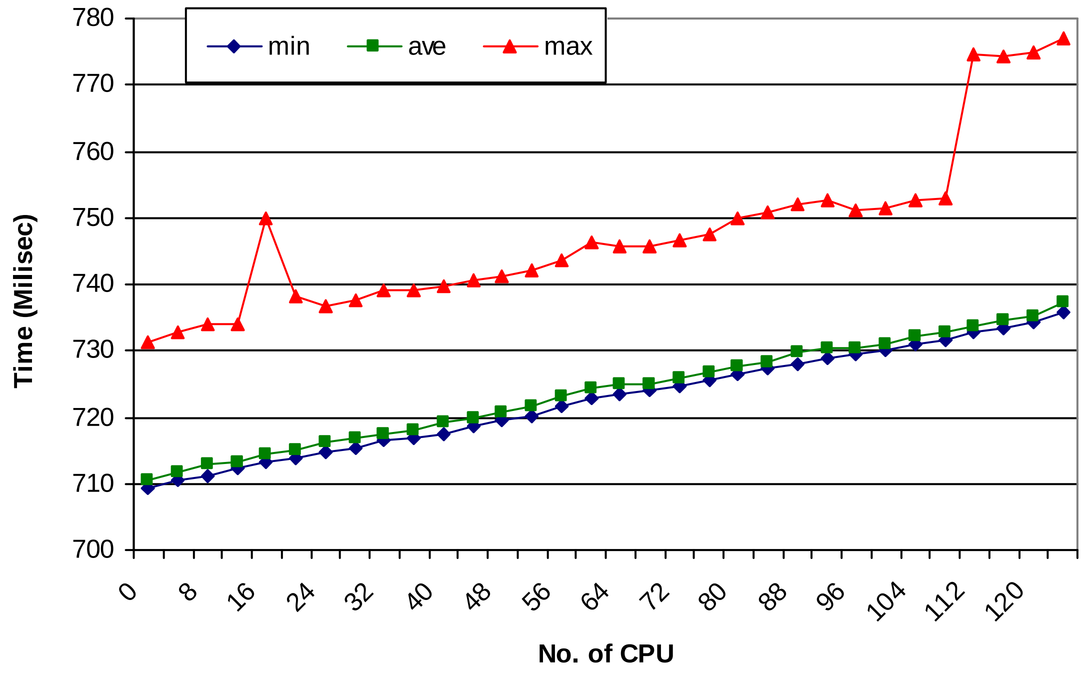

Figure 5.28 : Minimum, Maximum and Average Time for each CPU on Ethernet for 128 CPUs at 64 KByte

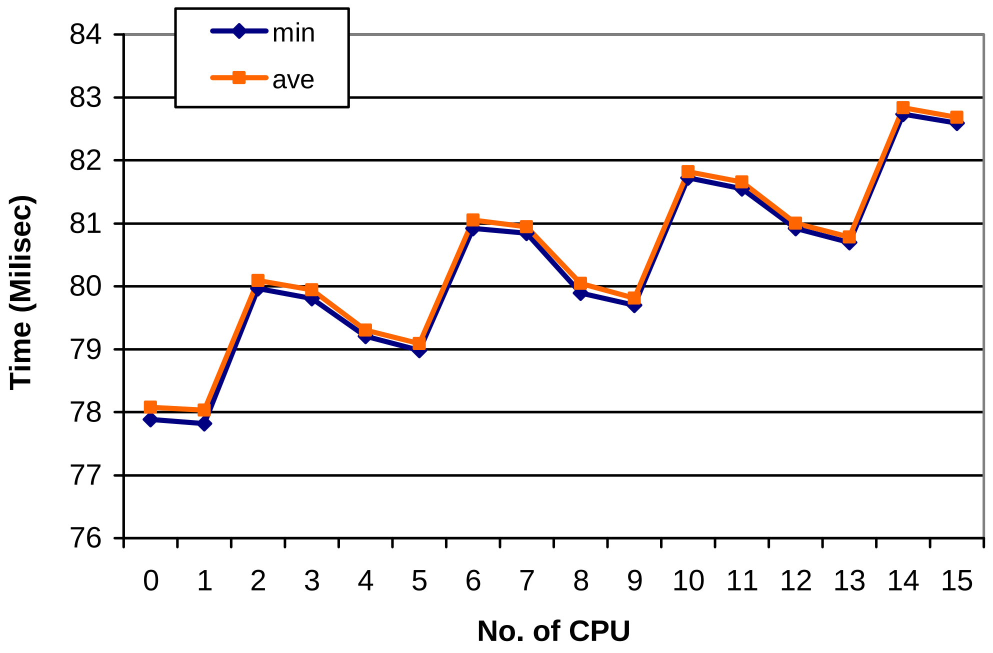

Figure 5.29 : Minimum and Average Time for each CPU on Ethernet for 16 CPUs at 64 KByte

5.7.2 Gather

The performance of MPI_Gather is mainly determined by how much data is received by the root process, which is the bottleneck in this operation. Hence the time taken is expected to be roughly proportional to the total data size for a fixed number of processors, with the time being slower for larger numbers of processors due to serialization and contention effects. Figure 5.30 and Figure 5.31 shows the average completion time for MPI_Gather on Myrinet and Ethernet for 4 to 200 CPUs. The average times for both networks increase proportionally as the message size and number of processes is increased.

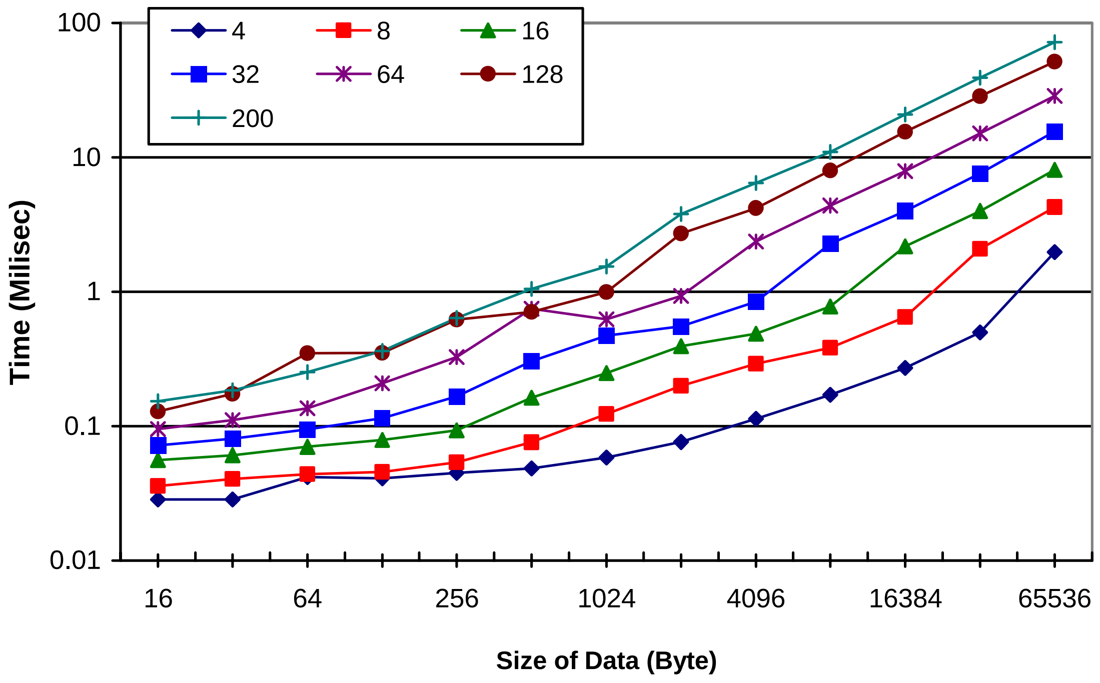

Figure 5.30 : Average time for MPI_Gather on Myrinet

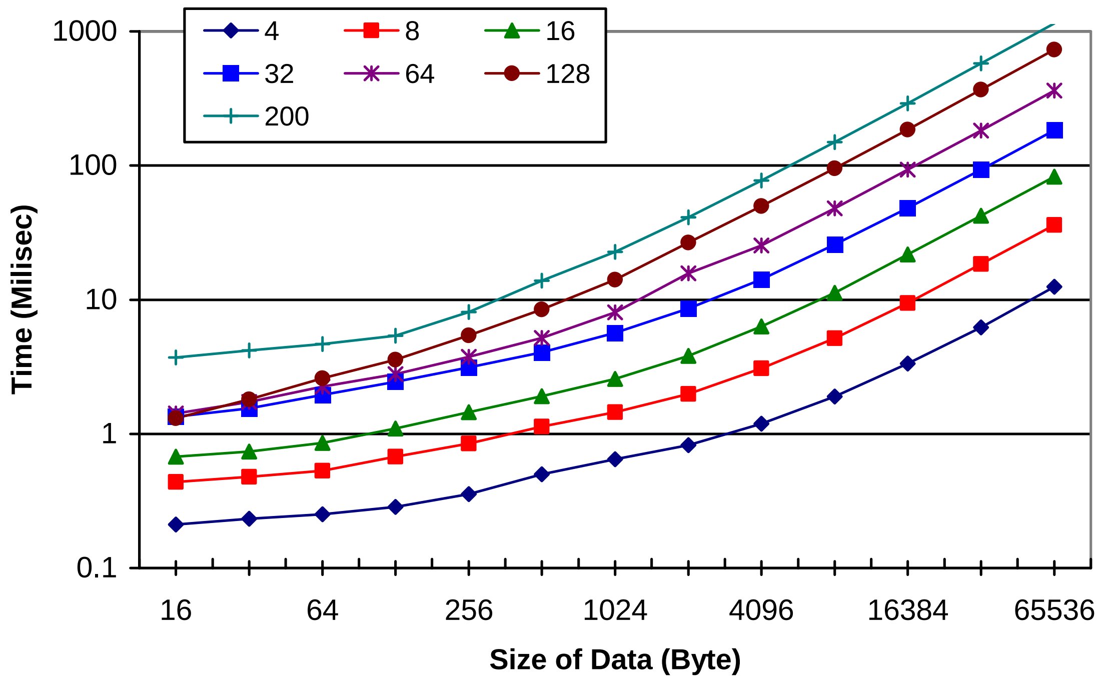

Figure 5.31 : Average time for MPI_Gather on Ethernet.

The spread of the distributions for both networks indicate different completion times for different processes rather than any wide variation in completion times for each process or the effect of RTOs for Ethernet. Although MPI_Gather does the reverse process from MPI_Scatter, Figure 5.33, Figure 5.35 and Figure 5.36 show that the pattern of average times for each process is clearly very different to scatter. The capability of MPIBench to show average times for each process clearly illustrates the inverse binary tree algorithm used for gather, particularly in Figure 5.35 and Figure 5.36. Each of the odd numbered processes sends their data to an intermediate even numbered process, which combines this data with its own. Then this combined data will be forwarded to the next intermediate process which has a rank that is a multiple of 4, where it is combined again and forwarded to an intermediate process with rank a multiple of 8, and so on, until finally all the data is gathered on process 0 [69].

The broad distribution of times for Myrinet in Figure 5.32 is mostly due to the difference in average completion times for processes at different levels of the binary tree, as well as some variation between the average and maximum completion times of each process, as shown in Figure 5.33. There are a small number of outliers between 80ms and 100ms. Figure 5.33 shows that these are from the maximum times for the last processes to complete, i.e. 64 and 0.

This distribution of times for Ethernet in Figure 5.34 shows a similar scenario. While the clusters of times at 200ms, 400ms and 800ms appear as though they may be due to RTOs, they actually represent the different completion times for processes at different levels of the binary tree, as seen in Figure 5.35. This analysis shows that RTOs have no effect on the performance of scatter and gather. This is because for scatter and gather, the size of the data being sent by each process decreases as the number of communicating processes in the binary tree increases, which will reduce the bottleneck problem at the central switch on the Ethernet network.

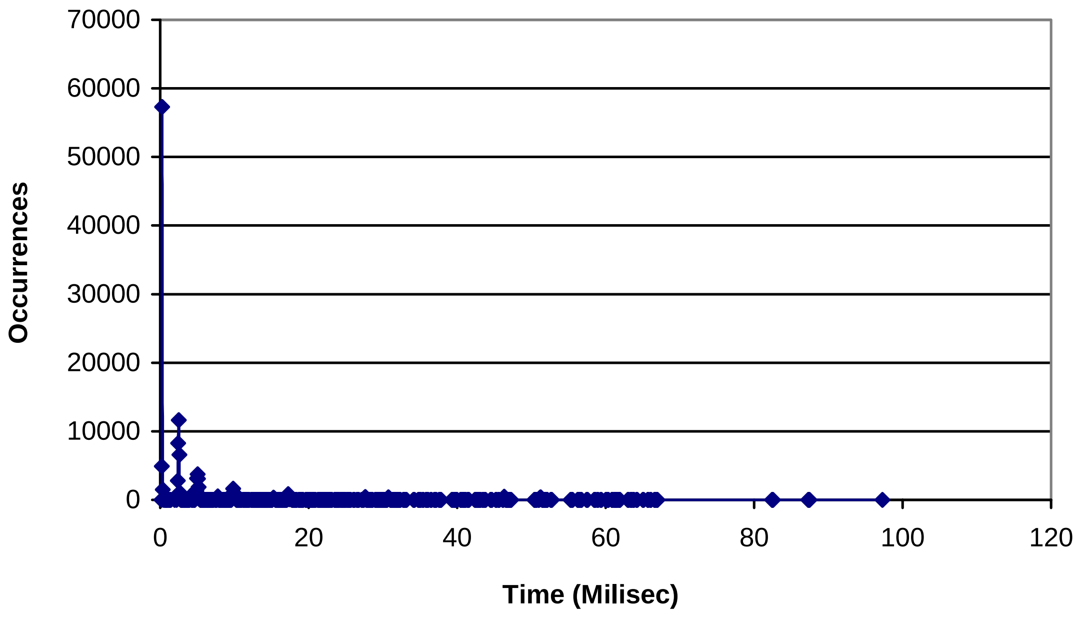

Figure 5.32 : Myrinet for 128 CPUs at 64 KByte.

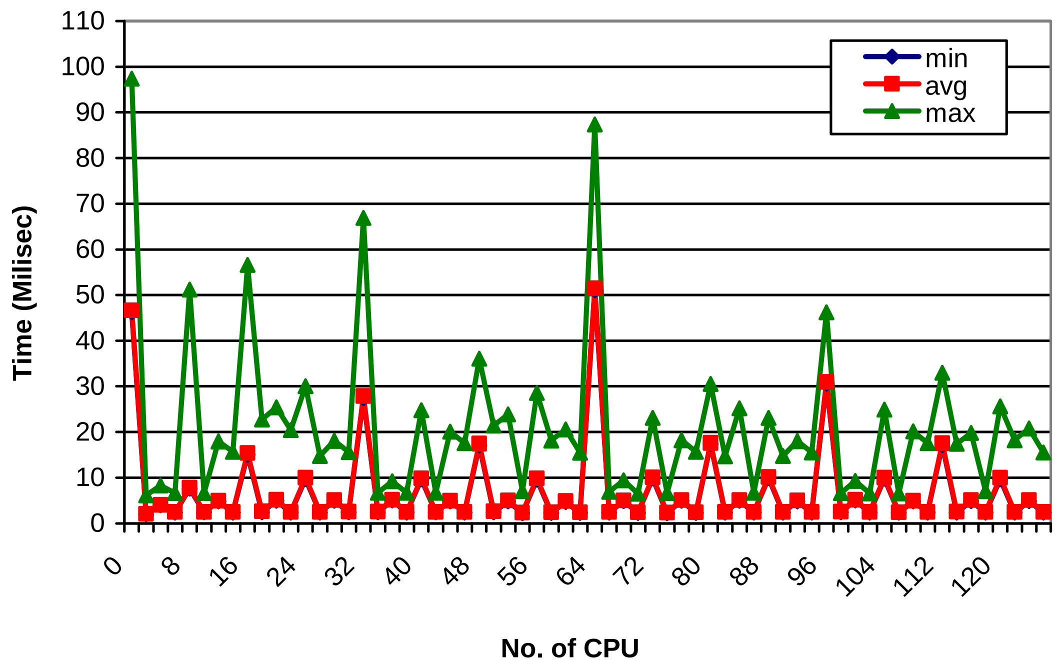

Figure 5.33 : Minimum, Maximum and Average Time for each CPU on Myrinet for 128 CPUs at 64 Kbyte.

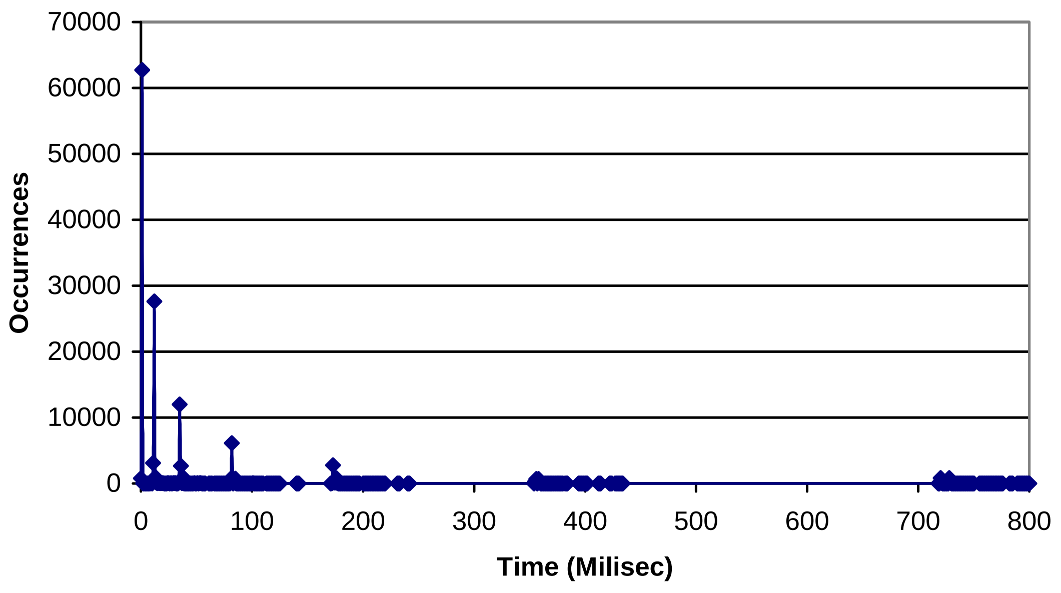

Figure 5.34 : Ethernet for 128 nodes at 64 KByte

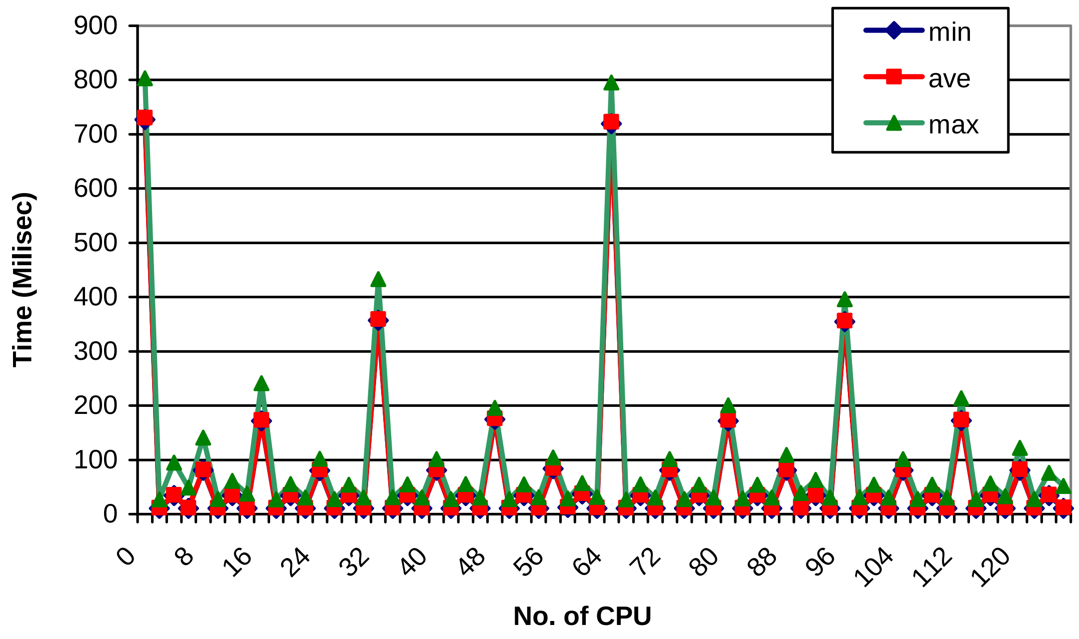

Figure 5.35 : Minimum, Maximum and Average Time for each CPU on Ethernet for 128 CPUs at 64 KByte

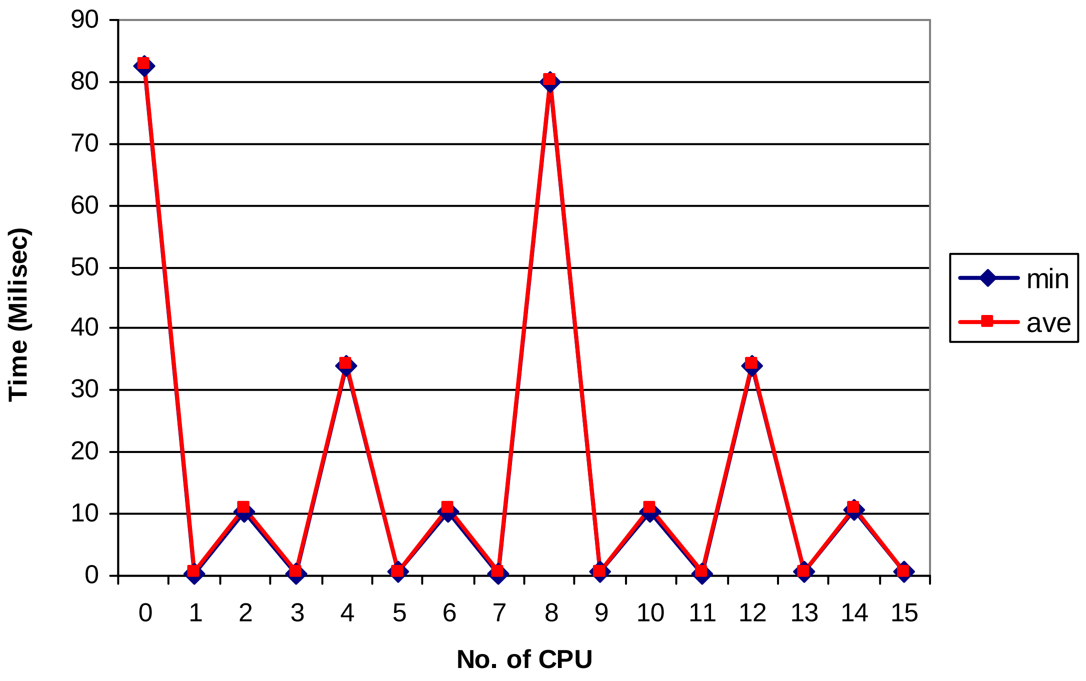

Figure 5.36 : Minimum and Average Time for each CPU on Ethernet for 16 CPUs at 64 KByte.

5.8 Alltoall

The final collective communication that has been measured is MPI_Alltoall, where each process sends its data to every other process. This provides a good stress test of the communications network. MPICH 1.2.6 uses four algorithms for MPI_Alltoall [11,12,69]. For short messages (<256Bytes) and (number of processes >= 8) it uses store-and-forward algorithm. For medium size messages (256Bytes<medium<32KBytes) and (short messages for number of processes < 8) it uses an algorithm that posts all irecvs and isends and then does a waitall, then scatter the order of sources and destinations among the processes. For long messages and power-of-two number of processes, it uses a pair-wise exchange algorithm. For a non-power-of-two number of processes, it uses an algorithm in which, in step i, each process receives from (rank-i) and sends to (rank+i).

Figure 5.37 shows that average completion time for Myrinet increases gradually with message size and number of processes (for 128 and 200 cpus the size of data is only until 32Kbytes due to time contraints). However, in Figure 5.38 the communication times for Ethernet increase markedly for more than 32 CPUs, which shows the effect of Retransmit Timeouts (for 128 and 200 cpus the size of data is only until 16Kbytes due to time contraints). Note that there is often a hump (indicating slower times for smaller message sizes) where the algorithm changes, particularly after 256Byte and 32KByte.

Figure 5.39 shows the distribution for 128 CPUs for 2 KByte on Myrinet. Besides the single point at around 28 ms, which is due to a slow result only for a single iteration, it can be seen that even for a large number of CPUs, Myrinet has a single, fairly narrow peak at a much smaller communication time than Ethernet, and with a short tail, so the performance of All-to-All is excellent. In contrast, Figure 5.40 shows that RTOs occur for Ethernet at 128 CPUs even for relatively small message sizes. Figure 5.40 to Figure 5.45 show the distribution of times for 128 CPUs on Ethernet for message sizes from 64 Byte to 2 KByte. These figures show the increase of RTO with the increasing of message sizes. At 2 KByte there are several peaks which are due to occurrences of multiple RTOs. Above this message size the RTOs are hard to detect, since the average time is much larger than the 200 ms RTO delay and the distribution of times becomes very broad.

Figure 5.41 and Figure 5.43 shows the minimum, average and maximum time for 128 CPUs on Ethernet for 64 Byte and 256 Byte message sizes, respectively. For 64 Byte messages, there are only a small number of RTOs which are only experienced by 1 CPU, which is CPU rank 80, while for 256 Byte message sizes RTOs are experienced by all CPUs.

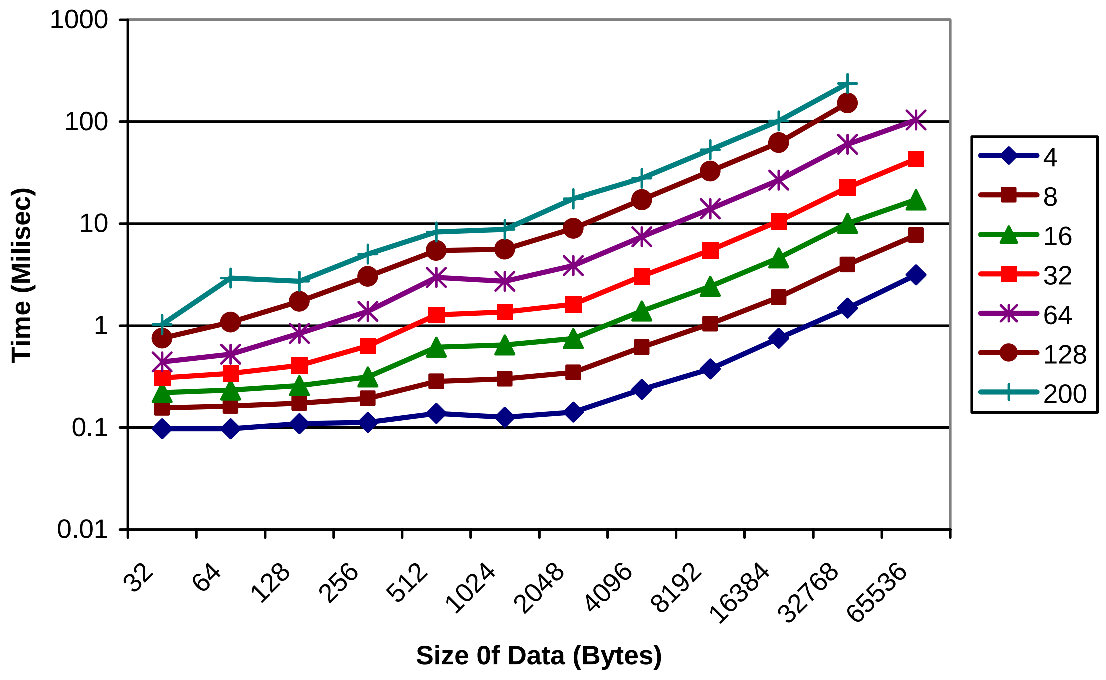

Figure 5.37 : Average time for MPI_Alltoall on Myrinet for 4 to 200 CPUs.

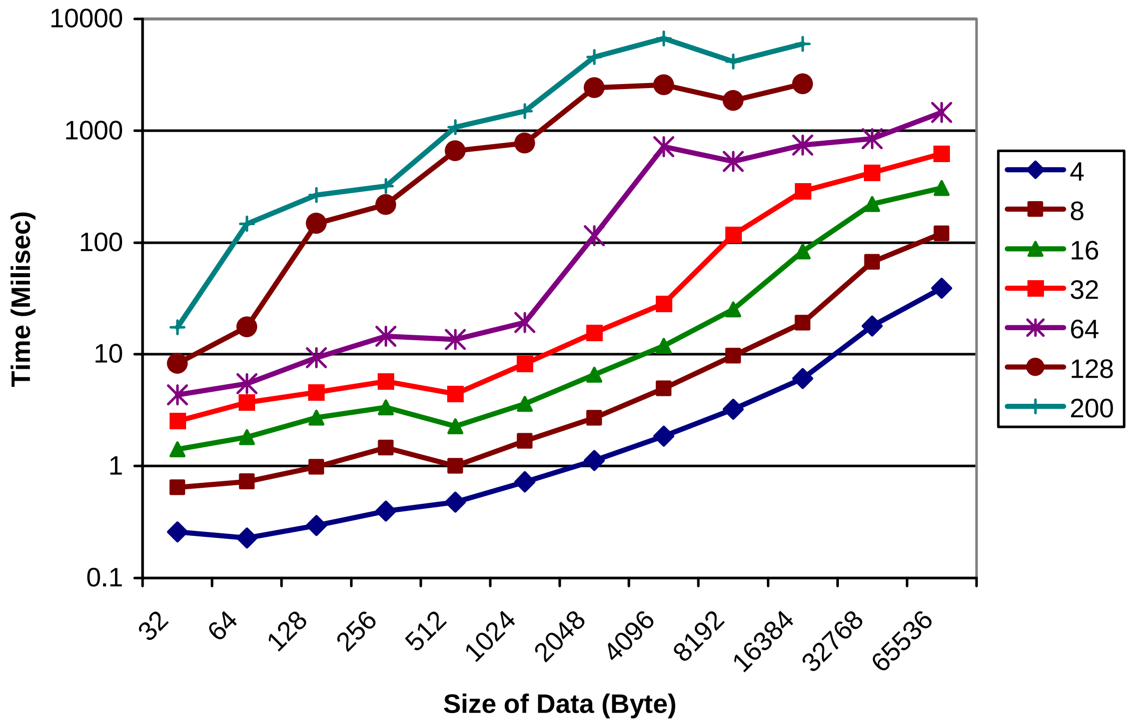

Figure 5.38 : Average time for MPI_Alltoall on Ethernet for 4 to 200 CPUs.

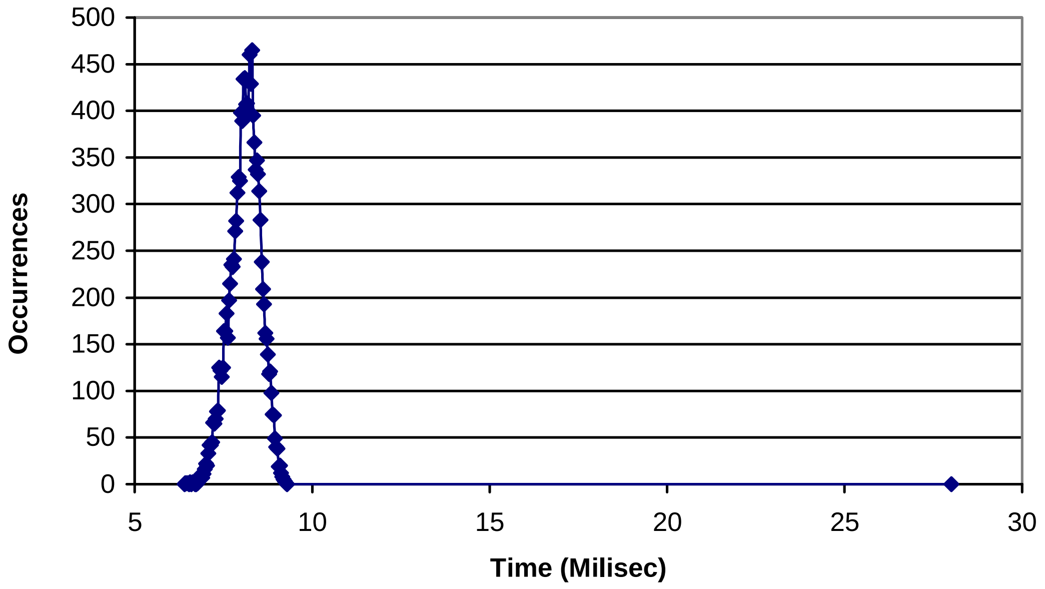

Figure 5.39 : Myrinet at 128 CPUs for 2KByte.

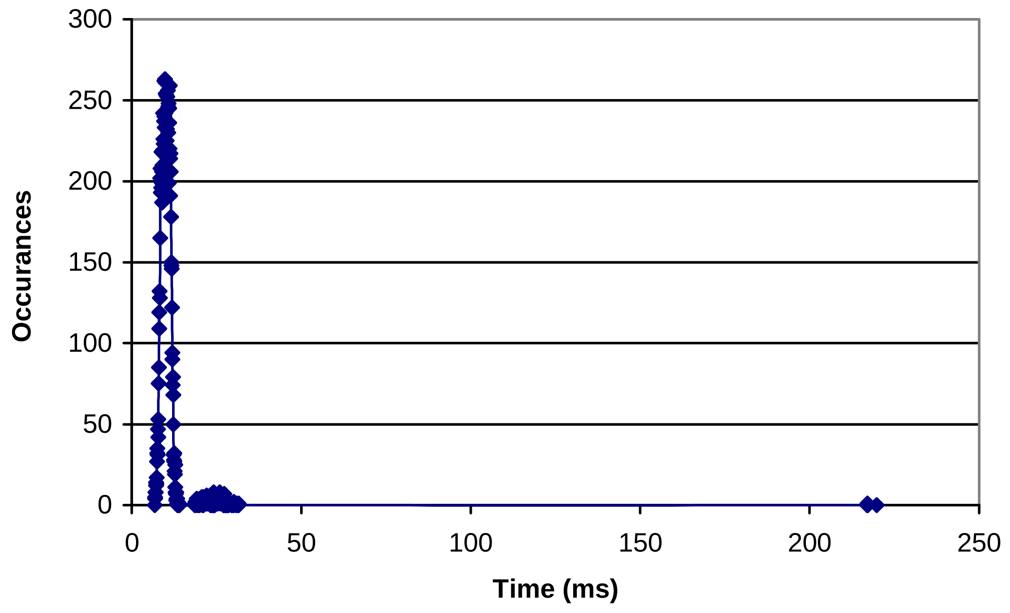

Figure 5.40 : Ethernet at 128 CPUs for 64Byte.

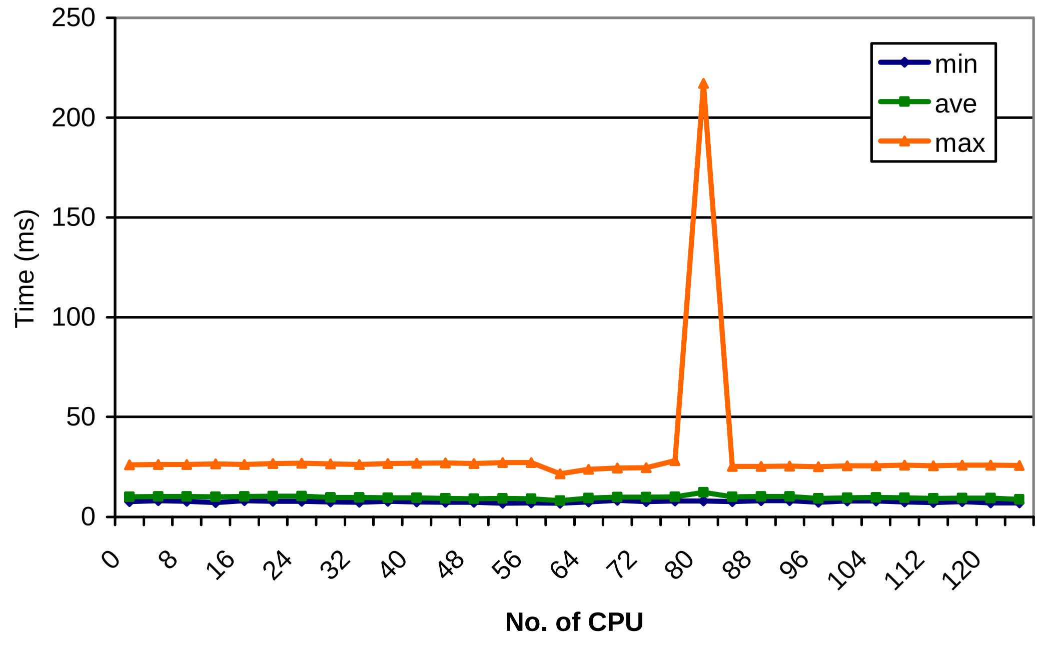

Figure 5.41 : Minimum, Maximum and Average time on Ethernet for 128 CPUs at 64 Byte

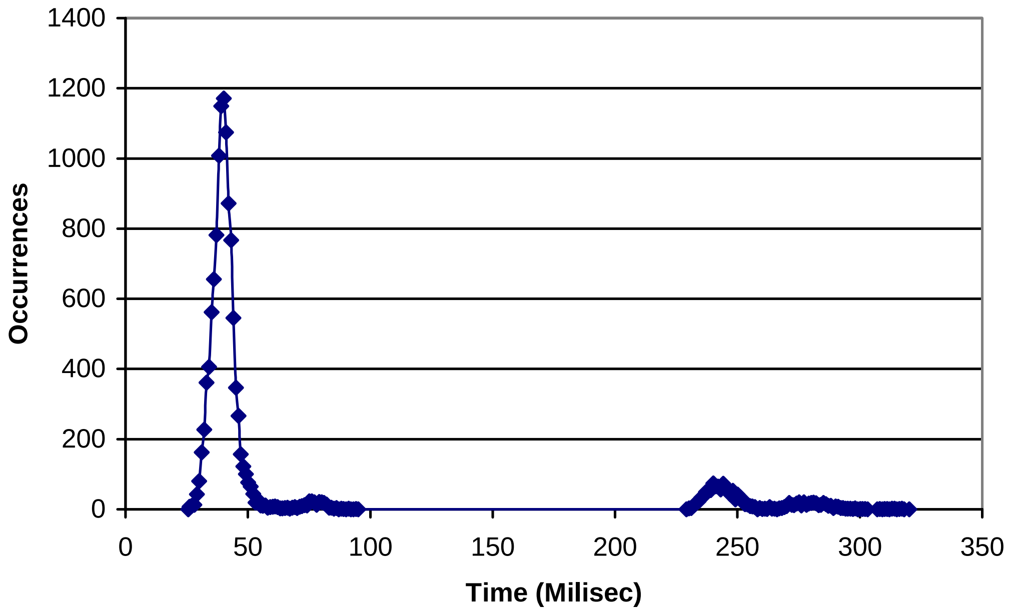

Figure 5.42 : Ethernet at 128 CPUs for 256 Byte.

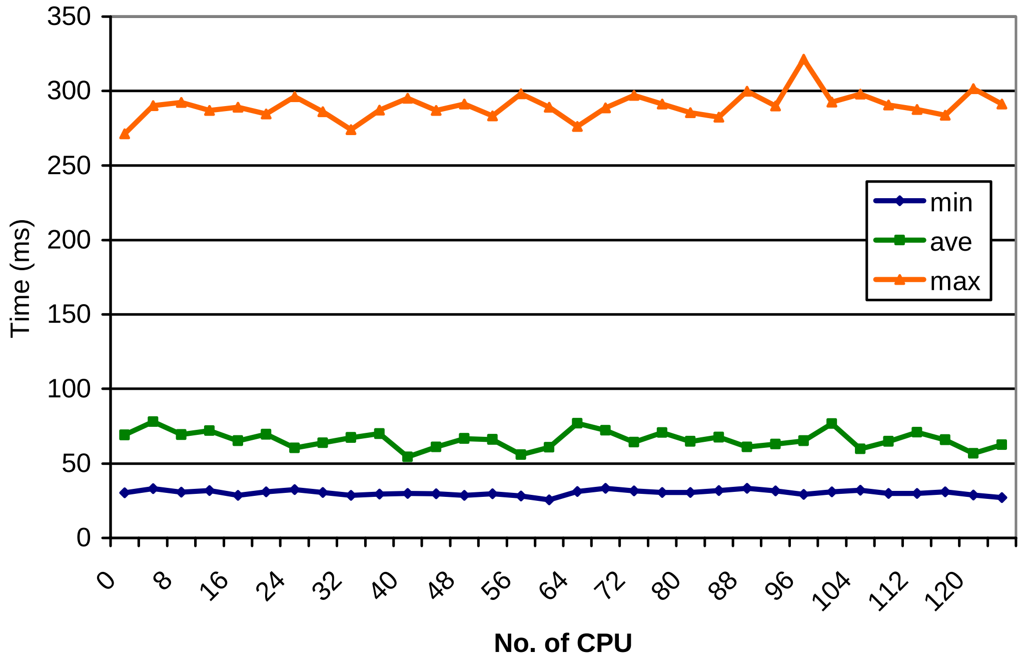

Figure 5.43 : Minimum, Maximum and Average time on Ethernet for 128 CPUs at 256 Byte

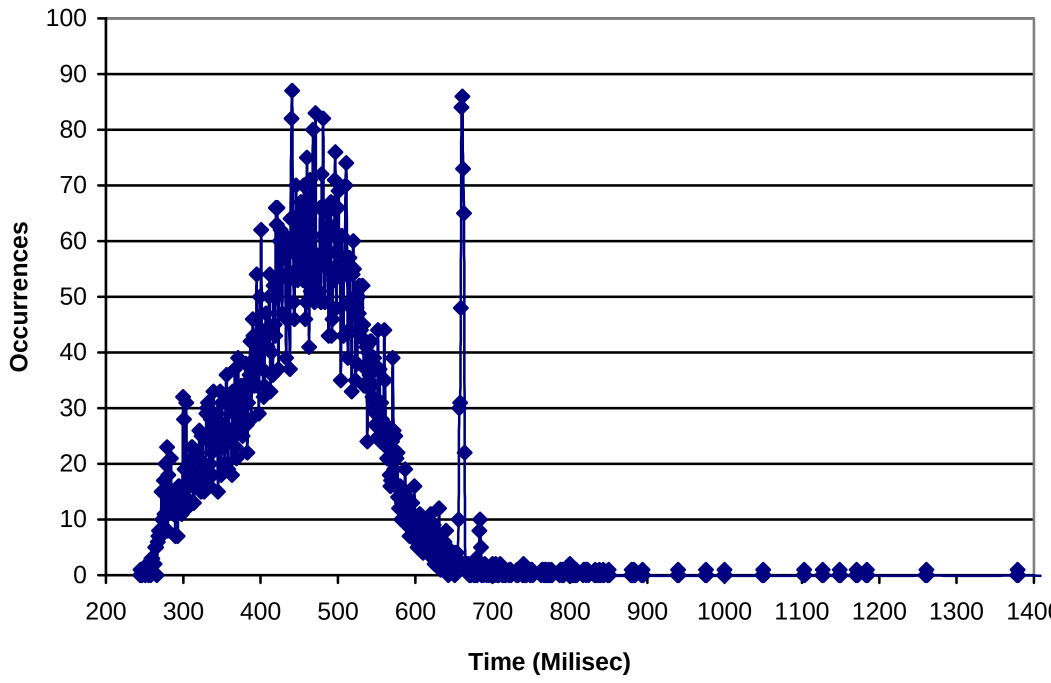

Figure 5.44 : Ethernet at 128 CPUs for 1KByte.

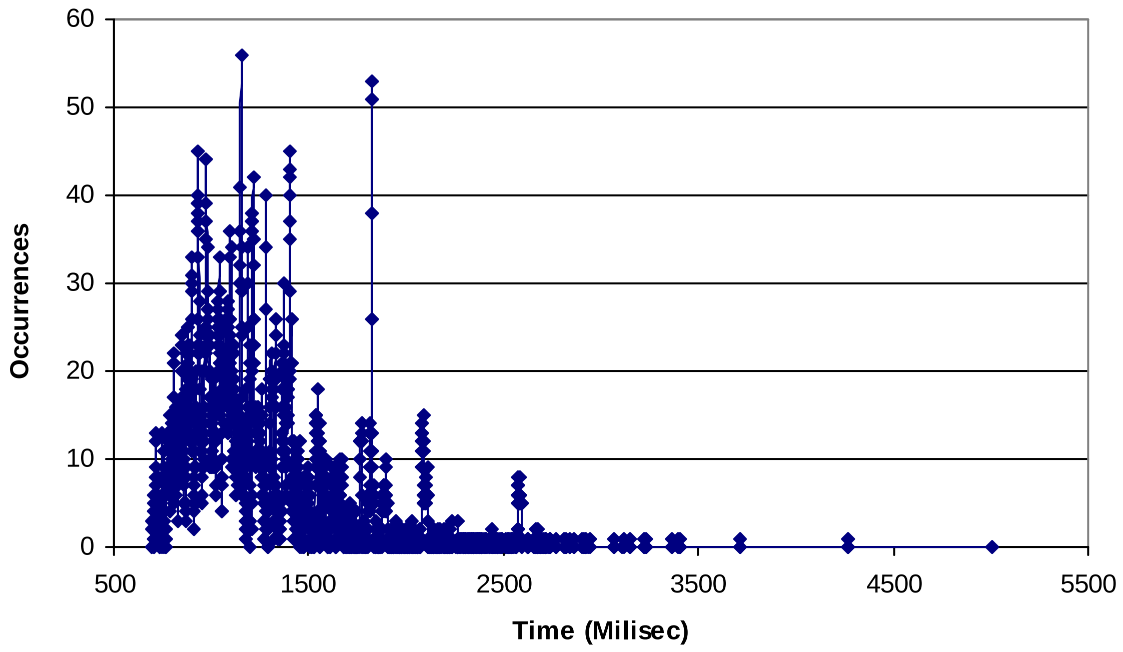

Figure 5.45 : Ethernet at 128 CPUs for 2KByte.

Similarly with point-to-point and broadcast, on Fast Ethernet the performance for 64 CPUs was checked in order to see if there were any occurrences of RTO without the effect of communication problems at the central switch. As seen in the previous sections, point-to-point and broadcast communication suffered little or no RTOs for 64 CPUs and below, where there is no inter-switch communication. However, for MPI_Alltoall, which involves all processes communicating at the same time, it is more likely that RTOs would occur even for communication within a switch. It was found that RTOs did indeed occur for 64 CPUs, starting at message sizes as small as 2 KBytes. Figure 5.46 to Figure 5.50 show the distribution of times for 64 CPUs from 2 to 32KByte. The figures show that the number of RTOs is increasing as the message size increases. For larger data sizes, Figure 5.48 shows a large number of multiple RTOs above the first significant peak at 200ms. The small number of results below 200ms occur only for certain processors. Once the message size gets large enough, as in Figure 5.49 and Figure 5.50, the distribution becomes so broad that it is impossible to identify delays due to RTOs. Myrinet still performs well for large message sizes and similar number of CPUs, as shown in Figure 5.51.

Table 5.7 shows the percentage of occurrences of RTO for All-to-All for different message sizes and numbers of CPUs. The percentage of single and multiple RTOs increases rapidly as the message size increases, until the point where the average time is large enough, and the distribution is broad enough, that it is not possible to identify the number of RTOs (as in Figure 5.50). Table 5.7 also shows the estimated average communication time if there were no RTOs. Again, this shows that RTOs have a significant effect on the communications performance of MPI over Ethernet networks.

Grove et al. [8,9] did a similar performance measurement on MPI_Alltoall using a Beowulf-type cluster and MPICH 1.2.0. Based from the paper, the TCP RTO badly affected the results at a smaller number of processors (for example 16 CPUs) and smaller message sizes (for example 4 KByte) and above certain message sizes the communication failed completely. They found that one of the problems was due to the contention for buffer access by the switches input queueing processes and that MPICH 1.2.0 had an inefficient implementation of the algorithm for MPI_Alltoall, which was fixed in subsequent versions. However the analysis presented in this section found the effects of RTOs to be much less serious than for these previous measurements. The improvement is probably due to both improved algorithms and implementations of the collective communications routines used in MPICH, and improved Ethernet network in the machine used in this study, which had switches with more ports and a faster backplane.

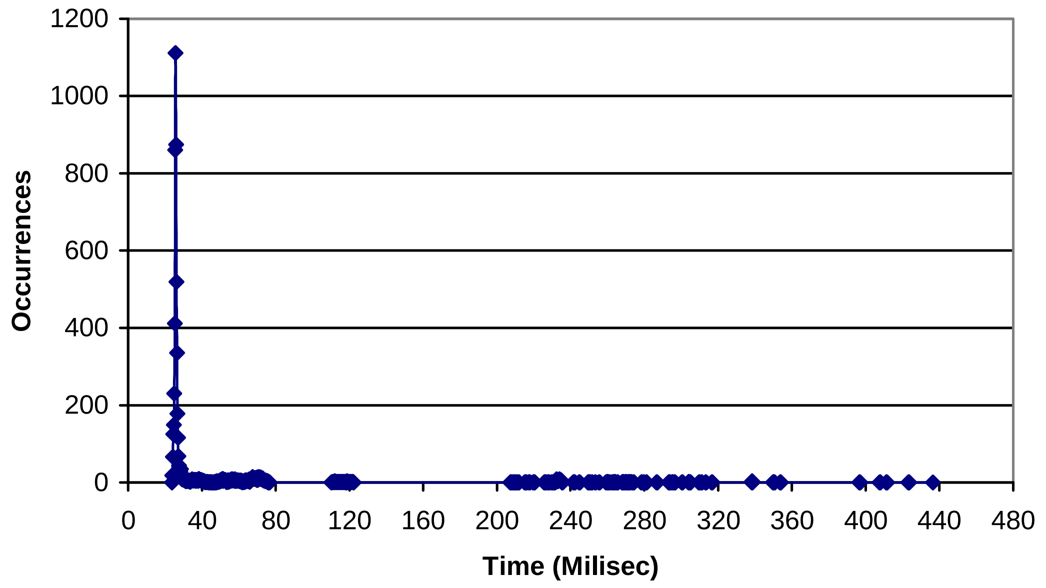

Figure 5.46 : Ethernet at 64 CPUs for 2 KByte.

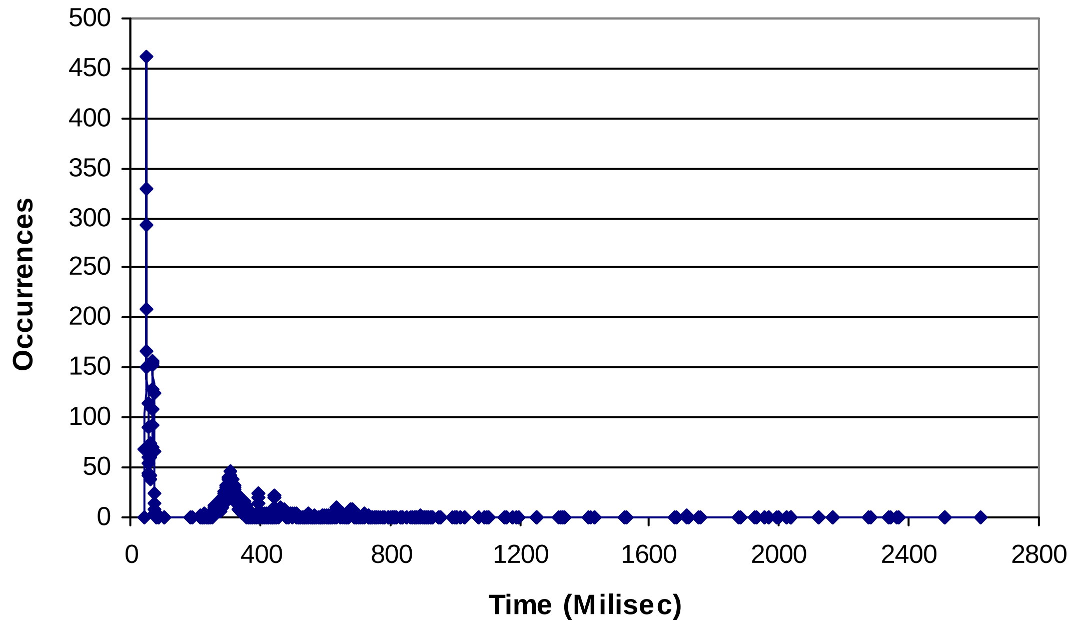

Figure 5.47 : Ethernet at 64 CPUs for 4 KByte.

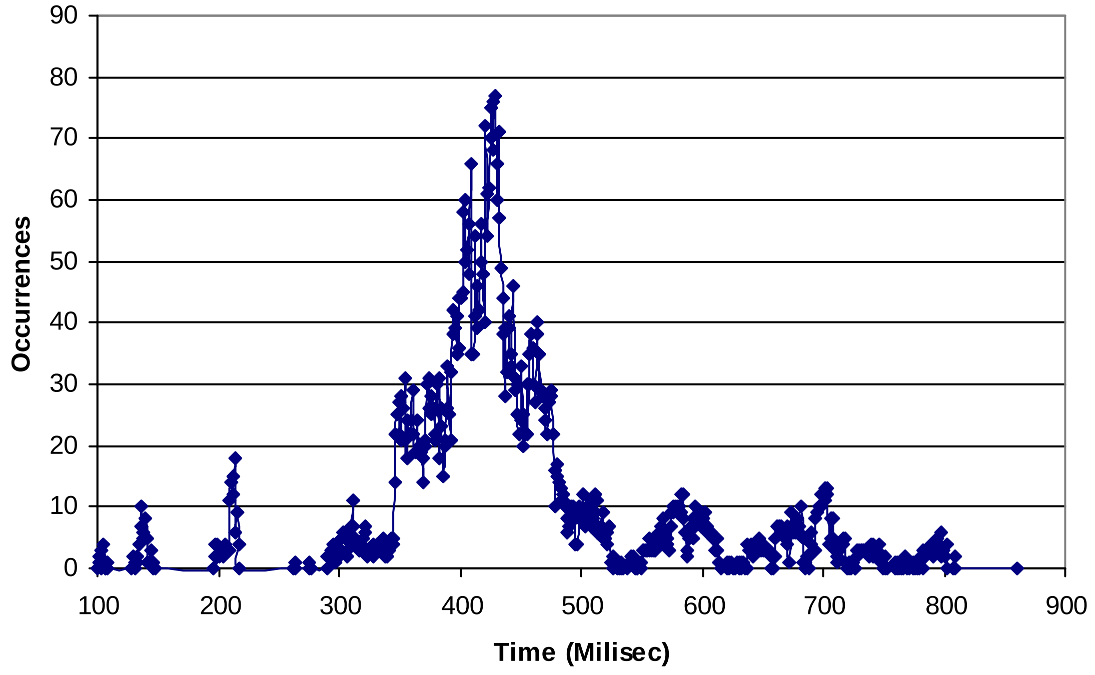

Figure 5.48 : Ethernet at 64 CPUs for 8 KByte

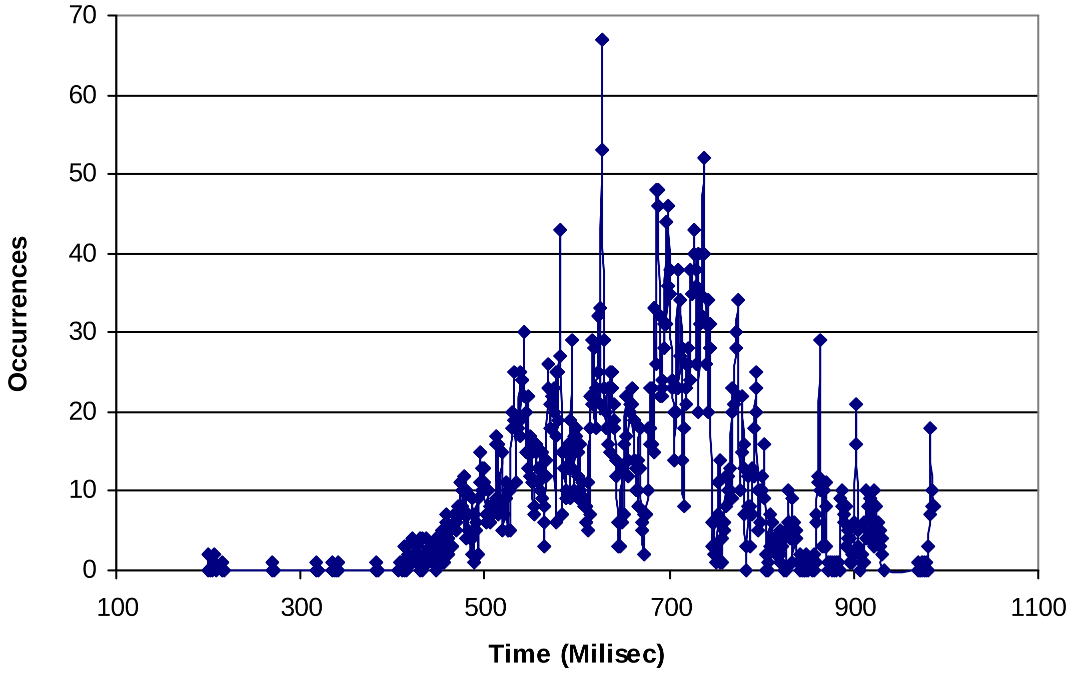

Figure 5.49 : Ethernet at 64 CPUs for 16 KByte

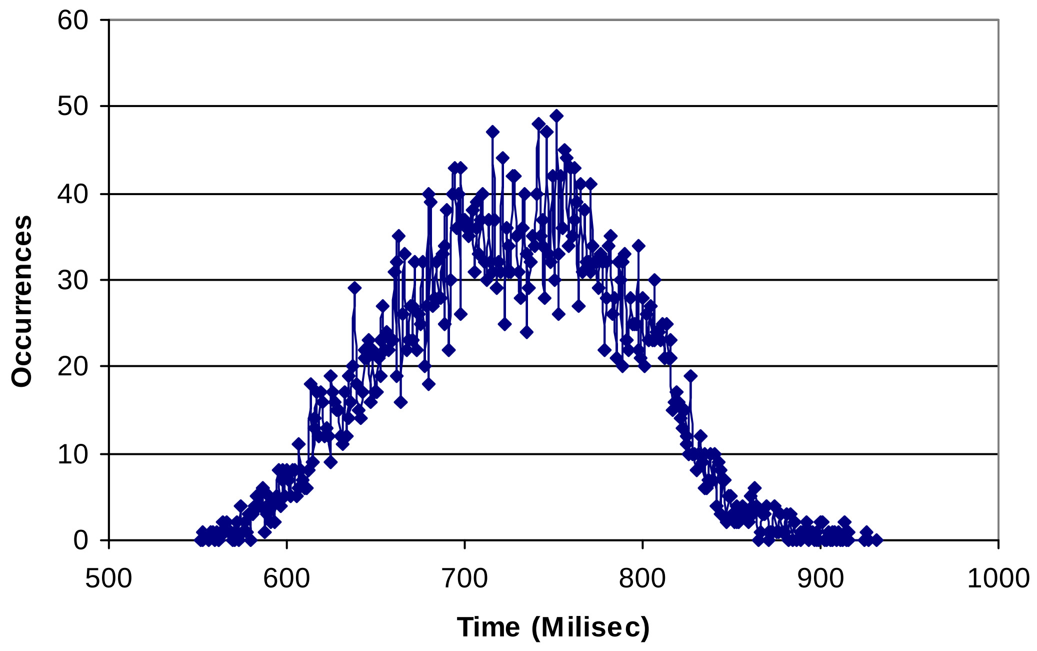

Figure 5.50: Ethernet at 64 CPUs for 32 KByte.

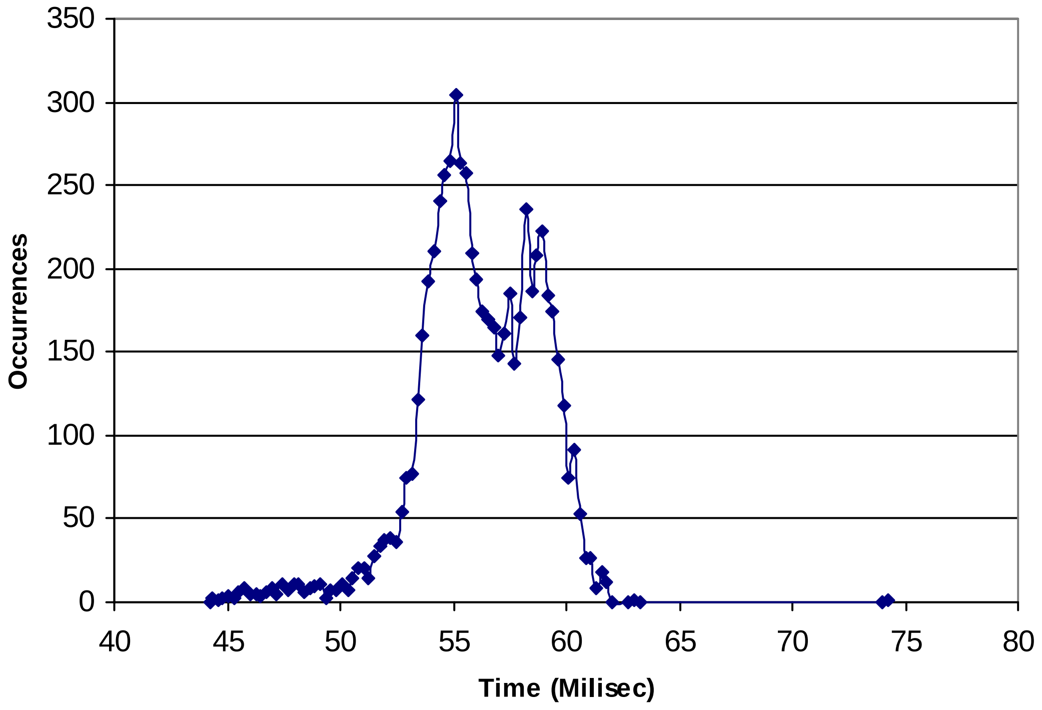

Figure 5.51 : Myrinet at 64 CPUs for 32 KByte.

	No RTO	1 x RTO	2 x RTO	≥ 3 x RTO	Avg. Time (msec)	Est. Avg. Time Without RTO (msec)
32 CPU						
2KB	99.9	0.06	0	0	15.6	15.6
4 KB	100	0	0	0	28.4	28.4
8 KB	97.2	2.63	0	0	117.3	84.41
16 KB	81.19	18.81	0	0	287.1	239.91
32 KB	NA	NA	NA	NA	420.5	NA
64 CPU						
2KB	97.6	2.4	0	0	115.5	112.24
4 KB	55.7	33.0	6.9	4.4	718.6	205.83
8 KB	3.5	78.4	13.1	5.0	528.7	282.23
16 KB	0.2	28.6	61.5	9.6	742.8	381.14
32 KB	NA	NA	NA	NA	844.3	NA
128 CPU						
256 B	89.28	10.72	0	0	217.1	75.13
512 B	96.59	3.29	0.10	0	660.7	407.12
1 KB	95.5	4.37	0.05	0.08	773.7	425.86
2 KB	NA	NA	NA	NA	2425	NA
4 KB	NA	NA	NA	NA	2572	NA

Table 5.7 : Percentage of RTO Occurrences for Alltoall for 32, 64 and 128 CPUs

5.9 Summary

This chapter has compared the performance of Fast Ethernet and Myrinet networks for MPI communications on a commodity Linux PC cluster. In particular, the analyses have investigated the effects of network contention (including Ethernet retransmit time-outs) by measuring and analyzing distributions of communication times for point-to-point and collective communications, and how they scale with increasing message sizes and numbers of processes. As expected, the Myrinet network performs significantly better than Fast Ethernet. The TCP RTO on the Ethernet network does affect communications performance, but only for large message sizes and large numbers of processors (especially where multiple Ethernet switches are needed for communication), where the network becomes saturated. Importantly, in the case for lots of small messages are sent very quickly between lots of processors, which is what happens with MPIBench tests, RTOs do not occur. Hence they should only affect parallel applications that communicate very large messages. So even fine-grained applications that are dominated by lots of communications should not be affected unless the message sizes are large. So, it can have significant impact on the performance of collective communications, particularly MPI_Bcast and MPI_Alltoall. Earlier measurements by Grove et al. [8,9] for older versions of MPICH showed that TCP RTO can greatly reduce the performance of MPI_Bcast and MPI_Alltoall, and even cause them to fail at large message sizes. However the analysis presented in this chapter found the effects of RTOs to be much less serious than for these previous measurements, probably due mostly to improvements in the collective communications routines used in MPICH, although the improved Ethernet network in the machine used in this study, which had switches with more ports and a faster backplane, would also have helped.

This chapter also presented an analysis on the distributions of the communication times for each MPI collective operation, particularly focused on the causes of the slowest communication times. For Ethernet these are often due to RTOs. Even for Myrinet, there were some infrequent occurrences of very slow communications that may be due to problems on the nodes, perhaps operating system interrupts. In other cases, such as Scatter and Gather, RTOs had no effect on the communication times, and the slowest processes in the distribution of communication times were just due to the communication pattern of the algorithm used to implement the MPI operation.

This study also found that MPIBench is a very useful tool for detailed analysis of communications performance, particularly the capability to provide distributions of communication times to enable study of the effects of contention and the occurrence and impact of TCP RTO. It should be useful for researchers who are working on approaches for improving Ethernet performance, such as selecting the best approach for tuning the TCP RTO times, or analyzing the performance of new communication protocols for Ethernet networks.

CHAPTER 6

Analysis of Algorithm Selection for Optimizing Collective Communication with MPICH for Ethernet and Myrinet Networks

6.1 Introduction

This work was motivated by the collective communication results in Chapter 5, particularly for MPI_Bcast and MPI_Alltoall. The plots of average times for MPI_Bcast and MPI_Alltoall for various numbers of CPUs shows a gap or hump at certain message sizes, so the communication times are faster for larger message sizes just above the hump. So, more work has been done in analyzing MPICH in order to understand these unexpected results. MPICH is one of the main implementations of MPI. Recently MPICH research group released a new version, MPICH2. MPICH2 is an all-new implementation of MPI, designed to support research into high-performance implementations of MPI-1 and MPI-2 functionality. All the latest improved algorithms and new support and also the new functionality are included in it.

In MPI, the communication is divided into point-to-point and collective communication. The point-to-point involves communication between two processes, while collective communication involves communication for many processes at the same time. For years the collective communication algorithms have been the main concern of MPI researchers in improving the performance of message passing programming. There have been many papers improving on existing collective communication algorithms, either by suggesting a new algorithm or by identifying which algorithms are suitable for small or large message sizes. Recently, MPICH developers have released implementations of new collective communication algorithms and these have been reported in Thakur et al. [11]. The new algorithms have been applied in MPICH 1.2.6, the following versions, and also in MPICH2 [136].

The new MPICH implementations combine the best algorithms known for each MPI collective communication, and those multiple algorithms are differentiated based on message sizes. The message sizes mainly divide into two, the short-message algorithms aim to minimize latency, while the long-message algorithms aim to minimize the bandwidth [11]. For example the broadcast algorithm has changed from using the standard binomial tree algorithm to a combination of three algorithms, which are binomial tree for small message sizes and scatter followed by allgather for large message sizes, and for allgather either using recursive doubling or ring algorithm.

Currently, the message sizes where the algorithm changes in MPICH are the experimentally determined change-over points based on the work of Thakur et al. [11] which used an IBM SP and a Linux cluster machine connected with Myrinet, both with one process per node. In the paper, they did acknowledge having a plan to determine automatically the algorithm change-over points based on system parameters, since the optimum change-over point probably will be different for parallel computers with different architecture, and particularly with different networks. However, the MPICH 1.2.6 and MPICH2 1.0.4 source code shows that the message sizes where the algorithm is changed are still defined as constants and hard coded.

The aim of this study is to investigate the feasibility of using MPI benchmarks to provide an automated process for selecting the optimal choice of collective communication algorithms for a particular parallel computer and communication network, and to see if this approach is worthwhile by comparing the performance of the optimized MPICH implementation with the current MPICH implementation where the algorithm selection is hard coded.

So, this study measured performance over a range of message sizes for all of the different algorithms for all of the collective communication routines in MPICH that use multiple algorithms, which are:

- MPI_Bcast – binomial tree, recursive doubling and ring with scatter algorithms;
- MPI_Alltoall - store-and-forward and pairwise exchange algorithms;
- MPI_Allgather - a variant of the distribution algorithm for barrier, recursive doubling and ring algorithms;
- MPI_Reducescatter - recursive halving, recursive doubling and pairwise exchange algorithms.

- MPI_Reduce and MPI_Allreduce – binomial tree algorithm or reducescatter (using recursive halving) followed by allgather using binomial tree or recursive doubling algorithms.

Thakur et al. [11] provide a detailed description of all of these algorithms.

Measurements were done on a cluster of dual processor machines using two different networks, Myrinet with GM and Ethernet with TCP. In order to compare the different algorithms for all message sizes, the MPICH code was modified so that the change-over points were no longer constants, but variables that were initialized to the current static values in MPICH, which could then be overridden by reading from an environment variables or a configuration file. For each collective communication routine, an MPI benchmark such as MPIBench can be run to measure the performance for each possible algorithm, by varying the change-over parameters (e.g. by setting them to be the shortest or longest message sizes possible) to ensure that only a single algorithm is used for each benchmark run. Then the benchmark results for all the different algorithms for a particular collective communication routine can be compared and the optimal change-over points for that particular parallel computer can be determined. In future, this approach could be developed further to create automated software for configuring MPICH to provide optimal change-over points for a particular parallel computer, based on benchmark results.

There were four main outcomes from this study. Firstly to demonstrate that it is feasible to use MPI benchmark results to vary the message sizes where the algorithms change from the fixed values in MPICH, and that this can provide a significant improvement in some cases. Secondly, a comparison of results between different interconnects, Myrinet with GM and Ethernet with TCP, showing that the change-over points for different networks can be quite different. Thirdly, comparison of results using the new MPICH2 with MPICH 1.2.6. Finally, comparing algorithms using two processes per node instead of one process per node which was used by Thakur et al.[11], since the advent of multi-core processors means that all modern clusters have multiple CPUs per node.

6.2 Related Work

Thakur et al. [11] reported on improved implementations of collective communication algorithms in recent versions of MPICH and MPICH2. They compared the performance of different algorithms over a range of message sizes, for a Linux cluster with a Myrinet network and an IBM SP. In both cases the measurements were done using one process per node. The results from these measurements were used to fix the selection of algorithms for different message sizes in MPICH and MPICH2. Our work does similar measurements on a machine with more recent processors, and for more than one process per node, which is typical of modern parallel computers. Thakur et al. say that in future work they aim to develop models to allow the selection of changeover points between algorithms to be customized based on system parameters, whereas our work enables customization to be done based on benchmark results.

Recently there have been several efforts aimed at automatically tuning the performance of collective communications algorithms for the particular parallel computer being used [4,5]. These approaches are primarily aimed at tuning the implementation of each collective communication algorithm, for example selecting the optimum buffer size or communication topology for a particular machine.

The most relevant of these studies to our work is by Vandhiyar et al.[139]. They have developed automatically tuned collective communication by conducting several experiments on the system to obtain the optimum algorithm and optimum buffer size for a given collective communication using HARNESS FT-MPI [146], which is a fault tolerant MPI implementation. Since the buffer is the closest temporary storage with the processor, the optimized use of buffer will help in improving the communication performance. They also developed a set of algorithms implementing some of the MPI collective communication routines, for example for broadcast their algorithms are sequential, chain, binary and binomial. Basically, their approach followed three phases. In the first phase, the best buffer size for a given algorithm and for a given number of processors is determined by evaluating the performance of the algorithm for different buffer sizes. For the second phase, the best algorithm for a given message size is chosen by repeating the first phase with a known set of algorithms and choosing the algorithm that gives the best results. In the third phase, the first and the second phase are repeated for different number of processors. Thus, the best algorithm will be chosen for the system and in certain cases there will be several algorithms for each collective communication which are differentiated by different message size, either small or large message sizes. Based on their results, the use of the tuned collective communication resulted in about 30% to 650% improvement in performance over the native implementation on a variety of architectures including an IBM SP2 and clusters connected by Ethernet, Giganet and Myrinet networks. The difference between the work reported in this section and Vandhiyar et al. is that our work used MPICH rather than HARNESS FT-MPI, up to 64 processors rather than 8, and the recent best known algorithms. Also the work of Vandhiyar et. al was primarily focused on finding the optimal buffer size whereas our work is mainly on finding the choice of best algorithm for different message sizes.

Other related research is more focused on suggesting new algorithms or combination of algorithms for each collective communication. For example Van de Geijn [12,13] suggested the best algorithm for short and long vector message sizes for most of the common collective communication such as broadcast, scatter and gather. Rabenseifner [140] suggest a new algorithm for reduce and allreduce, Kale et al. [141] developed a new algorithm for alltoall, Bruck et al. [142] proposed algorithms for allgather and alltoall that are mainly efficient for small message sizes. Those previous works either suggested new algorithms or analyzed the algorithms used in old versions of MPICH.

This study will compare the results between the different algorithms for collective communication used in the latest version 1.0.4 of MPICH2. It is valuable to investigate ways to improve the performance of MPICH since it is widely used. Furthermore, this study will compare the result from different interconnects which are Myrinet with GM and also commodity Ethernet with TCP, and also will experiment using two processors per node instead of one process per node, so the effect of shared memory will be analyzed too. It is expected that the change-over points for Myrinet will be similar to the defaults set in MPICH, but different for Ethernet which has higher latency and lower bandwidth. This is because Thakur et al. [11] used Myrinet as interconnect for their experimental works to determine the change-over points for collective communications that are used in MPICH 1.2.6.

6.3 Methodology

The measurements for the work in this chapter were done using the same parallel computer as for Chapter 5, which was done on an IBM eServer 1350 Linux cluster with 128 compute nodes connected by a Myrinet 2000 network as well as a 100 Mbit/s Fast Ethernet network. So, all the setting and systems configuration were the same. This analysis used the latest MPICH, which is MPICH2 1.0.4 for Ethernet, however MPICHGM 1.2.7 is used for Myrinet since MPICH2 was not available yet for GM when the test was done. However, the collective communication algorithms used are the same in each case.

In order to allow different change-over points in collective communications algorithms for MPICH some changes have been done to the MPICH code, as well as a few lines of code for the setting of environment variables, as shown in the following code fragments. The first code fragment shows the constant value in the existing MPICH program for specifying the short message size in MPI_Bcast. This is followed by some of the modified code to enable dynamic change-over points to be specified using environment variables. The constant MPIR_BCAST_SHORT_MSG_DEFAULT is set to the fixed value used by MPICH. Then, MPIR_BCAST_SHORT_MSG is declared as a variable and initialized to the default value, however it can obtain the new change-over value from the environment variable, using the setConstant function. If the environment variable is not set, there is no change from the fixed default value. The next code fragment shows the environment variable setting to change the fixed value from 1Kbyte to 16 Kbyte for broadcast. Note that, these changes have been made to MPICH2 1.0.4 for Ethernet with TCP and MPICH 1.2.7 for Myrinet with GM.

Fixed values for change-over points in the existing MPICH code

```
(in mpiimpl.h)

#define MPIR_BCAST_SHORT_MSG  1024

…
```

Part of modified code to enable dynamic change-over points

```
(in mpiimpl.h )

#define MPIR_BCAST_SHORT_MSG_DEFAULT 1024
#define MPIR_BCAST_LONG_MSG_DEFAULT   524288
#define MPIR_BCAST_MIN_PROCS_DEFAULT  8
…

extern int MPIR_BCAST_SHORT_MSG;
extern int MPIR_BCAST_LONG_MSG;
extern int MPIR_BCAST_MIN_PROCS;
…..

(in init.c)

int MPIR_BCAST_SHORT_MSG = MPIR_BCAST_SHORT_MSG_DEFAULT;
int MPIR_BCAST_LONG_MSG = MPIR_BCAST_LONG_MSG_DEFAULT;
int MPIR_BCAST_MIN_PROCS = MPIR_BCAST_MIN_PROCS_DEFAULT;
…..

void setConstant(int *constant, char *envVar ) {
        char *envStr = getenv(envVar);
        int constantValue = 0;

        if (envStr != 0) {
                constantValue = atoi(envStr);
                *constant = constantValue;
              printf("EnvironmentVar %s defined with value: %d\n", envVar, constantValue);
             }
      else {
                // do nothing
              printf("EnvironmentVar %s not defined.\n", envVar);
             }
}
void setThresholds() {
   setConstant(&MPIR_BCAST_SHORT_MSG, "MPIR_BCAST_SHORT_MSG");
   setConstant(&MPIR_BCAST_LONG_MSG, "MPIR_BCAST_LONG_MSG");
   setConstant(&MPIR_BCAST_MIN_PROCS, "MPIR_BCAST_MIN_PROCS");
….
```

Environment Variable Setting

```
export MPIR_BCAST_SHORT_MSG=16384
```

The MPI benchmark used for measurements was SKaMPI 4.1 [21, 62] and all measurements were done using the default settings for SKaMPI. SKaMPI was chosen since it has a bigger variety of collective communication routines compared to other MPI benchmarks. It was not possible to run the measurements with dedicated access to the cluster. In order to ensure the accuracy of the results, the measurements of different algorithms were taken one after another and using the same set of CPUs. For sanity checking there were at least three measurements taken for each test and at least one measurement for the same test was also taken using PMB and MPIBench, where the MPI routine was provided by both MPI benchmarks. The measurements were done up to 16 nodes (32 CPUs) with 2 CPUs per node and using 100 repetitions as a default setting for SKaMPI. There were also some preliminary results using 64 CPUs (32 nodes) and using one processor per node, and for numbers of CPUs which are not power of two in order to check for unusual results.

In the following subsections, the formulas for the approximate time expected for the different algorithms are taken from Thakur et al.[11], or (in some cases where that paper does not specify a formula) from comments in the MPICH2 1.0.4 source code [136]. The latency and bandwidth values used in the formulas are based on internode point-to-point communication using SKaMPI. The latency for Myrinet is 15μs, while on Ethernet it is 97μs. The bandwidth for Myrinet is 200 Byte/μs, while on Ethernet it is 11Byte/μs. The analysis of Thakur et al. is done using one process per node, while the analysis here will also consider the performance for shared memory nodes since there are two CPUs per node used for the measurements.

6.4 Broadcast

MPICH2 1.0.4 [136] and MPICH-GM 1.2.7 [69] use some new broadcast algorithms. For small message size (<12KByte) or for less than 8 CPUs the standard binomial tree algorithm is used and the time taken for this algorithm is approximately $T_{tree} = [\lg p](\alpha+n\beta)$, where p is the number of processors, α is the latency, β is the bandwidth and n is the message size. For medium (12KByte < medium < 512KByte) and for long (>512KByte) message sizes it uses scatter followed by allgather algorithm, and for allgather algorithm for medium message size and for power of two (POF2) number of processes uses recursive doubling algorithm and the time taken is $T_{recursive}= \lg p\ \alpha + (p-1)/p\ n\beta$, while for medium message size and for non power of two number of processes and also for long message size, a ring algorithm is used and the time taken is $T_{ring}=(\lg p + p-1)\alpha + 2(p-1)/p\ n\beta$.

0	12KByte	512KByte
Message Size > 12 KByte or CPU < 8 use Binomial Tree Algorithm	POF2 use Scatter with recursive doubling algorithm for Allgather. Not POF2 use Ring Algorithm for Allgather	Scatter with Ring Algorithm for Allgather

Table 6.1 : Summary of Algorithms used by MPICH for Broadcast

Figure 6.1 shows the performance of MPI_Bcast for different message sizes for 8 CPUs on Myrinet using the three different algorithms used by MPICH: binomial tree algorithm, scatter with recursive doubling and scatter with ring algorithm. Figure 6.3 shows the same plot for Ethernet. For 8 processes (4 nodes with 2 ppn) on both Myrinet and Ethernet the performance of scatter and allgather becomes close to the binomial tree algorithm when the message size increases to around 16 Kbytes. However for both networks, the binary tree algorithm remains the best algorithm for all message sizes measured (up to 1 Mbyte), so on this machine the binary tree algorithm should be chosen when the number of processes is less than or equal to 8, rather than less than 8 as in standard MPICH. It is also interesting to note that for medium message sizes (between 12 and 512 Kbytes) where MPICH uses recursive doubling for the allgather, using the ring algorithm for allgather gives up to 50% better performance for Myrinet and even more for Ethernet.

The same effect can also be seen for larger numbers of processors. Figure 6.2 and Figure 6.4 show the performance of MPI_Bcast for different message sizes for 32 CPUs (16 nodes with 2 ppn) on Myrinet and Ethernet using different algorithms. Again, the time for the binomial tree algorithm starts to exceed that of the other algorithms at approximately 16 Kbytes. For Ethernet, the binomial tree and the scatter with ring allgather perform better than scatter with recursive doubling allgather for all message sizes. For medium message sizes where recursive doubling is the default in MPICH, the improvement is approximately 40% to 50%.

For Myrinet, scatter with recursive doubling is (marginally) the best algorithm for message sizes in the range 8 KByte to 32 Kbyte. The change-over point to using the ring algorithm for allgather for large message is therefore much lower than the MPICH default of 512 Kbytes, and the improvement in using the ring algorithm rather than recursive doubling between 32 and 512 Kbytes is approximately 30% to 40%. This result is a bit surprising, since the fixed change-over values in MPICH were taken from measurements on a Linux cluster with Myrinet. The only difference is that the experimental results of Thakur et al., and their theoretical models for estimating the communications time, are all based on one process per node, whereas our measurements were using two processes per node, since the cluster had dual processor nodes.

In order to check this theory, the same test was also run with 8 CPUs for one process per node using Ethernet and 32 CPUs for one processor per node on Myrinet. Figure 6.7 and Figure 6.8 shows the comparison between 1 and 2 processes per node (ppn) with the model for 32 processors on Myrinet and Figure 6.9 and Figure 6.10 shows the same for large message sizes. The model is a close fit to the measured results for 1 process per node, but a poor match to the results for 2 processes per node, particularly for medium message sizes. The results show that recursive doubling is relatively much better for 1ppn than 2ppn. This indicates that in order to be able to specify customized change-over points based on system parameters, as suggested by Thakur et al. [1], a new model is needed that will provide good time estimates for multiple processes per node.

The results in Figure 6.5 and Figure 6.6 show a comparison of results between one and two processors per node for medium message size (between 16 KByte until 1 MByte) for 8 CPUs on Ethernet. The results show a similar trend to those for Myrinet, in that for 2 processors per node the recursive doubling has the worst performance compared to other algorithms for all message sizes, while for 1 ppn recursive doubling shows good performance after 16 KByte and until approximately 128 KByte. The change-over values for small message sizes at 1 ppn agree with the MPICH defaults, however for medium message sizes the change-over occurs earlier than the fixed MPICH value (refer to Table 6.6).

It is also noticeable that for 2 ppn the ring algorithm performs very well for all message sizes. The excellent performance of ring algorithm may be due to the new facility in MPICH2, for example the *mpd* [136] which pre-forms a ring to facilitate rapid process startup for the communication. The ring algorithm will take advantage of the nearest neighbor communication patterns, whereas for recursive doubling processes communicate much farther apart. The performance comparison between MPICH 1.2.6 and MPICH2 1.0.4 on Ethernet for 8 CPUs is shown in Table 6.7. It shows MPICH2 1.0.4 has an improvement of around 1% to 37% compared with MPICH 1.2.6. Note that for non-power-of-two number of processors the performance is not much different to the default settings, since only binomial tree and ring algorithm are used.

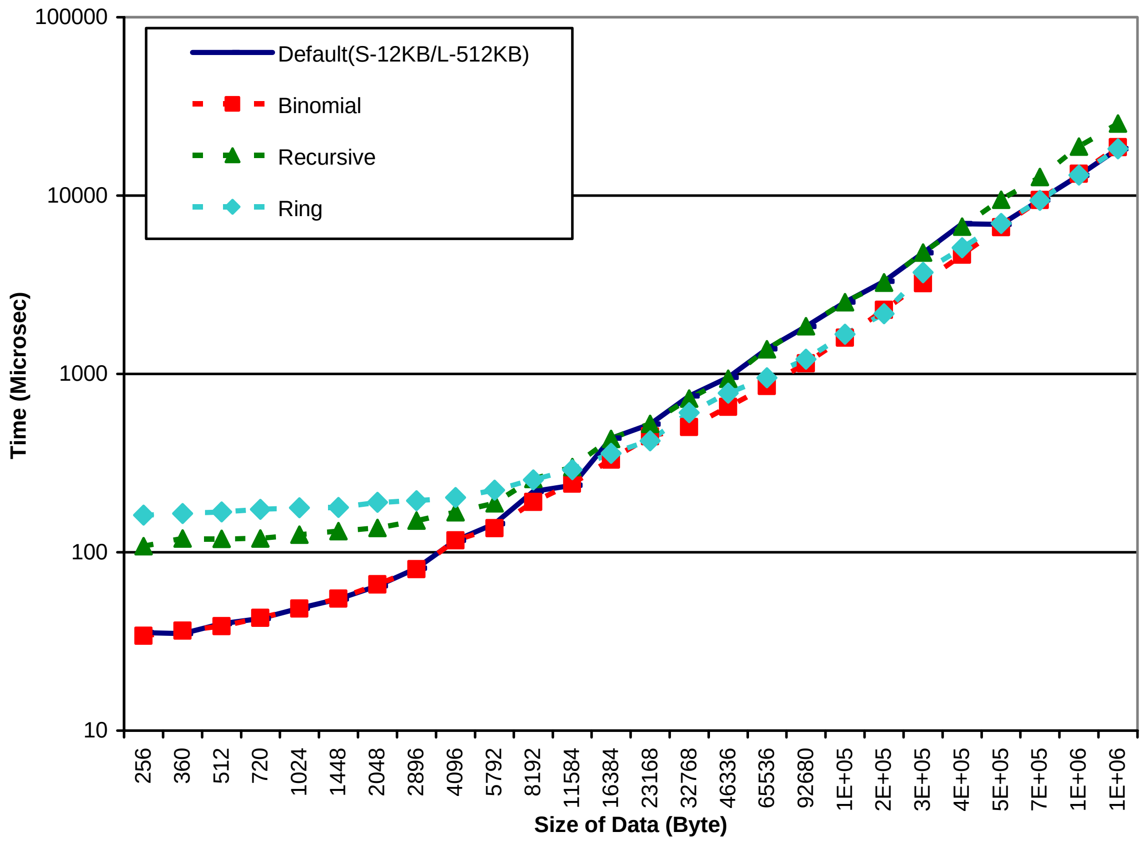

Figure 6.1: 8 CPUs broadcast on Myrinet.

Message Size (Bytes)	**Default(S-12KB/L-512KB)**	**Binomial**	**Recursive**	**Ring**
8192	218	190	257	254
11584	236	241	299	290
16384	435	330	432	359
23168	522	446	523	421
32768	750	503	728	606
46336	954	652	941	781
65536	1378	855	1376	951
92680	1842	1147	1852	1209
131072	2527	1593	2529	1667
185360	3312	2286	3256	2178
262144	4750	3220	4770	3703
370728	6924	4642	6669	5080
524288	6876	6637	9473	6972
741456	9403	9419	12686	9368
1048576	12949	13242	18792	13019

Table 6.2 : Results for 8 CPUs for broadcast on Myrinet.

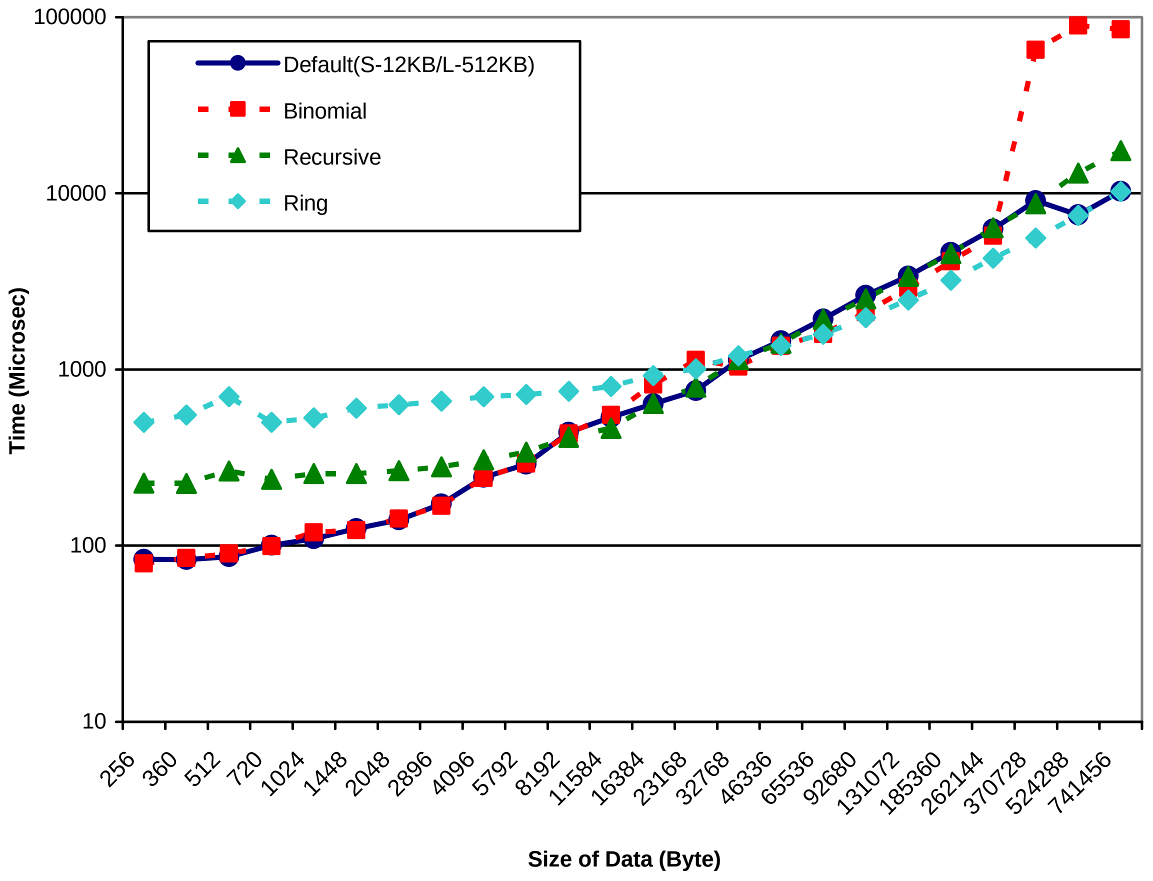

Figure 6.2 : 32 CPU Broadcast on Myrinet.

Message Size (Bytes)	Default(S-12KB/L-512KB)	Binomial	Recursive	Ring
8192	439	432	412	750
11584	535	551	464	800
16384	637	819	640	921
23168	756	1133	791	1006
32768	1130	1034	1135	1196
46336	1452	1362	1405	1364
65536	1935	1589	1923	1584
92680	2636	2110	2521	1963
131072	3393	2904	3380	2480
185360	4615	4102	4540	3215
262144	6270	5757	6384	4283
370728	9075	65135	8737	5583
524288	7557	89719	13031	7493
741456	10257	85287	17406	10198
1048576	17663	51897	238326	17519

Table 6.3 : Results for 32 CPUs for broadcast on Myrinet

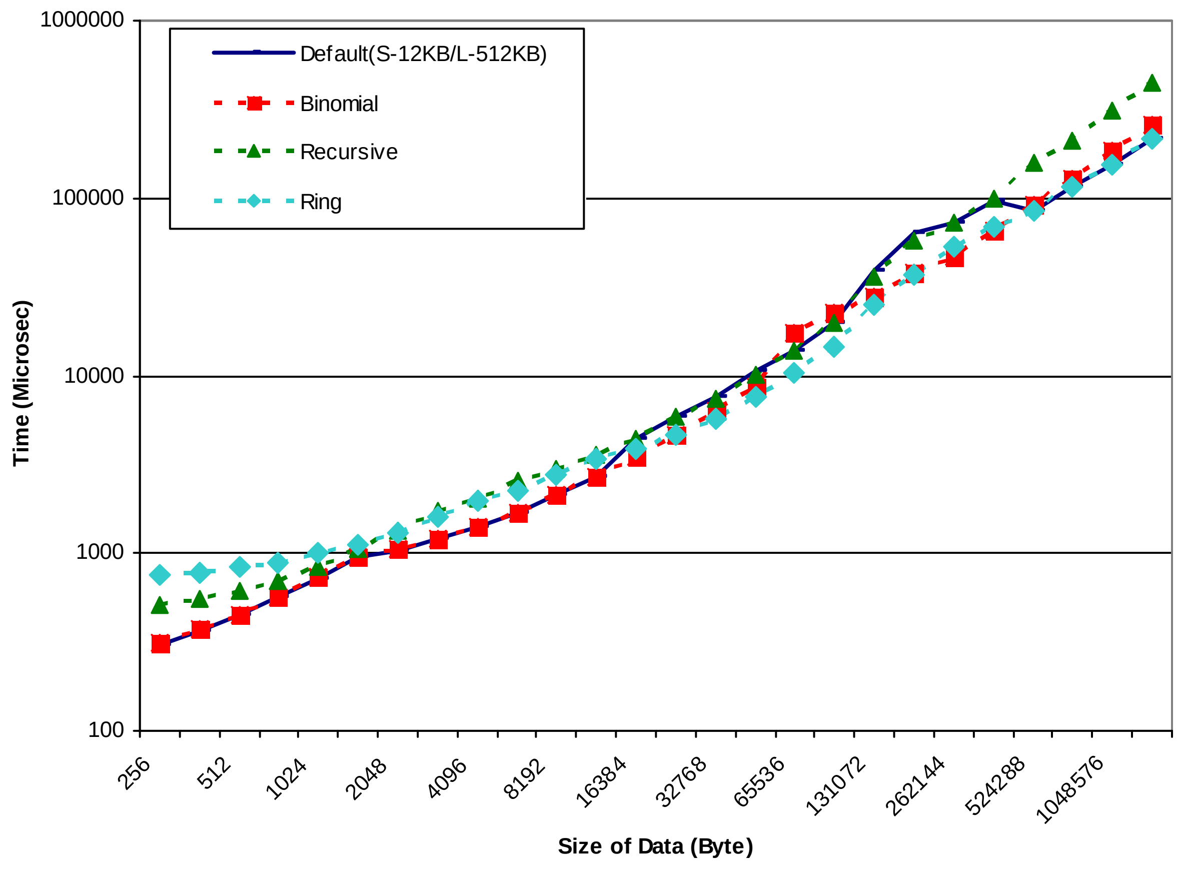

Figure 6.3 : 8 CPU Broadcast on Ethernet.

Message Size (Bytes)	Default(S-12KB/L-512KB)	Binomial	Recursive	Ring
8192	2109	2109	2955	2733
11584	2693	2702	3627	3399
16384	4402	3525	4405	3910
23168	5849	4696	5819	4697
32768	7529	6358	7515	5750
46336	10618	8713	10140	7678
65536	13783	17320	13853	10316
92680	19637	22477	20031	14646
131072	38543	28214	36072	25269
185360	64552	38381	58071	37453
262144	72284	46800	72153	52977
370728	95923	65740	99160	69376
524288	86080	92542	156638	86139
741456	116839	130327	213474	116845
1048576	154694	183758	313106	154704

Table 6.4 : Results for 8 CPUs for broadcast on Ethernet.

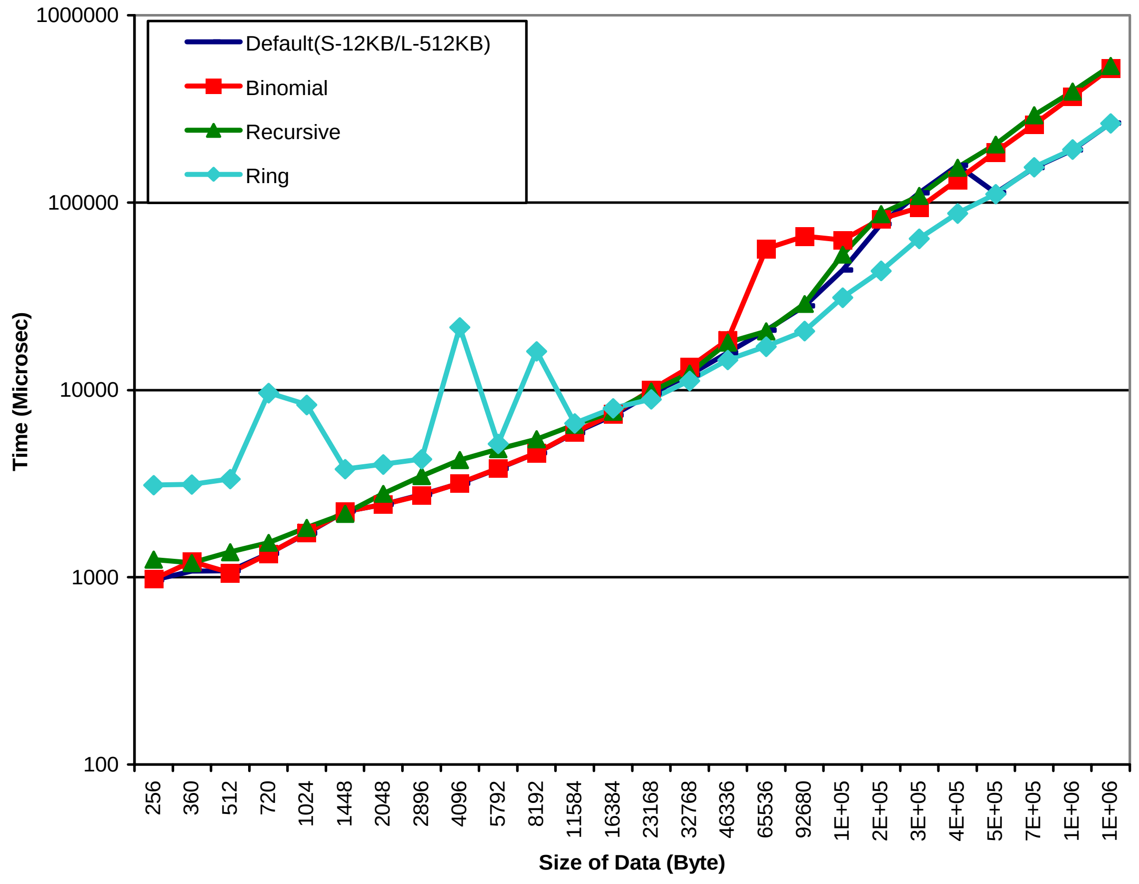

Figure 6.4 : 32 CPU Broadcast on Ethernet.

Message Size (Bytes)	**Default(S-12KB/L-512KB)**	**Binomial**	**Recursive**	**Ring**
8192	4608	4609	5459	16117
11584	5910	5950	6512	6644
16384	7306	7452	7659	7992
23168	9580	10030	9828	8919
32768	12041	13230	12263	11177
46336	15823	18500	17961	14497
65536	20809	56731	20605	17071
92680	28077	66079	28863	20684
131072	43687	63074	52855	31201
185360	76610	81804	87138	43309
262144	112449	93977	108673	64177
370728	157913	132165	154155	87682
524288	112351	185899	204248	111132
741456	153483	261935	293599	154442
1048576	191141	369545	391933	192628

Table 6.5 : Results for 32 CPUs for broadcast on Ethernet

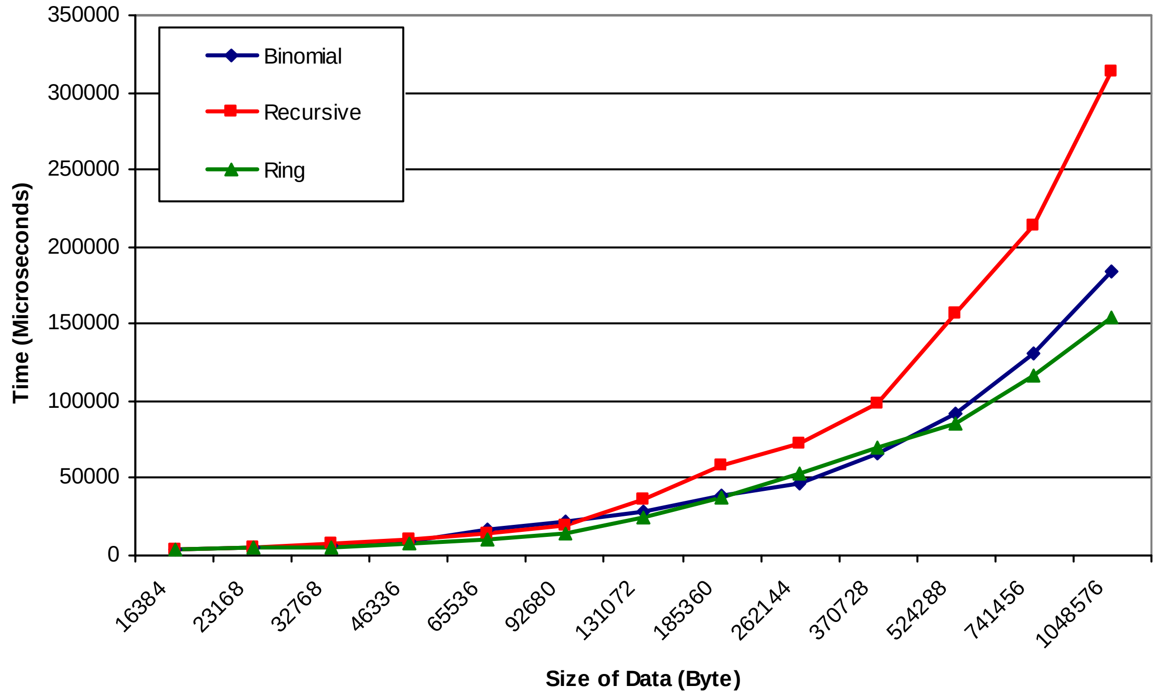

Figure 6.5 : Broadcast for 8 CPUs with 2 ppn for 16 KByte to 1 Mbyte on Ethernet.

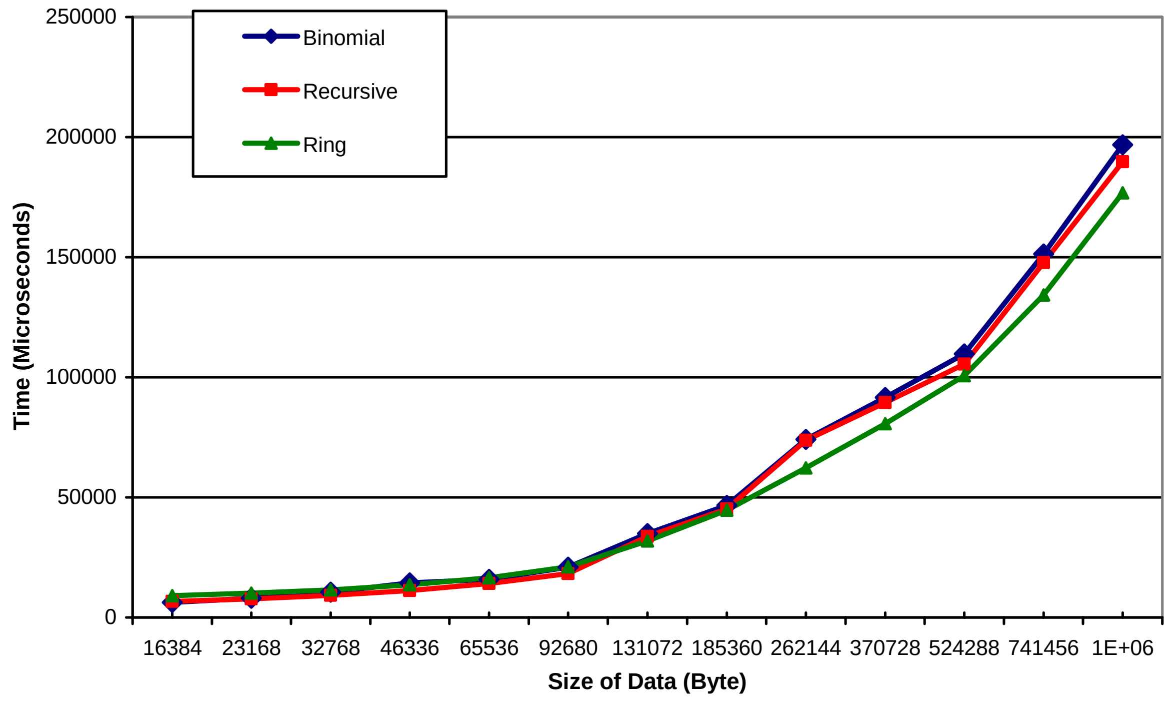

Figure 6.6 : Broadcast for 8 CPUs with 1 ppn for 16 KByte to 1 Mbyte on Ethernet.

Message Sizes (Byte)	**8 CPUs with 2ppn**			***8 CPUs with 1ppn***		
	Binomial	**Recursive**	**Ring**	***Binomial***	***Recursive***	***Ring***
16384	3525	4405	3910	*6278*	*6673*	*9061*
23168	4696	5819	4697	*8020*	*7738*	*10114*
32768	6358	7515	5750	*10502*	*9166*	*11525*
46336	8713	10140	7678	*14368*	*11254*	*13638*
65536	17320	13853	10316	*15924*	*14143*	*16469*
92680	22477	20031	14646	*21066*	*18248*	*21066*
131072	28214	36072	25269	*34573*	*33733*	*31883*
185360	38381	58071	37453	*45665*	*45190*	*44602*
262144	46800	72153	52977	*62215*	*73715*	*62215*
370728	65740	99160	69376	*80651*	*89404*	*80651*
524288	92542	156638	86139	*100702*	*100532*	*100702*
741456	130327	213474	116845	*141273*	*167739*	*134273*
1048576	183758	313106	154704	*196857*	*235789*	*176857*

Table 6.6 : Comparison results between 8p and 2ppn with 8p and 1ppn for Broadcast on Ethernet.

Message Sizes (Byte)	**MPICH2 1.0.4**	**MPICH1.2.6**	**Differences between MPICH2 - MPICH1.2.6**
1024	726	770	5.7 %
2048	1043	1058	1.4 %
4096	1399	1415	1.1 %
8124	2109	2140	1.4 %
12288	2693	2739	1.7 %
16384	4402	4452	1.1 %
32768	7529	7674	1.9 %
65536	13783	18035	23.6 %
131072	38543	39149	1.5 %
262144	72284	80364	10.1 %
370728	69376	110097	36.9 %
524288	86139	108738	20.8 %
741456	116845	138908	15.9 %
1048576	154704	186145	16.9 %

Table 6.7 : Comparison between MPICH2 1.0.4 with MPICH 1.2.6 for Broadcast on Ethernet.

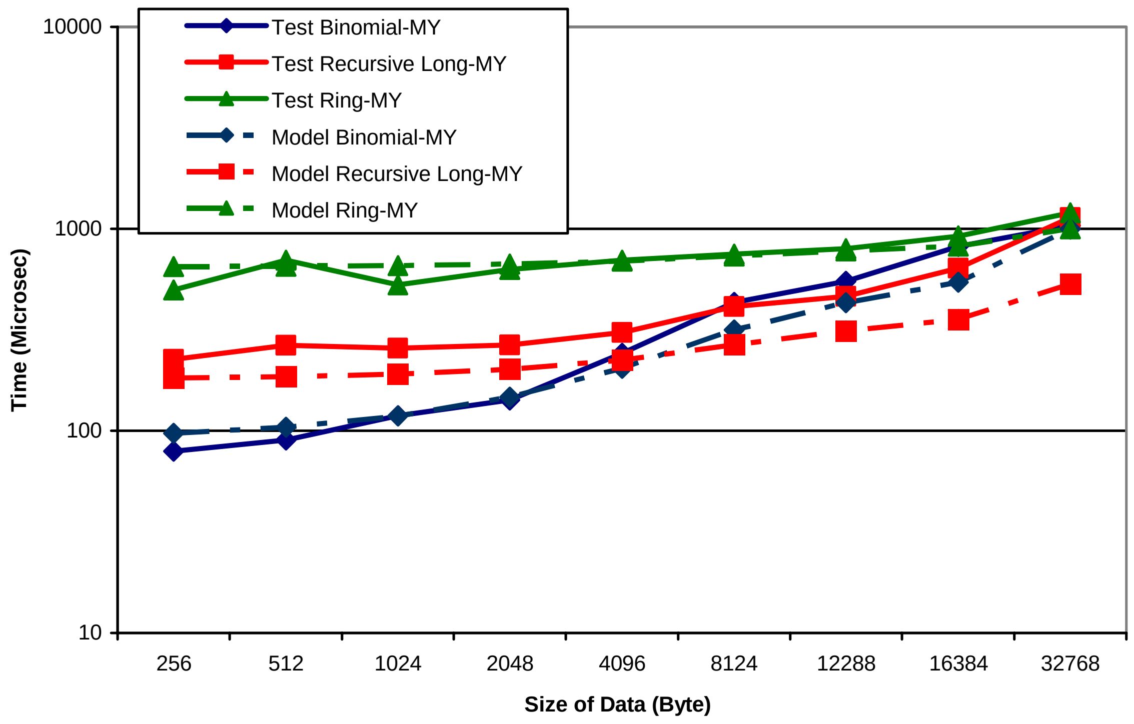

Figure 6.7 : Comparison between test results and model for 2 ppn for 32 CPUs for medium message size on Myrinet.

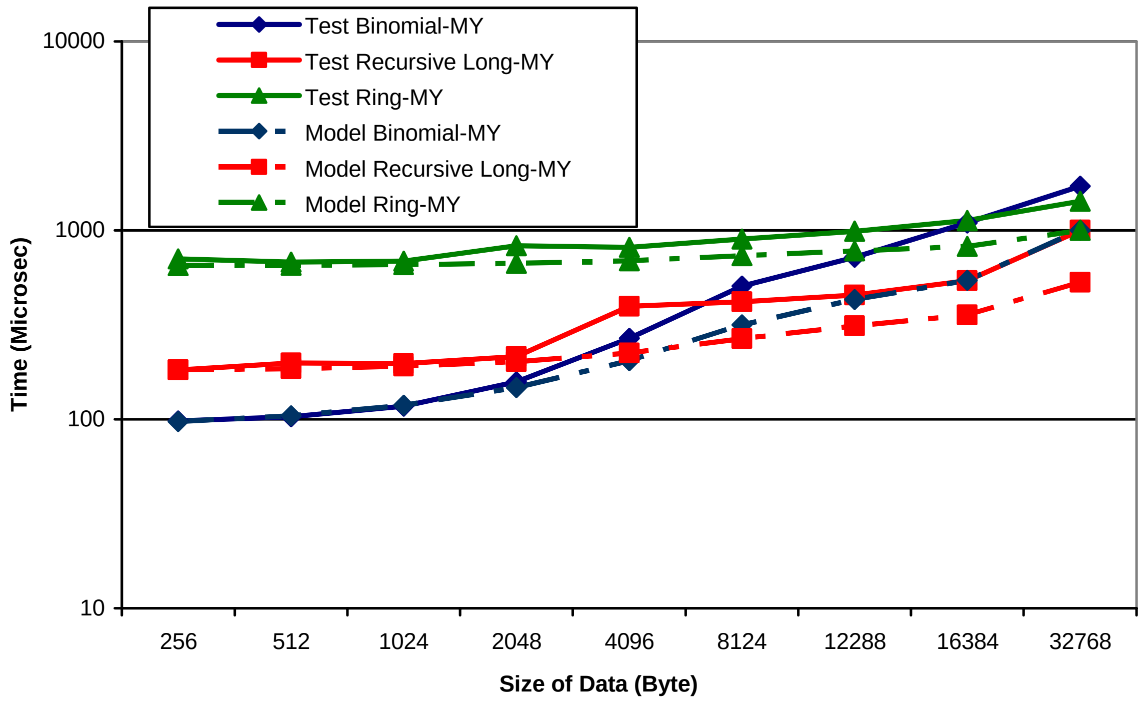

Figure 6.8 : Comparison between test results and model for 1 ppn for 32 CPUs for medium message size on Myrinet.

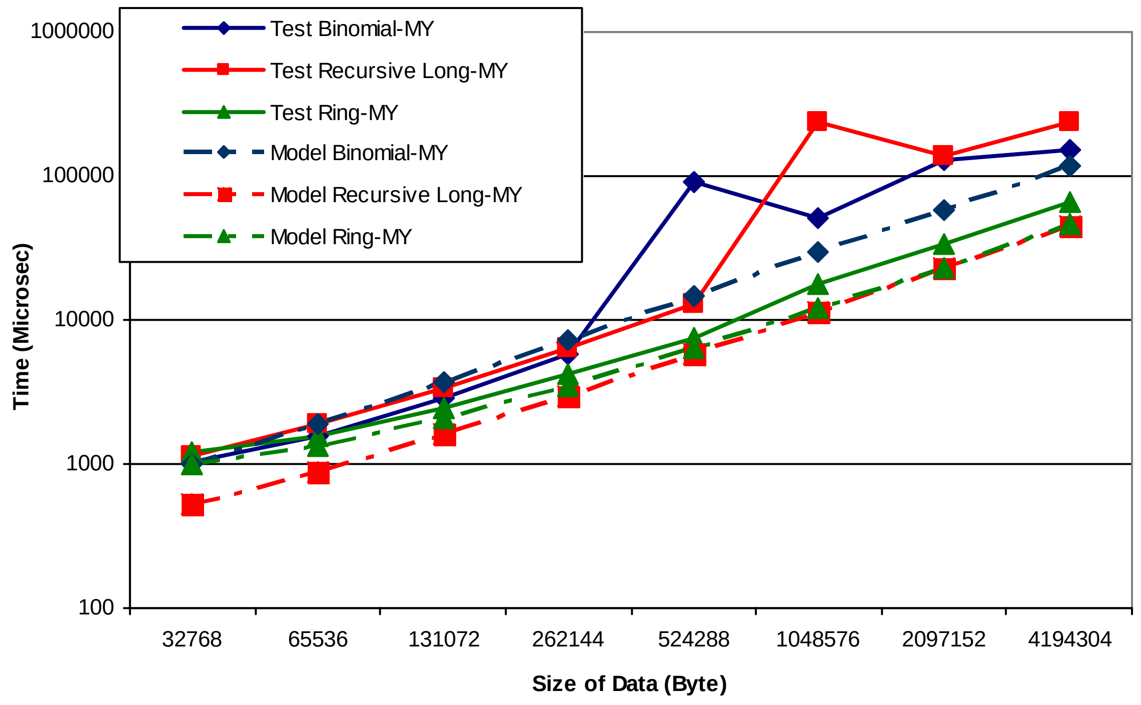

Figure 6.9 : Comparison between test results and model for 2ppn for 32 CPUs for large message size on Myrinet.

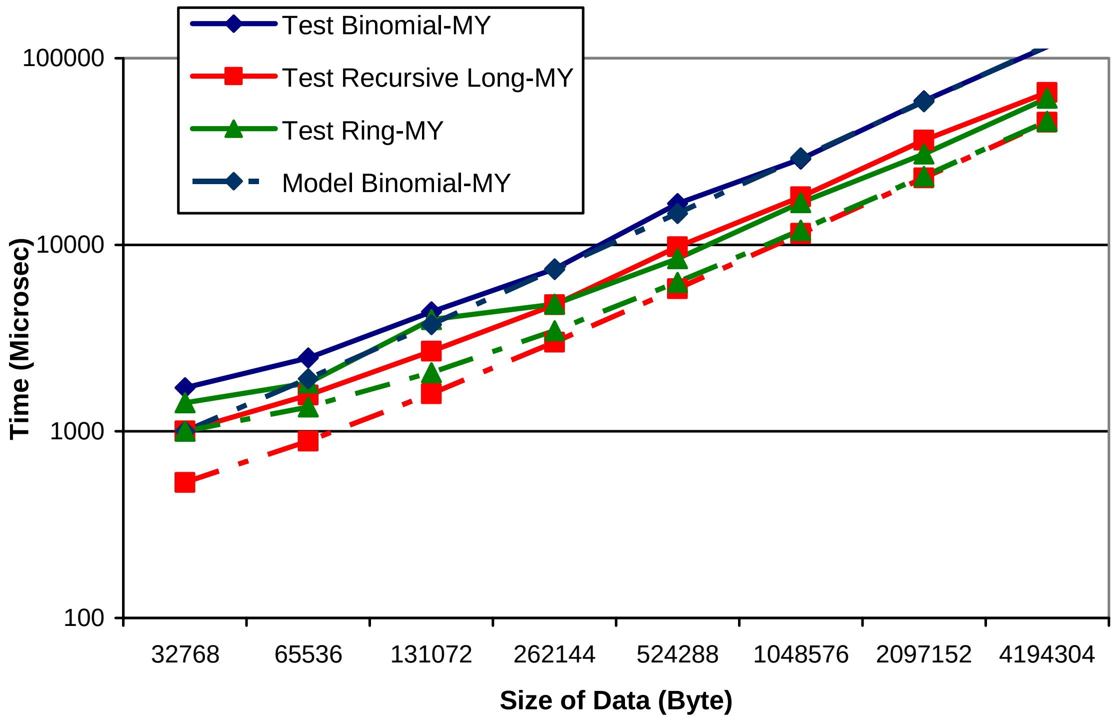

Figure 6.10 : Comparison between test results and model for 1 ppn for 32 CPUs for large message size on Myrinet.

6.5 Alltoall

MPICH2 1.0.4 uses four algorithms for MPI_Alltoall [136]. For short messages (<256Bytes) and (number of processors >= 8) it uses store-and-forward algorithm, which takes [lg p] steps at the expense of some extra data communication ($n/2 \lg p \, \beta$ instead of $n\beta$, where n is the total amount of data to be sent or received by any process) and the time taken is $T_{storeforward} = \lg p.\alpha + (n/2).\lg p.\beta$. Therefore, it is a good algorithm for very short messages where latency is an issue. For medium size messages (256Bytes =< medium message size =< 32768Bytes) and (short messages for number of processes < 8) it uses an algorithm that posts all irecvs and isends and then does a waitall, then scatter the order of sources and destinations among the processes, so that all processes will not be sending and receiving to or from the same process at the same time.

For long messages and power-of-two number of processes, it uses a pairwise exchange algorithm, which takes p-1 steps. In each step *k*, 1<= *k* < *p*, each process calculates its target process as (rank ˆ k) (exclusive-or-operation) and exchanges data directly with that process. This algorithm, however, does not work if the number of processes is not a power of two. So, for the non-power-of-two number of processes, it uses an algorithm in which, in step k, each process receives from (rank-k) and sends data to (rank+k). In both these algorithms, data is directly communicated from source to destination, with no intermediate steps. The time taken by these algorithm is given by $T_{long} = (p-1).\alpha + n\beta$. For more detailed explanation on the algorithms refer to Thakur et al. [11].

0 —— 256Bytes	- - - - 32 KByte	- · - · - · -
For CPU >=8 use store-and-forward algorithm. For CPU < 8 uses irecvs and isends and then does a waitall.	For CPU >=8 uses irecvs and isends and then does a waitall.	For POF2 uses pairwise exchange algorithm.

Table 6.8 : Summary of Algorithms used for Alltoall in MPICH.

Figure 6.11 and Figure 6.13 show the average time for 8 CPUs and 32 CPUs on Myrinet for the default settings, store and forward, isend and irecv and pairwise exchange algorithms for MPI_Alltoall. Figure 6.12 and Figure 6.14 shows the same plot for Ethernet. Measurements using Myrinet and 2 ppn show that the default settings are close to optimal, although moving the small message change-over point from 256 Bytes to 512 Bytes gives an improvement of 30% to 50% for messages in that range. The transition from medium to larger message sizes should occur at 32 KByte, where the isend and irecv algorithm should change to the pairwise exchange algorithm. However the results show that the pairwise exchange algorithm performs about the same as isend and irecv algorithms.

The improvement for Ethernet with 2 ppn is much greater. In that case, the store-and-forward algorithm turns out to be slower than isend/irecv even for small messages, so using isend/irecv improves performance by around a factor of 2 for messages of size 256 bytes or less. The pronounced hump in the results for 32 CPUs is probably due to RTOs (see Section 5.8) which do not seem to have as much of an effect for the store and forward algorithm.

Figure 6.15 shows the results from running the benchmarks using 1 process per node for 32 processors on Myrinet, and the change-over values for small message sizes agrees with the MPICH defaults. Figure 6.16 and Figure 6.17 shows the comparison between 1 and 2 processes per node (ppn) with the model for 32 processors on Myrinet. The results for isend/irecv and pairwise exchange are a close fit to the measured results for 1 process per node, but a poor match to the results for 2 processes per node. However, for store/forward model, the results for 1 processor per node only have a slight improvement compared with 2 processes per node measured results.

Based on the above results and discussions, it is suggested that for Myrinet on this cluster only two algorithms are needed, which are the store-and-forward and isend and irecv or pairwise exchange, and the change over points for smaller message sizes is the same as the default settings. On Ethernet, it shows that only one algorithm is needed which is either the isend and irecv algorithms or the pairwise exchange, since the performance is almost the same. The improvement using the suggested algorithms and change-over point for MPI_Alltoall is nearly 50% for small message sizes on Ethernet, while for Myrinet there was little change. For large message sizes there was no improvement for both networks since the result between isend and irecv and pairwise exchange is almost the same.

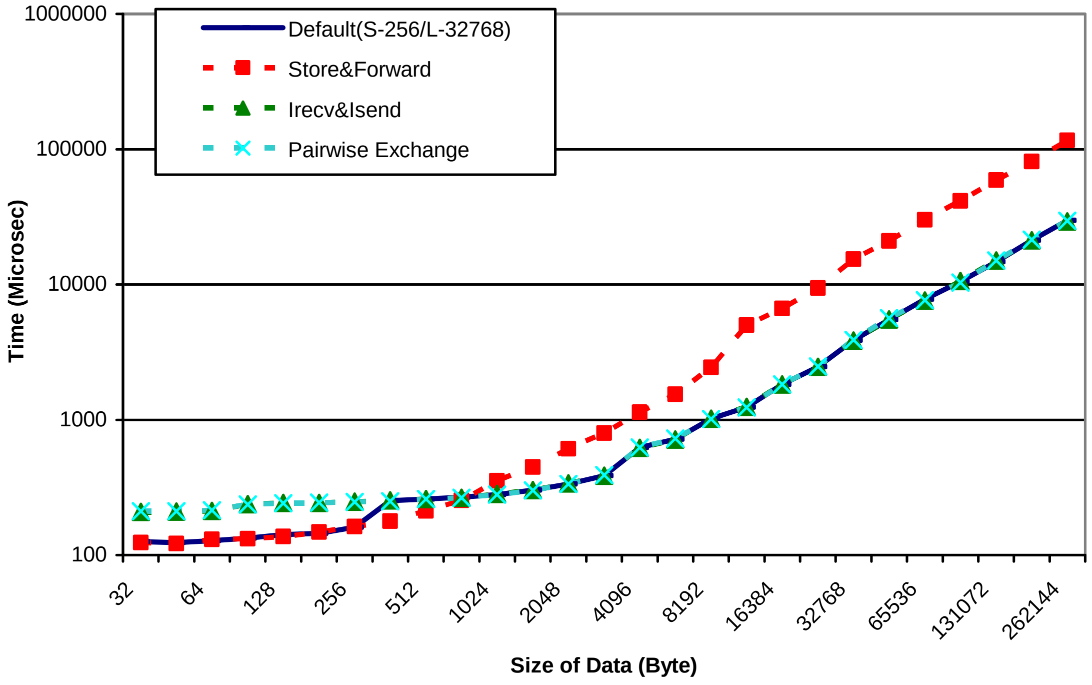

Figure 6.11 : 8 CPU and 2 ppn for Alltoall on Myrinet.

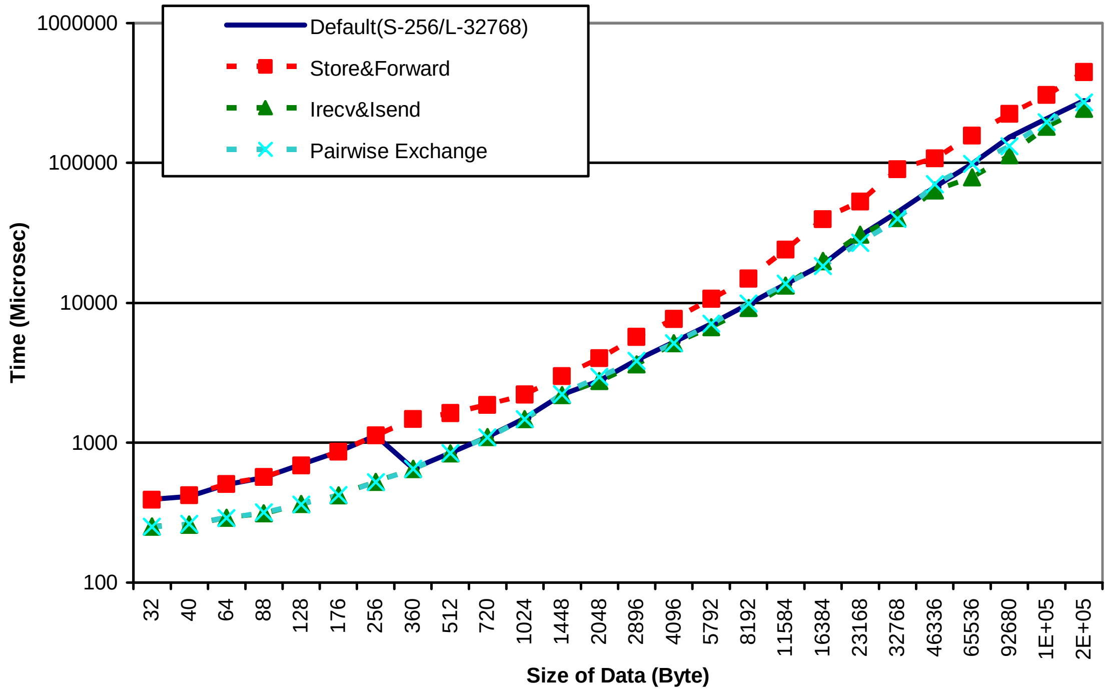

Figure 6.12 : 8 CPU and 2ppn for Alltoall on Ethernet.

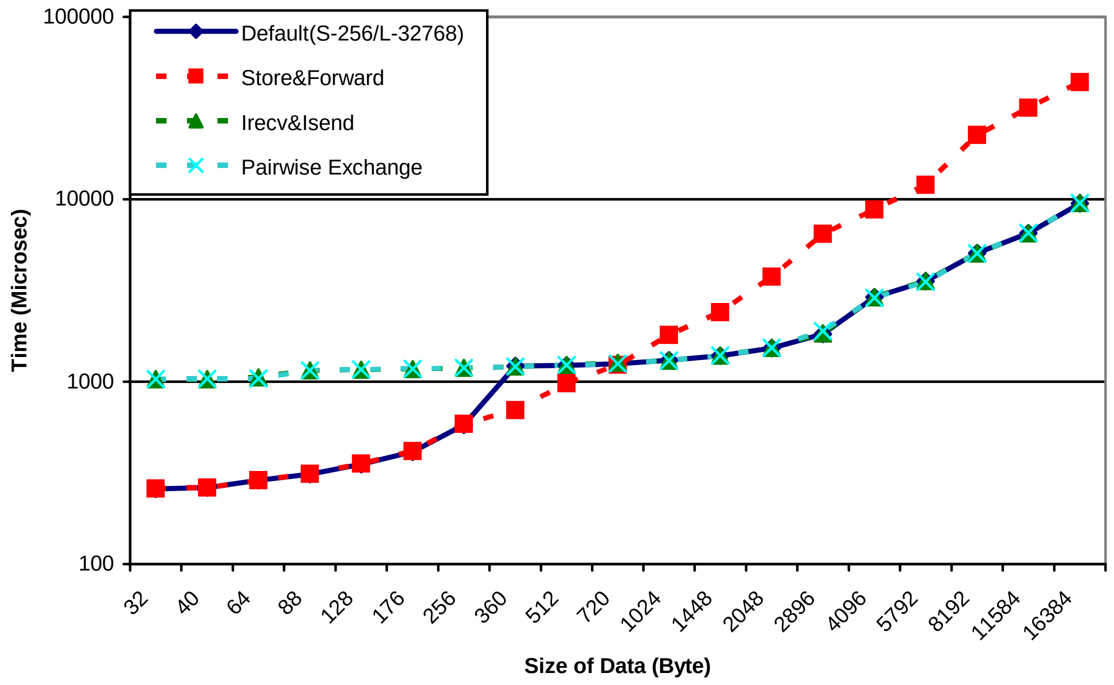

Figure 6.13 : 32 CPU and 2 ppn for Alltoall on Myrinet

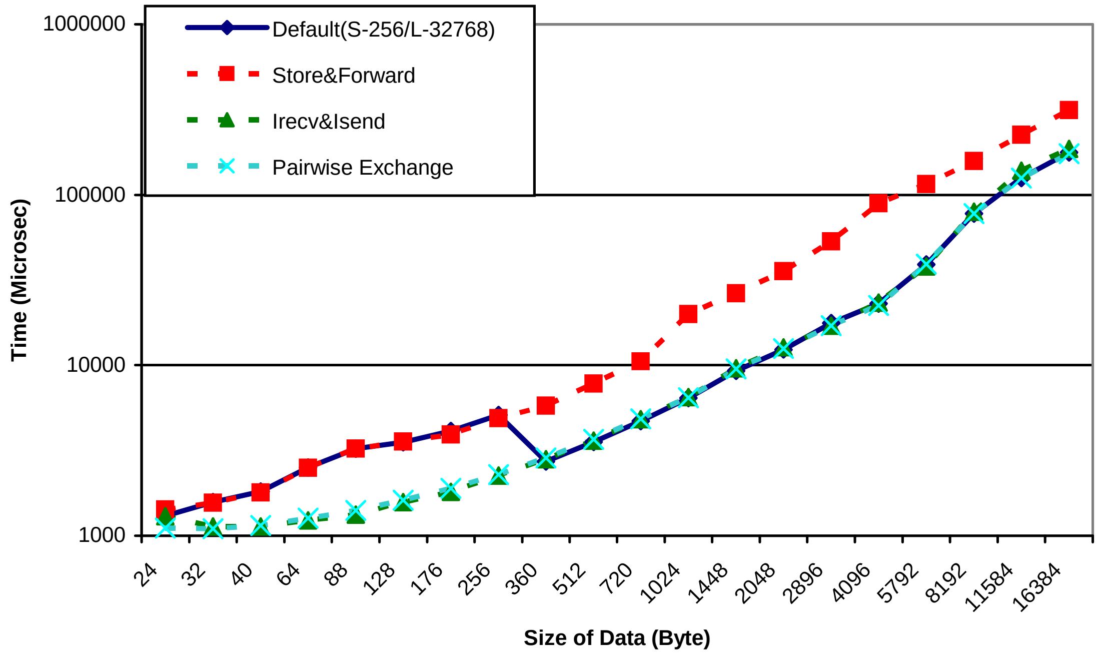

Figure 6.14 : 32 CPU and 2 ppn Alltoall on Ethernet.

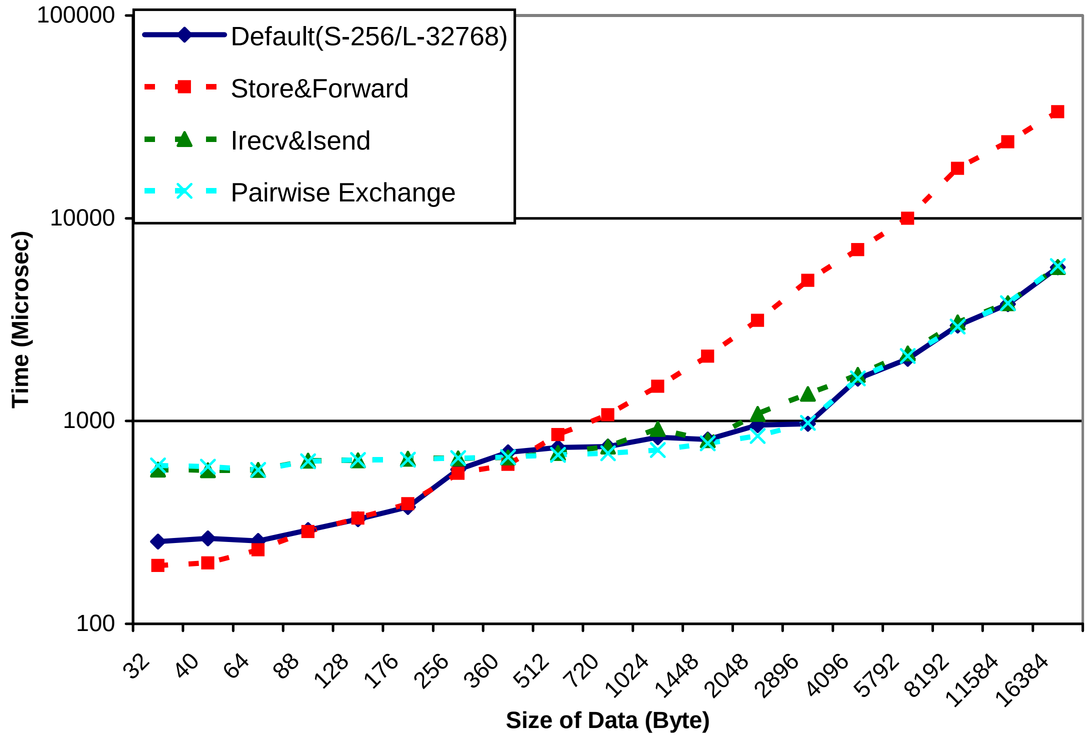

Figure 6.15: 32 CPU and 1 ppn for Alltoall on Myrinet

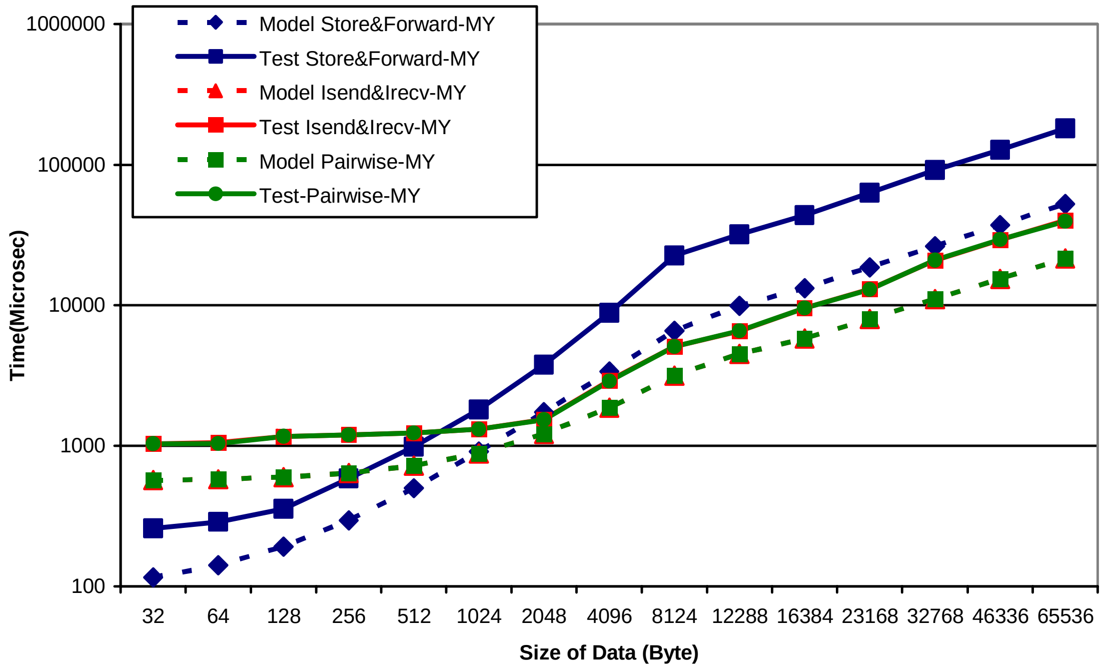

Figure 6.16 : Comparison between test results and model for 2ppn for 32 CPUs on Myrinet.

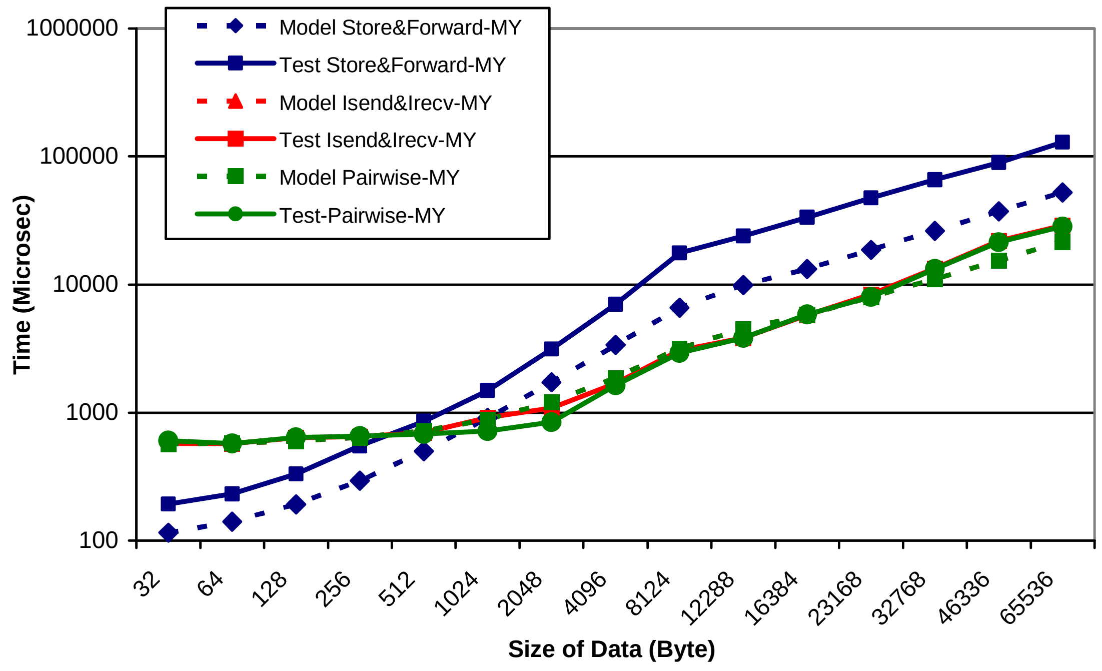

Figure 6.17 : Comparison between test results and model for 1ppn for 32 CPUs on Myrinet.

6.6 Reduce Scatter

The algorithm for reduce scatter considers two types of reduction operation, either commutative or non-commutative. A binary operation is *commutative* if *ab* = *ba* for any possible *a* and *b*. Commonly, addition and multiplication are commutative operations, whereas multiplication of matrices is generally non-commutative. If the operation is commutative, for short and medium-size messages, MPICH uses a recursive-halving algorithm in which the first p/2 processes send the second n/2 data to their counterparts in the other half and receive the first n/2 data from them. This procedure continues recursively, halving the data communicated at each step, for a total of lgp steps. So the time taken will be T_{rec_half} = lgp.α + ((p-1)/p)n.β + ((p-1)/p)n.γ. The time required for a typical arithmetic operation such as multiple or add is indicated by γ. If the number of processes is not a power-of-two, it will convert to the nearest lower power-of-two by having the first few even-numbered processes send their data to the neighboring odd-numbered process at (rank+1). Those odd-numbered processes compute the result for their left neighbor as well in the recursive halving algorithm, and then at the end send the result back to the processes that do not participate. The time taken for non-power-of-two is

$T_{rec\_halv} = ([\lg p] + 2)\alpha + 2n\beta + (1+(p-1)/p)n.\gamma$. However, the above cost in the non power-of-two case is approximate because there is some imbalance in the amount of work each process does since some processes do the work of their neighbors as well [11].

For commutative operations and very long messages a pairwise exchange algorithm is used, similar to the one used in MPI_Alltoall. At step i, each process sends n/p amount of data to (rank+i) and receives n/p amount of data from (rank-i) and the time taken is $T_{long} = (p-1)\alpha + ((p-1)/p)n.\beta + ((p-1)/p)n.\gamma$.

For non-commutative operations a recursive doubling algorithm is used for very short message sizes, which takes lgp steps. At step 1, processes exchange (n-n/p) amount of data; at step 2, (n-2n/p) amount of data; at step 3, (n-4n/p) amount of data, and so forth. So the time taken will be $T_{short} = \lg p.\alpha + n(\lg p-(p-1)/p)\beta + n(\lg p-(p-1)/p)\gamma$. The time required for the reduction operation is indicated by γ and this thesis takes 0.1 microsecond as an estimate for γ. It is hard to estimate the value for γ. Therefore, we choose 0.1 as the value since it gives the best match between the predicted and measured values. The algorithm for medium and long messages uses the same algorithm as commutative for long message sizes, which is pairwise exchange, and the time taken will be the same as above.

0	512B	512KB
- For Commutative uses recursive-halving algorithm. - For Not Commutative uses recursive doubling algorithm.	- For Commutative uses recursive-halving algorithm. - For Not Commutative uses pairwise exchange algorithm.	- For Commutative and Not Commutative uses pairwise exchange algorithm.

Table 6.9 : Summary of Algorithms used for Reduce Scatter in MPICH.

The default setting of SKaMPI is using non-commutative operations, so the change-over point should have an effect at smaller message sizes. SKaMPI Reduce Scatter operation performs a tree-wise data reduction operation (Bitwise OR[1]) on all participating processes and then distributes the result partially to all participating nodes, with every node receiving a different part of the result array.

[1] A **bitwise OR** takes two bit patterns of equal length, and produces another one of the same length by matching up corresponding bits (the first of each; the second of each; and so on) and performing the logical OR operation on each pair of corresponding bits. In each pair, the result is 1 if the first bit is 1 **OR** the second bit is 1 (or both), and otherwise the result is 0

In order to check for commutative operations, measurements using PMB were performed, which uses MPI_FLOAT as the data type and MPI_SUM as the MPI operation. The results for PMB show that if the change-over point is decreased to 100 KByte from 512 KByte (the fixed setting for long message size for commutative operation), then in that range the pairwise exchange algorithm gives approximately 5% to 10% improvement compared to using the recursive-halving algorithms. So, for commutative operations there is little benefit in using a different change-over point.

Figure 6.18 to Figure 6.21 show the average time for the default settings, recursive doubling and pairwise exchange on Myrinet and Ethernet for 8 and 32 CPUs, for the non-commutative operation measured by SKaMPI. The results for Myrinet for 8 CPUs show that recursive doubling performs better than pairwise exchange up to 8 Kbytes, rather than the MPICH default of 512 Bytes. As the number of CPUs is increased, the cross-over between the two algorithms increases also, as shown in Figure 6.20 for 32 CPUs, where the change-over point is 16 KByte. The improvement in performance from using the optimum change-over point increases as the number of CPUs increases. At 32 CPUs it is more than a factor of 4 between 512 bytes and 4 Kbytes, and more than a factor of 2 up to 8 Kbyes.

On Ethernet there is little difference in performance between the two algorithms for 8 CPUs, but as with Myrinet, as the number of CPUs is increased the improvement by using recursive doubling is increased. For 8 CPUs the change-over occurs at 2 KByte instead of 512 Byte, while at 32 CPUs the change-over point is at 8 KByte. As shown in Figure 6.21, the performance improvement from moving the change-over value is quite significant, although not as large as for Myrinet.

Figure 6.22 shows the performance for 32 CPUs on Myrinet using one process per node (1ppn) rather than two, which again shows results consistent with the default change-over point in MPICH, so the difference in the optimal change-over points from the MPICH defaults is again due to the use of 2ppn rather than 1ppn. Figure 6.23 and Figure 6.24 shows the comparison between 1 and 2 processes per node (ppn) with the model for 32 processors on Myrinet. The cross over based from the calculated value from the model is approximately at 1 KByte, which is very near to the default cross over at 512Byte. The measured results for 1ppn shows a very close fit with recursive doubling algorithm until larger message sizes. However, for 2ppn the measured result shows a poor performance particularly for recursive doubling algorithm.

Based on this analysis it is suggested that the change over point for smaller message sizes should increase from 512 Byte to 16 KByte for both networks on this cluster. Although for 8 CPUs and Ethernet the cross-over between the algorithms occurs at less than 16 KByte, the performance is almost the same between both algorithms until 16 KByte, So it is still worth using recursive doubling until 16 KByte for at least 32 CPUs for non-commutative operations.

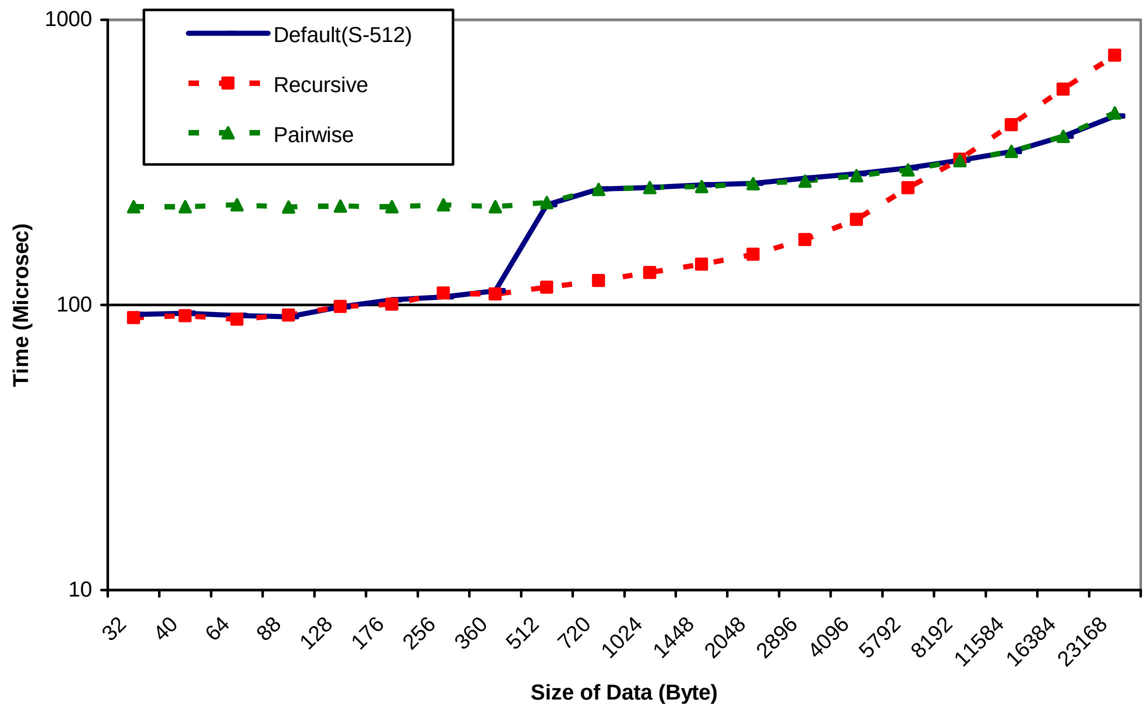

Figure 6.18 : 8 CPU on Myrinet for Reduce Scatter

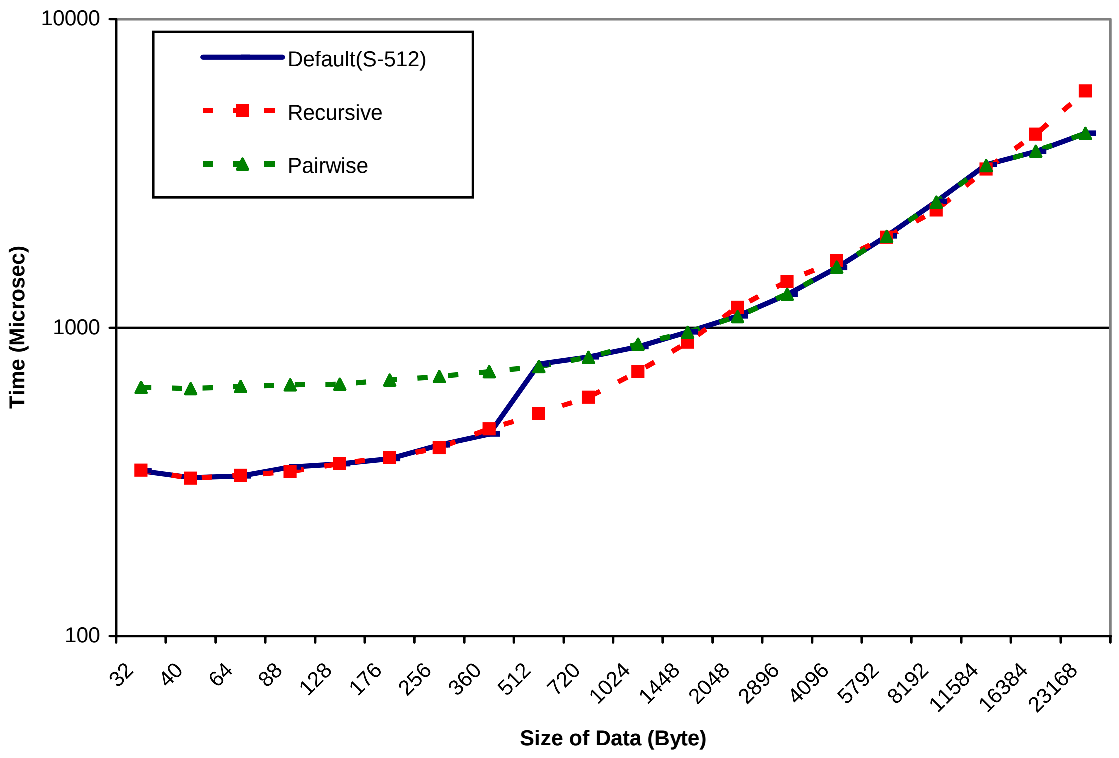

Figure 6.19 : 8 CPU on Ethernet for Reduce Scatter

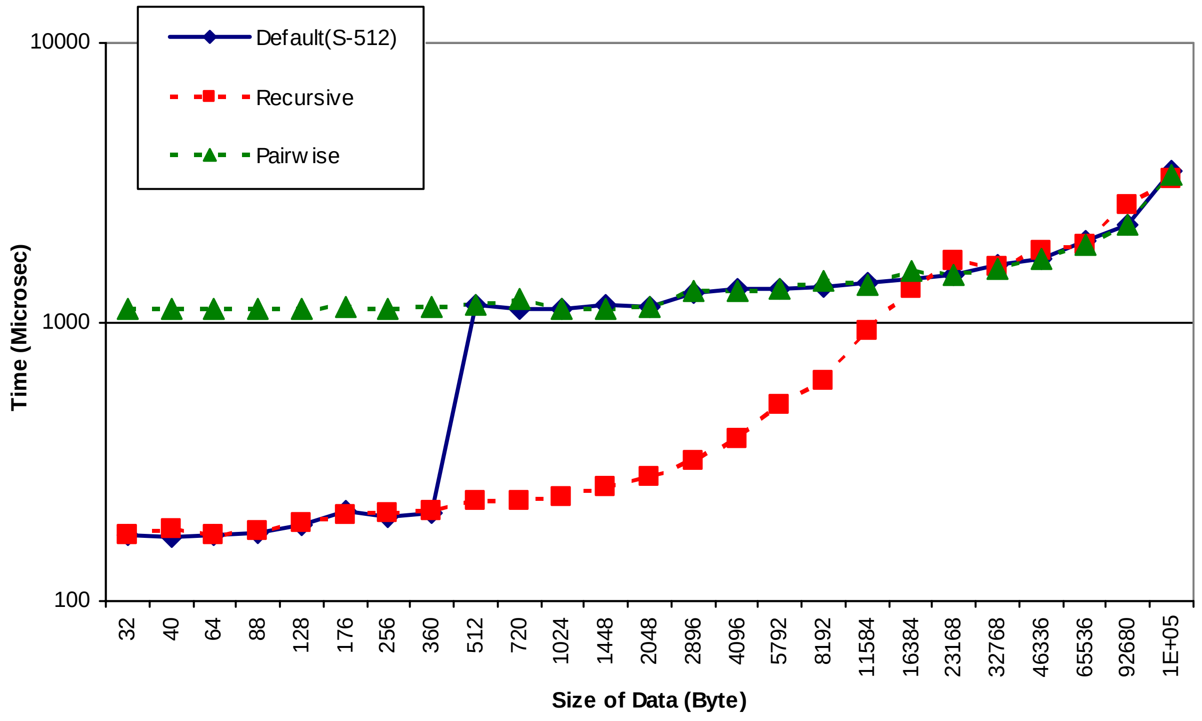

Figure 6.20 : 32 CPU on Myrinet for Reduce Scatter

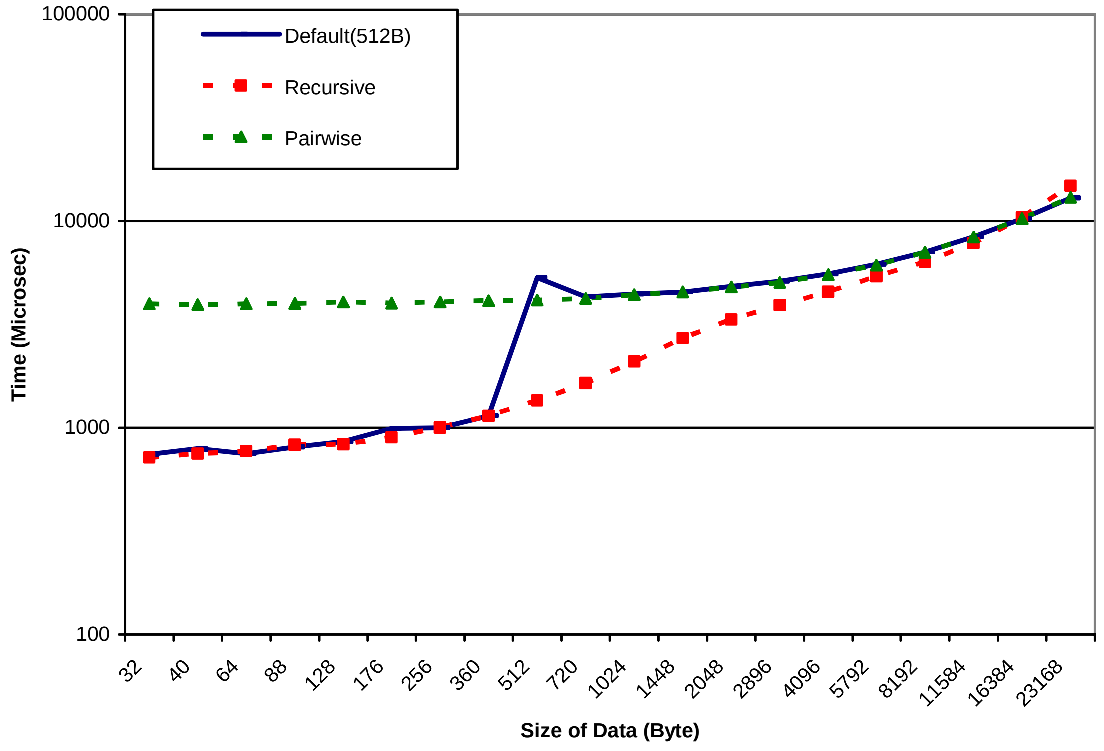

Figure 6.21 : 32 CPU and 2 ppn on Ethernet for Reduce Scatter

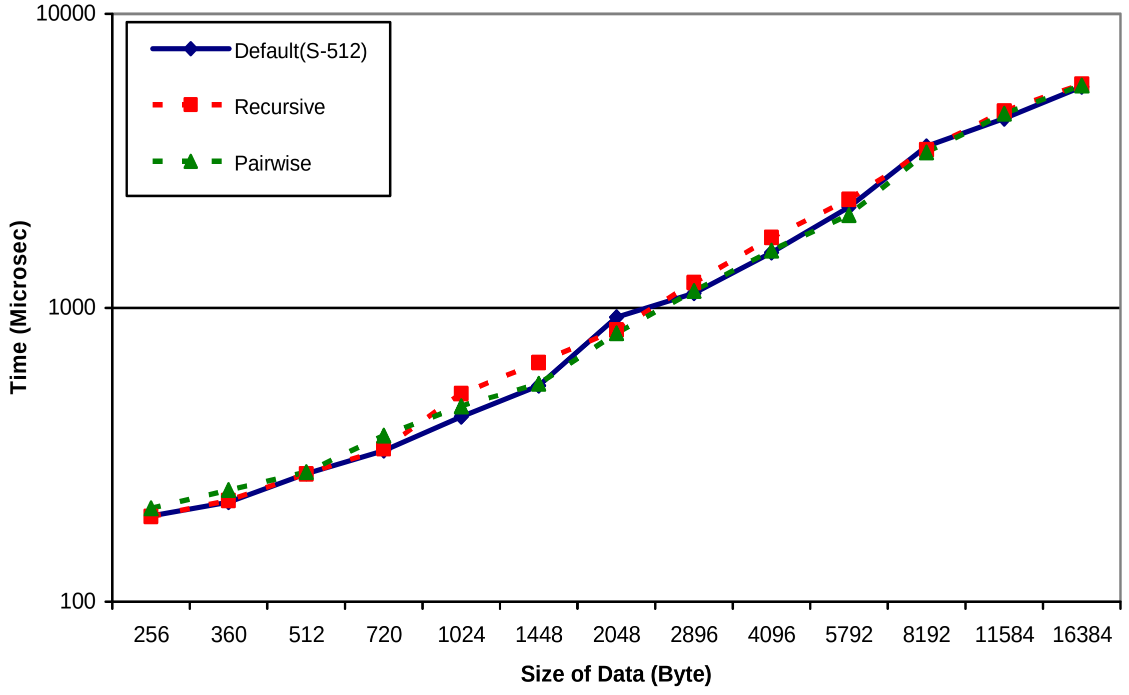

Figure 6.22 : Results for 32 CPUs and 1 ppn for Reduce Scatter on Myrinet.

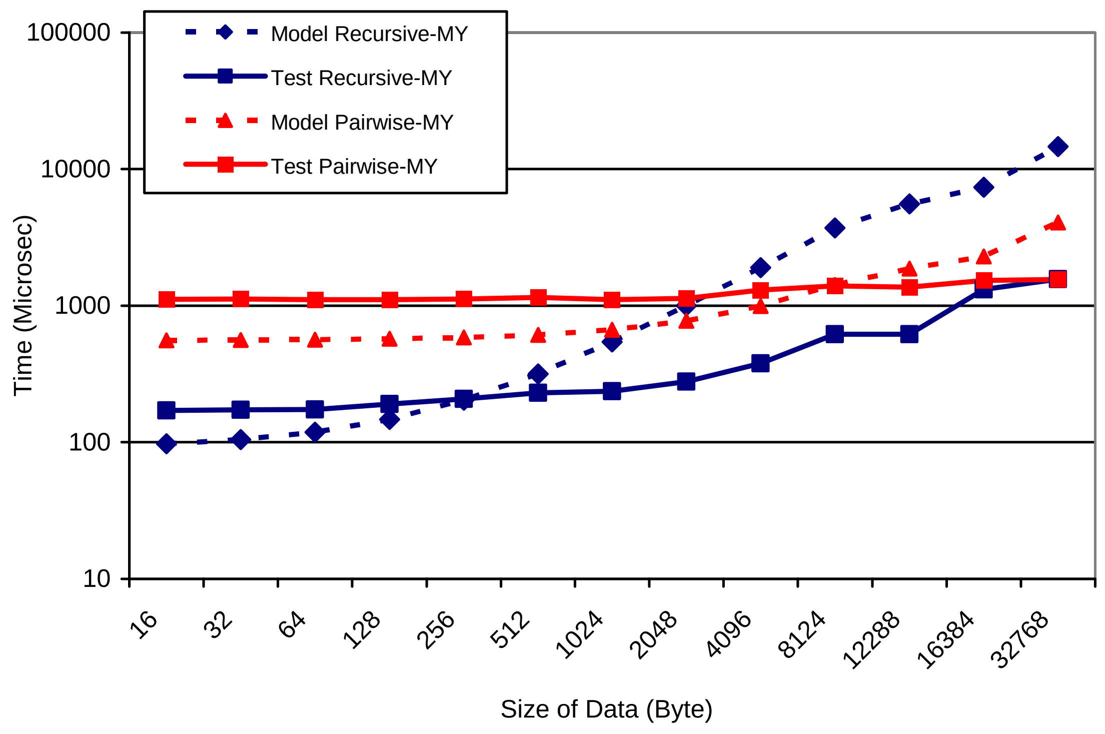

Figure 6.23 : Comparison between test results and model for 2ppn for 32 CPUs on Myrinet.

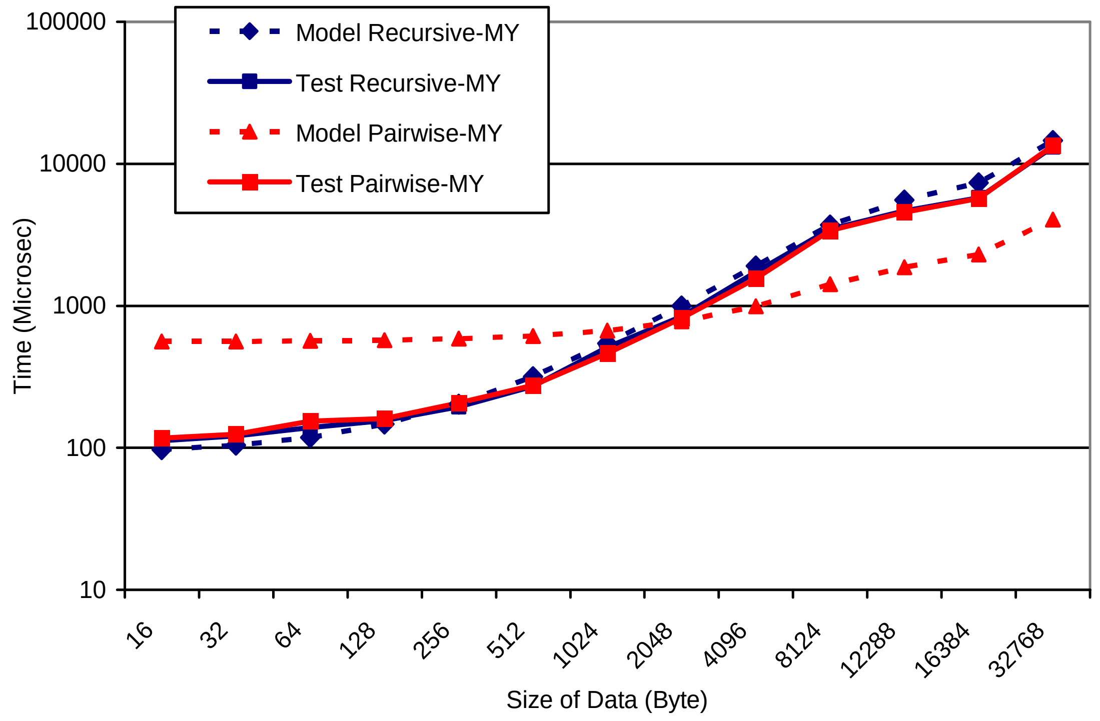

Figure 6.24 : Comparison between test results and model for 1ppn for 32 CPUs on Myrinet.

6.7 Allgather

In allgather, for short messages and non-power-of-two number of processes MPICH uses the algorithm by Bruck et al. [142] which is a variant of the distribution algorithm for barrier, and takes ceiling(lg p) steps and time $T_{distribution}$= lgp.α + n.((p-1)/p).β. For short or medium-size messages and power-of-two number of processes MPICH uses the recursive doubling algorithm and the time taken is the same as the Bruck algorithm which is $T_{distribution}$= lgp.α + n.((p-1)/p).β. For long messages or medium-size messages and non-power-of-two number of processes a ring algorithm is used. At the first step, each process i sends its contribution to process i+1 and receives the contribution from process i-1. From the second step onwards, each process i forwards to process i+1 the data it received from process i-1 in the previous step. This takes a total of p-1 steps and the time taken is T_{ring}= (p-1).α + n.((p-1)/p).β. This algorithm is used instead of recursive doubling for long messages because the nearest neighbor communication pattern used in ring algorithm performs twice as fast as recursive doubling for long messages particularly on Myrinet and IBM SP[11]. The change over point is occurring based on the equation *comm_size*type_size<MPIR_ALLGATHER_LONG_MSG* in the MPICH code, where *comm_size* is the number of processors used and *type_size* is the message size to be sent. So if the change-over point for long message size is set at 512 KByte and number of CPUs used is 8, the algorithm changes will occur at 64 KByte (8 x 64 KByte = 512 KByte) and for 32 CPUs the occurrence will be at 16KByte.

0	*comm_size*type_size* < 80KB	*comm_size*type_size* < 512KB
- For POF2 uses recursive doubling algorithm. - For Non POF2 uses variant of the disemmination algorithm for barrier. It takes ceiling(lg p) steps.	- For POF2 uses recursive doubling algorithm. - For Non POF2 uses ring algorithm.	- Ring Algorithm for POF2 and Non POF 2.

Table 6.10 : Summary of Algorithms used for Allgather in MPICH.

The average time for the default settings, recursive doubling and ring algorithm is shown in Figure 6.25 to Figure 6.28 on Myrinet and Ethernet network for 8 and 32 CPUs.

The figures show that the change over point should occur earlier since the ring algorithm is performing well compared to recursive doubling for smaller message sizes. Both 8 and 32 CPUs show a similar change-over point approximately at 1 KByte, however due to the above equation the change-over point should be different between 8 and 32 CPUs. Based from the equations the long message sizes should be set at 128 Byte for 8 CPUs and 32 Byte for 32 CPUs. It is also noticeable that on Ethernet the improvement is much bigger compared to Myrinet network, mainly for larger number of CPUs. This occurrence has a similarity with broadcast, where the ring algorithm performs better than recursive doubling. The improvement from using a different change-over point to the MPICH default can be quite significant, up to approximately 40% to 50% for Myrinet and around 50% or more for Ethernet, particularly for larger number of CPUs.

For non-power-of-two number of CPUs, the results show that even for small message sizes, the ring algorithm is better than using the variant of the dissemination algorithm for barrier by Bruck et al.[142]. However the improvements are fairly small, around 10% to 20%. Note that, similarly to broadcast, for one process per node there was not much difference between the different algorithms, and the default cross-over points work well.

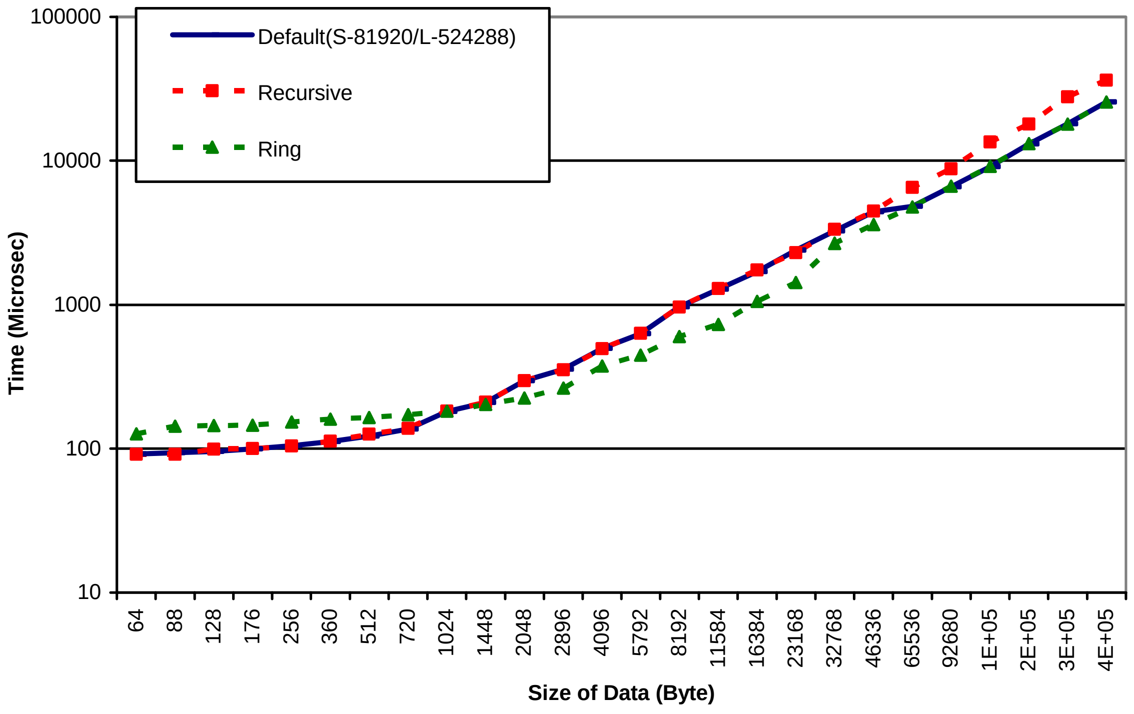

Figure 6.25 : 8 CPU and 2ppn on Myrinet for Allgather.

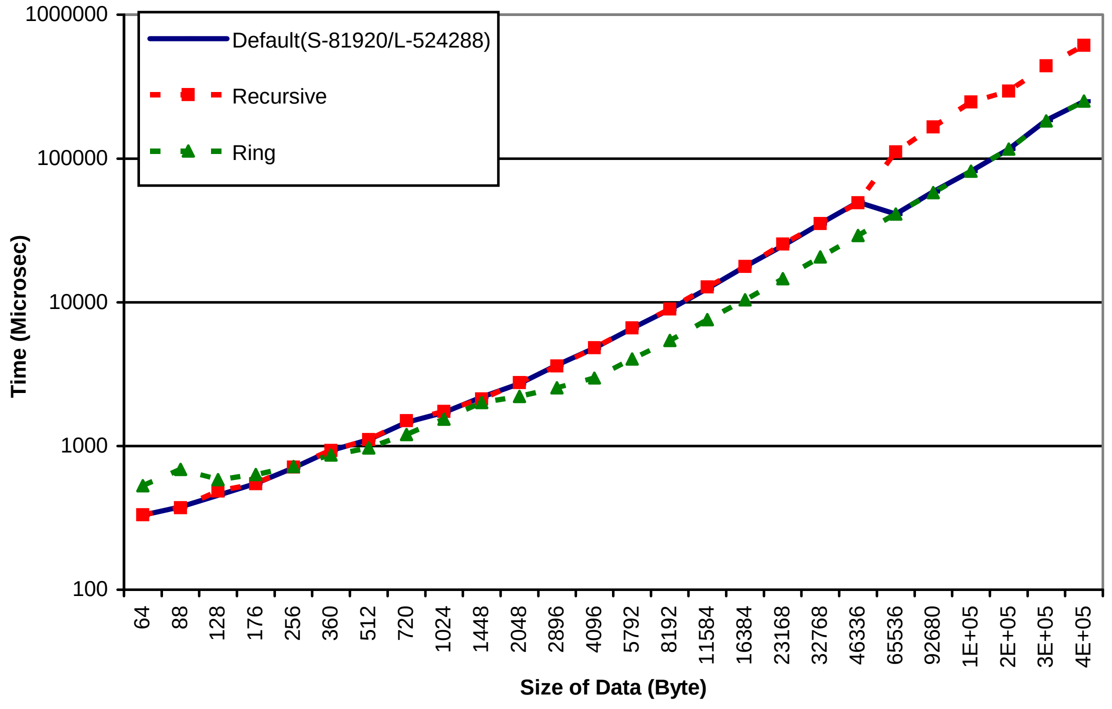

Figure 6.26 : 8 CPU and 2ppn on Ethernet for Allgather

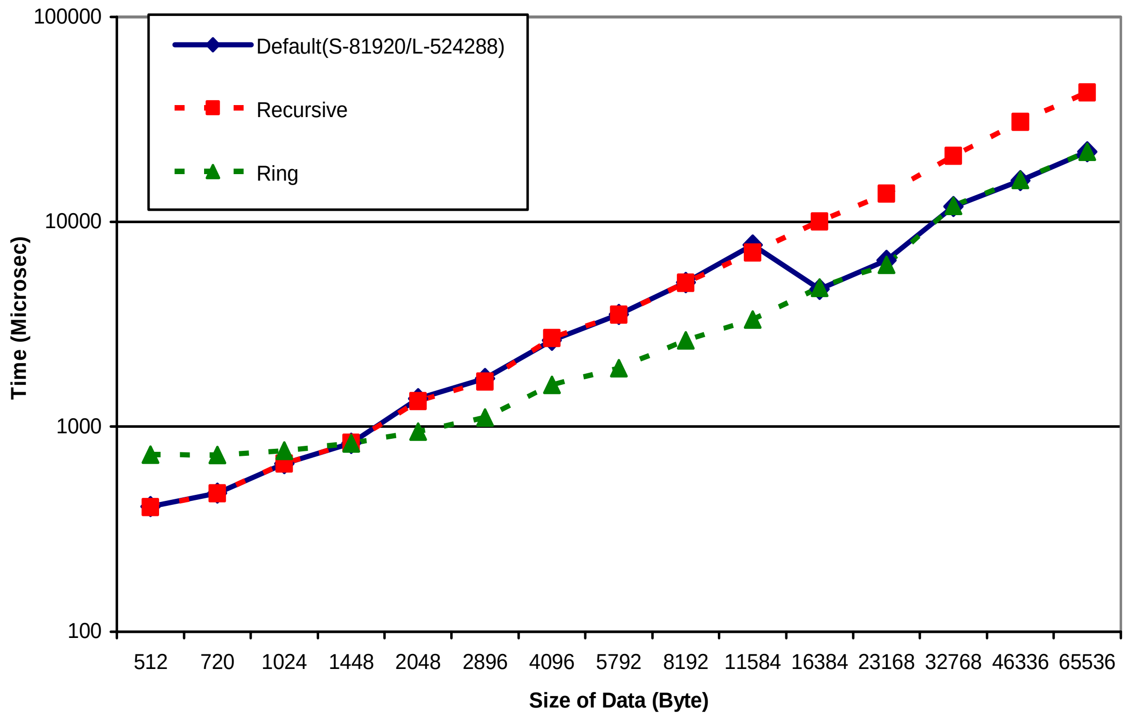

Figure 6.27 : 32 CPU and 2 ppn on Myrinet for Allgather

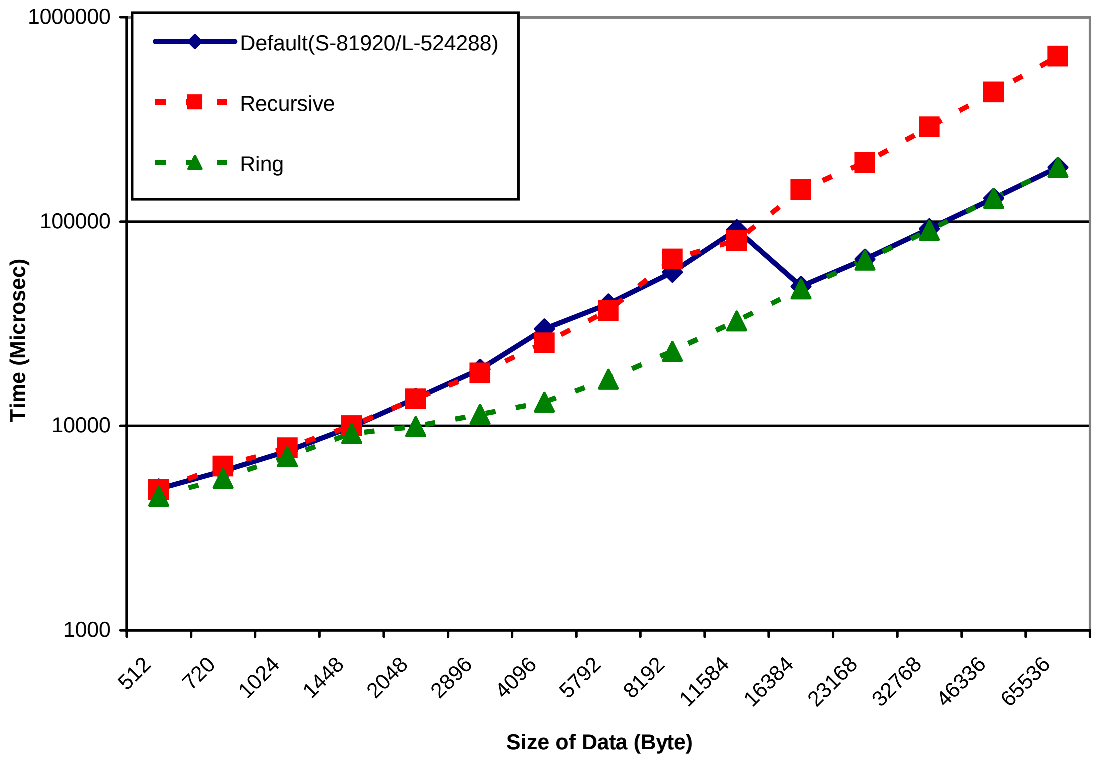

Figure 6.28 : 32 CPU and 2 ppn on Ethernet for Allgather

6.8 Other Collective Communication

This section describes other collective communications that have changes in the new MPICH implementation but have very small percentage differences between the different algorithms. The small difference is also observed for one processor per node and non-power-of-two number of CPU. The MPI collective communications discussed in this section are allreduce and reduce. All of these routines only have two different algorithms and the change between them all occurs at the same message size, which is 2 KByte. The performances for all of these routines only have about 2% to 3% differences between different algorithms from modifying the change-over point.

6.8.1 Allreduce

In Allreduce the algorithm is divided into two major components, which are for predefined (built-in) reduction operation and user-defined reduction operation. For Allreduce the recursive doubling algorithm is used for short and long message sizes with user-defined reduction operation. The same algorithm is used for short message sizes for built-in reduction operation, while for long messages sizes, Rabenseifner's algorithm [138] is used. This algorithm implements the allreduce in two steps, firstly a reduce-scatter, followed by an allgather. A recursive-halving algorithm (beginning with processes that are distance 1 apart) is used for the reduce-scatter, and a recursive doubling algorithm is used for the allgather.

The non-power-of-two case is handled by dropping to the nearest lower power-of-two: the first few even-numbered processes send their data to their right neighbors (rank+1), and the reduce-scatter and allgather happen among the remaining power-of-two processes. At the end, the first few even-numbered processes get the result from their right neighbors.

In SKaMPI, the Allreduce is measured using a built-in operation, which is the MPI_SUM operation. So the effect of different algorithms is between recursive doubling and reduce-scatter followed by allgather.

0	2 KB
- For built-in ops uses recursive doubling algorithm. - For user-defined ops uses recursive doubling algorithms.	- For built-in ops uses reduce-scatter followed by an allgather. (A recursive-halving algorithm for the reduce-scatter, and a recursive doubling algorithm for the allgather.) - For user-defined ops uses recursive doubling algorithms.

Table 6.11 : Summary of Algorithm uses in Allreduce.

6.8.2 Reduce

Similarly with allreduce, in reduce for long messages and for built-in operations, Rabenseifner's algorithm [138] is used, while for short message sizes the binomial tree algorithm is used. The Rabenseifner's algorithm implements the reduce in two steps, firstly a reduce-scatter, followed by a gather to the root. A recursive-halving algorithm (beginning with processes that are distance 1 apart) is used for the reduce-scatter, and a binomial tree algorithm is used for the gather. The non-power-of-two case is handled by dropping to the nearest lower power-of-two, the first few odd-numbered processes send their data to their left neighbors rank-1, and the reduce-scatter happens among the remaining power-of-two processes. If the root is one of the excluded processes, then after the reduce-scatter, rank 0 sends its result to the root and exits; the root now acts as rank 0 in the binomial tree algorithm for gather.

For short messages and long message sizes, the binomial tree algorithm is used for user-defined operations. Similarly with Allreduce, the reduce test in SKaMPI uses the built-in MPI_SUM operation. So the effect of different algorithms is between binomial tree algorithms and reduce-scatter followed by allgather. The small difference in performance in using different change-over-points is suspected to be due to the use of binomial tree algorithms, which is also used for long message sizes under the allgather algorithm.

0	2 KB
- For builtin ops uses binomial tree algorithms. - For user-defined ops uses binomial tree algorithms. .	- For builtin ops uses reduce-scatter followed by gather. (A recursive-halving algorithm for the reduce-scatter, and a binomial tree algorithm for the gather.) - For user-defined ops uses binomial tree algorithms.

Table 6.12 : Summary of Algorithm uses in Reduce

6.9 Summary

MPICH provides a mechanism for selecting between different algorithms for a particular collective communication routine based on whether the message size is greater than or less than a specified change-over point. In current versions of MPICH this value is hard-coded, based on experimental results and theoretical models that assume a single process per node.

This study has demonstrated that it is straightforward to modify the MPICH code to allow the change-over points between different algorithms to be customized. This enables the change-over values to be set so that MPI benchmarks can be run to measure the performance of each different algorithm over any range of message sizes, and to therefore be able to find the optimal change-over points for any parallel computer. These customized change-over points can then be set in a configuration file and used by MPICH, in order to optimize the performance of collective communications for a particular parallel computer.

This study has shown that the values of the optimal change-over points can vary significantly for different networks and different numbers of CPUs per node, and that using these customized change-over points can provide significant performance improvements for collective communications routines in the range of message sizes between the default MPICH change-over point and the optimum change-over point for the particular machine. All of the collective communications routines for which MPICH implements multiple algorithms showed improvements of over 50% for some message sizes, and in some cases improvements of a factor of 2 or more.

One of the main factors in determining the change-over point was the number of processes per node. With the advent of multi-core processors, all modern clusters will have more than one CPU per node, so it will be useful to be able customize collective communications rather than use the default change-over points that are based on measurements for a single process per node.

Additionally, it is noticeable that in most cases the change over point on Ethernet occurs at lower message sizes than Myrinet, this is probably due to the effect of the higher latency and lower bandwidth. Figure 6.29 shows the expected performance for different algorithms on broadcast with 32 CPUs for Gigabit Ethernet which has similar latency with 100 Mbit/s Ethernet (90 microsec) and similar bandwidth with Myrinet (180Byte/microsec). The performance is calculated based on the broadcast formula, so the results should be assumed to only hold for one processor per node. The results indicate that the change over-value will occur higher than the fixed value used by MPICH, with the cross-over value for small and medium message size increasing from 12 KByte to 32 KByte and from 512 KByte to 4 MByte approximately. So, it shows that if the bandwidth is increased by factor of 10, with the latency remaining constant, then the cross-over point will be increased too in this case.

In conclusion, this study provides information on better change over points for two processors per node on Myrinet with GM and Ethernet with TCP on IBM eServer 1350 Linux cluster. This study also provides comparison results between MPICH2 1.0.4 and MPICH 1.2.6 and it shows that MPICH2 has improvement for certain algorithms. Besides, this study also shows that with better change over point and MPICH2, Ethernet can improve the performance until 50% in some cases. Finally, it is possible to get significant performance improvement by allowing tuning of the change over point between difference collective communication algorithms based on measurements from MPI benchmarks, rather than have then set to fixed values

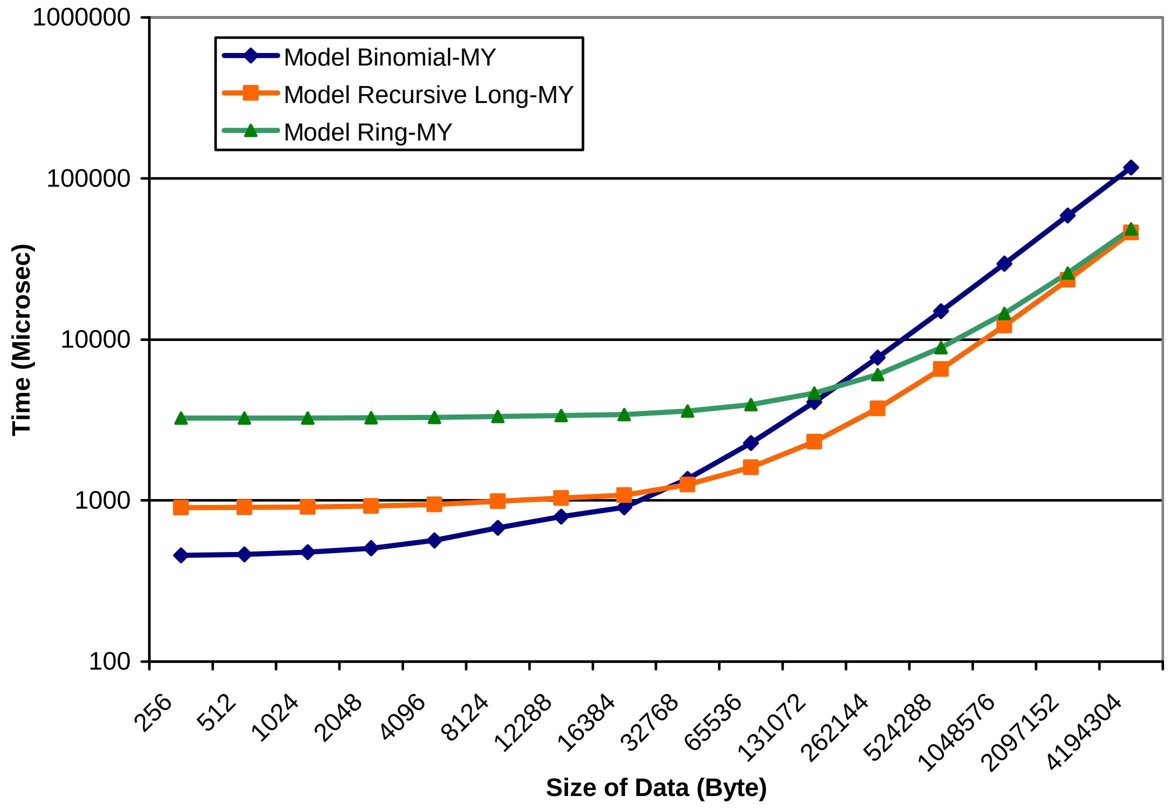

Figure 6.29 : Expected performance for 32 CPU and 1ppn for Gigabit Ethernet

6.10 Future Work

In future a fully automated mechanism for configuring the change-over points for each collective communication to maximize the performance will be developed. The idea is run an MPI benchmark for each algorithm used in each collective communication routine using the smallest and largest message sizes possible. Then the results will be processed to compute the optimal change-over points, which will be written to a configuration file for use by MPICH.

With the move to multi-core CPUs, new clusters are likely to have many cores per node. We plan to repeat these measurements on a new cluster with dual quad-core processors (i.e. 8 CPUs) per node, to see if there is an even greater variation from the default values in MPICH which are based on measurements with 1 CPU per node, and also to obtain results for Infiniband and Gigabit Ethernet networks. It would also be interesting to do a detailed performance comparison between MPICH2 and MPICH1 on the same network, e.g. for Myrinet with GM and Ethernet with TCP, particularly to analyse the performance for ring algorithm, since it is suspected to have improvement in MPICH2 by using the new *mpd* facilities. Finally, more dynamic change-over points used in the equation for Allgather algorithm need to be revised and suggested.

CHAPTER 7

Performance Evaluation on ccNUMA Shared Memory Machine SGI Altix 3000

7.1 Introduction

The SGI Altix [70,27] is a cache coherent, non-uniform memory architecture (ccNUMA) shared memory multiprocessor system that is a popular machine for high-performance computing, with several large systems now installed, including the 10,160 processor Columbia machine at NASA. In Australia, a 1680 processor Altix (the APAC AC) has recently replaced an ageing AlphaServer SC with a Quadrics network (the APAC SC) as the new peak national facility of the Australian Partnership for Advanced Computing (APAC) [84], and was number 26 in the June 2005 list of the Top 500 super-computers [91]. There are several other Altix machines at APAC partner sites, including two systems with 160 processors and another with 208 processors.

Most parallel programs used for scientific applications on high-performance computers are written using the Message Passing Interface (MPI), so the performance of MPI message passing routines on a parallel supercomputer is very important. Shared memory machines such as the Altix typically have very high-speed data transfer between processors, however this will only translate into good MPI performance if the MPI library can efficiently translate the distributed memory, message-passing model of MPI onto shared memory hardware. It is therefore of interest to measure the performance of MPI routines on a shared memory machine such as the SGI Altix, and to compare it with a distributed memory supercomputer with a high-end communications network. This section will provide results for MPI performance on the SGI Altix, and comparisons with similar measurements on the AlphaServer SC [72] with a Quadrics network [20].

This work was done in early 2005, and was of particular interest to users of the APAC National Facility, due to the change from the Alphaserver SC to the SGI Altix as the peak Australian supercomputing facility. Note that, Quadrics had the best MPI performance of any network used in large clusters at that time. For example Grove [8] did a comparison between Fast Ethernet, Myrinet and Quadrics network, which showed that for point-to-point communication using 32 nodes and 2 processors per node for 64 KByte messages, Quadrics obtained 0.5 ms, while Myrinet was approximately 1.5 ms.

However, at the current time Quadrics is no longer the best performance network. This is because the recent commodity interconnect known as Infiniband now offers better performance. Liu et al. [130] compared the performance between Infiniband, Myrinet and Quadrics network and found that for 8 node clusters Infiniband can provide significant performance improvements for applications compared with Myrinet and Quadrics.

A number of benchmark programs have been developed to measure the performance of MPI on parallel computers, including Pallas MPI Benchmark (PMB) [65], SKaMPI [21,62], MPBench [19,64], Mpptest [17,63], and the most recently developed, MPIBench [1,2,8]. The measurements reported here used MPIBench, which is the only MPI benchmark that takes into account the effects of contention in point-to-point communications, and can also generate distributions of communication times, not just averages. These are the first results of using MPIBench on a large shared memory machine.

7.2 MPI Benchmark Experiments on the Altix

The benchmark results reported in this chapter were carried out on Aquila, an SGI Altix 3000 managed by the South Australian Partnership for Advanced Computing (SAPAC) [131]. Details about Aquila, the Altix architecture and benchmark methodology are given in section 3.6.1.

7.3 Selection of Processors for Benchmarking

The Altix documentation suggests that applications should avoid using processor 0, particularly for parallel jobs, since it is used to run system processes. Preliminary test runs using processor 0 showed that communication involving this processor was indeed slower than for other CPUs, although the effect is fairly small, just a few percent. Figure 7.1 shows two peaks for 8 processors using processor number from 0 to 7, with the peak at the larger time corresponding to processor 0, but for processor number from 16 to 23 there is only one peak.

Therefore the benchmark runs reported here have avoided using processor 0, with all measurements being done using processors 32 to 159. The benchmarks started with processor number 32 in order to maintain the hierarchical pattern of 32 processor groups shown in Figure 3.6.

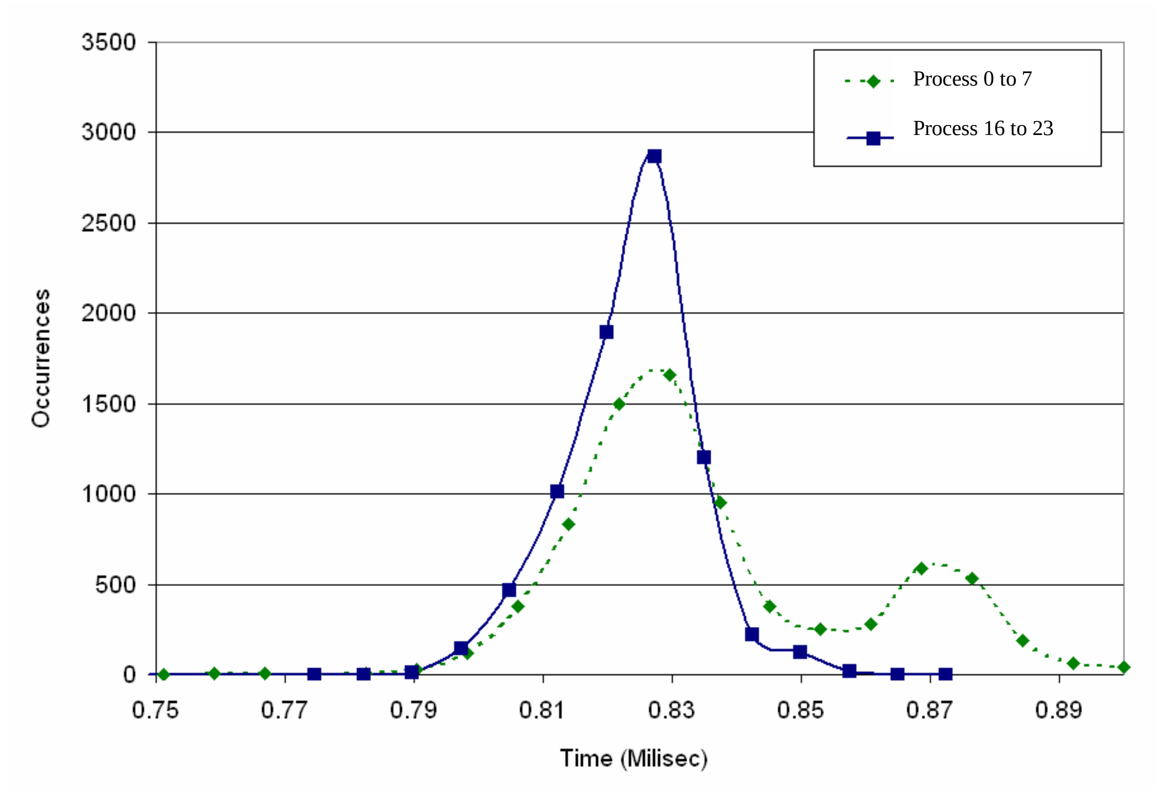

Figure 7.1 : 8 CPUs for Point-to-Point at 256 KBytes using processor number from 0 to 7 and 16 to 23

Another issue to investigate is the difference in communication time between 32 processors that have a direct link to each other or otherwise. Figure 3.6 shows that each group of C-Bricks is linked by R-Bricks (router brick) and there is a direct link between groups of two R-Bricks. For example, R1A is connected with a direct link to R2A, however R2A has to communicate with R3A using meta-router MR1A. So, it is interesting to know the difference in performance between communications that go directly between routers such as R1A and R2A or those that need to use an intermediate router such as R2A and R3A.

Figure 7.2 shows the different results for communications within groups of 32 processors for 256KByte. The communications for 0 to 31, 32 to 63 and 64 to 95 are the communications that have direct connection between the R-Bricks or router brick, while 16 to 47 and 48 to 79 need to use the intermediate router for communications. Noticeably, for 0 to 31 there are two peaks and the peak at the shortest time is from process 4,5,6,7 and their partners 20,21,22,23 (refer to section 3.7 on MPIBench point-to-point pattern). The direct link between R1A and R2A is probably the cause of the faster time for those processes. Unfortunately, we do not have more precise explanation on the reason for the faster time that obtain by those processors. The main peak still overlaps with the results from the communication between processors 32 to 63 and 64 to 95, which also have direct links between the R-Brick for a processor and the R-Brick for its communication partner. However, if we look closely at the communication between processors 16 to 47 and processors 48 to 79 there is about 25 μs difference with the other group of communication. This shows that the communication is faster when using the direct link between two R-Bricks. However, the overhead of having to go through an intermediate router is very small, around 2.5%. This means that users do not need to understand the communication architecture of the parallel computer, or be concerned about the placement of MPI processes to particular processors, in order to get good MPI performance.

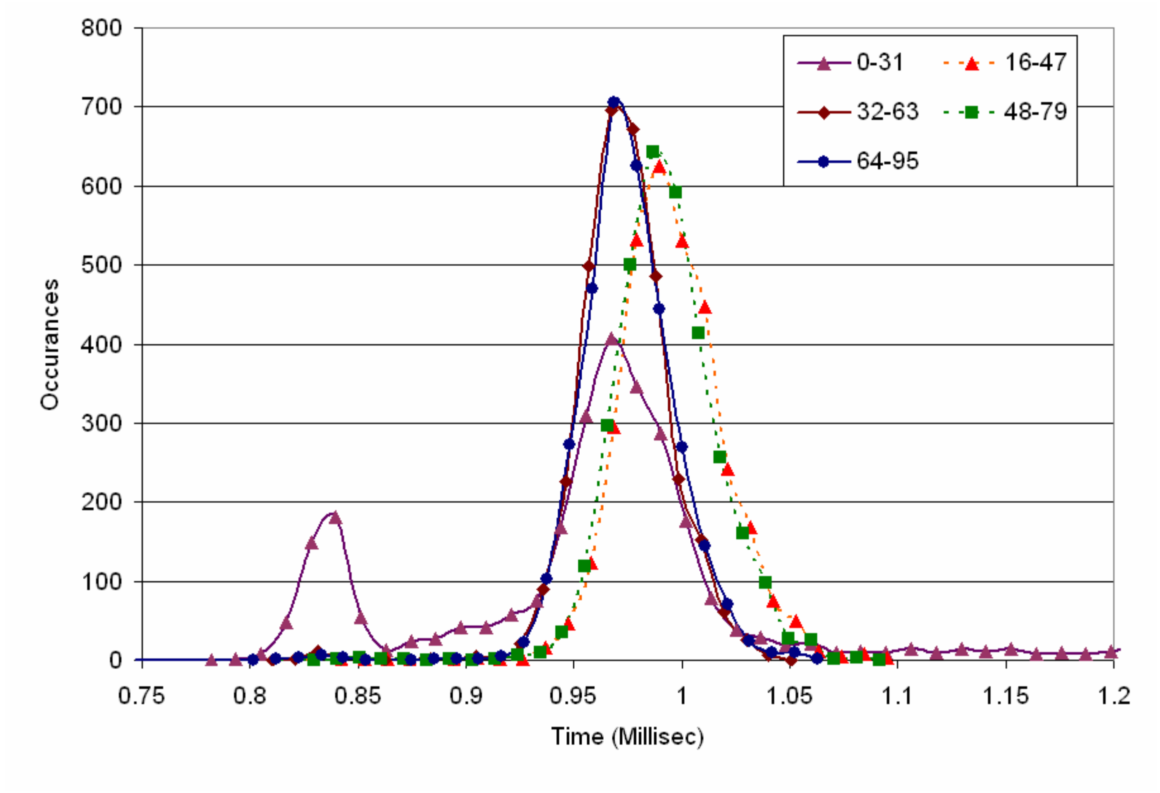

Figure 7.2 : Communication for 32 processor for different group of processor for 256KBytes.

7.4 MPI_Send with Default Settings and Single Copy

The issue of buffered (default) and non-buffered (Single Copy) options in the SGI MPI library has been discussed in section 3.6.1. This section gives a more detailed comparison of Point-to-Point communication results between default setting and Single Copy. Figure 7.3 shows the average time for MPI_Send using default settings and Single Copy. It shows that the communication time using Single Copy is decreased by more than a factor of ten for large numbers of CPUs. The performance of Single Copy does not depend as much on the network architecture of the SGI Altix as with the default settings. However, for message size larger than 512 KByte the communication starts to show the hierarchical pattern of 32 processors groups shown in Figure 3.6. Figure 7.4 shows the bandwidth comparison for the same pattern and the bandwidth starts to show the hierarchical pattern of 32 processors groups for message sizes larger than 512 Kbytes too.

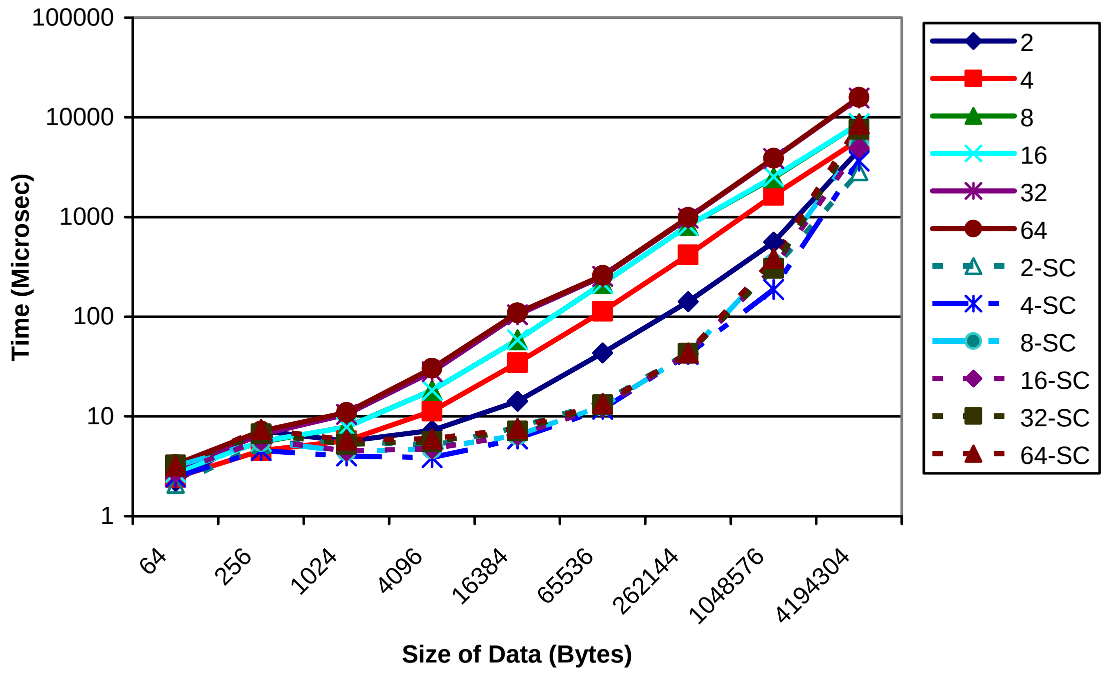

Figure 7.3 : Average time for point-to-point using the default setting and single copy.

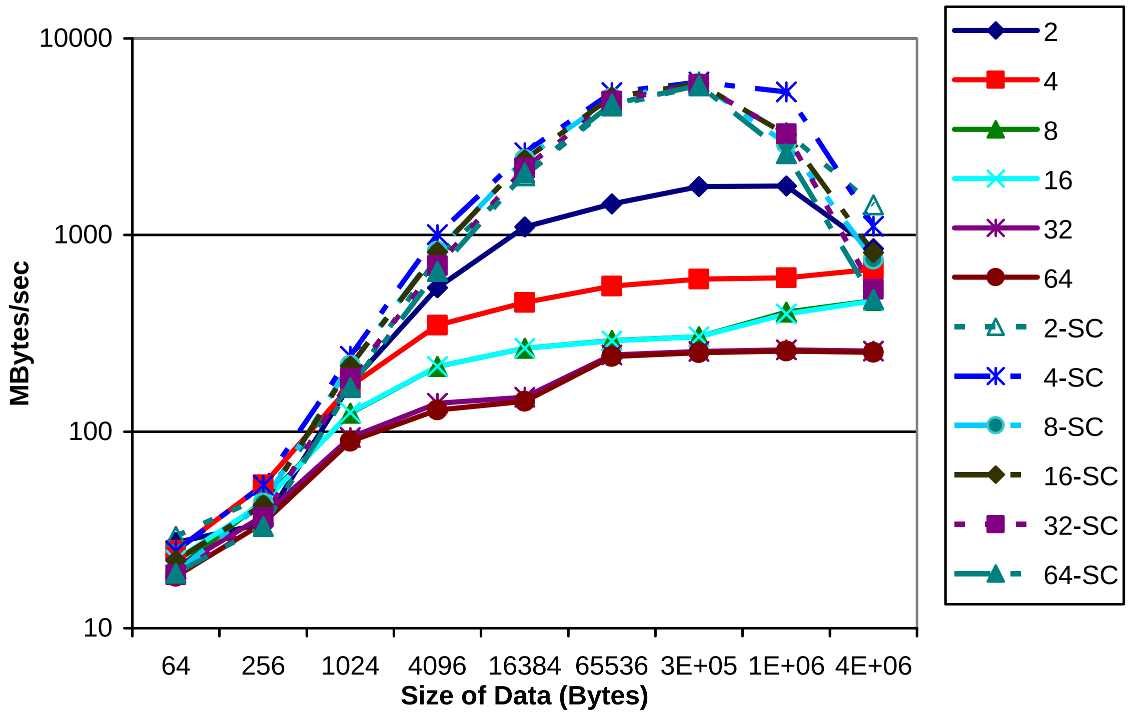

Figure 7.4 : Bandwidth for point-to-point using the default setting and single copy.

7.5 Point-to-Point Communications

MPIBench uses MPI_Isend and MPI_Recv to measure point-to-point communication, so it uses non-blocking sends and blocking receives. For the comparison between the other MPI benchmark applications, the MPI_Isend was changed to MPI_Send in order to standardize the comparison methodology, however the results were essentially unchanged. This section concentrates on analysis of the point-to-point communications performance of the SGI Altix 3700 based on measurements using MPIBench. Firstly, the performance for different numbers of processors is analysed, to determine the different communication times due to the memory hierarchy of the Altix ccNUMA architecture.

Number of Processors	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Latency (MPIBench)	1.96 us	1.76 us	2.14 us	2.21 us	2.56 us	2.61 us	2.70 us
Latency (MPBench)	**1.76 us**	**2.07 us**	**2.48 us**	**2.41 us**	**2.53 us**	**3.06 us**	**3.01 us**
Bandwidth (MPIBench)	851 MByte/s	671 MByte/s	464 MByte/s	462 MByte/s	256 MByte/s	256 MByte/s	248 MByte/s
Bandwidth (MPBench)	**831 MByte/s**	**925 MByte/s**	**562 MByte/s**	**562 MByte/s**	**549 MByte/s**	**532 MByte/s**	**531 MByte/s**

Table 7.1 : Measured latency (for sending a zero byte message) and bandwidth (for a 4 MByte message) for different numbers of processes on the Altix. Results for MPIBench are for all processes communicating concurrently, so include contention effects. Results for MPBench (in bold font) are for only two communicating processes (processes 0 and N-1) with no network or memory contention.

Table 7.1 shows latency and bandwidth data for the Altix, obtained by running the MPI benchmarks on different numbers of processors, which gives an indication of the performance of the different levels of memory hierarchy in the Altix. The results from MPBench give the best possible results, where only two processors are communicating with no contention. The results from MPIBench show the more realistic case where all processors are communicating at the same time, and therefore show the effects of contention in the communications network. The results within a C-Brick (2 and 4 processors) show very good performance, although for 2 processors the bandwidth for smaller messages (around 512 KB) is about twice as large, which is surprising. The results between C-Bricks (more than 4 processors) show remarkably little degradation in performance as the number of processors is increased, indicating that the routers are very fast. Note that the bandwidth measurements are for buffered MPI_Send, which is the default for SGI MPI. Using a single copy send gives significantly higher bandwidth, as shown in section 7.4, giving results that are much closer to the theoretical NUMAlink network speed of 3.2 Gbytes/sec.

In comparison, measurements with MPIBench on the AlphaServer SC with Quadrics network [18] gave a latency of around 5 microseconds for internode communication with a single process per node, however this increased to around 10 microsec when all 4 processors per node were communicating. The latency for shared memory communication within a node was also around 5 microsec. The bandwidth within a node was 740 MBytes/sec, while the bandwidth over the Quadrics network was 262 MBytes/sec. So in all cases, the performance of MPI point-to-point message passing performance of the Altix is significantly better than the AlphaServer SC.

Figure 7.5 shows the performance for point-to-point communications for small message sizes and Figure 7.6 shows the results for larger message sizes. The results for different numbers of processors in Figure 7.5 and Figure 7.6 clearly illustrate the non-uniform memory architecture of the SGI Altix. For 2 processors the time is for intranode communication, which is approximately 0.14 ms for a 256 KByte message. We are not sure what it causing the strange results for 2 processors in Figure 7.5. The result for 4 processors represents internode communication within a C-Brick, which takes approximately 0.42 ms for the same message size. The results for 8 processors and 16 processors are about the same, around 0.82 ms, since both communicate between C-Bricks and in the same R-Brick. Communication between 32 processors is done directly between R-Bricks, and takes around 0.95 ms. Results for 64, 96 and 128 processors all involve communication between R-Bricks through a meta-router, which is only marginally slower than direct communication between R-Bricks, taking approximately 1.0 ms for a 256 Kbyte message.

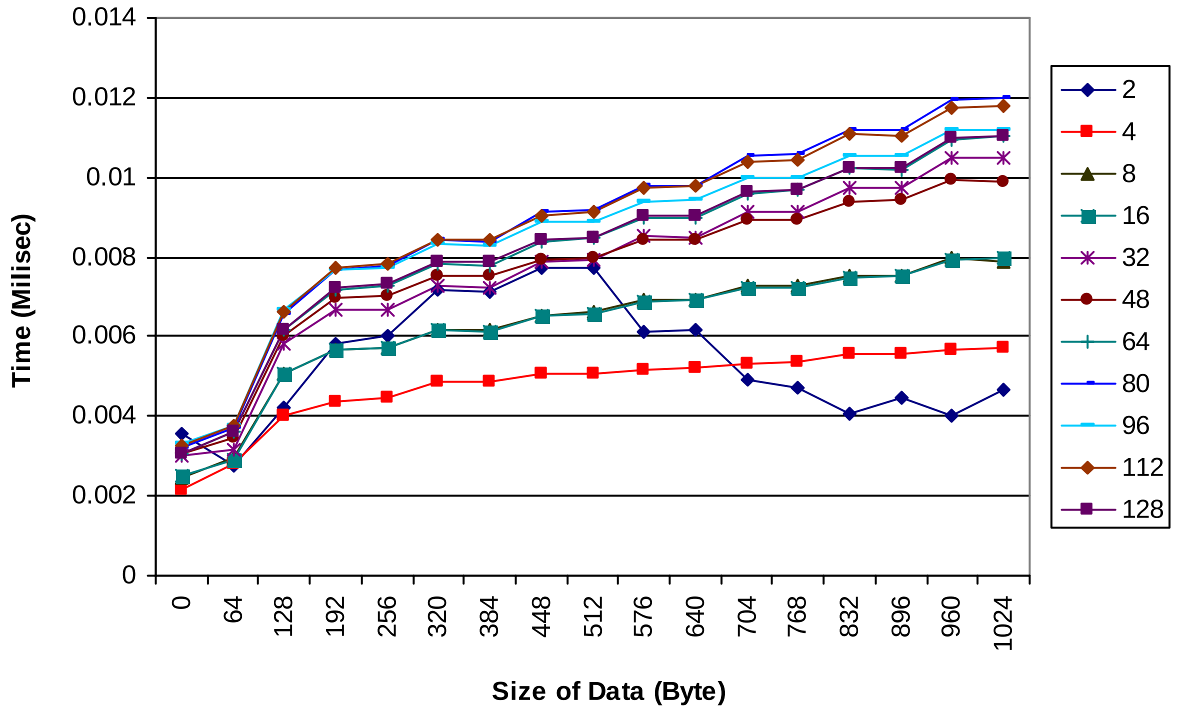

Figure 7.5 : Point-to-Point performance for small message sizes.

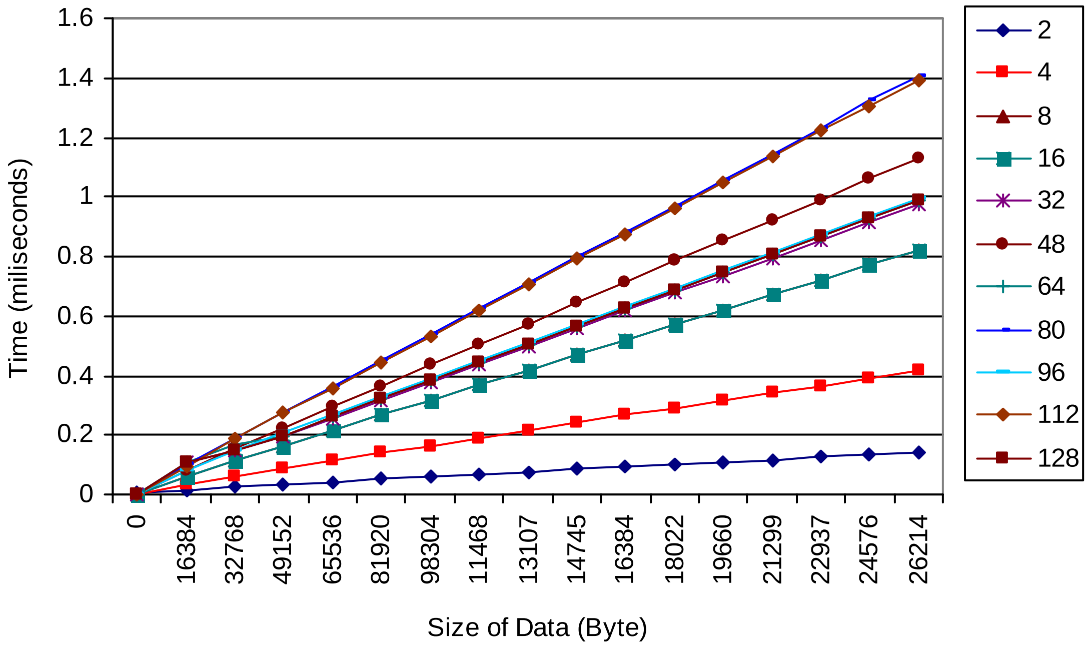

Figure 7.6 : Point-to-Point performance for large message sizes.

Figure 7.6 show that performance is essentially bandwidth limited for large message sizes. However it also displays a curious anomaly in the results for 48, 80 and 112 processors, which are all slower than for 128 processors. For 48 nodes the bandwidth is around 15% worse than might be expected. However this may be explained by a reduction in effective bandwidth due to the impost of additional router latency for some of the traffic, since the point-to-point test distributes the participating pairs evenly across the nodes. The division of 48 nodes across the Altix will place the nodes on three separate 16 processor sub-clusters, where each sub-cluster is connected by a single router. Two of the three routers will be interconnected directly by two of their NUMAlink ports, whilst the third router can only access the other two routers via two meta-routers, to which it only has a single connection each. To maintain bandwidth the third router is dependant upon the dual port connection to its peer router, and that router's additional connections to the meta-routers. Hence, half the traffic to the third sub-cluster must transit one additional router hop. This accounts for one third of the traffic in the test. If we examine the bandwidth difference between 16 and 32 nodes for this test we can see a similar issue. One half of the traffic in the 32 node test transits the meta-routers, and the bandwidth impost of the additional router step is roughly 64MB/s for the whole test, or 128MB/s for the half of the traffic affected. The reduction in performance seen in the 48 node test is consistent with this.

To explore this anomalous behaviour in more detail, Figure 7.7 shows the distribution of communication times for 48 and 64 processors, which shows a significant difference in performance. For 64 processors it shows the expected result of a single peak centred at the average communication time through the meta-router for this message size. However for 48 CPUs there are multiple peaks and a very long tail, which is often indicative of contention effects, something that is unexpected in the SGI Altix design. The cause of the anomalous behavior is suspected to be that the grouping of the 32 processors is not maintained, as discussed in the above section and also section 7.4, which might affect the buffering settings. The buffering is suspected to be affected because in Single Copy the anomalous behaviour is not observed as shown in Figure 7.8. In this figure both peaks overlap within each other.

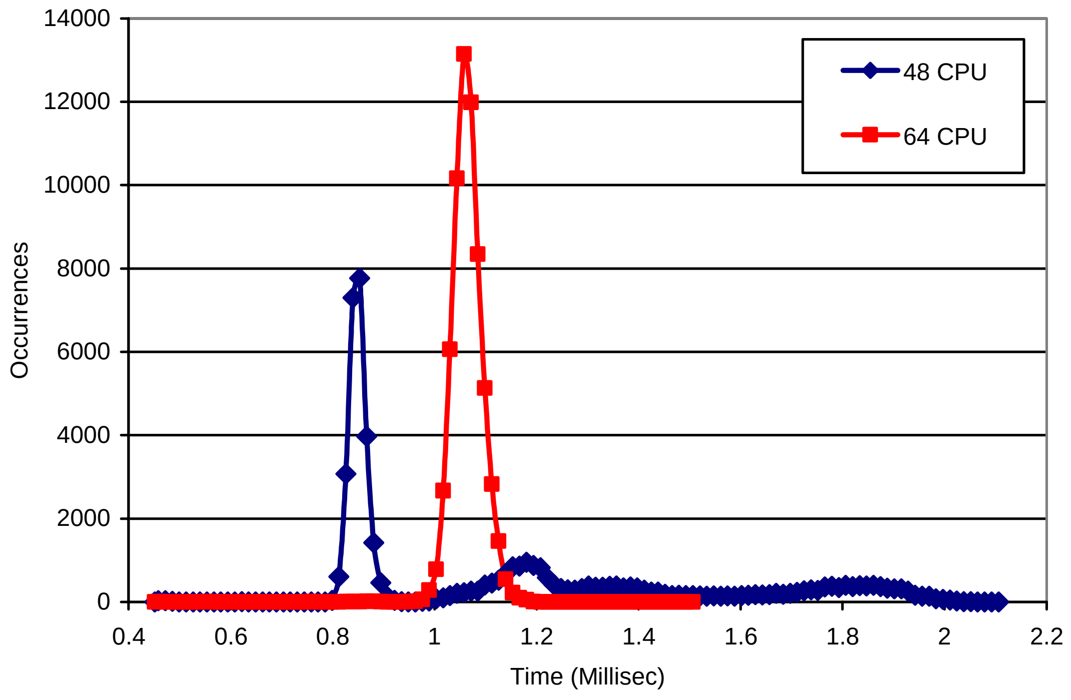

Figure 7.7 : Probability distributions for MPI point-to-point communications using 48 and 64 processors for 256 KByte message size.

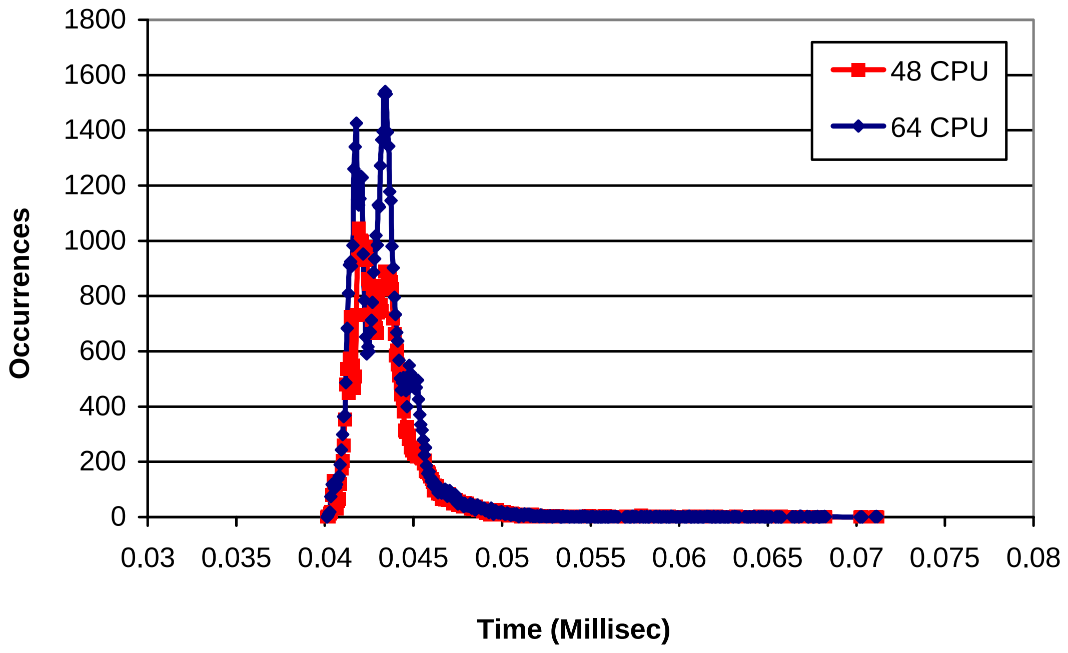

Figure 7.8 : Probability distributions for MPI point-to-point communications using 48 and 64 processors for 256 KByte message size using Single Copy options.

7.5.1 MPI_Sendrecv

As discussed in section 3.8, results for MPI_Sendrecv using the Point-to-Point communication pattern of MPIBench has proven that SGI Altix provides full bidirectional bandwidth, since all the results of MPI_Sendrecv are similar to the results of MPI_Send/Recv for Single Copy options. However, measurements by Grove [8] showed that the Quadrics network on the AlphaServer SC does not provide the expected bidirectional bandwidth facilities [8]. Petrini et al. [143] suggest that PCI bottlenecks and DMA contention between system memory and the network interface are the cause of the unexpectedly poor performance.

7.6 Broadcast

Figure 7.9 shows average times measured by MPIBench for MPI_Bcast for different data sizes for 2 up to 128 processors. Above 16 Kbytes (which is the page size on the Altix) the results increase almost linearly with the data size.

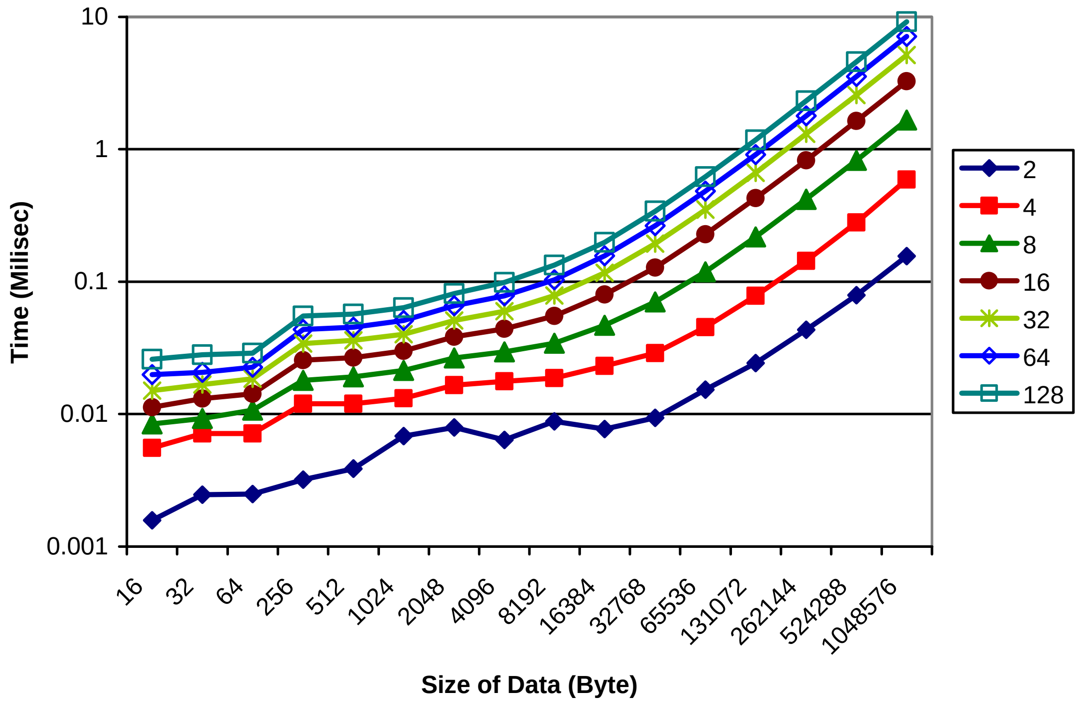

Figure 7.9 : Performance of MPI_Bcast as a function of data size on 2 to 128 CPUs.

The Quadrics network on the AlphaServer SC provides a very fast hardware broadcast, but only if the program is running on a contiguous set of processors. Otherwise, a standard software broadcast algorithm is used. A simple comparison of broadcast performance on the two machines is difficult, since for smaller numbers of processors (around 32 processor or less, but this depends somewhat on the message size) the Altix does better due to its higher bandwidth, whereas for larger numbers of processors the AlphaServer starts to do better since the hardware broadcast of the Quadrics network scales really well (much better than logarithmic) with the number of processors. For example, hardware-enabled broadcast of a 64 KByte message on the AlphaServer SC takes around 0.40 ms for 16 CPUs and 0.45 on 128 CPUs [18], while on the Altix is takes approximately 0.22 ms on 16 CPUs, 0.34 ms on 32 CPUs, 0.45 ms on 64 CPUs, and 0.62 ms for 128 CPUs. If the processors for an MPI job on the AlphaServer SC are not contiguous, which will often be the case on a shared machine running many jobs, the software broadcast is a few times slower than the hardware-enabled broadcast and doesn't scale as well, so broadcast on the Altix will always beat it.

Figure 7.11 show the distribution results for MPI_Bcast on 32 CPUs for smaller and larger messages sizes, respectively. Analysing this data is more difficult than for a cluster due to the non-uniform memory hierarchy on the Altix and since there is no documentation on what broadcast algorithms the SGI MPI libraries are using. However, MPIBench allows distributions to be generated individually for each processor, so we are able to check that the overall distribution shown in Figure 7.11 shows peaks that are consistent with a binary tree broadcast algorithm, with the first peak corresponding to completion time for processors 0 and 1, the second peak is for 2 and 3, the third peak around 0.65 ms is for 4,5,6,7, the next group between 0.8 and 1.0 ms is for 8-15, and the final clump is for 16-31.

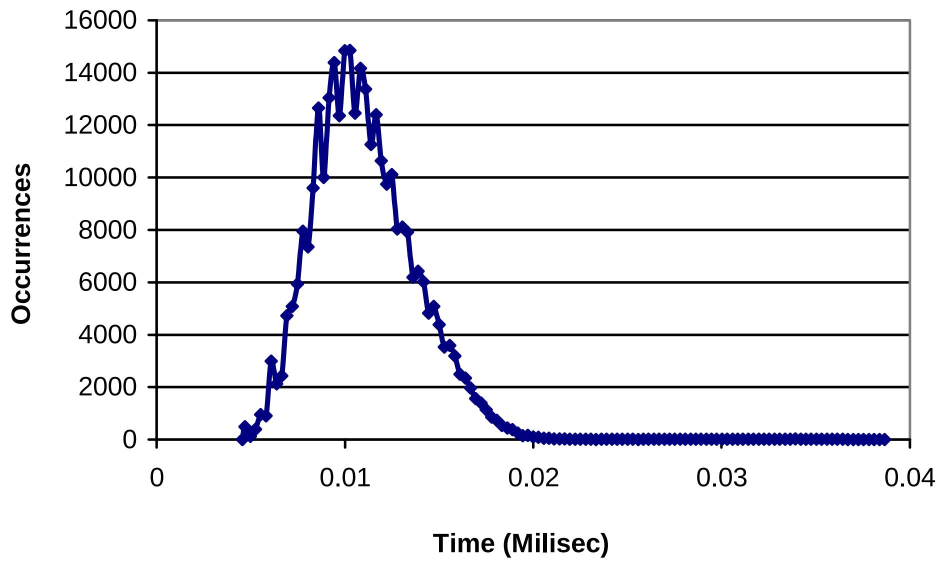

Figure 7.10 : Distribution results for MPI_Bcast at 64 Bytes on 32 cpus.

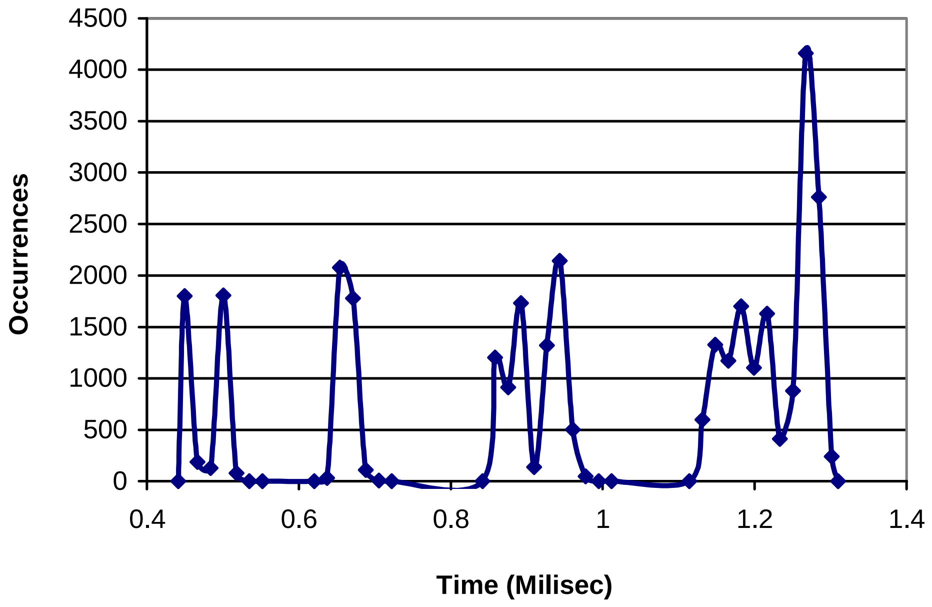

Figure 7.11 : Distribution result for MPI_Bcast at 256Kbytes on 32 cpus.

7.7 Barrier

Results for the MPI_Barrier operation for 2 to 128 processors are shown in Figure 7.12. As expected, the times scale logarithmically with the numbers of processors. The hardware broadcast on the Quadrics network means that a barrier operation on the AlphaServer SC is very fast and takes almost constant time of around 5-8 microseconds for 2 to 128 processors, which is similar to the Altix.

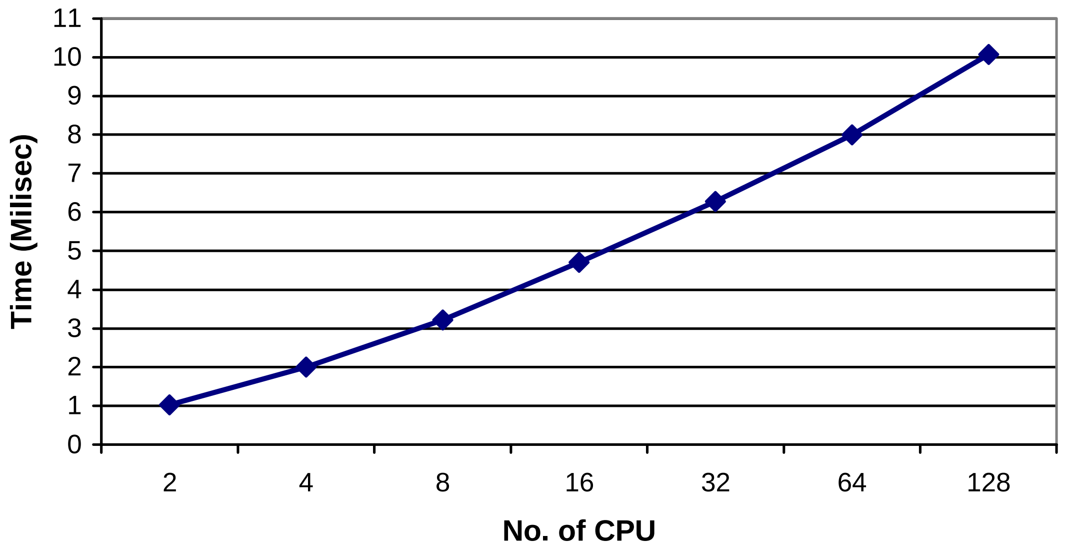

Figure 7.12 : Average time for an MPI barrier operation for 2 to 128 processors.

7.8 Scatter and Gather

Scatter and gather are typically used to distribute data at the root process (e.g. a large array) evenly among the processors for parallel computation, and then recombine the data from each processor back into a single large data set on the root process. The performance of MPI_Scatter is dependent on how fast the root process can send all the data, since it is a bottleneck. However the root process can use asynchronous sends, which means that the overall performance of the scatter operation is also dependent on the overall communications performance of the system and the effects of contention. Figure 7.13 shows the average communication time for an MPI_Scatter operation for different data size per processor on different numbers of processors. The results show an unexpected hump at a data sizes between 128 bytes and 2 KBytes per process, so that the time for scattering larger data sizes than this is actually lower. This is presumably due to the use of buffering for asynchronous sends for messages of these sizes. Note that overall, the time for an MPI_Scatter operation grows remarkably slowly with data size. In the worst case, at 1 Kbyte per process, the Altix is around 4 to 6 times faster than the APAC SC, while at 4 Kbytes per process it is around 10 times faster.

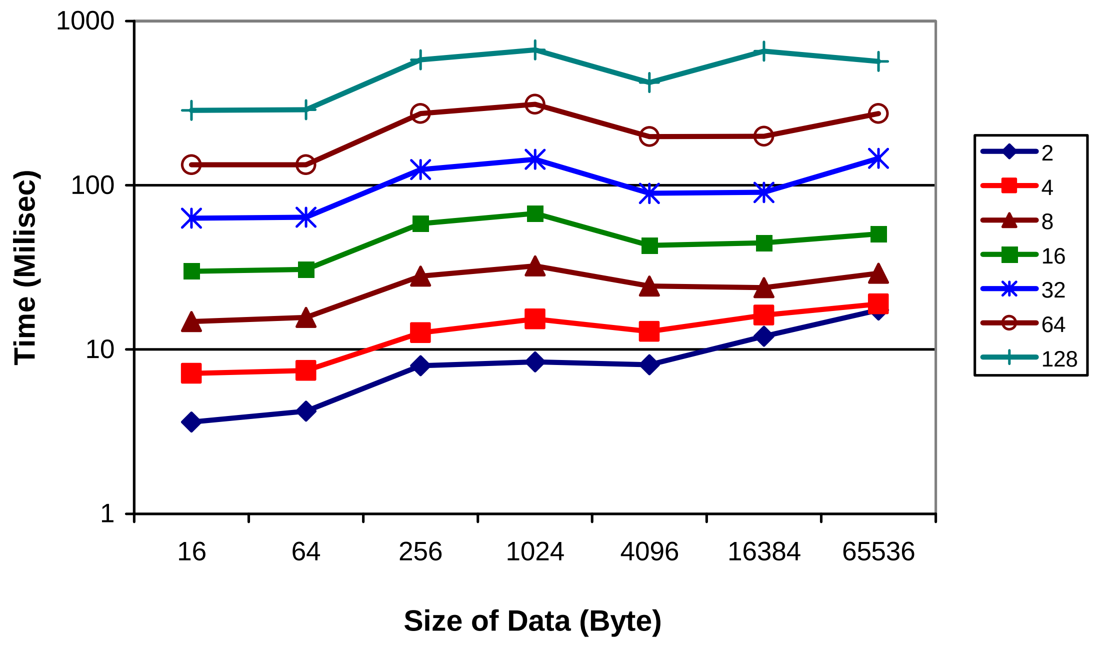

Figure 7.13 : Performance for MPI_Scatter for 2 to 128 processors

Figure 7.14 shows the probability distribution for 64 processors and at 256 Kbytes per process. Each processor completes the scatter operation in the order that they receive the data from the root processor. The root process is the last to complete (shown by the small peak at the right of the plot) since it needs to receive an acknowledgement from all of the processors that they received the data.

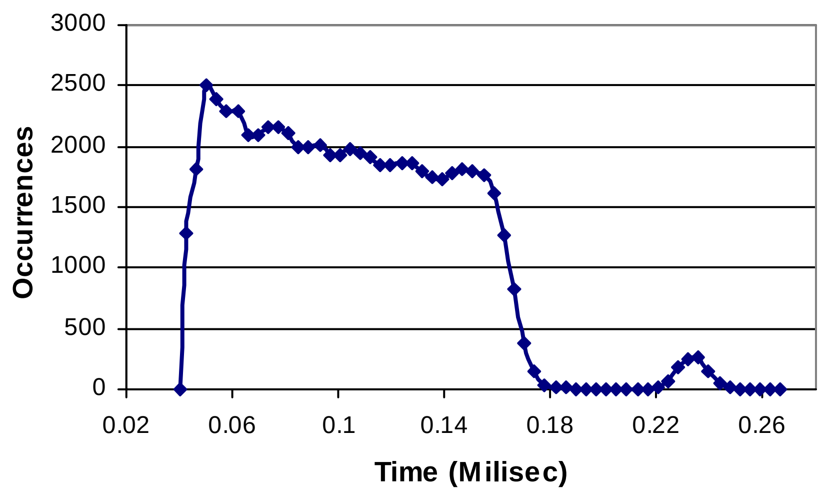

Figure 7.14 : Distribution for MPI_Scatter for 64 processors at 256Kbytes

The performance of MPI_Gather is mainly determined by how much data is received by the root process, which is the bottleneck in this operation. Hence the time taken is expected to be roughly proportional to the total data size for a fixed number of processors, with the time being slower for larger numbers of processors due to serialization and contention effects. Figure 7.15 shows the results from MPIBench for average times to complete an MPI_Gather operation. The times are roughly proportional to data size, at least for larger sizes. The Altix gives significantly better results than the APAC SC. In the worst case, at 1 Kbyte per process, it is around 2 to 4 times faster, while at 2 Kbytes per process it is around 10 times faster. Above 2 Kbytes per process the implementation on the AlphaServer SC became unstable and crashed, whereas the Altix continues to give good performance.

Figure 7.16 shows the probability distribution for 64 processors and at 4 Kbytes per process. Process 0 is by far the slowest process to complete, since it has to gather and merge results from all other processors. Process 1 is the first to complete (the small peak at the left in Figure 7.16) since it is on the same node as the root process, and therefore has a much faster communication time.

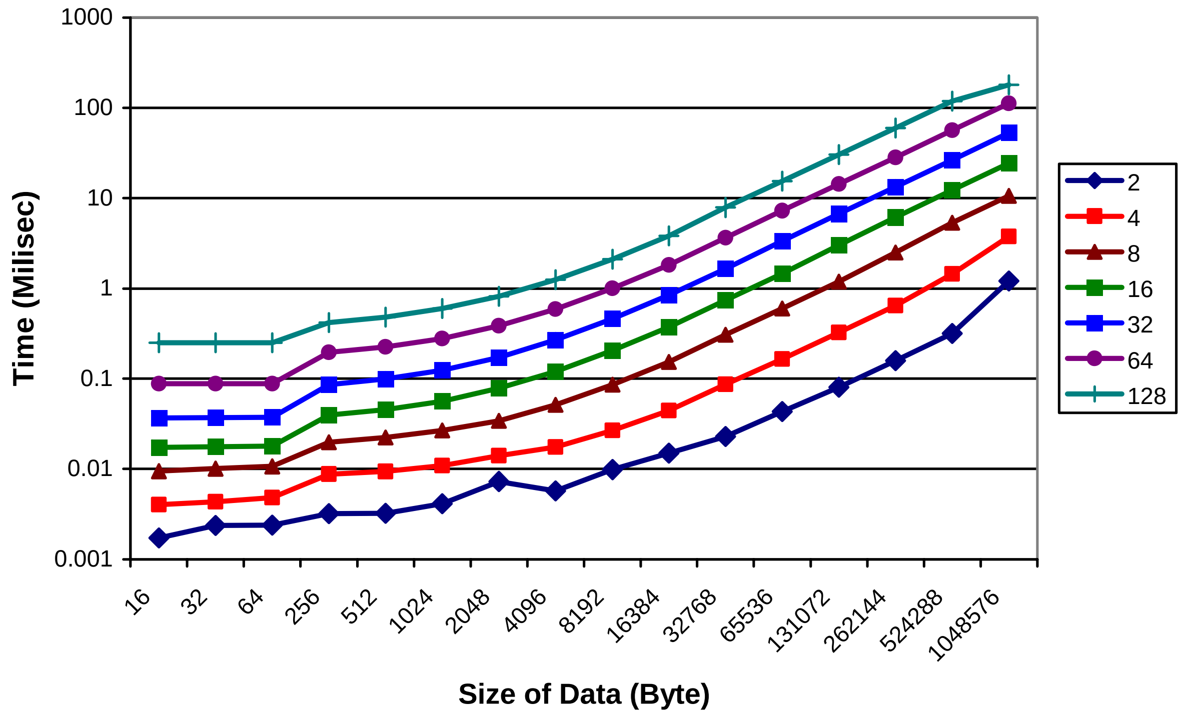

Figure 7.15 : Performance for MPI_Gather for 2 to 128 processors

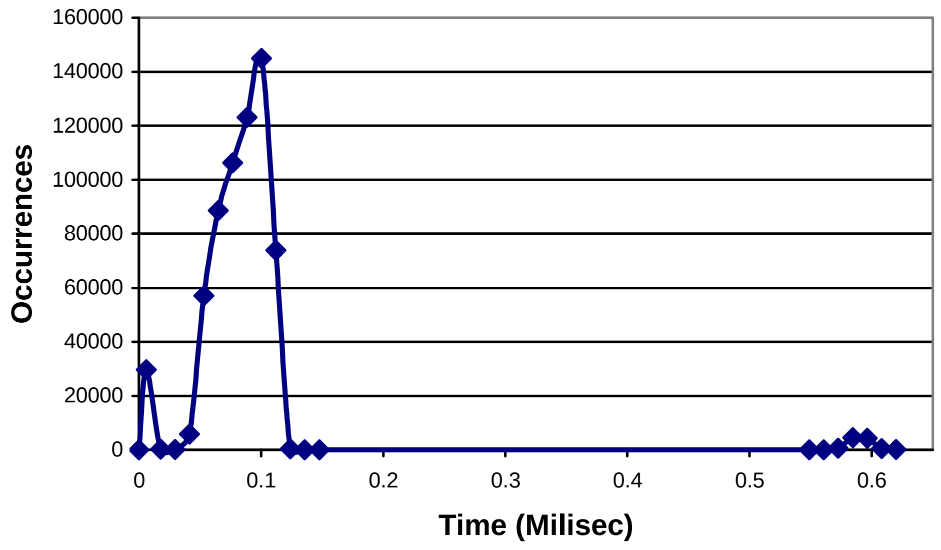

Figure 7.16 : Distribution for MPI_Gather for 64 processors at 4Kbytes

7.9 Alltoall

The final collective communication operation that we measured is MPI_Alltoall, where each process sends its data to every other process. This provides a good test of the communications network. We might expect the communication times to be roughly linear in the data size, however Figure 7.17 shows the results are more complex than that, with the same broad hump around 1 Kbyte per processor that was seen MPI_Scatter, again presumably due to the use of buffered communications for messages of this size. Figure 7.18 shows that for large messages, there is a wide range of completion times, due to contention effects.

The times for MPI_Alltoall are significantly better on the Altix than the AlphaServer SC. In the worst case, for 1 Kbyte per processor, the Altix is around 2 to 4 times faster than the results measured on the APAC SC [18]. It is around 20 times faster for 4 Kbytes per process and around 30 times faster for 8 Kbytes per process. This is partly because the MPI implementation on the AlphaServer SC did not appear to be optimized for SMP nodes [18].

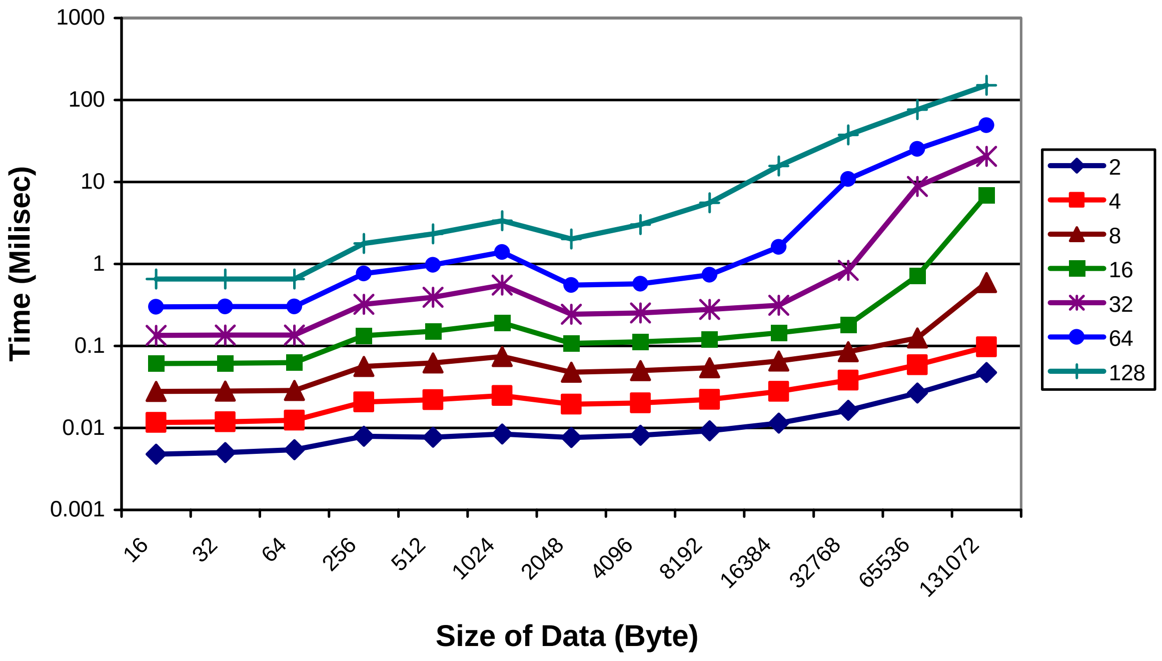

Figure 7.17 : Performance for MPI_Alltoall for 2 to 128 processors

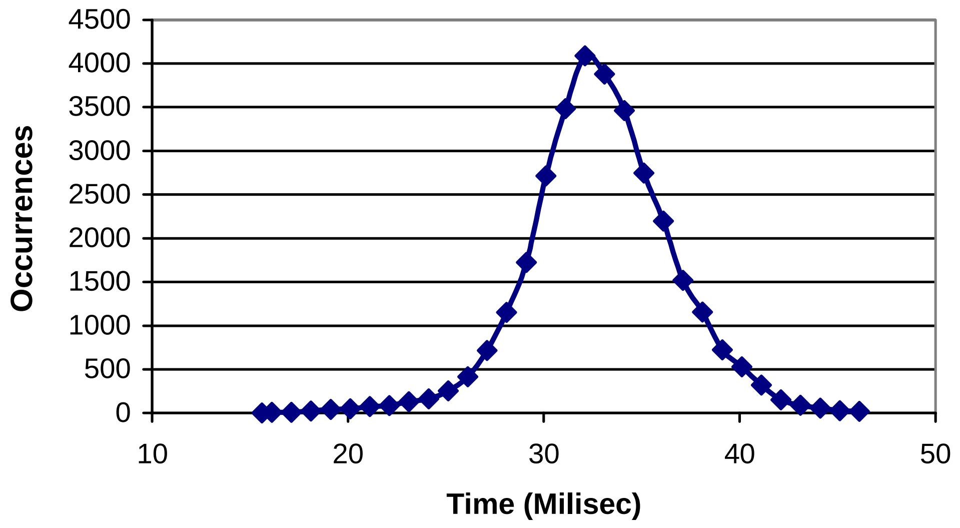

Figure 7.18 : Distribution for MPI_Alltoall for 32 processors at 256Kbytes

7.10 Discussion

The SGI Altix shows very good MPI communications performance that scales well up to 128 processors. Overall the performance was significantly better than the measured performance of the AlphaServer SC with Quadrics network, which has been replaced by a large SGI Altix as the Australian national supercomputer facility. The Altix provides higher bandwidth and lower latency for point-to-point MPI communication than the Quadrics network on the AlphaServer SC, with significantly better collective communications performance, except for broadcast and barrier operations on contiguous nodes, where the Quadrics network provides very fast hardware-enabled broadcast.

The performance of some communications routines on the Altix can be significantly improved by using the Single Copy option provided in the SGI MPI library rather than a buffered copy, in the cases where it is not already used as the default.

CHAPTER 8

Conclusion and Further Work

This thesis provides information on several different aspects of the performance analysis of Message Passing Interface (MPI) implementations on both distributed memory and shared memory parallel computers. A major focus of this thesis was the use of MPIBench, a new MPI benchmark program that provides some useful new functionality compared to existing MPI benchmarks. The work presented in this thesis involved a detailed comparison of MPIBench with other MPI benchmarks, making a number of improvements to MPIBench, and then using MPIBench to investigate MPI performance for a variety of commonly used architectures for parallel computing. Measurements and comparisons were done for a Linux PC cluster with Ethernet and Myrinet networks, and for ccNUMA shared memory machines. Particular attention was given to the variability of communication times and how the performance scales to large numbers of processors, and to identifying performance problems and mechanisms for improving performance, particularly for commodity cluster computers. MPIBench proved to be a useful tool for investigating MPI performance, particularly the capability of providing distributions of communication times for each processor.

The work in chapter 3 is the first detailed comparison of the functionality of different MPI benchmarks and the results they produce on both distributed memory and shared memory machines. The analysis involved five widely used MPI benchmarks: SKaMPI; PMB; Mpptest; MPBench and MPIBench. The analysis showed that different MPI benchmarks can give significantly different results for certain MPI routines, particularly on the SGI Altix. This is primarily due to the SGI Altix having a hierarchical ccNUMA architecture, which can enhance the variations due to different measurement techniques employed by the different benchmarks. The variations for point-to-point communications are due to the different communications patterns used by the different benchmarks, differences in how averages are computed and errors are handled, and how bandwidth is reported. There are also significant differences in measurements of some collective communications routines, particularly broadcast, due to differences in the use of cache for message data and in synchronizing the calls to the routines on each processor.

The differences in the results of the MPI benchmarks is understandable, since the MPI benchmarks were designed primarily for use on distributed memory machines, and the analysis shows that some of the different design decisions made for the different benchmarks can significantly affect the results for ccNUMA shared memory machines. In contrast, the results on distributed memory machines show not much difference between the benchmarks either for point-to-point or collective communication. The users of MPI benchmarks on shared memory machines should therefore be careful in the interpretation of the benchmark results, and developers of some of the MPI benchmarks may need to make some minor modifications to their codes to provide more accurate results for shared memory machines.

Chapter 4 explains the improvements and additional functionality that were provided for MPIBench, based on the comparison between MPI benchmarks from Chapter 3, and experiences in using MPIBench on a variety of machines. Improvements were made to the functionality, ease of use, robustness and portability of MPIBench. There are a number of improvements in functionality, for example addition of ring communication pattern and user-specified communication pattern for point-to-point communication, options for messages to be in cache or not, and analysis of results over an arbitrary set of processes. For ease of use, part of the changes is the auto configuration, which eases the compilation task and the choices of message sizes for the measurements. In terms of portability, there were some errors that have been fixed, particularly for collective communication, memory allocation error and array problems which fixed the problem with non-buffered communication in the SGI Altix. Tests were also done to compare the results of measurements using the global clock synchronization provided by MPIBench with results obtained using MPI_Wtime on the SGI Altix MPI library, which uses the Altix hardware to provide an accurate synchronized clock. This comparison showed that the approach used by MPIBench gave reliable results. Based on the current uses of MPIBench and analysis of other MPI benchmark, Chapter 4 presents a few suggestions for further improvements to MPIBench, for example, the adaptive refinement of message sizes in order to find results for additional message sizes where the results are changing rapidly, which has been provided by Mpptest and SKaMPI.

One of the reasons for undertaking the detailed comparison between MPI benchmarks, and the improvements to MPIBench, was is to ensure that it could be easily and reliably used for analysis of MPI performance in a number of different situations. Chapter 5 compares the performance of Fast Ethernet and Myrinet networks for MPI communications on the same commodity Linux PC cluster. In particular, the analyses have investigated the effects of network contention (including Ethernet packet loss and subsequent Retransmit Time-Outs) by measuring and analyzing distributions of communication times for point-to-point and collective communications, and how they scale with increasing message sizes and numbers of processes. As expected, the Myrinet network performs significantly better than Fast Ethernet. The TCP RTO on the Ethernet network does affect communications performance, but only for large message sizes and large numbers of processors, where the network becomes saturated so that packets are dropped at a fairly high rate. In that case, it can have significant impact on the performance of collective communications, particularly MPI_Bcast and MPI_Alltoall. Earlier measurements by Grove et al. [8,9] for older versions of MPICH showed that TCP RTOs and congestion control mechanisms can greatly reduce the performance of MPI_Gather and MPI_Alltoall, and even cause them to fail at large message sizes. However the new analysis described in Chapter 5 found the effects to be much less serious than for these previous measurements, probably due to improvements in the collective communications routines used in the latest versions of MPICH. The results presented in this chapter also found some anomalous results for all-to-all and broadcast routines, where communication was sometimes faster for larger message sizes.

Chapter 6 analysed the anomalous results that were presented in Chapter 5 and found that they were caused by the changeover points between different algorithms for collective communication for the new version of MPICH. The experiments done in this chapter found that for a number of collective routines, the best changeover points between algorithms for Myrinet and Ethernet on a Linux PC cluster with two processors per node are quite different to the fixed settings in MPICH. The experiments demonstrated that tuning the changeover points to a particular architecture can give a significant improvement in performance, particularly for Ethernet networks. The main reason for the improvements seems to be that the best changeover points can be very different depending on whether 1 or 2 CPUs per node are used, and the MPICH values are on the measurements of Thakur et al. [11] on clusters with 1 CPU per node. This study provides values for better cutoff points for a Linux PC cluster with dual processor nodes connected by Myrinet with GM and Fast Ethernet with TCP. This study also provides a comparison of results between MPICH2 1.0.4 and MPICH-GM 1.2.7 and it shows that MPICH2 shows improvement for certain algorithms.

Finally, chapter 7 used MPIBench to analyse the MPI performance of a large ccNUMA shared memory machine, the SGI Altix 3000, and compared the results with an AlphaServer SC, a high-end cluster of SMP nodes connected by a high-speed network, Quadrics QsNet. This is an interesting contrast of MPI performance between shared memory and distributed memory machines. It is particularly of interest to users of the Australian national computing facility, where an AlphaServer was replaced with an Altix. The results show that the Altix has very good MPI communications performance that scales well up to 128 processors. Overall the performance was significantly better than the measured performance of the AlphaServer SC. The Altix provides higher bandwidth and lower latency than the Quadrics network, with significantly better collective communications performance, except for broadcast and barrier operations on contiguous nodes, where the Quadrics network provides very fast hardware-enabled broadcast. It was found that the Single Copy (non-buffered) communication option significantly improved performance over buffered copy. Single Copy is not always the default option, but should be used wherever possible in order to get best performance.

In completing this thesis work, there were many obstacles that had to be passed. A major problem was to get dedicated access to large parallel computers in order to test the MPI performance of different networks for large numbers of processors. The machines we had access to were an IBM Linux cluster and SGI Altix at SAPAC. Since these machines are a shared resource with many users, it was difficult to get dedicated access to the whole machine for large periods of time. Tests on smaller numbers of processors were easily done. Dedicated access was arranged during times after the machines were taken down for upgrades or reboots. The architecture of SGI Altix took some time to understand, since at the time the research was done the SGI Altix architecture was still new and not many references could be found, this made it difficult to understand some of the results from this machine, such as differences in results between the MPI benchmarks for certain MPI routines, and the problem with Single Copy option with MPIBench. A problem on the IBM Linux cluster was to find on how to analyse the performance using the same machine but different networks, Ethernet and Myrinet. This required understanding the configurations of the switches for Ethernet and Myrinet. A different code have been created to refer either the switches are for Ethernet or Myrinet, for example node1-m and m is referring to Myrinet. Another issues that take longer time to understand is the anomalous performance results that were obtained from some of the MPI routine from both machines. In general, it was often difficult to explain the results obtained from the MPI benchmarks in terms of the machine architecture and the details of the MPI implementation, collective communication algorithm and the MPI benchmark used. Implementation details of the different MPI benchmarks were often not available from the papers and user documentation for the benchmarks, so an understanding of the coding from all MPI benchmarks involved in this thesis was required, which was one of the hardest tasks in completing this thesis work.

In summary, the main contributions of this thesis are:

i. Obtained results from MPIBench on several different types of parallel computers, and demonstrated that it is a very useful tool for detailed MPI performance analysis, particularly for machines with SMP nodes

ii. Significantly improved the functionality, ease of use, robustness and portability of MPIBench.

iii. Provided the first detailed comparison of the functionality and results of different MPI benchmarks, and showed that results can differ significantly between the different benchmarks in some situations, particularly for shared memory architectures. Importantly, some of these differences could also affect results from future clusters with many cores per node.

iv. Comparison of Myrinet and Ethernet performance using MPIBench for large numbers of nodes, particularly analysis of variations in communication times, including the effects of TCP retransmit timeouts on Ethernet performance. The results showed that new versions of MPICH using improved collective communication algorithms no longer have the major problems that were found with older versions on Ethernet clusters. They also showed that RTOs were only a problem with large message sizes or collective operations such as Alltoall that have a large amount of aggregate communication.

v. Demonstrated that significant performance improvements can be obtained by converting MPICH to enable tuning of changeover points between different algorithms for collective communications, particularly for clusters with multiprocessor SMP nodes.

vi. The first comprehensive analysis of MPI performance on the SGI Altix shared memory architecture, providing results for point-to-point and collective communications and for large numbers of processors. There is also the comparison with an AlphaServer SC, a distributed memory cluster with a high-speed communications network

The following are a few suggestions for improving MPIBench and further work that has not been done in this thesis.

i. In improving MPIBench,
   a. The adaptive message refinement tools in order to fix results for certain message sizes which have been applied by Mpptest and SKaMPI.
   b. More MPI communications routines should be added to give more choice to user, particularly for the collective communication.
   c. Making available a variety of common communication patterns would also be useful.
   d. The user-specified calculation of average time for the ring pattern for testing MPI_Sendrecv, either it is calculated based on the average time of all processes or the average time of the slowest process.

ii. Investigate current research aiming to make TCP policies more flexible and dynamic, which could allow improved performance on Ethernet clusters.

iii. In future, software could be developed to automate the process of selecting the optimal changeover points between different algorithms for MPI collective communication routines. When a portable MPI library such as MPICH is installed on a machine, a configuration script could run MPI benchmarks for a wide range of message sizes for all of the multiple algorithms used by MPI collective routines. Then the results could be automatically compared and the default changeover points can be replaced with new improved changeover points.

iv. Use MPIBench for additional analysis and comparison of different communication networks (e.g. Infiniband and Gigabit Ethernet) and MPI libraries (e.g. OpenMPI), particularly for SMP clusters with many CPUs per node.

v. Use MPIBench for measuring the effects of RTOs and MPI performance on Gigabit Ethernet with Jumbo Frame.

REFERENCES

[1] D.A. Grove and P.D. Coddington. *Precise MPI Performance Measurement Using MPIBench*, in Proc. of HPC Asia, September 2001.

[2] MPIBench, http://www.dhpc.adelaide.edu.au/projects/MPIBench

[3] M. Matsuda, T. Kudoh, Y. Kodama, R. Takano, and Y. Ishikawa. *TCP Adaptation for MPI on Long-and-Fat Networks*, in Proc. of IEEE Cluster, 2005.

[4] S. Majumder and S. Rixner. *Comparing Ethernet and Myrinet for MPI Communication*, in Proc. of 7th Workshop on languages, compilers, and run-time support for scalable systems, 2004, Houston Texas.

[5] P. J. Sokolowski and D. Grosu. *Performance Considerations For Network Switch Fabrics On Linux Clusters*, in Proc. of 16th IASTED International Conference on Parallel and Distributed Computing Systems, November 2004, MIT Cambridge, USA.

[6] M. Lobosco, V. S. Costa and C. L. de Amorim. *Performance Evaluation of Fast Ethernet, Giganet and Myrinet on a Cluster*, in Proc. of International Conference on Computational Science-Part I, 2002.

[7] G. R. Luecke, J. Yuan, S. Spanoyannis and M. Kraeva. *Performance and Scalability of MPI on PC Clusters.* Journal of Concurrency and Computation : Practice and Experience, 2004: vol 16, pages 79-107.

[8] Duncan A. Grove, *Performance Modelling of Message-Passing Parallel Programs,* PhD Thesis, University of Adelaide, 2003. Technical report DHPC-138.

[9] Francis A. Vaughan, Duncan A. Grove and Paul D. Coddington, *Communication Performance Issues for Two Cluster Computers*, in Proc. of 26th Australasian Computer Science Conference (ACSC 2003), Adelaide, February 2003, Conferences in Research Practice and Information Technology, Vol 16, 2003.

[10] D.A. Grove and P.D. Coddington, *Analytical Models of Probability Distributions for MPI Point-to-Point Communication Times on Distributed Memory Parallel Computers*, in Proc. 6th Int. Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2005), Melbourne, Oct 2005, Lecture Notes in Computer Science, Volume 3719, pp 406-415, Springer, 2005.

[11] Rajeev Thakur, Rolf Rabenseifner, and William Gropp, *Optimization of Collective Communication Operations in MPICH*, Int. Journal of High Performance Computing Applications, (19)1:49-66, 2005.

[12] Mohak Shroff and Robert A.van de Geijn. *CollMark: MPI collective communication benchmark*. Technical Report, Dept. of Computer Sciences, University of Texas at Austin, December 1999.

[13] M. Barnett, S. Gupta, D. Payne, L. Shuler, A. van de Geijn, and J. Watts. *Interprocessor collective communication library (Intercom)*. In Proceedings of Supercomputing '94, November 1994.

[14] C. Pope, H. Detmold, et al. *Adapting to New Environments: Rethinking the TCP/IP Stack*. Proceedings of the International Conference on Internet Computing, Las Vegas, CSREA Press. June 2003.

[15] F.Petrini, D.J.Kerbyson, et al. (2003). *The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q*. Proceedings of the 2003 ACM/IEEE conference on Supercomputing, IEEE Computer Society.

[16] Wu-chun Feng, Peerapol Tinnakornrishuphap. *The Failure of TCP in High-Performance Computational Grids*. Supercomputing 2000.

[17] W. Gropp, E. Lusk. *Reproducible Measurements of MPI Performance Characteristics*. In Proc. of the PVM/MPI Users' Group Meeting (LNCS 1697), pages 11-18, 1999.

[18] D.A. Grove and P.D. Coddington. *Performance Analysis of MPI Communications on the AlphaServer SC*. Proc. of APAC'03, Gold Coast, 2003.

[19] P.J. Mucci, K. London, and J. Thurman. *The MPBench Report*. Technical Report UT-CS-98-394, University of Tenessee, Department of Computer Science, November 1988.

[20] F. Petrini, W.-C. Feng, A. Hoise, S. Coll and E. Frachtenberg. *The Quadrics network: High-performance clustering technology*. IEEE Micro **22**(1), 46-57 (2002).

[21] R. Reussner, P. Sanders, L. Prechelt, and M. Muller. *SKaMPI: A Detailed, Accurate MPI Benchmark. In Parallel Virtual Machine and Message Passing Interface*, Proc. of 5th European PVM/MPI Users' Group Meeting, 1998.

[22] N.A.W Abdul Hamid, P.D. Coddington and F. A. Vaughan. *Performance Analysis of MPI Communications on the SGI Altix 3700*, Proc. Australian Partnership for Advanced Computing Conference (APAC'05), Gold Coast, Australia, September 2005.

[23] H. Mierendorff, K. Cassirer and H. Schwamborn. *Working with MPI Benchmarking Suites on ccNUMA Architectures,* Proc. of the 7th European PVM/MPI Users' Group Meeting, 2000.

[24] T. Worsch, R. Reussner and W. Augustin. *On Benchmarking Collective MPI Operations,* Proc. of 9th European PVM/MPI Users' Group Meeting, 2002.

[25] H. Chen and P. Wyckoff. *Simulation studies of Gigabit Ethernet versus Myrinet using real application cores.* In Proc. of High-Performance Computer Architecture, Toulouse, France, January 2000.

[26] P. Patarasuk, A. Faraj and X. Yuan. *Pipelined Broadcast on Ethernet Switched Clusters.* In 20th International Parallel and Distributed Processing Symposium, April 2006.

[27] N.A.W Abdul Hamid, P.D. Coddington and F. A. Vaughan. *Comparison of MPI Benchmark Programs on an SGI Altix ccNUMA Shared Memory Machine.* In 20th International Parallel and Distributed Processing Symposium, April 2006.

[28] B.Wilkinson and M.Allen (1999). *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, ALAN APT.

[29] D.Jiang and J. P.Singh (1998). *A methodology and an evaluation of the SGI Origin2000*. Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, Madison, Wisconsin, United States, ACM Press.

[30] Stevens, W. R. *TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms*. USA, National Optical Astronomy Observatory. January 1997.

[31] J.Piernas, A.Flores, et al. *Analyzing the Performance of MPI in a Cluster of Workstation Based on Fast Ethernet*. In Parallel Virtual Machine and Message Passing Interface." Lecture Notes in Computer Science **1332** (4th European PVM/MPI Users' Group Meeting): 17-24 (1997).

[32] Jacobson, V. *Congestion Avoidance and control*. ACM Computer Communication Review 18(4): 316-329 (1988).

[33] M. Baker, A. Farrell, et al. *VIA Communication Performance on a Gigabit Ethernet Clusters*. *Proceedings of the International Euro-Par Conference*, Manchester, UK. 2001.

[34] Mark Allman, V. P., and W. Richard Stevens. "TCP congestion control." *The Internet Society*. (April 1999).

[35] IBM eServer 1350 Linux cluster.
http://www3.ibm.com/systems/clusters/hardware/1350.html

[36] P. Loic and T. Bernard. *BIP : A New Protocol Designed for High Performance Networking on Myrinet*, in 1st Workshop on Personal Computer based Networks of Workstation, 1998.

[37] G.R.Luecke, Z.Guan, et al. *Scalability and Performance of MPI, HPF, and OpenMP on an SGI Origin 2000.* (2002).

[38] G. Chiola, G. Ciaccio. *Porting MPICH ADI on GAMMA with Flow Control*, in Proc. of MWPP'99, 1999 Midwest Workshop on Parallel Processing, Kent, Ohio, August 11 - 13, 1999.

[39] G. Chiola and G. Ciaccio. *GAMMA: a Low-cost Network of Workstations Based on Active Messages*, in Proc. of PDP'97, 5th EUROMICRO workshop on Parallel and Distributed Processing, London, UK, January 1997.

[40] G. Chiola and G. Ciaccio. *Architectural Issues and Preliminary Benchmarking of a Low-cost Network of Workstations based on Active Messages*, in Proc. of ARCS'97, 14th ITG/GI conference on Architecture of Computer Systems, Rostock, Germany, September 1997.

[41] R. Morrison, D. Balasubramaniam, M. Greenwood, G. Kirby, K.Mayes, D. Munro and B. Warboys. *An Approach to Compliance in Software Architectures. Computing and Control Engineering,* in Journal. 11:4 (August 2000) 195-200.

[42] O. Hong, A. F. Paul. *Performance Comparison of LAM/MPI, MPICH and MVICH on a Linux Clusters connected by a Gigabit Ethernet Networks*, in 4th Annual Linux Showcase & Conference, Atlanta, Georgia. October 10-14, 2000.

[43] E. Thorsten, B. Anindya, B. Vineet, V. Werner. *U-Net: A User-Level Network Interface for Parallel and Distributed Computing,* in Proc. of the 15th ACM Symposium on Operating System Principles, Copper Mountain, Colorado, December 3-6 1995.

[44] H. Liu, Z. Du, Q. Ma, Y. Chen, C. Xie. *Design and Test of MVICH Device Layer of MPICH for VIA*, in Proc of 2002 International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES2002), pp370~373, Dec, 2002.

[45] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. *Active Messages: A A Mechanism for Intergrated Communication and Computation,* in Proc. Of the 19th ISCA, pages 256-266, May 1992.

[46] Jeffrey S. Vetter, Sadaf R. Alam, etal. *Early Evaluation of the Cray XT3.* In 20th International Parallel and Distributed Processing Symposium, April 2006.

[47] R. Riesen. *Communication Patterns.* In 20th International Parallel and Distributed Processing Symposium, April 2006.

[48] R. Fatoohi, S. Saini and R. Ciotti. *Interconnect Performance Evaluation of SGI ALTIX 3700 BX2, CRAY X1, Cray Opteron Cluster, and Dell PowerEdge.* In 20th International Parallel and Distributed Processing Symposium, April 2006.

[49] S. Saini, R. Ciotti et al. *Performance Evaluation of Supercomputer using HPCC and IMB Benchmarks*. In 20th International Parallel and Distributed Processing Symposium, April 2006.

[50] Z. Yingchou, M. Dan and M. Jie. *Dual-Layered File Cache On cc-NUMA System.* In 20th International Parallel and Distributed Processing Symposium, April 2006.

[51] P.H Carns, W.B. Lignon III, S.P. McMillan, and R.B. Ross. *An evaluation of message passing implementations on Beowulf workstation*. In Proceedings of the IEEE Aerospace Conference, March 1999.

[52] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. *High Performance, portable implementation of the MPI Message Passing Interface standard*. *Parallel Computing,* 22(6):789-828, 1996.

[53] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, Massachusettes, 1994.

[54] Lars Paul Huse. *Collective communication on dedicated cluster of workstations*. In Proceedings of the 6th European PVM/MPI Users' Group Meeting, pages 469-476, September 1999.

[55] L. Brakmo, S. O'Malley and L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*. In Proceedings of the 1994 SIGCOMM Symposium, pages 24-35 1994.

[56] K. Fall and S. Floyd. *Simulation-based comparisons of Tahoe, Reno and SACK TCP*. ACM Computer Communication Review, 26(3):5-21, 1997.

[57] S. Floyd. *Congestion control principles*. Technical Report RFC 2914, The Internet Society, September 2000.

[58] S. Floyd, J. Mahdavi, M. Mathis and M. Podolsky. *An extension to the Selective Acknowledgement (SACK) option for TCP*. Technical Report RFC 2883, The Internet Society, July 2000.

[59] R. Ludwig and R. H. Katz. *The Eifel algorithm: Making TCP robust against spurious retransmission*. ACM Computer Communication Review, 30(1), January 2000.

[60] R. Ludwig and R. H. Katz. *The Eifel retransmission timer*. ACM Computer Communication Review, 30(3), July 2000.

[61] J. Mo, R. J. La, V. Anantharam and J. C. Walrand. *Analysis and comparison of TCP Reno and Vegas*. In Proceedings of IEEE INFOCOM, pages 1556-1563, March 1999.

[62] SKaMPI. http://liinwww.ira.uka.de/~skampi/.

[63] Mpptest. http://www-unix.mcs.anl.gov/mpi/mpptest/.

[64] MPBench. http://icl.cs.utk.edu/projects/llcbench/mpbench.html

[65] Intel MPI Benchmark (IMB) Homepage.
http://www.intel.com/cd/software/products/asmo-na/eng/cluster/219847.htm

[66] E. W. Chan, M. F. Heimlich, A. Purakayastha and R. A. van de Geijn. *On optimizing collective communication.* In Proceddings of the 2004 IEEE International Conference on Cluster Computing, September 2004.

[67] E.A Brewer and B.C Kuszmaul. *How to get good performance from the CM-5 data network.* In Proceedings of the 8th International Parallel Processing Symposium, 1994.

[68] Gregory D. Benson, Cho-Wai Chu, Qing Huang, and Sadik G. Caglar. *A comparison of MPICH allgather algorithms on switched networks*. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users' Group Meeting, pages 335--343. Lecture Notes in Computer Science 2840, Springer, September 2003.

[69] MPICH – A portable implementation of MPI.
http://wwwunix.mcs.anl.gov/mpi/mpich.

[70] SGI Altix 3000. http://www.sgi.com/products/servers/altix/.

[71] Myricom Incorporated. http://www.myri.com.

[72] Hewlett-Packard, AlphaServer SC supercomputer,
http://h18002.www1.hp.com/alphaserver/sc/sys_sc45_features.html

[73] Cosimo Anglano. Cluster benchmarks.
http://www.di.unito.it/~mino/cluster/becnhmarks.

[74] Massimo Bernaschi, Giulio Iannello , Mario Lauria, *Experimental Results about MPI Collective Communication Operations*, Proceedings of the 7th International Conference on High-Performance Computing and Networking, p.774-783, April 12-14, 1999

[75] Thilo Kielmann , Rutger F. H. Hofman , Henri E. Bal , Aske Plaat , Raoul A. F. Bhoedjang, *MagPIe: MPI's collective communication operations for clustered wide area systems*, Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming, p.131-140, May 04-06, 1999, Atlanta, Georgia, United States

[76] PCI-X Networked Bus Analyzers from VMetro. http://www.pcixanalyzer.com/.16 December 2005.

[77] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and W. Su. *Myrinet : A Gigabit per second Local Area Network*. IEEE Micro, 15(1):29-36, February 1995.

[78] A. Faraj, P. Patarasuk and X. Yuan. *Bandwidth Efficient Alltoall Broadcast on Switched Cluster*. IEEE Cluster, 27-30, September 2005.

[79] A. Proskurowski. *Minimum Broadcast Trees*. IEEE TC, c-30, pp. 363-366, 1981.

[80] gettimeofday().
http://www.opengroup.org/onlinepubs/007908799/xsh/gettimeofday.html

[81] The MPIForum. *The MPI-2: Extension to the Message Passing Interface*, July 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[82] S. S. Vadhiyar, G. E. Fagg and J. Dongarra. *Automatically tuned collective communication*, In Proceedings of SC'00: High Performance Networking and Computing 2000.

[83] N. Nupairoj and L. Ni. *Performance Evaluation of Some MPI Implementation on Workstation Clusters*. In Proceedings of the Scalable Parallel Libraries, October 1994.

[84] Australian Partnership for Advanced Computing (APAC). http://nf.apac.edu.au.

[85] D. Bailey, E. Barszcz, J. Barton, D. Browning, et al. *The NAS parallel benchmarks*. International Journal of Supercomputing Applications, 5(3):63-73, 1991.

[86] S. Browne, J. Dongarra and K. London. *Review of performance analysis tools for MPI parallel programs*. Technical report, University of Tenessee, Department of Computer Science, December 1997.

[87] F. Cappello, O. Richard and D. Etiemble. *Understanding performance of SMP clusters running MPI programs.* Future Generation Computer Systems, 17(6):711-720, 2001.

[88] L. A. Cowl. *How to measure, present and compare parallel performance*. Parallel and Distributed Technology, 2(1):9-25, 1994.

[89] G. Cybenko, L. Kipp, L. Pointer and D. Kuck. *Supercomputer performance evaluation and the Perfect benchmark.* Technical Report 965, University of Illinois Center for Supercomputing R&D, March 1990.

[90] B. R. de Supinski and N. T. Karonis. *Accurately measuring MPI broadcasting in a computational grid.* In Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing, pages 29-37, August 1999.

[91] J. Dongarra, H. Meuer and E. Strohmaier. *Top 500 Supercomputer Sites*. http://www.top500.org.

[92] J. L. Gustafson, D. Heller, R. Todi and J. Hsieh. *Cluster Performance: SMP versus Uniprocessor nodes.* In Proceedings of Supercomputing, November 1999.

[93] K.A. Hawick, D.A. Grove, P.D Coddington and M.A. Buntine. *Commodity cluster computing for computational chemistry*. Internet Journal of Chemistry, 3:article 4, 2000.

[94] R. W. Hockney. *Performance parameters and benchmarking of supercomputers*. Parallel Computing, 17(10), December 1991.

[95] J. Hsieh. *Design choices for a cost-effective, high performance Beowulf-cluster*. Dell Power Solutions, 3, 2000.

[96] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk and J. Bresnahan. *Exploiting Hierarchy in parallel computer networks to optimize collective operation performance*. In Proceedings of the 14th International Parallel and Distributed Processing Symposium, pages 377-384, May 2000.

[97] D.E. Knuth. *Big Omicron and Big Omega and Big Theta*. ACM SIGACT News, 8(2):18-23, 1976.

[98] W. E. Leland, M.S. Taqqu, W. Willinger and D. V. Wilson. *On the self-similar nature of Ethernet Traffic (extended version).* IEEE/ACM Transaction on Networking, 2:1-15, 1994.

[99] R. P. Martin, A. M. Vahdat, D. E. Culler and T. E. Anderson. *Effects of communication latency, overhead and bandwidth in a cluster architecture.* In Proceedings of the 24th International Symposium on Computer Architecture, pages 85-97, 1997.

[100] V. Paxson and M.Allman. *Computing TCP's retransmission timer*. Technical Report RFC 2988, The Internet Society, November 2000.

[101] X. Qin and J. Baer. *A performance evaluation of cluster architectures*. IEEE SIGMETRICS Performance Evaluation Review, 25(1):237-247, June 1997.

[102] J. Radajewski. *Beowulf supercomputer HOWTO draft*, January 1998.

[103] R. Reussner, P. Sanders and J. Larsson Traff. *SKaMPI: A comprehensive benchmark for public benchmarking of MPI.* Scientific Computing, 10, 2001.

[104] D. B. Skilicorn. *A taxonomy for computer architectures*. IEEE Computer, 21(11):46-57, November 1988.

[105] T. Sterling, Donald J. Becker and D. Savarese. *Beowulf: A parallel workstation for scientific computation.* In Proceedings of the International Conference on Parallel Processing, 1995.

[106] T. B. Tabe, J. Hardwick and Q. F. Stout. *Statistical analysis of communication time on the IBM SP2*. Computing Science and Statistics, 27:347-351, 1995.

[107] W. B. Tan and P. Strazdins. *The analysis and optimization of collective communications on a Beowulf cluster*. In Proceedings of the International Conference on Parallel and Distributed Systems, December 2002. (analysis for allgathe, reduce scatter and allreduce)

[108] M. S. Warren, D. J. Becker, M. Patrick Goda, J. K. Salmon and T. Sterling. *Parallel supercomputing with commodity components*. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, pages 1372-1381, 1997.

[109] M. J Flynn. *Some Computer Organizations and Their Effectiveness*. IEEE Transaction on Computers. C-211(9):948-960.

[110] Myrinet. http://www.netlib.org/utk/papers/advanced-computers/myrinet.html

[111] Compaq, Intel, and Microsoft. VIA Specification 1.0. http://www.viarch.org

[112] G. Chiola and G. Ciaccio. *Implementing a Low Cost, Low Latency Parallel Platform*, in Proc. of DAPSYS'96, Miskolc, Hungary, October 1996.

[113] MVICH. MPI for Virtual Interface architecture. Available at http://www.nersc.gov /research/FTG/mvich/. 4 December 2003.

[114] Micro-Benchmark Level Performance Comparison of High-Speed Cluster Interconnects. Available at http://nowlab.cis.ohio-state.edu/projects/mpi-iba/ publication /hoti01.pdf. 4 December 2003.

[115] G.Ciaccio. *Messaging on Gigabit Ethernet: Some experiments with GAMMA and Other Systems*, in Cluster Computing 6, 2003, pages 143-151.

[116] S. Cozzini. *Network Hardware/Software for Cluster Computing.* Available at http://www.cecalc.ula.ve/HPCLC/slides/day_03/network.pdf. 19 January 2004.

[117] R. Mraz. *Reducing the Variance of Point-to-Point Transfer in the IBM 9076 Paralllel Computer*, in Proc. of Supercomputer '94, pages 620-629, Washington, D.C, November 14-18, 1994

[118] C. SchaubschlÄager. *Automatic testing of nondeterministic programs in message passing systems.* Master's thesis, Johannes Kepler University Linz, Department for Computer Graphics and Parallel Processing, 2000.

[119] S. Pakin, V. Karamcheti, and A. Chien. *Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors*. IEEE Concurrency, 5, April--June 1997, pp. 60--73.

[120] The Abstract Device Interface. Available at http://www-unix.mcs.anl.gov/mpi/ mpich/papers/mpicharticle/node22.html. 18 February 2004.

[121] Design, Implementation and Evaluation of User-Level Protocol. Available at http://www.cis.ohio-state.edu/~surs/Slides/2c-2e.pdf. 4 December 2003.

[122] Programming the Infiniband Network Architecture. Available at http://www.mpi-softtech.com/company/publications/files/prog_ib_2003-02-13.pdf. 16 December 2003

[123] S. S. Vadhiyar, G. E. Fagg, and, J. J. Dongarra, *Towards an Accurate Model For Collective Communications*, International Journal of High Performance Computing Applications, vol. 17, no. 4, Fall 2003.

[124] M. Woodacre, D. Robb, D. Roe, and K. Feind, *The SGI Altix 3000 Global Shared-Memory Architecture*, White Paper, 2005, available from http://sc.tamu.edu/whitepapers/altix/altix_shared_memory.pdf.

[125] S. Neuner, *Scaling Linux to New Heights : the SGI Altix 3000 System*, Linux Journal, Vol. 2003, Issue 106, Pg. 3, February 2003.

[126] H. J. Wassermann, O. M. Lubeck, Y. Lou, and F. Bassetti, *Performance Evaluation of the SGI Origin2000 : A Memory-Centric Characterization of LANL ASCI Applications*, in Proceedings of the 1997 ACM/IEEE Conference on Supercomputing, pg. 1-11, San Jose, CA, 1997.

[127] J. Laudon and D. Lenoski, *The SGI Origin : A ccNUMA Highly Scalable Server*, in Proceedings of the 24th Annual International Symposium on Computer Architecture, pg. 241-251, Denver, Colorado, United States, 1997.

[128] T. Sterling, D. Savarese, P. MacNeice, K. Olson, C. Mobarry, B. Fryxell, and P. Merkey, *A Performance Evaluation of the Convex SPP-1000 Scalable Shared Memory Parallel Computer*, in Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, pg. 55, San Diego, California, United States, 1995.

[129] 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*, in Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Dallas, Texas, 2000.

[130] J. Liu, B. Chandrasekaran, etal. (2003). *Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics*. Proceedings of the 2003 ACM/IEEE conference on Supercomputing, IEEE Computer Society.

[131] South Australian Partnership for Advanced Computing (SAPAC). http://sapac.edu.au.

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

[133] S. Charles's Glossary of Technical Terminology. http://www.mission-cons.k12.tx.us/CECS5030/scharles/terms_html.htm. July 2004.

[134] 100 Mbit/s Fast Ethernet network. http://www.ethermanage.com/ethernet/100mbps.html

[135] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message Passing Interface*. Cambridge, MA: MIT Press, 1999.

[136] MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2/

[137] Overview of recent supercomputers, 2007. http://www.phys.uu.nl/~steen/

[138] Rabenseifner's algorithm. http://www.hlrs.de/organization/par/ sevices/models/mpi/ myreduce .html

[139] S. Vandhiyar, G. E. Fagg and J. Dongarra. *Automatically tuned collective communication.* In Proceedings of SC99: High Performance Networking and Computing, November 1999.

[140] R. Rabenseifner. *New optimized MPI reduce algorithm.* http://www.hlrs.de/organization/par/services/models/mpi/myreduce.html

[141] L. V. Kale, S. Kumar and K. Vardarajan. *A framework for collective personalized communication.* In Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03), 2003.

[142] J. Bruck, C. T. Ho, S. Kipnis, E. Upfal and D. Weathersby. *Efficient algorithms for all-to-all communications in multiport message-passing systems*. IEEE Transactions on Parallel and Distributed Systems, 8(11):1143-1156, November 1997.

[143] F. Petrini, S. Coll, E. Frachtenberg and A. Hoisie. *Performance evaluation of the Quadrics interconnection network*. Journal of Cluster Computing, 2002.

[144] Infiniband Trade Association. *Infiniband Architecture Specification, Release 1.0.* October 24 2000.

[145] Quadrics. Quadrics, Ltd. http://www.quadrics.com

[146] Beowulf. http://www.beowulf.org/

[147] HPC Architeture. http://www.phys.uu.nl/%7Esteen/web06/sm-mimd.html

[148] Is 10 Gigabit Ethernet catching up with Infiniband? http://searchdatacenter.techtarget.com/news/article/0,289142,sid80_gci1280975,00.html

[149] D.V. James, A.T. Laundrie, S. Gjessing, and G.S. Sohi, *Scalable Coherent Interface*, IEEE Computer, 23, 6, (1990), 74--77.

[150] Scalable Coherent Interface, http://sunrise.scu.edu/.

[151] Microsoft Solution Guide for Migrating High Performance Computing (HPC) Applications from UNIX to Windows. http://www.microsoft.com/technet/solutionaccelerators/cits/interopmigration/unix/hpcunxwn/ch01hpc.mspx

[152] Thomas BrÄeanl. *Parallel Programming - An Introduction*, Chapter 3. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[153] R.W. Hockney and C.R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger, Bristol, 1988.

[154] David Skillicorn. *Foundations of Parallel Programming*, chapter 8, pages 123-169. Cambridge University Press, Cambridge, 1994.

[155] David B. Skillicorn and Domenico Talia. *Models and languages for parallel computation*. ACM Computing Surveys, 30(2), June 1998.

[156] D.B. Skillicorn. *A taxonomy for computer architectures*. IEEE Computer, 21(11):46-57, November 1988.

[157] http://www.redbooks.ibm.com/abstracts/sg245161.html

[158] http://www.cray.com/

[159] http://www.ibm.com/us/

[160] Geoffrey C. Fox, Roy D. Williams and Paul C. Messina. *Parallel Computing Works!*. Morgan Kaufmann, 1994.

[161] http://www.cs.uiuc.edu/homes/snir/

[162] https://computing.llnl.gov/tutorials/parallel_comp/

[163] Geoffrey C. Fox and etal., *Solving problems on concurrent processors*, Prentice Hall, 1988.

[164] Jumbo Frame. http://en.wikipedia.org/wiki/Jumbo_Frame

[165] Gigabit Ethernet Jumbo Frame. http://sd.wareonearth.com/~phil/jumbo.html

[166] http://www.top500.org/2007_overview_recent_supercomputers/networks

[167] Quadrics QsTenG for HPC Interconnect Product Family. 13 November 2007. http://www.quadrics.com/quadrics/QuadricsHome.nsf/