Modeling the Communication and Computation Performance of the IBM SP2

Gheith A. Abandah
(313) 936-2917
gabandah@eecs.umich.edu

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122

Advisor: Edward S. Davidson

May 8, 1995

Contents

Preface

In Summer 1994 our research group started a project for studying scientific applications performance on message-passing multi-computers. The objectives of this project included developing techniques for porting scientific applications to message-passing multi-computers, and developing techniques for performance estimation, evaluation, and improvement. The initial test case was porting a large commercial finite-element method application for car crash simulation to the IBM SP2. This application was originally written for a scalar uniprocessor and then ported and tuned for parallel vector supercomputer with shared memory. Now that we have completed this porting we are in a position to use this application to carry out a wide range of experiments to test different techniques and ideas for achieving the project objectives. This report focuses on the development of a series of performance models, together with some examples of using these models, which provide a basis for these experiments and for future porting and tuning efforts on message-passing systems in general, and the IBM SP2 in particular.

We have realized the need to develop models for characterizing the performance of the different aspects of a message-passing computer. The performance models should be able to predict execution time given the high-level source code of a scientific application. Such models are useful to explain the achieved performance and to help in selecting appropriate techniques and where to apply them for performance improvement and tuning.

The achieved performance of a message-passing application depends on the application itself, the guidance provided by the programmer, the quality of the compiler used to generate the object code, the computational performance of the processors, the performance of the memory hierarchy, and the performance of the interconnection network.

In this report we present the results of some of our efforts in developing performance models for the IBM SP1 and the IBM SP2. These efforts started in the Summer of 1994, with a directed study to develop models for the IBM SP1 and the IBM SP2 communication performance using the MPL message-passing library. Since that time we have refined these models, and developed models for some other computers and message-passing libraries.

Since February 1995 we have been developing models for the computational performance of the IBM SP2 processor nodes and their memory hierarchy. Parts of the material presented in this report were also used as term projects for the EECS 570, EECS 587, and EECS 598 courses.

This report is organized in 6 chapters. After the introductory chapter 1, chapter 2 presents the development of the communication performance models for the SP1 and the SP2 using the MPL message-passing library. Chapter 3 shows an example of how the SP2 communication performance models are used in performance evaluation and improvement.

The performance of six broadcast algorithms are evaluated and a methodology for developing an efficient broadcast algorithm is presented. Chapter 4 presents the communication performance models for the IBM SP2 and the Convex SPP-1000 using the PVM message-passing library, and compares their performance based on the communication performance models. Chapter 5 presents the models for the SP2 processor nodes, namely a Machine-Application Performance Bound Model. Some test cases are presented to illustrate the use of this model to obtain computational performance evaluation and improvement. Chapter 6 concludes this report by summarizing what has been achieved and outlining our plans for future work.

Chapter 1

Introduction

In this introductory chapter we specify the problems addressed in this report, we outline our approach for modeling the communication performance of a message-passing multi-computer, we outline our approach for modeling the computation performance of the processor nodes, and we describe how these models are integrated to model the overall performance of a message-passing multi-computer.

1.1 Problem Statement

A Distributed-Memory Multi-computer consists of multiple processor nodes interconnected by a message-passing network. Each processor node is an autonomous computer consisting of a central processing unit (CPU), memory, communication interface adapter, and, for at least some nodes, mass storage and I/O devices.

Figure 1.1 shows a general model for a message-passing multi-computer. The Interconnection Network provides the communication channels through which the processor nodes exchange data and coordinate their work in solving a parallel application. Different types of Interconnection Networks vary in topology and throughput. Hypercubes and Multistage Interconnection Networks, MINs, are two of the commonly used topologies. The Communication Adapter provides the interface between the processor node and the Interconnection

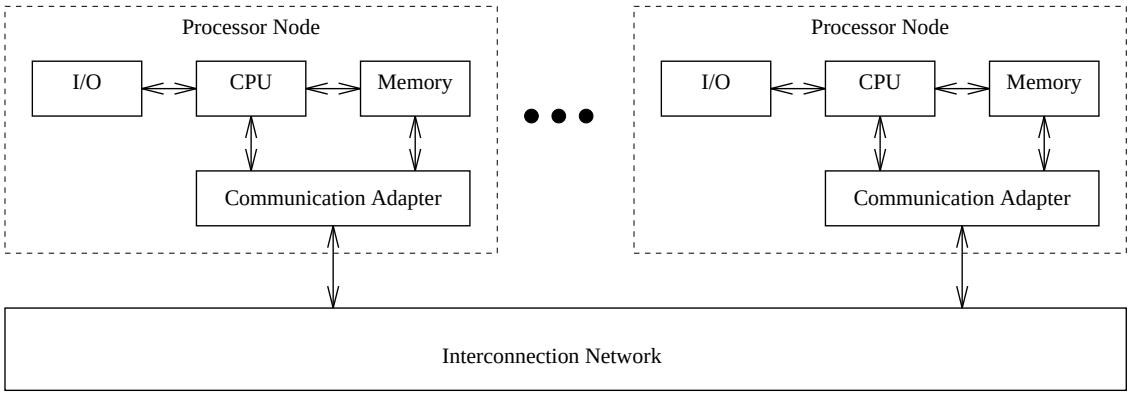

Figure 1.1: Model of a message-passing multi-computer

Network. Some simple Communication Adapters are directly controlled by the CPU, more sophisticated adapters do communication processing, error checking and correction, and direct memory access, DMA. The number of communication links in each Communication Adapter depends on the topology of the Interconnection Network. MINs usually have one communication link per processor node, and hypercubes have $\log_2 p$ communication links, where p is the number of processor nodes.

The performance of a multi-computer is a function of the performance of the processor node components and the Interconnection Network. The objective of this report is to develop models that characterize the performance of the different aspects of a multi-computer to facilitate the following activities:-

- Explain the achieved performance of message-passing applications.
- Develop efficient message-passing applications.
- Compare the performance of different multi-computers and application code implementations.

The models should estimate the execution time of a message-passing application given its high-level source code. Such models are the most useful for engineers who are involved in writing and tuning message-passing applications. Their utility results from two factors. First, the problem of estimating the execution time of a message-passing application for a range of problem sizes becomes an analytical problem of evaluating the performance models. Second, selecting the best performance algorithms and implementation techniques can be done with some confidence at an early stage when a draft of the source code becomes available for the different implementations.

1.2 Communication Modeling

Estimating the time needed to carry out a communication process in a message-passing multi-computer is a complex problem. The communication time depends on the characteristics of the processor nodes and the Interconnection Network, the used message-passing library, the communication pattern, the message size, the number of processor nodes, and the distance between the processor nodes.

Our approach in developing communication performance models is to model the performance of a set of common communication patterns. This set contains the basic communication patterns, and is selected in such a way that the complex communication patterns can be constructed from these basic patterns. Hence the time of a complex communication pattern is estimated by summing the times of its basic components. The set of the basic communication patterns include Point-to-point, Exchange, One-to-many, Many-to-one, and Many-to-many.

To develop performance models for the basic communication patterns we develop programs to perform and time these patterns. Each program is run for varying message lengths and varying number of processors. The timing data gathered from executing these programs is analyzed, and, using curve-fitting techniques, we develop formulas that give the time as a function of the message length and the number of processors.

In the SP1 and the SP2, the communication latencies vary by only a few percent as the distance between the processor nodes is varied. Although we have characterized the distance effect on the SP1 and the SP2 communication times, we do not include the distance in their communication models for simplicity.

1.3 Computation Modeling

Our approach for modeling the computation performance of the processor nodes is through using a Machine-Application Performance Bound (MA Bound) [1, 2, 3, 4]. The MA Bound of an application is an upper bound for the performance that can be achieved for the application on a certain machine. The MA Bound is found by counting the essential operations in the application. The operations are then classified according to the functional unit(s) that will be involved in their execution. After summing up the instructions that will be executed by each functional unit, and assuming a perfect instruction schedule, we find the time that each functional unit will be busy in executing its share of instructions. The MA time is simply the time of the busiest functional unit in each region of the code.

Although the measured time is usually larger than the MA time, experience has shown that the gap can be reduced by better quality compilers, hand coding, or modeling more aspects of the machine.

In this report we develop the MA Bound model for the IBM POWER2 processor which is used in the IBM SP2. This development is done by identifying the POWER2 functional units that affect the performance of scientific applications, and developing a model for each functional unit to translate the number of instructions into time. The memory hierarchy performance is part of the MA Bound. The memory hierarchy is considered as one of the functional units, and is responsible for executing the load and store operations. To model the memory functional unit we use techniques to measure its effective access time, and we develop methods for estimating the number of essential cache misses of an application.

1.4 Model Integration

The execution time of a message-passing application depends on the computation time and the communication time. The multi-computers that use the main CPU for communication processing and data transfer, as in the IBM SP2, do not provide overlap for the computation time and communication time. So in this case the execution time is the sum of the computation and communication times.

In our approach for modeling the performance of the SP1 and the SP2 we find an application execution time by summing the computation time and the communication time. A lower bound on the computation time is found by counting the essential operations in the high-level source code and using the MA Bound model. The communication time is found by identifying the explicit calls to the message-passing library routines in the high-level source code, relating these calls to the basic communication patterns, and using the communication performance models.

Chapter 2

Communication Modeling

2.1 Introduction

In order to write efficient message-passing applications, programmers need good performance models for wide variety of communication patterns. There isn't enough work on developing performance models for message-passing routines. The available models either do not cover all the important communication patterns, or do not give information that is directly usable by a programmer tuning a message-passing application.

Most of the work in developing performance models for message passing multi-computers is centered around benchmarks. The NAS Parallel Benchmarks (NPB) [5] are developed to study the performance of parallel supercomputers. These benchmarks consist of five parallel kernels and three simulated application benchmarks. Together they mimic the computation and data movement characteristics of large scale computational fluid dynamics applications. Another benchmark suite is the PARKBENCH [6], this suite contains low level benchmarks for measuring basic computer characteristics, kernel benchmarks to test typical scientific subroutines, and compact applications to test complete problems.

Benchmarks like NPB are useful for comparing different machines, but they do not separate computation performance from communication performance. PARKBENCH has benchmarks for some of the communication patterns like the benchmark for the point-to-point communication, but it does not contain benchmarks for the other important communication patterns.

We have developed experiments to measure the time of the basic communication patterns on the IBM SP1 and the IBM SP2 using the MPL message-passing library. These communication patterns are common in message-passing applications, and the other more complex communication patterns are constructed from them. We have developed experiments for timing Point-to-point, Exchange, One-to-many, Many-to-one, Many-to-many, Broadcast, and Combine. Also we have studied the Synchronization time on these multi-computers. The gathered data from these experiments were used to develop performance models for these types of communication patterns, and to compare the performance of the two multi-computers.

For each experiment, we have prepared a FORTRAN program that calls the MPL message-passing routines. Each program performs one communication pattern many times,

and reports the minimum time, the average time, and the standard deviation. These programs are executed on a varying number of processors (p) with varying message lengths (n).

All the experiments were carried out on an exclusively reserved machine; where there are no processes for other users. The measured data have low standard deviation; the measured standard deviation divided by the measured time is usually only a few percent.

Section 2.2 describes the IBM SP1 and SP2. The following sections describe the experiments used to study the different communication patterns, show the measured data, analyze it, and develop the performance models. Finally, section 2.11 states the conclusions of this chapter.

2.2 IBM SP1 and SP2

The IBM Scalable POWERparallel Systems SP1 and SP2 connect RISC System/6000 processors via the communication subsystem [7]. This subsystem is based upon a low latency high bandwidth switching network called the *High Performance Switch*. The SP1 systems offer switch connectivity from 8 to 64 POWER nodes, the SP2 systems offer switch connectivity from 4 to 128 POWER or POWER2 nodes [8].

The SP1/SP2 networks are bidirectional multistage interconnection networks (MIN's). In a bidirectional MIN each communication link comprises two channels which carry data in opposite directions. MIN's are capable of scaling bisection bandwidth linearly with the number of nodes while maintaining a fixed number of communication ports per switching element.

The bidirectional MIN is constructed from 4-way to 4-way bidirectional switch elements. The switching elements-physically 8 input, 8 output devices- are wired as bidirectional 4-way to 4-way elements, see figure 2.1. Each switching element can forward packets from any input port to any output port as directed by the packet's route information.

Each node of an SP1/SP2 is incorporated into a *logical frame*, which encompasses up to 16 nodes connected to one side of a switch board. The switch board has two stages, each stage has four switch elements, the first stage is connected to the processor nodes, and the second stage is used to connect with other frames.

Nodes send messages to other nodes by breaking messages into *packets* and injecting these packets into the network. Each packet contains route information examined by switching elements to forward the packet correctly to its destination. The smallest unit on which flow control is performed is called a *flit*, which is one byte in the SP1/SP2 Switch. The width of the data transmitted by an output port is also one byte. Each packet has one length flit followed by one or more route flits then data flits; packet length is at most 255 flits.

The Switch flow control method is *buffered wormhole routing*, each flit of a packet is advanced to the appropriate output port as soon as it arrives at a switching element input port. When the head of a packet is blocked, the flits are buffered in place. As soon as the output port is free, packet transfer resumes.

An SP1/SP2 system is composed of frames containing up to 16 processors and one switch board assembly that implements the two stage network building block. The switching element of the board is the *Vulcan switch chip* [9]. The Switch operates at 40 MHz, providing

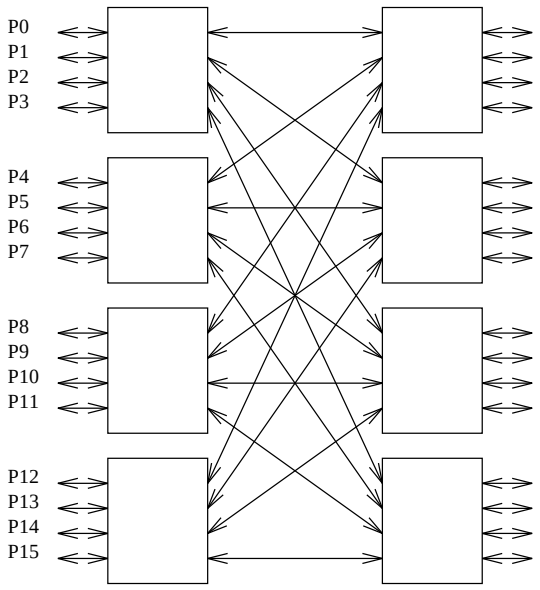

Figure 2.1: An SP1/SP2 logical frame—a 16 node bidirectional MIN

peak bandwidth of 40 megabytes per second over both byte-wide channels of each communication link.

The Vulcan switch chip contains 8 receiver modules and 8 transmitter modules, an unbuffered crossbar, and the central queue. All ports are one flit (one byte) wide. In the absence of contention, packet flits incur 5 cycles of latency cutting through the chip via the crossbar path.

In SP1 system, processor nodes are attached to the switching elements of the network via *Communication Adapters*.

In SP2 system, processor nodes are attached to the switching elements of the network via *Enhanced Communication Adapters*. This adapter incorporates an Intel i860 XR 64-bit microprocessor, 2KB input FIFO, 2KB output FIFO, and two DMA engines. The adapter carries communications coprocessing, data checking, and DMA between the Micro Channel and the two FIFO's.

The SP1 system used for these experiments has 32 POWER nodes operating at 62.5 MHz, with 32 KB instruction cache and 32 KB data cache.

The SP2 system used for these experiments has 32 POWER2 Thin nodes operating at 66.7 MHz, with 32 KB instruction cache and 64 KB data cache.

The IBM SP2 supports three message passing libraries, MPL [10], PMVe [11], and MPI. PVMe is IBM's implementation of the popular PVM [12] on SP2, the current version is 3.2.6. The IBM PVMe is compatible with PVM, but its internal structure is different. The IBM PVMe does not interface directly with the TCP/IP to perform data communication between processors. Instead, it interfaces with the Communication Subsystem (CSS); the communication software that runs on the High Performance Switch.

SP1/SP2 can be configured without the High Performance Switch, in this case communication is carried out using the Internet Protocol (ip) over slower interconnects like the Ethernet and FDDI networks.

2.3 Point-to-Point Communication

Th Point-to-point communication experiment is intended to measure the basic communication properties of a message-passing computer. In this experiment a message is sent from processor A to processor B. Processor B receives the message and immediately returns it back to processor A. The time of the complete trip is divided by two to get the Point-to-point Communication time.

Figure 2.2 shows the minimum Point-to-point communication time on the SP1, the SP2, and the SP2 when using the Internet Protocol (ip) over the Ethernet network.

The latency of short messages is 43 microseconds for the SP1, and 47 microseconds for the SP2. The transfer rate of large messages reaches 35.1 MBytes/sec in the SP2 (see figure 2.3), and it is only about 8.7 MBytes in the SP1. The transfer rate is found by dividing the message length by the minimum communication time. The latency of short messages for the SP2 using the Internet Protocol (ip) is 506 microseconds, and the transfer rate of large messages is 1.08 MBytes/sec.

Table 2.1 summarizes this data for the three cases, it gives the asymptotic latency in microseconds for small messages, and the transfer rate in MBytes/sec for large messages (1 MByte).

Configuration	Latency	Transfer Rate
IBM SP1	43	8.7
IBM SP2	47	35.1
IBM SP2 (ip)	506	1.08

Table 2.1: Latency and Transfer Rate for Point-to-point Communication

More detailed experiments were done to find out more about the Point-to-point com-

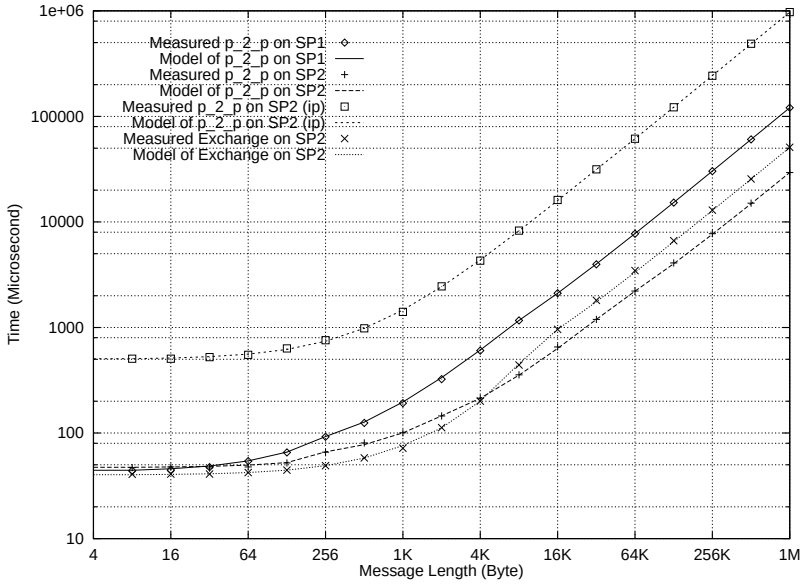

Figure 2.2: Point-to-point and Exchange Communication Times

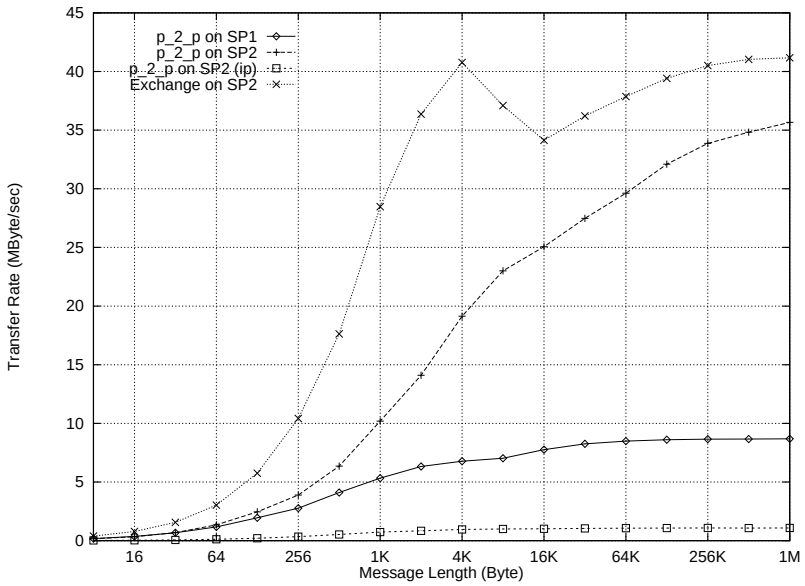

Figure 2.3: Point-to-point and Exchange Transfer Rates

munication time characteristics. The experiment described above was repeated for every message length in the range form 0 to 2000 bytes, the results are shown in figure 2.4. This experiment was repeated twice, the first time with two nodes from the same frame (Near nodes), and the second time with two nodes from different frames (Far nodes).

The results show that Point-to-point communication time is about 1 to 3 microseconds longer when the communication is done between far nodes. One can notice that there are discontinuities in the communication time at periodic sizes of the message length due to packetization. The discontinuities are at $217 + 232i$ Bytes; where $i \in \{0, 1, 2, \ldots\}$

The Point-to-point communication time often can be expressed by a few simple parameters [13]; r_∞ is the *asymptotic transfer rate* in MBytes/sec, and t_0 is the *asymptotic zero message length latency* in microseconds. The time (in microseconds) for Point-to-point communication as a function of n, $T_{pp}(n)$, is given by the following equation:-

$$T_{pp}(n) = t_0 + \frac{n}{r_\infty}$$

For small messages, the setup time t_0 is dominant, while for large messages the transfer time governed by r_∞ is dominant. The transfer rate, $r(n)$, is found by the following equation:-

$$r(n) = \frac{n}{T_{pp}(n)}$$

After measuring the Point-to-point communication time for different configurations, we have observed that there is no single set of r_∞ and t_0 that cover a wide range of n. This suggests splitting the n domain into regions, where each region has its own set of r_∞ and t_0. The resulting values for r_∞ and t_0 are shown below.

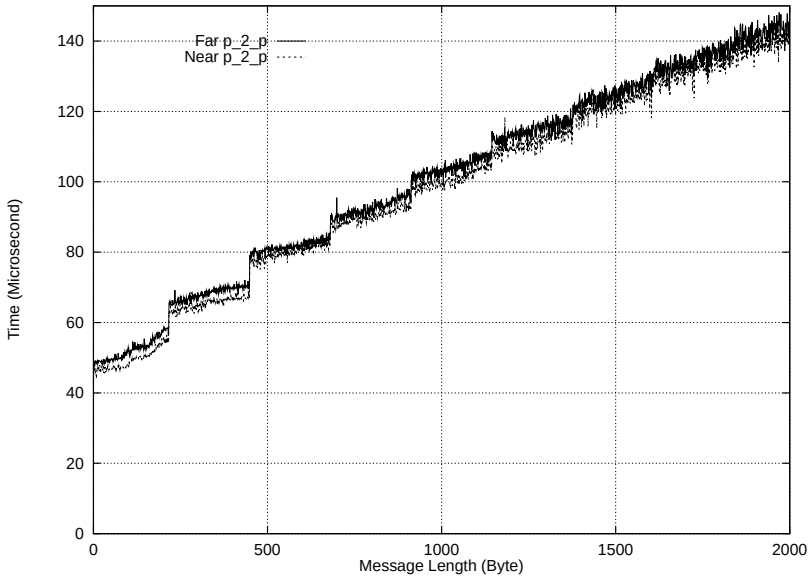

Figure 2.4: Point-to-point Communication time for the SP2, the effects of Packetization and Distance

SP1

The following are the parameters and regions for the Point-to-point Communication time on the SP1 using the MPL message-passing library.

$$t_0 = \begin{cases} 43 & n < 217 \\ 58 & 217 \leq n \leq 8\text{KB} \\ 228 & n \geq 8\text{KB} \end{cases}$$

$$r_\infty = \begin{cases} 5.56 & n < 217 \\ 7.39 & 217 \leq n \leq 8\text{KB} \\ 8.70 & n \geq 8\text{KB} \end{cases}$$

The boundary between the first and second regions is due to the limit on the maximum packet length (255 flits).

SP2

The following are the parameters and regions for the Point-to-point Communication time on the SP2 using the MPL message-passing library.

$$t_0 = \begin{cases} 47 & n < 217 \\ 55 & 217 \leq n \leq 2048 \\ 74 & 2048 < n < 64\text{KB} \\ 399 & n \geq 64\text{KB} \end{cases}$$

$$r_\infty = \begin{cases} 23.5 & n < 217 \\ 22.6 & 217 \leq n \leq 2048 \\ 29.3 & 2048 < n < 64\text{KB} \\ 36.2 & n \geq 64\text{KB} \end{cases}$$

The boundary between the second and the third regions is due to the size of the communication adapter's FIFO buffer (2 KBytes). Although the SP2 has slightly longer latency than the SP1, its transfer rate is about 4 times higher.

SP2 (ip)

The following are the parameters and regions for the Point-to-point Communication time on the SP2 using the MPL message-passing library with the Internet Protocol over the Ethernet.

$$t_0 = \begin{cases} 499 & n < 64\text{KB} \\ 746 & n \geq 64\text{KB} \end{cases}$$

$$r_\infty = \begin{cases} 1.06 & n < 64\text{KB} \\ 1.08 & n \geq 64\text{KB} \end{cases}$$

These parameters show clearly that the High Performance Switch has lower latency and its transfer rate is about 36 times higher than the Ethernet network.

2.4 Exchange Communication

This experiment is intended to measure the performance of the processor when it is using its communication link in both directions simultaneously. In this experiment we have two processors, each processor sends a message to the other processor and receives the message sent to it. The time needed to complete receiving one message and sending one message is the Exchange Communication time.

Figure 2.2 shows the minimum Exchange communication time on the SP2. Figure 2.3 shows the transfer rate for the Exchange communication. The transfer rate is found by dividing the message length by one half the minimum Exchange communication time. The transfer rate, $r(n)$, is found by the following equation:-

$$r(n) = \frac{2n}{T_{exch}(n)}$$

where

$$T_{exch}(n) = t_0 + \frac{n}{r_\infty}$$

The Exchange time for small messages is 41 microseconds, and the transfer rate for 1 MByte messages is 41.2 MBytes/sec.

The following are the parameters and regions for Exchange Communication time on the SP2 using the MPL message-passing library.

$$t_0 = \begin{cases} 40 & n < 4\text{KB} \\ -52 & 4\text{KB} \le n \le 16\text{KB} \\ 167 & n > 16\text{KB} \end{cases}$$

$$r_\infty = \begin{cases} 28.4 & n < 4\text{KB} \\ 16.2 & 4\text{KB} \le n \le 16\text{KB} \\ 20.7 & n > 16\text{KB} \end{cases}$$

When using the communication link in both directions in the Exchange communication we get about 17% higher transfer rate than when using it in one direction only as in the Point-to-point communication.

2.5 One-to-many Communication

This experiment is intended to measure the outbound performance of a message-passing computer. The sender processor sends p different messages to p different processors. The time taken by the sender processor to finish sending these messages is the One-to-many Communication time.

Figure 2.5 shows the average One-to-many communication time on the SP1 and the SP2. This data shows that the One-to-many Communication time is proportional to n and p, and the SP2 times are better than the SP1 times.

In One-to-many communication there are two variables, the message length, n, and the number of destination processors, p. So the models for this communication pattern are

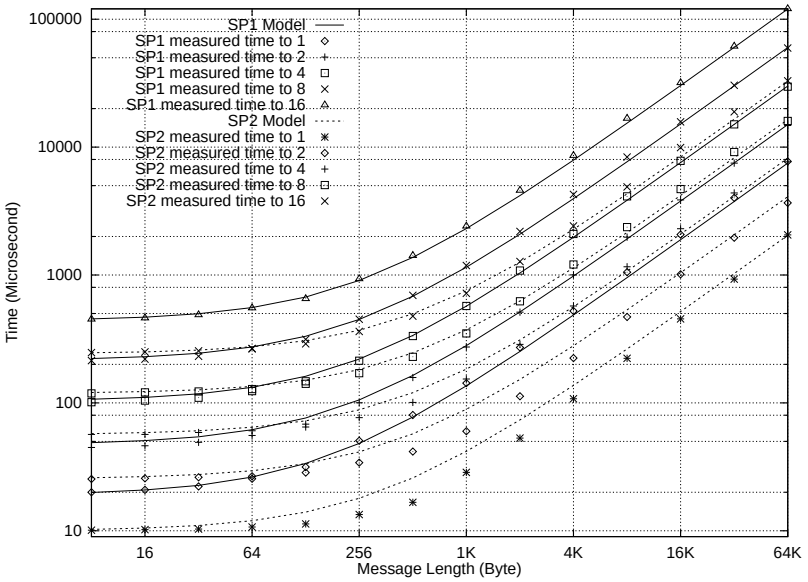

Figure 2.5: One-to-many Communication Time

functions of n and p, $T_{1m}(n,p)$. In order to have models that are natural and simple, we use the following model:-

$$T_{1m}(n,p) = t_{1m}(p) + \pi_{1m}(p)n$$

Here $t_{1m}(p)$ gives the *setup time* for p messages, and $\pi_{1m}(p)$ gives the *transfer time per byte* to p destinations. The following are the models for the two systems.

SP1

The model for the One-to-many Communication time on the SP1 using the MPL message-passing library is:-

$$T_{1m}(n,p) = (-9 + 28p) + (0.114p)n$$

Figure 2.5 shows the model and the measured data.

SP2

The model for the One-to-many Communication time on the SP2 using the MPL message-passing library is:-

$$T_{1m}(n,p) = (-5.5 + 15.5p) + (0.031p)n$$

Figure 2.5 shows the model and the measured data. The model shows a loose fit for small p at n about 1/2 KBytes. Nevertheless; the model can be made more accurate by splitting it into more regions as in the Point-to-point communication model. The SP2 transfer time is about 1/3 the SP1 transfer time.

2.6 Many-to-one Communication

This experiment is intended to measure the inbound performance of a message-passing computer. The receiver processor receives p different messages from p different processors. The time taken by the receiver processor to finish receiving these messages is the Many-to-one Communication time. Figure 2.6 shows the average Many-to-one communication time on the SP1 and the SP2, the SP2 times are better than the SP1 times.

In Many-to-one communication there are two variables, the message length, n, and the number of destination processors, p. So the models for this communication pattern are functions of n and p, $T_{m1}(n,p)$. In order to have models that are natural and simple, we use the following model:-

$$T_{m1}(n,p) = t_{m1}(p) + \pi_{m1}(p)n$$

Here $t_{m1}(p)$ gives the *setup time* for p messages, and $\pi_{m1}(p)$ gives the *transfer time per byte* from p sources. The following are the models for the two systems.

SP1

The model for the Many-to-one Communication time on the SP1 using the MPL message-passing library is:-

$$T_{m1}(n,p) = (-26 + 40\sqrt{p} + 7.6p) + (0.128 + 0.161 \log p)n$$

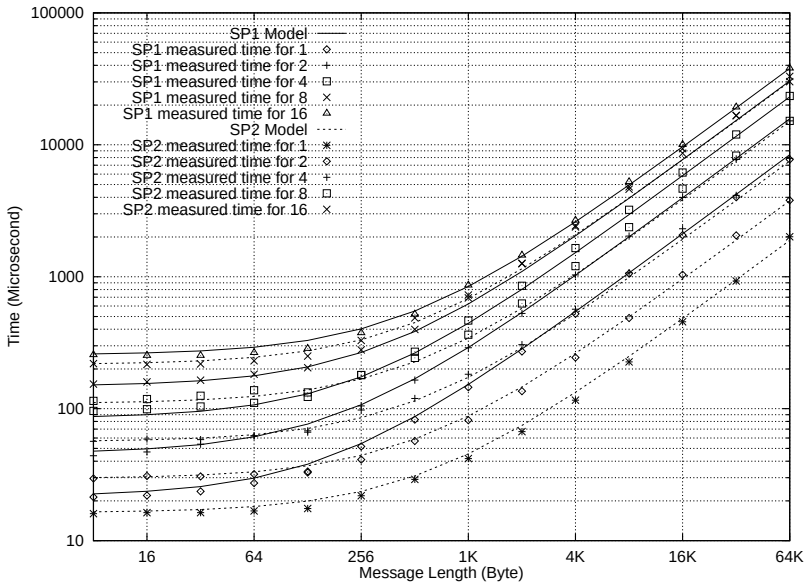

Figure 2.6: Many-to-one Communication Time

SP2

The model for the Many-to-one Communication time on the SP2 using the MPL message-passing library is:-

$$T_{m1}(n,p) = (3 + 13.3p) + (0.0285p)n$$

2.7 Many-to-many Communication

This experiment is intended to measure the total saturation bandwidth of a message-passing computer, and to find how this bandwidth scales with the number of processors. In this experiment each processor sends different messages of length n to the other $p-1$ processors, and receives $p-1$ messages. The time needed by a processor to send and receive its share of messages is the Many-to-many Communication time. Figure 2.7 shows the average Many-to-many communication time for SP1 and SP2.

In Many-to-many communication there are two variables, the message length, n, and the number of processors participating in this communication pattern, p. So the models for this communication pattern are functions of n and p, $T_{mm}(n,p)$. In order to have models that are natural and simple, we use the following model:-

$$T_{mm}(n,p) = t_{mm}(p) + \pi_{mm}(p)n$$

Here $t_{mm}(p)$ gives the *setup time* for sending $p-1$ messages and receiving $p-1$ messages, and $\pi_{mm}(p)$ gives the *transfer time per byte* with $p-1$ nodes. The following are the models for the two systems.

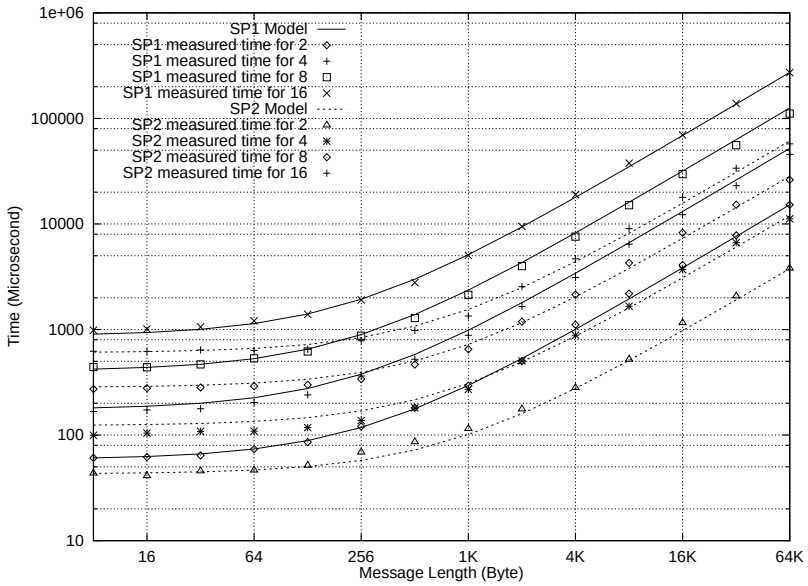

Figure 2.7: Many-to-many Communication Time

SP1

The model for the Many-to-many Communication time on the SP1 using the MPL message-passing library is:-

$$T_{mm}(n,p) = (59 + 58(p-2)) + (0.231 + 0.280(p-2))n$$

SP2

The model for the Many-to-many Communication time on the SP2 using the MPL message-passing library is:-

$$T_{mm}(n,p) = (43 + 40(p-2) + (0.057 + 0.062(p-2))n$$

The SP2 has smaller setup time and about one fifth the SP1's transfer time.

2.8 Broadcast

This experiment is intended to measure the Broadcast performance of a message-passing computer. The sender processor sends the *same* message to p different processors. The time taken by the sender processor to finish sending these messages is the Broadcast time. Figure 2.8 shows the average Broadcast time on the SP1 and the SP2.

In Broadcast communication there are two variables, the message length, n, and the number of destination processors, p. So the models for this communication pattern are functions of n and p, $T_{bc}(n,p)$. In order to have models that are natural and simple, we use the following model:-

$$T_{bc}(n,p) = t_{bc}(p) + \pi_{bc}(p)n$$

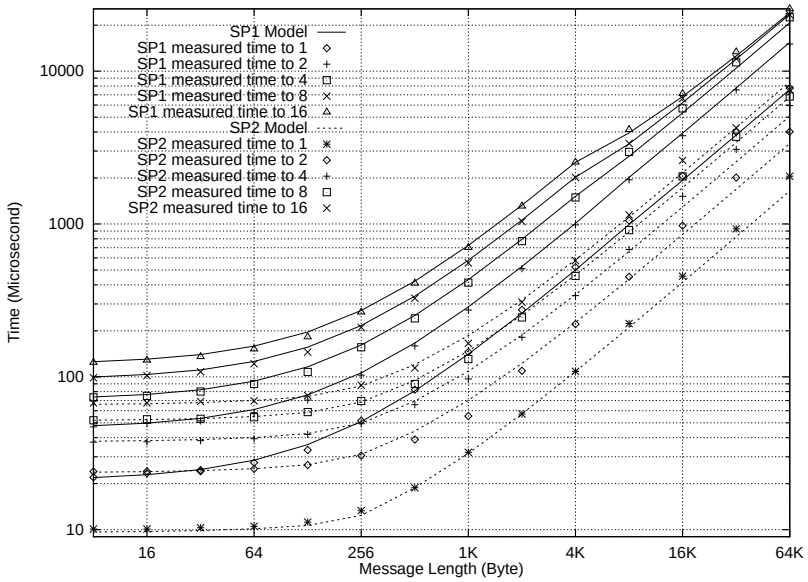

Figure 2.8: Broadcast Time

Here $t_{bc}(p)$ gives the *setup time* for p destinations, and $\pi_{bc}(p)$ gives the *transfer time per byte* to p destinations. The following are the models for the two systems.

SP1

The model for the Broadcast time on the SP1 using the MPL message-passing library is:

$$T_{bc}(n,p) = \begin{cases} (21+25D)+(0.117+0.118D)n & n \leq 4\text{KB} \\ (48p+1.2p^2)+(0.115+0.139D-0.02D^2)n & n > 4\text{KB} \end{cases}$$

where $D = \lfloor \log_2 p \rfloor$.

SP2

The model for the Broadcast time on the SP2 using the MPL message-passing library is:-

$$T_{bc}(n,p) = \begin{cases} (9.6+14D)+(0.0083+0.015D)n & n < 217 \\ (6+12D)+(0.025+0.026D)n & n \geq 217 \end{cases}$$

where $D = \lfloor \log_2 p \rfloor$.

2.9 Combine

This experiment is intended to measure the Combine performance of a message-passing computer. In Combine a reduction operation is applied on message data from, and the result is sent to, all participating processors. Figure 2.9 shows the average Combine time on the SP1 and the SP2. The reduction operation used in this experiment is double precision summation.

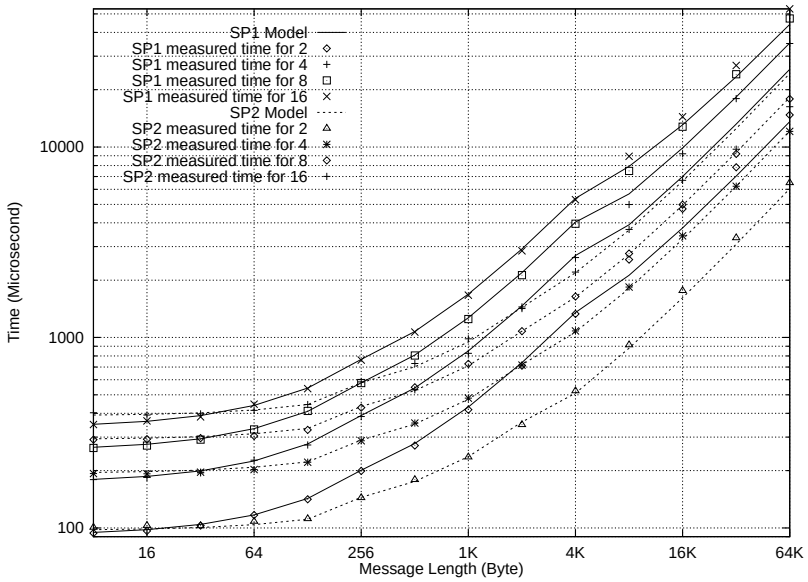

Figure 2.9: Combine Time

In Combine communication there are two variables, the message length, n, and the number of participating processors, p. So the models for this communication pattern are functions of n and p, $T_{bc}(n,p)$. In order to have models that are natural and simple, we use the following model:-

$$T_{cb}(n,p) = t_{cb}(p) + \pi_{cb}(p)n$$

Here $t_{cb}(p)$ gives the *setup time* for p processors, and $\pi_{cb}(p)$ gives the *combine time per byte* for p processors. The following are the models for the two systems.

SP1

The model for the Combine time on the SP1 using the MPL message-passing library is:-

$$T_{cb}(n,p) = \begin{cases} (9.5 + 82D) + (0.40D)n & n < 217 \\ (12 + 112D) + (0.30D)n & 217 \leq n \leq 4\text{KB} \\ (160 + 162p) + (-0.23 + 0.43\sqrt{D})n & n > 4\text{KB} \end{cases}$$

where $D = \lfloor \log_2 p \rfloor$.

SP2

The model for the Combine time on the SP2 using the MPL message-passing library is:-

$$T_{cb}(n,p) = \begin{cases} (97D) + (0.11D)n & n < 217 \\ (114D) + (0.12D)n & 217 \leq n \leq 2\text{KB} \\ (-50 + 191D) + (0.09D)n & n > 2\text{KB} \end{cases}$$

where $D = \lfloor \log_2 p \rfloor$.

2.10 Synchronization

This experiment measures the time to execute a barrier synchronization routine as a function of the number of processors taking part in the barrier.

Figure 2.10 shows the average Synchronization time for the two configurations. The SP1 and the SP2 times using the MPL message-passing library are similar.

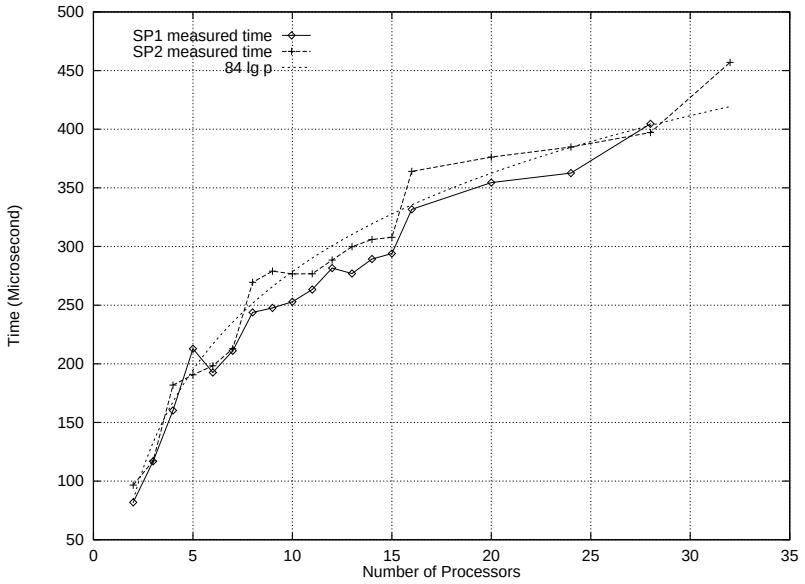

Figure 2.10: Synchronization Time

The model for the Synchronization time gives the time as function of the number of processors, p, involved in the synchronization barrier, $T_{sync}(p)$.

The data for the SP1 and the SP2 are similar and can be represented by the following model:-

$$T_{sync}(p) = 84 \log_2 p$$

This model implies an efficient implementation of the barrier, such good performance can be accomplished when the barrier is implemented using a binary tree method.

2.11 Conclusions

We have developed experiments to measure the communication performance on the IBM SP1 and the SP2. Each experiment is running and timing a program that do one communication pattern many times. The results of running these experiments enabled us to develop models for the common communication patterns. These models give the time of a communication pattern as functions of the message length, n, and the number of processors, p. The accuracy of these models is better than $\pm 8\%$, which is adequate for estimating execution times and tuning a message-passing application.

We have shown that it is possible to model the time of wide range of communication patterns by the following equation:-

$$T_{comm}(n,p) = t_{comm}(p) + \pi_{comm}(p)n$$

where $t_{comm}(p)$ is the *setup time*, and $\pi_{comm}(p)$ is the *transfer time per byte*.

The communication performance of the SP2 is better than that of the SP1. In Point-to-point communication the SP2 latency for short messages is a little bit longer than the SP1 latency, but the SP2 transfer rate for long messages is four times the SP1 transfer rate. In One-to-many communication the SP2 has shorter setup time and one fourth the transfer time. In Many-to-one communication the superiority of the SP2 is not as impressive. In Many-to-many communication the SP2 has shorter setup time and one fifth the transfer time. The SP1 and SP2 use efficient algorithms for implementing the Broadcast and Combine communications, and the SP2 performance is better than the SP1 performance. The two systems have similar Synchronization time.

These models are useful for programmers and engineers involved in writing high performance message-passing applications. We have successfully used these models to analyze and tune a Finite Element Application [4]. And we have used them to study the performance of six broadcast algorithms (see chapter 3).

These models can also be used to do comparisons between MPP's. We were able to conclude interesting results (see chapter 4) by developing performance models for the PVM message-passing library on the IBM SP2 and the Convex SPP-1000 [14].

Chapter 3

Broadcast Algorithms

3.1 Introduction

In message-passing applications, collective communication patterns can be time consuming. In this chapter we present a study for six broadcast algorithms implemented on the IBM SP2. The objective of this study is to find out the relative performance of these algorithms. This objective is achieved by developing a performance model for each algorithm. The developed models give the broadcast time as a function of the message length, n, and the number of participating processors, p. Given certain n and p these models can be evaluated to find out which algorithm has the best performance. We also give explanations for the relative performance of these algorithms according to the characteristics of the SP2 interconnection subsystem.

The IBM SP2 is a message-passing multi-computer, it uses an interconnection network that has a bisection bandwidth that scales linearly with the number of nodes, while maintaining a low communication latency. The broadcast algorithms presented in this chapter vary in complexity, and are chosen to exploit the SP2 interconnection network capabilities.

The algorithms are implemented by writing short FORTRAN programs using the MPL message-passing library routines. Each program calls a broadcast routine many times, and reports the minimum time, the average time, and the standard deviation. The measured time is the time needed to complete the broadcast operation globally; beginning at the start of the first activity by the sender and ending at the end of the last reception. To find this time experimentally, the broadcast is repeated p times. Processor 0 starts broadcasting, followed by the processor that receives the last, and so on until all the processors have broadcasted. The time for these p broadcasts is measured and divided by p to get the wanted *broadcast time*.

In the following sections we describe each of the six broadcast algorithms, describe their implementation on the SP2, present the measured timing data, develop their performance models, and explain the achieved performance. In section 3.8 we present a comparison for the performance of the six algorithms, finally, section 3.9 states the chapter conclusions.

3.2 MPL Broadcast (bc)

This algorithm simply calls the `mp_bcast` routine which is one of the MPL collective communication routines, these routines are optimized for doing collective communication. The `mp_bcast` is a blocking communication in which all the processors remain busy until the broadcast is completed. The implementation of this algorithm is straightforward; all the participating processors call the `mp_bcast` routine, the variable `source` specifies the sender processor, and the other processors become the receivers. The pseudo-code for this algorithm is as follows:-

```
begin
     call mp_bcast(buf, length, source, allgrp)
end
```

Figure 3.1 shows the measured **bc** minimum broadcast times for wide range of message lengths and varying number of processors. Roughly speaking, the **bc** broadcast time is proportional to n and $\log_2 p$.

IBM implemented the `mp_bcast` routine using a recursive doubling algorithm that uses low-level primitive sends and receives. Given the measured timing data for this routine and using curve-fitting techniques we found that the following equation gives a good estimate for the **bc** broadcast time:-

$$t_{bc}(n,p) = 23 + 27\log_2 p + \frac{n\log_2 p}{r_\infty}$$

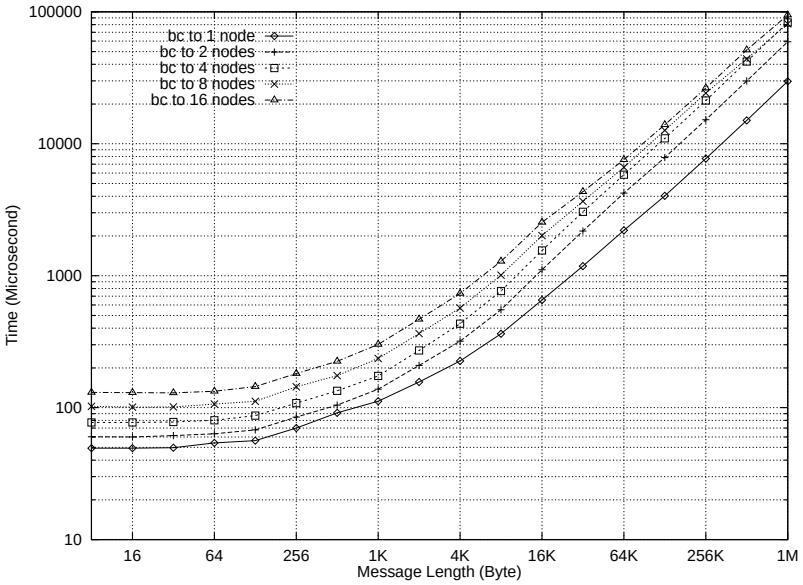

Figure 3.1: MPL Broadcast Time

3.3 One-to-many Broadcast (1m)

This is a naive algorithm, the sender sends to each of the other processors in turn, so it uses one communication link at each iteration. The pseudo-code for this algorithm is shown below, the sender processor calls the `mp_send` routine p-1 times to send the message to the p-1 receiver processors, and every receiver processor calls the `mp_brecv` routine to receive the message.

```
begin
     if (my_proc .eq.  source) then
        do j=1, nprocs-1
           dest = mod(my_proc+j, nprocs)
           call mp_send(buf, length, dest, msgtype, msgid1)
        end do
     else
        call mp_brecv(buf, length, source, msgtype, msgid2)
     end if
end
```

Figure 3.2 shows the measured **1m** minimum broadcast times for wide range of message lengths and varying number of processors. Roughly speaking, the **1m** broadcast time is proportional to n and p.

The broadcast time of this algorithm can be constructed by using the basic communication models that were developed in chapter 2. The **1m** broadcast time equals the time for the sender to do $p-2$ sends plus the time of the last Point-to-point communication, it is

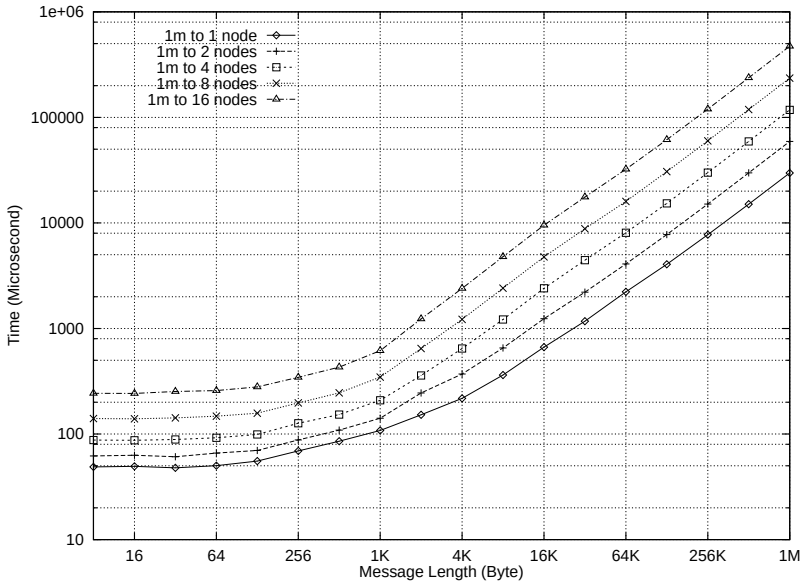

Figure 3.2: One-to-many Broadcast Time

approximated by the following equation:-

$$t_{1m}(n,p) = T_{1m}(n,p-2) + T_{pp}(n)$$

3.4 Recursive Doubling Broadcast (rd)

This algorithm was introduced for hypercube multi-computers [15], it has $\lceil \log_2 p \rceil$ stages, in stage i processors $0, 1, \ldots, 2^i - 1$ send to processors $2^i, 2^i + 1, \ldots, 2^{i+1} - 1$. The pseudo-code for this algorithm is as follows:

```
begin
     dim = int(lg(nprocs-1)) + 1
     me = mod(my_proc+nprocs-source, nprocs)
     do i=0, dim-1
        if (me .lt.  2**i) then
           dest = me + 2**i
           if (dest .lt.  nprocs) then
              dest = mod(dest+source, nprocs)
              call mp_bsend(buf, length, dest, msgtype, msgid1)
           end if
        else
           if (me .ge.  2**i .and.  me .lt.  2*2**i) then
              to = mod(me-2**i+source, nprocs)
              call mp_brecv(buf, length, to, msgtype, msgid2)
           end if
        end if
     end do
end
```

Figure 3.3 shows the measured **rd** minimum broadcast times for wide range of message lengths and varying number of processors. Roughly speaking, the **rd** broadcast time is proportional to n and $\log_2 p$.

In this algorithm the longest path has $\lceil \log_2 p \rceil$ hops of Point-to-point communication, so the broadcast time can be approximated by the following equation:-

$$t_{rd}(n,p) = \lceil \log_2 p \rceil T_{pp}(n)$$

3.5 Pipelined Recursive Doubling Broadcast (prd)

This algorithm is similar to the Recursive Doubling Broadcast, but in **prd** the message is broken into smaller *parts*, and the routine is repeated until all the parts are broadcasted. By doing experiments with different part sizes, we found that an 8 KBytes part size gives the best performance.

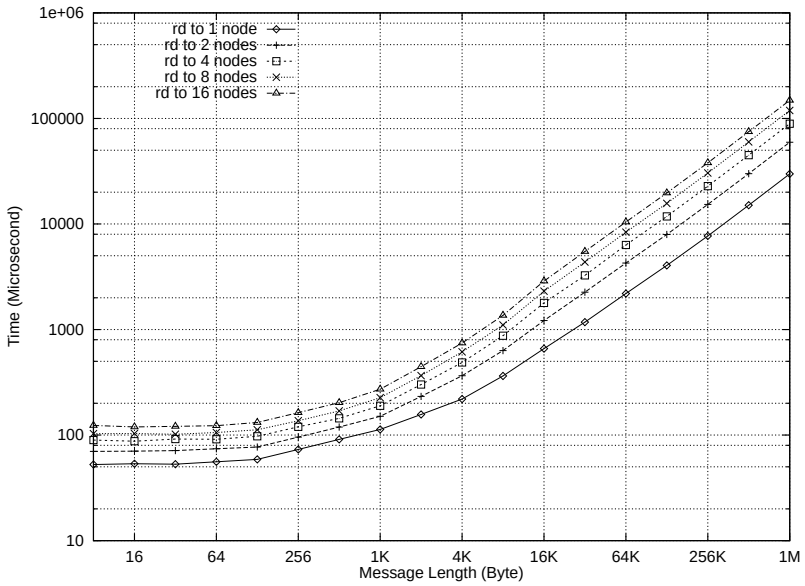

Figure 3.3: Recursive Doubling Broadcast Time

Figure 3.4 shows the measured **prd** minimum broadcast times for wide range of message lengths and varying number of processors. The measured broadcast times are a little bit higher than the **rd** broadcast times.

The SP2 communication links are bidirectional, so an SP2 node is capable of receiving and sending at the same time. In Point-to-point communication a link is used in one direction at a time, leading to r_∞ about 35 MBytes/sec. With bidirectional communication we can achieve higher transfer rates. Figure 2.3 shows that we can get about 41 MBytes/sec transfer rate in the Exchange communication which uses bidirectional communication.

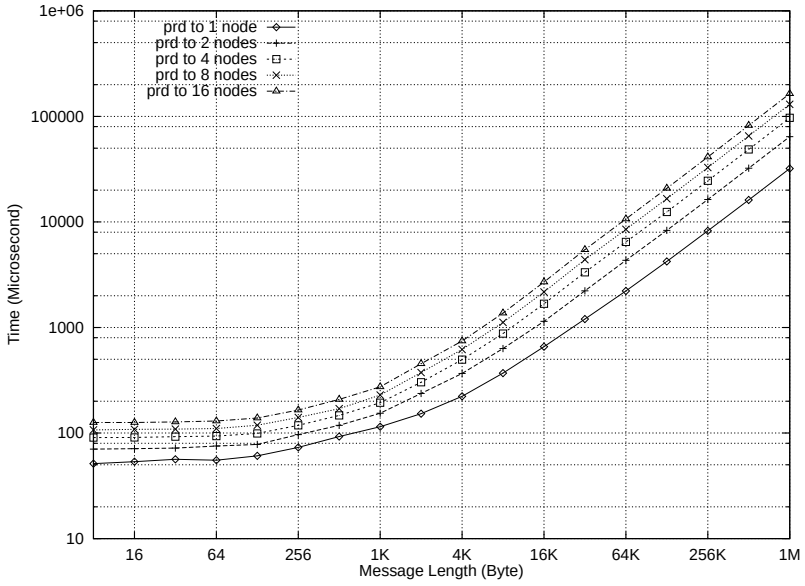

Figure 3.4: Pipelined Recursive Doubling Broadcast Time

The **prd** broadcast time is higher than the **rd** broadcast time because it fails to exploits this feature effectively due to the fact that a node receives 1 part and sends $\theta(\log_2 p)$ parts, and the gain in the partial usage of bidirectional communication does not compensate for the overhead of splitting the message into smaller parts. The broadcast time for this algorithm is approximated by the following equation:-

$$t_{prd}(n,p) = (\frac{n}{8192})\lceil \log_2 p \rceil T_{pp}(8192)$$

3.6 Binary Tree Broadcast (bt)

In this algorithm the processors are arranged in a logical binary tree, each processor receives from its parent (if it has one) and sends to its children (if it has any). The pseudo-code for this algorithm is as follows:-

```
begin
     levels = int(lg(nprocs))
     me = mod(my_proc+nprocs-source, nprocs)
     mylevel = int(lg(me+1))
     lc = 2*me + 1
     rc = 2*me + 2
     p = int((me-1)/2)
     if (me .ne.  0) then
        to = mod(p+source, nprocs)
        call mp_brecv(buf, length, to, msgtype, msgid2)
     end if
     if (lc .lt.  nprocs) then
        dest = mod(lc+source, nprocs)
        call mp_bsend(buf, length, dest, msgtype, msgid1)
     end if
     if (rc .lt.  nprocs) then
        dest = mod(rc+source, nprocs)
        call mp_bsend(buf, length, dest, msgtype, msgid1)
     end if
end
```

Figure 3.5 shows the measured **bt** minimum broadcast times for wide range of message lengths and varying number of processors. Roughly speaking, the **bt** broadcast time is proportional to n and $\log_2 p$.

In this algorithm there are $\lfloor \log_2 p \rfloor$ levels, a node receives from its parent, sends to its left child, then sends to its right child, the longest path is the one along the rightmost path, hence the broadcast time of this algorithm can be given by the following equation:-

$$t_{bt}(n,p) = \lfloor \log_2 p \rfloor \{T_{1m}(n,1) + T_{pp}(n)\}$$

One advantage of this algorithm is that a node needs to do at most 1 receive and 2 sends, resulting in a limited involvement in the broadcast process.

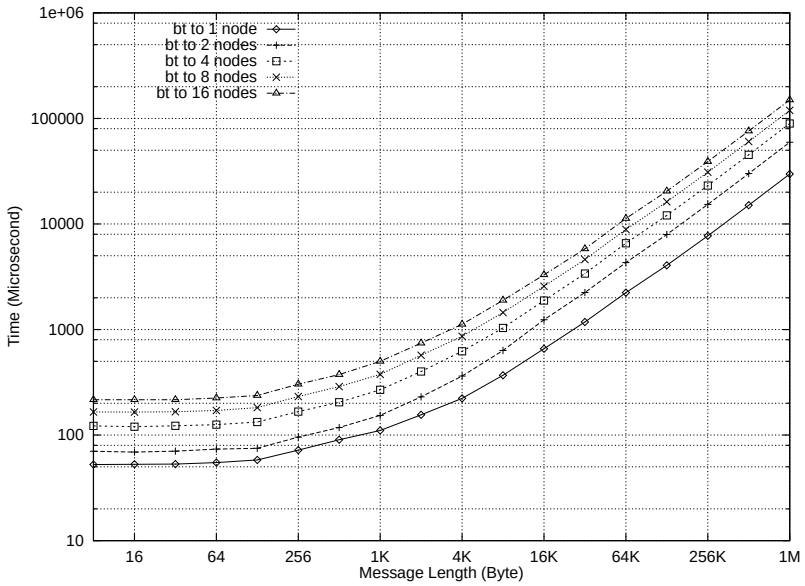

Figure 3.5: Binary Tree Broadcast Time

3.7 Pipelined Binary Tree Broadcast (pbt)

This algorithm is similar to the Binary Tree Broadcast, but in **pbt** the message is broken into smaller *parts*, and the routine is repeated until all the parts are broadcasted. By doing experiments with different part sizes, we found that an 8 KBytes part size gives the best performance.

Figure 3.6 shows the measured **pbt** minimum broadcast times for wide range of message lengths and varying number of processors. The measured broadcast times are similar to the

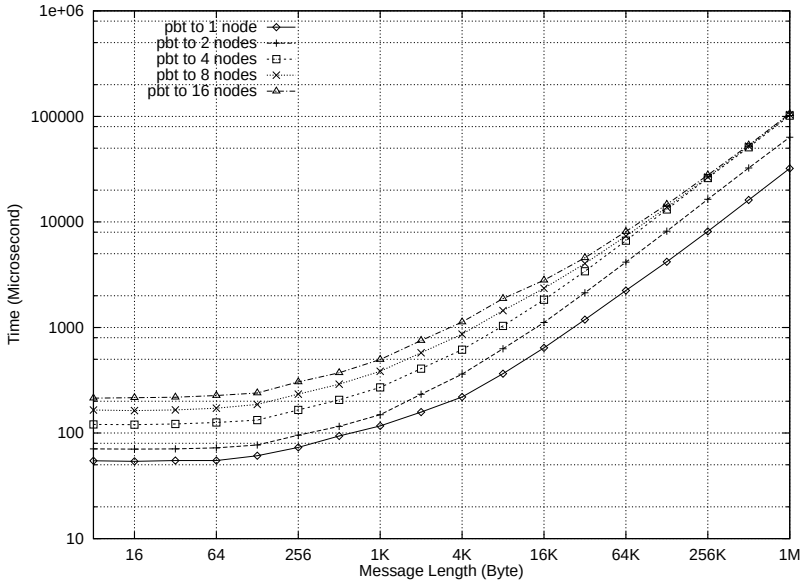

Figure 3.6: Pipelined Binary Tree Broadcast Time

rd broadcast times, but there is an improvement for $n > 8$KB.

The time of this algorithm is the time needed to transfer $n/8192$ parts in a Point-to-point communication plus the time taken for the last part to travel through the length of the tree, hence the time can be approximated by the following equation:-

$$t_{pbt}(n,p) = \{(\frac{n}{8192}) - 1\}T_{1m}(8192,2) + \lfloor \log_2 p \rfloor \{T_{1m}(8192,1) + T_{pp}(8192)\}$$

3.8 Comparison

Figures 3.7, 3.8, and 3.9 show comparisons for the performance of the six broadcast algorithms on the SP2. Figure 3.7 shows the broadcast times for short message (16 Bytes), figure 3.8 shows the broadcast times for medium message (8 KBytes), and figure 3.9 shows the broadcast times for long message (1 MByte). For $n \leq 8192$ the performance of **prd** is similar to **rd**, and the performance of **pbt** is similar to **bt**. In addition to the broadcast times of the six algorithms, the figures show the busy time of the sender processor in **bt**. The **bt** sender busy time is shown in order to highlight the **bt** feature where the processors are less involved in the broadcast operation, notice how the **bt** sender time is constant for three or more processors.

All the figures show that **bc** has the lowest broadcast time almost in all n and p ranges. The general shape of the **bc** time curve is similar to **rd** curve for short and medium messages, and is similar to the **pbt** curve for long messages. The marginal superiority of **bc** over **rd** for short and medium messages, and its marginal superiority over **pbt** for long messages can be attributed to the fact that it is implemented using lower level constructs, and better usage of the communication buffers.

For short messages the **rd** and **bt** broadcast times are similar, and are higher than the broadcast time of **1m** when the number of processors is less than 12. This is because the

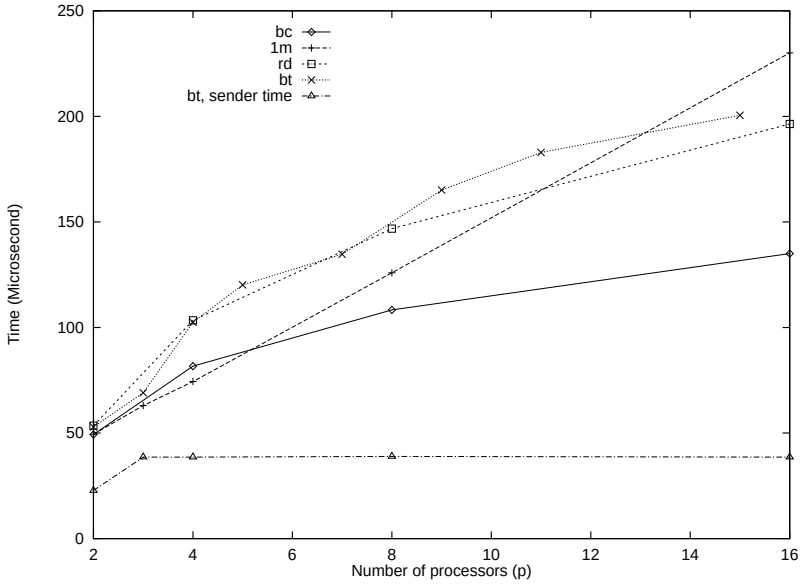

Figure 3.7: Broadcast Time comparison for n=16 Bytes

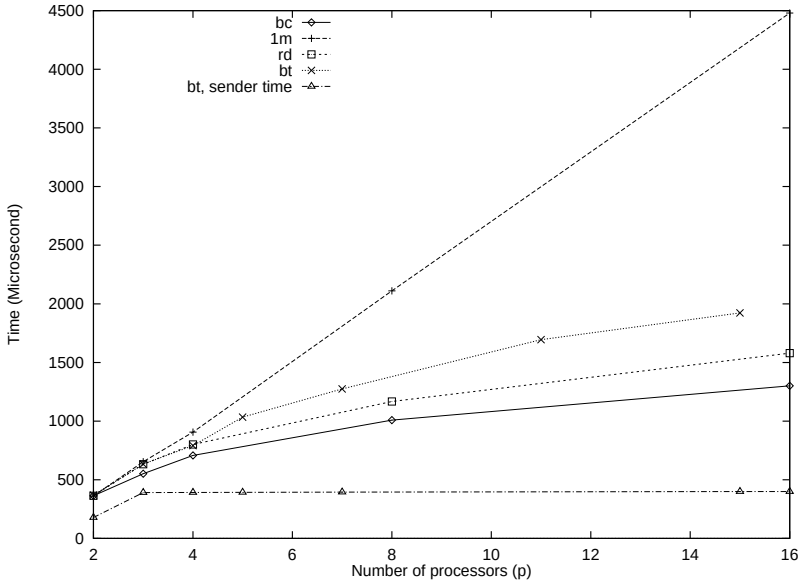

Figure 3.8: Broadcast Time comparison for n=8 KBytes

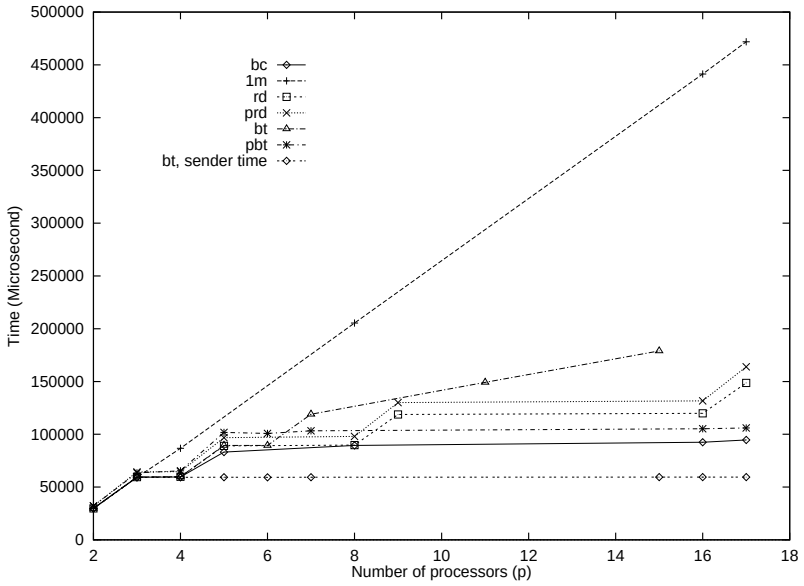

Figure 3.9: Broadcast Time comparison for n=1 MByte

sender communication adapter in **1m** is usually not saturated for short messages, so it can fire several short messages to several destinations fast. Contrast to **rd** and **bt**, in **1m** there is no dependence on the one-way trip time before starting a new send.

For medium messages the communication adapter in **1m** becomes saturated and **1m** has the worst performance. **rd** is better than **bt** because it uses more communications links simultaneously.

For long messages, **1m** remains the worst and **rd** remains better than **bt**. **prd** does not have better performance than **rd** because the overhead of sending many small parts is bigger than the gain of using bidirectional links. But **pbt** does has better performance than **bt**, it even has better performance than **rd** when the number of processors is bigger than 8.

3.9 Conclusions

In this chapter we have presented six broadcast algorithms, developed their performance models, analyzed the achieved performance, and given explanations.

We have observed that depending on n and p, one of {**1m**, **rd**, or **pbt**} is optimal. This suggests that the developed models can be used by a hyper-algorithm that contains these three algorithm and finds (during run time) which is the optimal algorithm and use it.

This approach of modeling the communication times of different algorithms can be used for tuning and optimizing other time-consuming communication problems, such as the collective communication patterns with medium and large messages.

The MPL broadcast routine, `mp_bcast`, is very efficient, it should be used whenever possible. But when there is a need for partial broadcast (to subset of the processors), then one of the above three routines can be used. Also if we are concerned about the time the processors spend in the broadcast operation then we might consider using **bt**.

The analysis done in this chapter can guide in implementing efficient broadcast algorithms for message-passing libraries that do not have collective communication routines, like the PVM message-passing library.

Chapter 4

PVM Comparison

4.1 Introduction

PVM is a popular message-passing library, it is available on many multicomputer systems. PVM stands for *Parallel Virtual Machine* [12]. It is a software package that allows heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource (a virtual machine). PVM consists of two parts: a *Daemon* process, and a *User Library* that contains routines for initiating processes on other machines, for communication between processes, and for changing the configuration of the virtual machine. The daemon coordinates between the tasks on the virtual machine.

In this chapter we develop performance models for the main communication patterns for the PVM message-passing library on the IBM SP2 (a Distributed Memory MPP) and the Convex SPP-1000 (a Shared Memory MPP). We also make a comparison between the communication performance of the two systems, and explain the achieved performance by the organization of the two systems.

We have developed experiments to measure the communication times on the IBM SP2 and the Convex SPP-1000 using the PVM message-passing library. Each experiment is timing a simple FORTRAN loop that do one communication pattern many times. We have studied the following communication patterns: Point-to-point, One-to-many, Many-to-many, and Broadcast. Also we have studied the Synchronization time. The gathered data from these experiments were used to develop the performance models, and to compare between the performance of the two systems.

A program written to carry out one of the timing experiments calls a routine that do a communication pattern many times, and reports the minimum time, the average time, and the standard deviation. Each program is executed on varying number of processors, p, and varying message lengths, n.

Section 4.2 describes the Convex SPP-1000 and its PVM implementation. The following sections describe the experiments for communication patterns, present the measured data, develop the performance models, and do comparisons for the performance of the two systems. Finally, section 4.9 states the conclusions of this chapter.

4.2 Convex SPP-1000

An SPP-1000 system (also called the Convex Exemplar) consists of 1 to 16 *Hypernodes* [14]. Each hypernode contains 4 *Functional Blocks*, each functional block contains 1 or 2 HP PA-RISC 7100 processors, memory, and some control devices, see figures 4.1 and 4.2. The functional blocks communicate across the hypernodes via four CTI (Convex Toroidal Interconnect) rings.

The CPUs have direct access to their own combined instruction and data cache, the size of this cache is 1 MByte and is located one clock away from the CPU, it is "direct-to-virtual-memory" mapped. The CPUs of the functional block communicate with the rest of the machine through the CPU *agent*. The Memory has two banks, that can be configured into three logical sections, *hypernode local*, *subcomplex global*, and *interconnect cache* (for holding copies of any off-hypernode data referenced the hypernode processors). The Convex Coherent Memory Controller (CCMC) provides the interface between the memory and the rest of the machine.

On the Exemplar systems, processes run on virtual machines called subcomplexes, which are arbitrary collections of processors.

Physical memory pages are interleaved across the memory banks in each functional block by interconnect cache lines. Contiguous interconnect cache lines are assigned in round robin fashion, first to the even bank, then to the odd. A processor cache line is 32 bytes wide, whereas the interconnect cache lines are 64 bytes wide, containing a pair of processor cache lines.

The Convex SPP-1000 supports ConvexPVM message-passing library, in this PVM implementation the data transfer is done through a shared buffer pool and lock primitives in the memory refered to as the Shared Memory Interconnect (SMI). Messages between two nodes use direct routing, while messages destined for another host go to the ConvexPVM

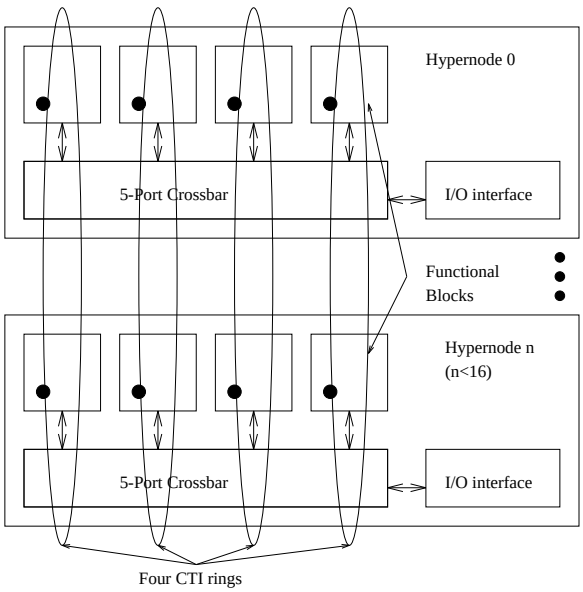

Figure 4.1: Convex Exemplar Hypernodes

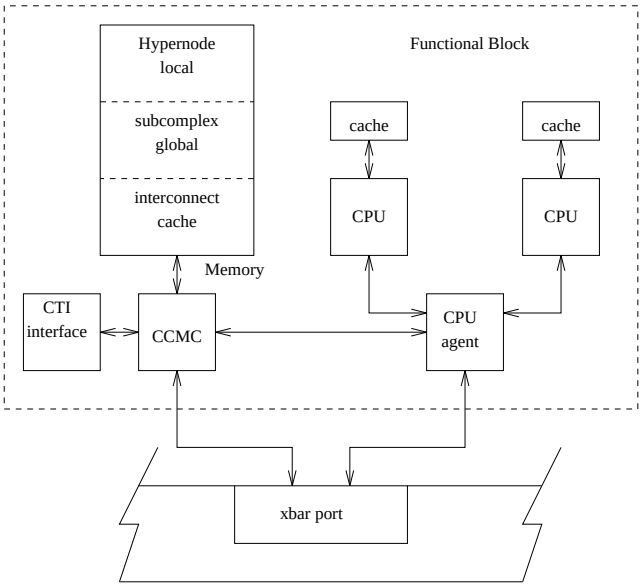

Figure 4.2: Convex Exemplar Functional Block

daemon for further routing, the ConvexPVM daemon should be running on one of the SPP-1000 processors. Intra-node communication between PVM clients and/or the daemon is done through the shared buffer. Inter-node communication between PVM tasks is done via a kernel CTI messaging library. Communication between a ConvexPVM client and a remote PVM daemon is done via the FDDI.

Our system has 4 hypernodes, with a total of 32 CPUs, and each hypernode is configured in a different subcomplex. However, this system is new and unstable and its configuration keeps changing. When the experiments described in this chapter were carried out, the CTI messaging was not working, so our experiments were possible only on a single hypernode for a maximum of 8 CPU's.

4.3 Point-to-point Communication

This experiment is intended to measure the basic communication properties of a message-passing computer. A message is sent from processor A to processor B. Processor B receives the message and immediately returns it back to processor A. The time of the complete trip is divided by two to get the Point-to-point Communication time.

Figure 4.3 shows the minimum Point-to-point communication time for the following three configurations:-

1. The IBM SP2 using the MPL message-passing library.
2. The IBM SP2 using the PVMe message-passing library.
3. The Convex SPP-1000 using the ConvexPVM message-passing library.

In the SP2 with MPL the latency for short messages is 43 microseconds, and the transfer rate for large messages reaches 35.1 MBytes/sec (see figure 4.4). With PVM, the IBM SP2

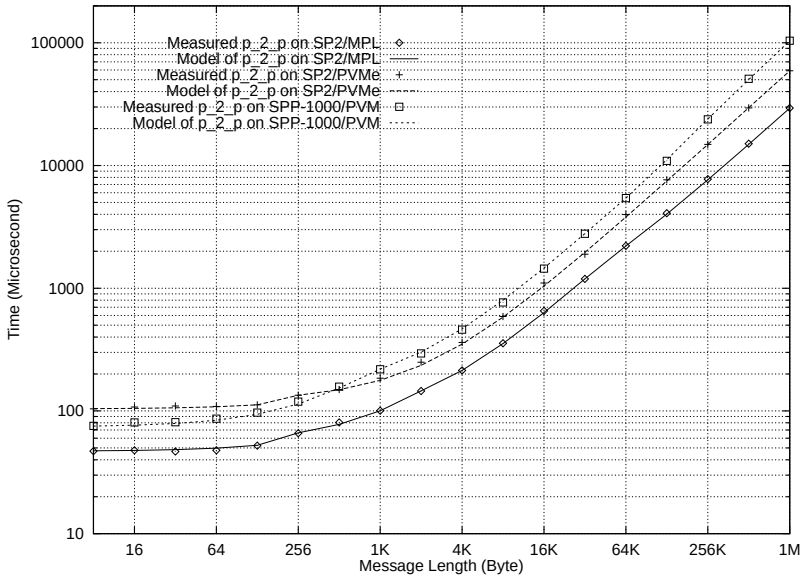

Figure 4.3: Point-to-point Communication Time

Configuration	Latency	Transfer Rate
IBM SP2/MPL	47	35.1
IBM SP2/PVMe	104	17.7
Convex SPP-1000	74	10.1

Table 4.1: Latency and Transfer Rate for Point-to-point Communication

has 104 microseconds latency, and the Convex SPP-1000 has 74 microseconds. The transfer rate for large messages in the SP2 is higher (17.7 MBytes/sec) than the SPP-1000 (10.1 MBytes/sec). In the SP2 the latency of short messages with PVMe is more than twice the latency with MPL, and the transfer rate with PVMe is about one half the MPL transfer rate. Since the SPP-1000 has shorter latency, it has better performance for short messages, but because it has lower transfer rate, its performance become worse starting after n=512 as shown in figure 4.3.

Table 4.1 summarizes the data for the three configurations, it gives the asymptotic latency in microseconds for small messages, and the transfer rate in MBytes/sec for large messages (1 Mbyte).

In cheater 2 we have shown that the Point-to-point communication can be approximated by the following equation:-

$$T_{pp}(n) = t_0 + \frac{n}{r_\infty}$$

In the following analysis we give the values for r_∞ and t_0 and the ranges where these parameters are valid.

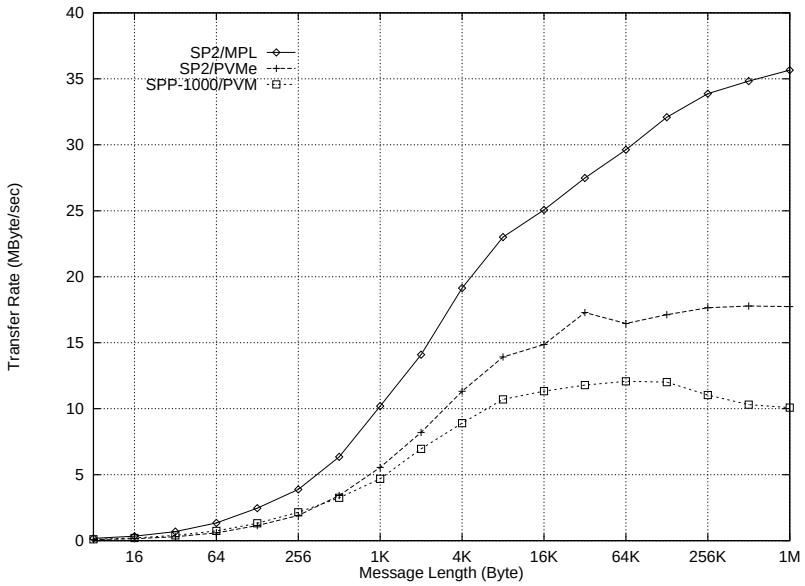

Figure 4.4: Point-to-point Transfer Rate

SP2/PVMe

The following are the parameters and regions for Point-to-point Communication time on the SP2 using the PVMe message-passing library.

$$t_0 = \begin{cases} 104 & n < 217 \\ 120 & n \geq 217 \end{cases}$$

$$r_\infty = \begin{cases} 15.9 & n < 217 \\ 17.8 & n \geq 217 \end{cases}$$

The boundary between the first and second regions is due to the limit on the maximum packet length (255 flits). One can observe an abrupt change for messages with n about 32 KBytes. Such changes can be attributed to changes in the message handling mechanisms of the message-passing library routines.

SP-1000/ConvexPVM

The following are the parameters and regions for Point-to-point Communication time on the SPP-1000 using the ConvexPVM message-passing library.

$$t_0 = \begin{cases} 74 & n \leq 512 \\ 136 & 512 < n \leq 64\text{KB} \end{cases}$$

$$r_\infty = \begin{cases} 6.44 & n \leq 512 \\ 12.38 & 512 < n \leq 64\text{KB} \end{cases}$$

For $n > 64$KB the simple model does not work, using curve-fitting techniques we found that the following equation approximates the Point-to-point communication time in this range:-

$$T_{pp}(n) = \frac{n}{9.39} - 7.78\sqrt{n}$$

Note the decrease in the transfer rate for $n > 64$KB as shown in figure 4.4, this behavior is due to the operating system interrupts that occur every 10 milliseconds.

4.4 One-to-many Communication

This experiment is intended to measure the outbound performance of a message-passing computer. The sender processor sends p different messages to p different processors. The time taken by the sender processor to finish sending these messages is the One-to-many Communication time.

Figure 4.5 shows the minimum One-to-many communication time on the SP2 using PVMe and on the SPP-1000 using ConvexPVM. This data shows that the One-to-many Communication time is proportional to n and p. The SP2 time has sudden increase at $n = 16$KB. The SP2 time is better than the SPP-1000 time for short and medium messages, and the SPP-1000 time is better than the SP2 time for long messages.

In One-to-many communication there are two variables, the message length, n, and the number of destination processors, p. So the models for this communication pattern are functions of n and p, $T_{1m}(n, p)$. In order to have models that are natural and simple, we use the following model:-

$$T_{1m}(n, p) = t_{1m}(p) + \pi_{1m}(p)n$$

Here $t_{1m}(p)$ gives the *setup time* for p messages, and $\pi_{1m}(p)$ gives the *transfer time per byte* to p destinations. The following are the models for the two systems.

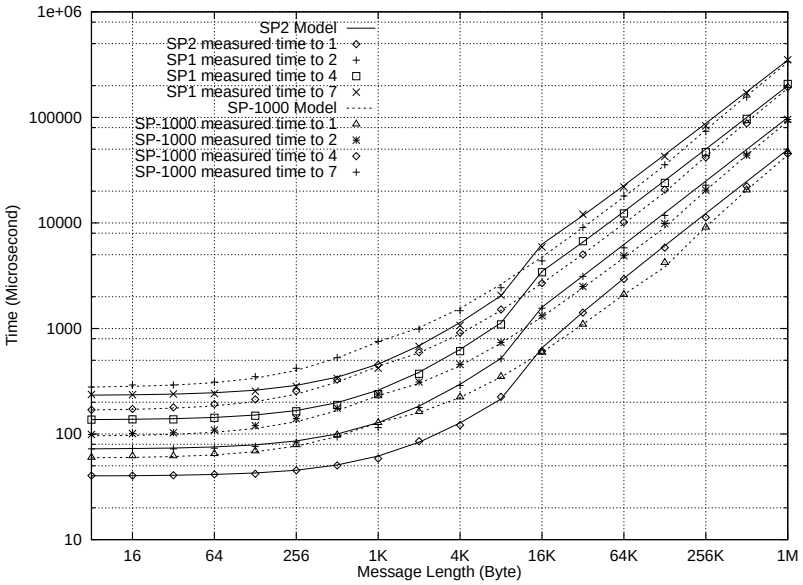

Figure 4.5: One-to-many Communication Time

SP2/PVMe

The model for the One-to-many Communication time on the SP2 using the PVMe message-passing library is:-

$$T_{1m}(n,p) = \begin{cases} (8+32p) + (\frac{p}{30} - 0.012)n & n \leq 8\text{KB} \\ (163p - 285) + (\frac{p}{21.1})n & n > 8\text{KB} \end{cases}$$

Figure 4.5 shows the model and the measured data.

SPP-1000/ConvexPVM

The model for the One-to-many Communication time on the SPP-1000 using the Convex-PVM message-passing library is:-

$$T_{1m}(n,p) = \begin{cases} (23+36p) + (\frac{p}{14.3})n & n \leq 512 \\ (33+64p) + (\frac{p}{26} - 0.008)n & 512 < n \leq 64\text{KB} \\ \frac{pn}{20.4} - (3.9+3.3p)\sqrt{n} & n > 64\text{KB} \end{cases}$$

Figure 4.5 shows the model and the measured data.

4.5 Many-to-one Communication

This test is usually intended to measure the inbound performance of a message-passing computer. The receiver processor receives p different messages from p different processors. The time taken by the receiver processor to finish receiving these messages is the Many-to-one Communication time.

This test is hard to implement accurately because the messages should be ready for reception at the receiver side at the moment the receiver starts to receive, also we need to separate the reception time from the wait and travel times.

4.6 Many-to-many Communication

This experiment is intended to measure the total saturation bandwidth of a message-passing computer, and to find how this bandwidth scales with the number of processors. In this experiment each processor sends different messages of length n to the other $p-1$ processors, and receives $p-1$ messages. The time needed by a processor to send and receive its share of messages is the Many-to-many Communication time.

Figure 4.6 shows the average Many-to-many communication time on the SP2 using PVMe and on the SPP-1000 using ConvexPVM. This data shows that the SP2 has good scalability, and the SPP-1000 has a dramatic increase in the time when the number of processors becomes 6 or more. The SPP-1000 times are generally better for short messages and few processors.

In Many-to-many communication there are two variables, the message length, n, and the number of processors participating in this communication pattern, p. So the models for this

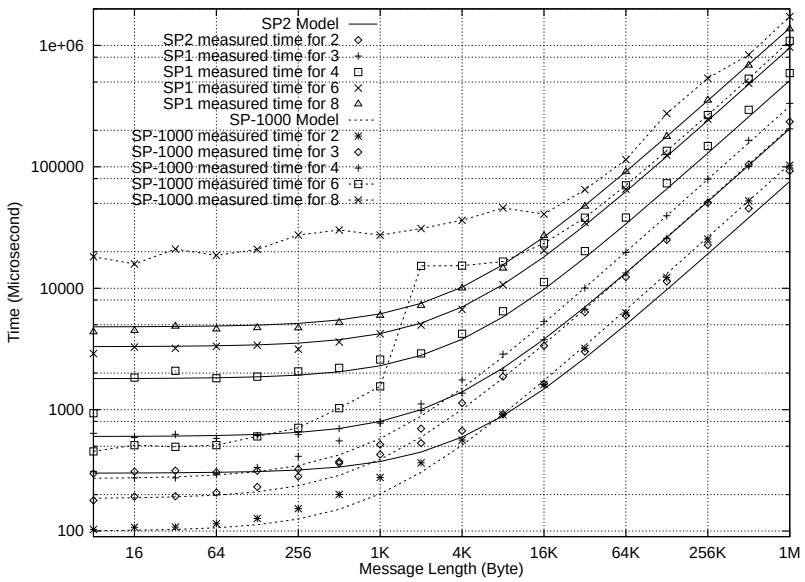

Figure 4.6: Many-to-many Communication Time

communication pattern are functions of n and p, $T_{mm}(n,p)$. In order to have models that are natural and simple, we use the following model:-

$$T_{mm}(n,p) = t_{mm}(p) + \pi_{mm}(p)n$$

Here $t_{mm}(p)$ gives the *setup time* for sending $p-1$ messages and receiving $p-1$ messages, and $\pi_{mm}(p)$ gives the *transfer time per byte* with $p-1$ nodes. The following are the models for the two systems.

SP2/PVMe

The model for the Many-to-many Communication time on the SP2 using the PVMe message-passing library is:-

$$T_{mm}(n,p) = \begin{cases} 600 + n/5.12 & \text{for } p = 3 \\ (300 + 750(p-2)) + (0.072 + \frac{p-2}{4.8})n & \text{otherwise} \end{cases}$$

Figure 4.6 shows the model and the measured data, note that we are using one model for all n, this is a good approximation, and even better models can be accomplished when having different parameters for different n regions as in the previous sections.

SPP-1000/ConvexPVM

The model for the Many-to-many Communication time on the SPP-1000 using the Convex-PVM message-passing library is:-

$$T_{mm}(n,p) \cong \begin{cases} (100 + 85(p-2)) + (\frac{p-1}{10})n & p \leq 4 \\ \text{Unscalable} & p > 4 \end{cases}$$

Figure 4.6 shows the application of this model to the data for p less than 5. This data shows the problem in the scalability of the SPP-1000 in the Many-to-many communication. When the number of processors is increased from 4 to 8, the latency for short messages increases by a factor of 128. This implies that there is a congestion on the Shared Memory Interconnect.

4.7 Broadcast

This experiment is intended to measure the broadcast performance of a message-passing computer. The sender processor sends the same message to p different processors. The time taken by the sender processor to finish sending these messages is the Broadcast time.

Figure 4.7 shows the minimum broadcast time on the SP2 using PVMe and on the SPP-1000 using ConvexPVM. This data shows that the Broadcast Communication time is proportional to n and p on the SP2 and proportional to n only on the SPP-1000. The SP2 time is similar to its time in the One-to-many Communication. The SPP-1000 time seems to has less dependence on the number of processors.

In Broadcast communication there are two variables, the message length, n, and the number of destination processors, p. So the models for this communication pattern are functions of n and p, $T_{bc}(n,p)$. In order to have models that are natural and simple, we use the following model:-

$$T_{bc}(n,p) = t_{bc}(p) + \pi_{bc}(p)n$$

Here $T_{bc}(p)$ gives the *setup time* for p destinations, and $\pi_{bc}(p)$ gives the *transfer time per byte* to p destinations. The following are the models for the two systems.

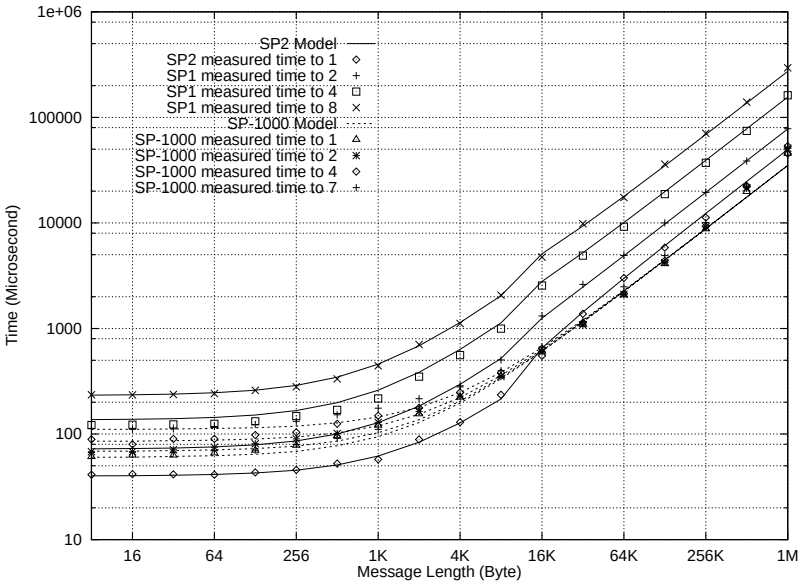

Figure 4.7: Broadcast Time

SP2/PVMe

The measured data for the Broadcast is very similar to the One-to-many communication data, so the models are similar. This implies that in the PVMe implementation there is no special handling for Broadcast.

SPP-1000/ConvexPVM

The model for the Broadcast Communication time on the SPP-1000 using the ConvexPVM message-passing library is:-

$$T_{bc}(n,p) = (52 + 8.4p) + \frac{n}{23}$$

Figure 4.7 shows the model and the measured data, note that we are using one model for all n, this is a good approximation, and even better models can be accomplished when having different parameters for different n regions as in the previous sections. The model shown implies that in the ConvexPVM implementation the data is copied to the SMI shared buffer pool once, and the p destination processors are notified.

4.8 Synchronization

This experiment measures the time to execute a barrier synchronization routine while varying the number of processors taking part in this barrier.

Figure 4.8 shows the minimum Synchronization time for the three configurations listed in section 4.3. The SPP-1000 time is the smallest but tends to increase sharply as p increases. The SP2 time using the MPL message-passing library is much smaller than its time when using the PVMe message-passing library.

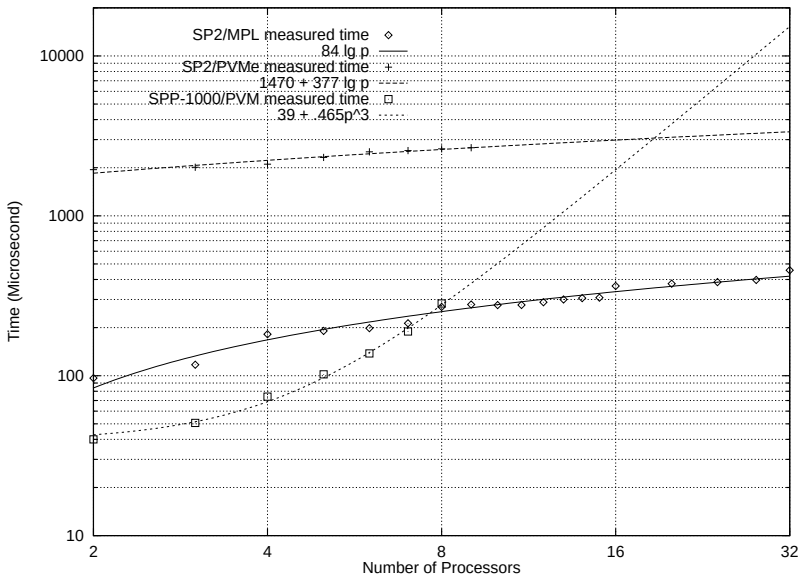

Figure 4.8: Synchronization Time

The model for the Synchronization time gives the time as function of the number of processors, p, involved in the Synchronization, $T_{sync}(p)$.

The data for the SP2 using MPL can be approximated by the following model:-

$$T_{sync}(p) = 84 \log_2 p$$

This model implies an efficient implementation of the barrier, such good performance can be accomplished when the barrier is implemented using a binary tree method.

The synchronization time for the SP2 using PVMe can be represented by the following model:-

$$T_{sync}(p) = 1470 + 377 \log_2 p$$

Although the $\log_2 p$ factor implies an efficient algorithm (like the binary tree method), but the constants are very large. We don't think that there is a fundamental reason for these large constants, this is just a bad implementation.

For the Convex SPP-1000 using ConvexPVM, the following model fits its Synchronization time:-

$$T_{sync}(p) = 39 + 0.465p^3$$

Here we see bad algorithm (due to the p^3 factor), but the small constants make this time better than the SP2 times. Nevertheless, the extension of this curve for p larger than 8 (see figure 4.8) raises a big concern about the scalability of the Synchronization time on the Convex SPP-1000.

4.9 Conclusions

In this chapter we have developed experiments to measure the communication performance of the IBM SP2 and the Convex SPP-1000 using the PVM message-passing library. We have developed models for the common communication patterns, these models give the time of a communication pattern as a function of the message length and the number of processors.

Some of the models presented in this chapter can be refined to get better accuracy by having more distinct n regions. Also, there is a need to develop appropriate experiments for measuring the Many-to-one communication time.

The comparisons between the communication performance of the two systems, as presented in this chapter, show that the two systems have close performance. This is an interesting result, because people usually expect that a distributed-memory machine like the SP2 to have higher communication time than a shared-memory machine like the SPP-1000.

We can draw the following conclusions about the communication performance of the two systems:-

- In Point-to-point communication, the SP2 has bigger latency and higher transfer rate.
- in One-to-many communication, the SP2 has better performance for small and medium messages.
- In situations where there are heavy communication, the SPP-1000 shows bad performance.

- The Broadcast time in the SPP-1000 is generally less than the time in the SP2, and this time does not increase much when p is increased.
- SPP-1000 has low Synchronization time for small number of processors.

Chapter 5

Modeling the Node Performance

The IBM POWER2 processor has multiple execution units, its instruction cache unit can fetch 8 instructions and issue six instructions per cycle, and it has an effective bandwidth of four double-words per cycle between the cache and the floating-point registers (a double-word is 8 bytes). Yet many scientific applications achieve less than 20% of the POWER2 peak performance.

This chapter tries to explain the achieved performance, and to recommend methods for performance improvement. The chapter presents a Machine-Application Performance Bound Model that takes into consideration the micro-architecture of the POWER2 processor and its memory hierarchy. This model is used to characterize the achieved performance of some test cases, to identify the problems that limited the achieved performance, and to guide in the performance improvement effort.

5.1 Introduction

The IBM RISC System/6000 processors use powerful RISC instructions and wide issue to achieve higher performance. The latest generation of these processors, the POWER2 series, has gone several steps forward in this direction. The POWER2 processors have new instructions that do square roots and instructions for loading or storing between two adjacent floating-point registers and two consecutive memory locations in one cycle.

The POWER2 processors are used in high-end workstations, servers, and in the IBM SP2. With their ability to execute four floating-point instructions per cycle, the POWER2 processors are gaining popularity for scientific applications. In many situations this promised performance is not attainable, and many scientific applications end up running on a small fraction of this peak performance.

Scientific applications are characterized by their intensive use of floating-point operations in loop dominated constructs. This chapter is a humble contribution to identify the problems that limit the achieved performance to a fraction of the peak performance. And to build methodology for improving the performance. In this chapter the Machine-Application Bound model is developed for the POWER2 processor and is extended to include the effect of the memory hierarchy. This model is the tool used for identifying the performance limiting problems and performance improvement.

This chapter uses experiments and analyses for reaching its targets. The experiments were done on one processor node of an IBM SP2. This processor node (Thin node) is a POWER2 processor with a 64 KByte data cache, a 32 KByte instruction cache, a 256 MByte main memory, and runs on a 67.7 MHz clock. The node runs the AIX 3.2.5 operating system. The test programs are written in FORTRAN and compiled by the IBM AIX XL FORTRAN Compiler/6000.

This chapter is organized in 8 sections. After this introductory section, section 5.2 describes the organization and micro-architecture of the POWER2 processor. Section 5.3 develops the Machine-Application Bound model for the POWER2 processor, then section 5.4 extends on it by giving more details for modeling the memory hierarchy performance. In section 5.5 we present and analyze the measured performance of some test cases, then in section 5.6 we give explanations for the performance. Section 5.7 develops a methodology for performance improvement. Conclusions drawn from this study are presented in section 5.8.

5.2 The IBM POWER2 RISC System/6000

The IBM POWER2 RISC System/6000 processor [8] is the second generation of IBM's implementation of the POWER Instruction Set Architecture, it was introduced in late 1993. The first generation, the POWER processor, was introduced in 1990. The POWER2 processor is a super-scalar of higher degree, has bigger caches, faster clock rate, wider busses, and more functional units.

The POWER2 processor philosophy of achieving higher performance is through using more powerful instructions and wider issue, while giving the clock rate a secondary degree of importance.

Figure 5.1 shows the block diagram of the POWER2 processor, the processor is a multichip module that contains 8 chips, partitioned in the following units:-

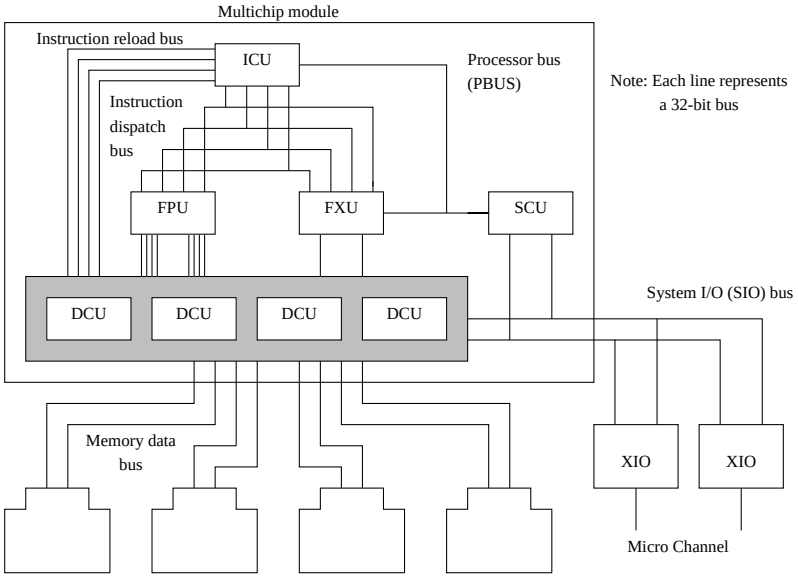

Figure 5.1: The IBM POWER2 RISC System/6000

- Instruction Cache Unit (ICU) [16], this unit contains the instruction cache, it prefetches the instructions form the cache and places them in the instruction buffers. The ICU control logic decodes the instructions in the buffers, executes the branch and condition-register instructions and dispatches the remaining instructions to the FXU and FPU. The ICU can fetch eight instructions per cycle form the I-cache. It can dispatch six instructions per cycle: two internally and four externally to the FXU and FPU.

- Fixed-Point Unit (FXU) [17], performs all storage references, integer arithmetic, and logical operations. The FXU contains the general-purpose registers, two fixed-point execution units, the data cache directory, and the data TLB. The two execution units enable the FXU to execute a total of two instructions per cycle.

- Floating-Point Unit (FPU) [18], includes the floating-point registers (FPRs) and two double-precision execution units. The two units allow it to execute two floating-point instructions per cycle. The FPU supports the compound multiply-add instruction, where an execution unit performs two operations with the same latency as a single multiply or add instruction. The two units enable the FPU to execute 2 double-precision multiply/add instructions every cycle, resulting in up to 4 floating-point operations per cycle. Some POWER2 models have dual quad-word interface to the data cache supporting the dual FPU execution units, dual units and quad-word storage references can load up to 4 FPRs per cycle. Thin nodes has dual double-word interface and can load up to 2 FPRs per cycle.

- Data Cache Unit (DCU) [17], is a four-way set-associative dual-ported D-cache that consists of four identical chips. The four data cache chips generate two single-word data buses to the FXU, two quad-word buses to the FPU (two double-word buses in the Thin node), a 4-word instruction reload bus to the ICU, and a 2-word System I/O (SIO) bus to the I/O subsystem for DMA data.

- Storage Control Unit (SCU) [17], handles the main memory references for I-cache and D-cache misses. When a data cache miss occurs, the FXU arbitrates for the processor bus (PBUS). After the FXU places the cache miss request on the PBUS, the SCU accepts the request and generates the corresponding memory control signals to start a memory operation. The returning data arrive at the DCU, which places the data in the D-Cache. When an instruction cache miss occurs, the ICU arbitrates for the PBUS. After the ICU places the cache miss request on the PBUS, the SCU accepts the request and generates the corresponding memory control signals to start a memory operation. The returning data arrive at the DCU, which forwards the data to the ICU on the instruction reload bus.

5.3 Machine-Application Bound Model

The Machine-Application (MA) Bound [1, 2, 3, 4] gives an upper bound on the performance that can be achieved on a certain machine for a certain application. The measured

performance, P_{measured}, on the other hand, can be represented by:

$$P_{\text{measured}} = \frac{\text{Floating-Point Operations}}{\text{Measured Execution Time}}$$

The MA Performance Bound, P_{MA}, can be found from the Clock Rate (f), and the MA Clocks Per Float bound (CPF).

$$P_{MA} = \frac{f}{\text{CPF}}$$

The CPF is the number of processor cycles, t_l, need to execute one iteration of a loop, or one pass of a program segment, divided by the number of essential floating-point operations in this loop or program segment. The number of essential floating-point operations equal the sum of the add operations, f_a, multiply operations, f_m, two times the number of multiply operations that can be combined with following add operations, f_{ma}, four times the number of divide operations, f_{div}, and four times the number of square root operations, f_{sqrt}. Divide and square root operations are weighted by a factor of four to reflect their complexity with respect to adds and multiplies (this is a common practice in scientific benchmarks [19]).

$$\text{CPF} = t_l/(f_a + f_m + 2f_{ma} + 4f_{div} + 4f_{sqrt})$$

The number of processor cycles, t_l, equals the time of the slowest (busiest) functional unit. The POWER2 is modeled as five independent functional units, each with lower bound time: floating-point unit, t_{fl}, fixed-point unit, t_{fx}, instruction issue unit, t_i, memory unit, t_m, and dependence pseudo-unit, t_d. The bound for each functional unit is calculated as a function of the number of essential operations found in the high-level source code that must be performed by that functional unit. The bound assumes that each functional unit needs to execute only these essential operations, and that it can execute them at its peak rate.

$$t_l = \max(t_{fl}, t_{fx}, t_m, t_i, t_d)$$

The number of essential floating-point loads, l_{fl}, equals the number of distinct values that appear on the right hand side of a high level code statement before they appear on the left hand side. The number of essential floating-point stores, s_{fl}, equals the number of distinct values that appear on the left hand side of a high level code statement that are not temporary values. For POWER2 models that have quadword bus to the D-cache, the number of loads and stores should be divided by two if the access stride equals one, this is because it is possible to employ quadword load and store instructions that do two floating-point loads or stores in one cycle. The following equation gives the bound for the FPU; the number of loads, stores, and arithmetic operations are all divided by two because the FPU has two execution pipelines for loads, two for stores, and two for arithmetic operations. The divide operation requires the FPU for 17 cycles and the square root operation require 27 cycles.

$$t_f = \max(l_{fl}/2, s_{fl}/2, (f_a + f_m + f_{ma} + 17f_{div} + 27f_{sqrt})/2)$$

The bound for the FXU models its impact on floating-point operations. An address calculation is required for each floating-point memory operation. The FXU can perform two address calculations and translations per cycle.

$$t_{fx} = (l_{fl} + s_{fl})/2$$

The memory unit bound is a function of two factors; the performance of main memory accesses on cache misses, and the performance of the D-cache on cache hits.

$$t_m = \max\left((\text{load miss time} + \text{store miss time}), (l_{fl}L_{eff} + s_{fl}S_{eff})/2\right)$$

The first term in the above equation reflects the fact that there is only one port between the D-cache and the main memory, so misses can be served one a time. The second term reflects the fact that there are two ports between the FPU and the D-cache. In this term the number of loads are multiplied by the effective load time, L_{eff}, and the number of stores are multiplied by the effective store time, S_{eff} (see section 5.4 for a discussion on how to find the effective access times). Note that the second term models the case when there is low miss rates, and assumes single outstanding cache miss at anytime. So while the memory services the miss, the other data cache port continues to service a single memory access per cycle.

The bound for the issue unit is specified by the ICU dispatch width. The ICU can dispatch 4 instructions per cycle to the FXU and FPU.

$$t_i = (f_a + f_m + f_{ma} + f_{div} + f_{sqrt} + l_{fl} + s_{fl})/4$$

The loop-carried dependence pseudo-unit models the performance of loops with a recurrence. This bound is found by summing the latencies of the longest path in the dependency graph. The latency is 1 cycle for floating-point adds and multiplies, 2 for multiply-adds, 17 for divides, and 27 for square roots.

$$t_d = \text{Total latency of loop-carried dependence}$$

5.4 Memory Hierarchy Model

For a memory system of n levels the Effective Access Time, T_{eff}, is:

$$T_{\text{eff}} = \sum_{i=1}^{n} f_i t_i$$

where f_i is the Access Frequency for level i, and t_i is the Access Time for that level. Given the hit ratios, h, or the miss ratios, m, then

$$\begin{aligned} f_i &= (1-h_1)(1-h_2)\cdots(1-h_{i-1})h_i \\ &= m_1 m_2 \cdots m_{i-1}(1-m_i) \end{aligned}$$

All POWER2 models have primary cache, some models have secondary cache. Also if we ignore page faults ($m_3 = 0$) we get the following model:

$$T_{\text{eff}} = (1-m_1)t_1 + m_1(1-m_2)t_2 + m_1 m_2 t_3$$

The POWER2 processor used in our experiments does not has a secondary cache, so $m_2 = 1$ and the Effective Access Time becomes:

$$T_{\text{eff}} = (1-m_1)t_1 + m_1 t_3$$

Experimental evidence showed that the Effective Access Time for loads is different than that of stores, so we have one formula for the Effective Load Time and one for the Effective Store Time:

$$L_{\text{eff}} = (1 - m_{1l})t_{1l} + m_{1l}t_{3l}$$
$$S_{\text{eff}} = (1 - m_{1s})t_{1s} + m_{1s}t_{3s}$$

Our experiments showed that $t_{1l} = t_{1s} = 1$, $t_{3l} = 16.3$, and $t_{3s} = 22.0$ (see subsection 5.4.1 for a description of these experiments), hence:

$$L_{\text{eff}} = 1 + 15.3m_{1l}$$
$$S_{\text{eff}} = 1 + 21.0m_{1s}$$

For the purpose of developing a bound model, we only consider Essential misses, which are Compulsory misses, m_{Comp}, and Capacity misses, m_{Cap}, so the miss rate can be found by:

$$m = \frac{m_{\text{Comp}} + m_{\text{Cap}}}{\text{Number of Essential Accesses}}$$

Compulsory misses equal the number of blocks in the Working Set, B, hence:

$$m_{\text{Comp}} = B$$

Capacity misses depend on the number of Working Set blocks, B, number of Cache Blocks, C, Cache Degree of Associativity, A, and Access Pattern, D. Assuming linear access pattern, as is often found in loops, a lower bound on the capacity misses can be calculated as follows:

$$m_{\text{Cap}} = \begin{cases} 0 & B < C \\ B\frac{B-C}{DC/A} & C \leq B \leq C(1 + D/A) \\ B & B > C(1 + D/A) \end{cases} \quad (5.1)$$

Here D is the Degree of Freedom in the access pattern which equals the number of distinct arrays in the working set, it is restricted to be between 1 and A. Equation 5.1 reflects the fact that the number of cache misses depends on the working set size relative to the cache size resulting in three distinct regions of behavior. See subsection 5.4.1 for more details on these regions, and see subsection 5.4.2 for more details on the estimation of the cache misses.

5.4.1 Finding the Memory Access Parameters

This subsection describes the experiments [20] used to characterize the behavior of the POWER2 memory system. Two experiments were done, the results are shown in figures 5.2 and 5.3. These figures show the time per access operation in clock cycles. The first experiment is timing a load kernel, in which the inner loop contains statements that load an array elements, the array size is varied between 1-128 KBytes, and the experiment is repeated for strides 1, 2, 4, 8, and 16.

The second experiment is timing a store kernel, in which the inner loop contains statements that store in an array, the array size is varied between 1-128 KBytes, and the experiment is repeated for strides 1, 2, 4, 8, and 16.

The two figures show three regions:-

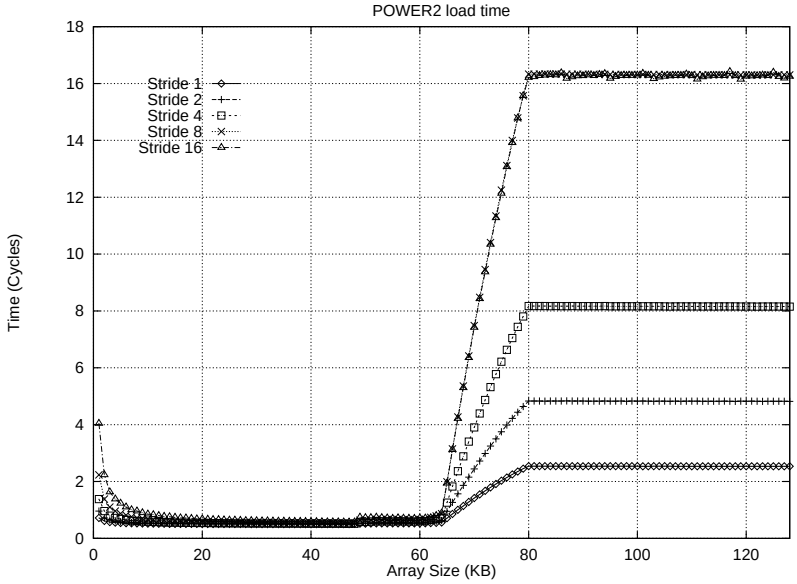

Figure 5.2: POWER2 Load Time

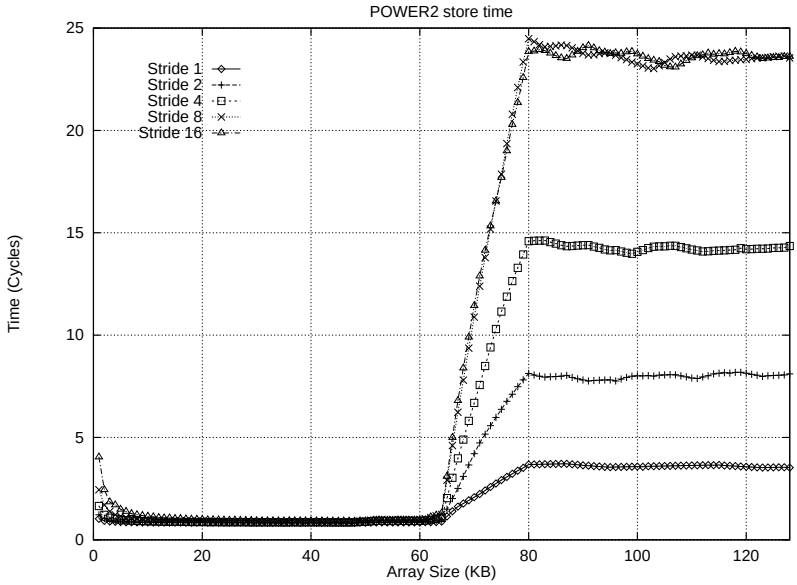

Figure 5.3: POWER2 Store Time

1. Hit Region; where the array size is smaller than the cache size, and every access is a hit. This region is supposed to be constant, but the high time for small array sizes is due to the loop overhead which has less effect for larger arrays.

2. Transition Region; where some of the accesses are hits and others are misses. The width of this region equals the cache size divided by the degree of the cache associativity.

3. Miss Region; where the array size is big enough that every access to a new cache block is a miss.

From these graphs we get the following information:-

1. The cache size is 64 KBytes; the Transition region starts at 64 KBytes.

2. The degree of the cache associativity is 4; the width of the transition region is 16 KBytes (64/(80-64)=4).

3. The cache block is 8 double-words (64 bytes); the performance of stride 8 is similar to the performance of stride 16 implying that with stride 8 or more every access is to a new cache block.

4. At the Hit Region the access time is 1/2 cycle, since we have two cache ports, then the time for one port is 1 cycle; and $t_{1l} = t_{1s} = 1$.

5. At the Miss region the load miss time is 16.3 cycles, and the store miss time is 22.0 cycles; hence $t_{3l} = 16.3$, and $t_{3s} = 22.0$.

5.4.2 Estimating Cache Misses

This subsection is intended to outline a recipe for estimating a lower bound on the cache misses, this bound is needed to find the MA bound of an application. As explained in section 5.4, we need to estimate compulsory and capacity misses, our means for this estimation is by looking at the application high-level source code. This approach assumes that the compiler will not do algorithmic changes to the application, like loop blocking, loop movement, or loop interchange.

This approach is suitable for loop dominated code, and it assumes that the arrays are accessed linearly, i.e. in any loop an array is accessed with a constant stride. The misses estimated by the methods introduced here can be used as lower bound for loops with non-linear access patterns if the access density is constant over all the array elements.

Estimating the compulsory misses is easy, the number of compulsory load misses equals the number of the array blocks that are used but not modified by the application. The number of compulsory store misses equals the number of the array blocks that are modified by the application.

Estimating the capacity misses can be done for the two cases capacity misses occur:-

- The application has an outer-most loop that iterates over the application's loop structure, in this case the working set is the sum of the array blocks and we can directly use equation 5.1. The number of the estimated cache misses should be split into load misses and store misses using the ratio of modified and read-only elements.

- The application has more than one loop structure that uses the same array, in this case we need to look at the loop's code and to estimate the cache status before it, if it is not possible to estimate the cache status at the beginning of this loop, then we can make an optimistic assumption that the cache holds the data needed by this loop. Then we find the size of the working set, use equation 1 to find the misses, and finally subtract the misses that could not happen because of the previous cache status.

5.5 How much are we getting?

Sections 5.3 and 5.4 have developed the MA model for the POWER2. This model gives an upper bound on the performance that can be achieved for a certain application. This section tries to answer the question of how much of this bound is achievable. This is done by a case study of a commercial application (FEMC) [4] that uses the finite element method, and of the Livermore kernels [19].

5.5.1 Finite Element Method Application (FEMC)

Some FEMC key routines were studied using the performance models developed in this chapter, the results of the analysis for two routines are presented here. The two routines (routine A and routine B) take considerable percentage of the total application execution time, they contain complex computations over many arrays. Routine A updates the node forces from element forces. Routine B finds the global kinetic energy of the nodes and elements.

The results of these analyses are listed in table 5.1. The table shows the percentage of the measured performance relative to the MA performance bound. These results show that for a well written application, we are getting less than one half of the MA bound performance.

Routine	$P_{measured}$
A	35%
B	47%

Table 5.1: FEMC Percentage of Achieved Performance

5.5.2 Livermore Kernels

The MA bound model was applied to the first 12 Livermore kernels, table 5.2 shows the analysis for these kernels.

The table shows the number of essential operations per one loop iteration, the number of clocks needed by each of the four functional units to execute one loop iteration, the maximum of these times t_l, the CPF, and the MA performance. The table shows that the MA performance ranges between 60 to 267 MFLOPS. All these kernels fit in the cache, and the table shows that they are either limited by the number of ports to the cache, or by the floating-point execution units (bold numbers are the performance limiting bounds).

LFK	f_m	f_a	f_{ma}	s_{fl}	l_{fl}	t_f	t_m	t_i	t_d	t_l	CPF	P_{MA}
1	1	0	2	1	2	**1.5**	**1.5**	1.50	0	1.5	0.300	222
2	0	0	2	1	2	1.0	**1.5**	1.25	0	1.5	0.375	178
3	0	0	1	0	2	0.5	**1.0**	0.75	0	1.0	0.500	133
4	0	0	1	0	2	0.5	**1.0**	0.75	0	1.0	0.500	133
5	1	1	0	1	2	1.0	**1.5**	1.25	3	1.5	0.750	89
6	0	0	1	0	2	0.5	**1.0**	0.75	0	1.0	0.500	133
7	0	0	8	1	3	**4.0**	2.0	3.00	0	4.0	0.250	267
8	0	6	15	6	9	**10.5**	7.5	9.00	0	10.5	0.292	228
9	1	2	7	1	10	5.0	**5.5**	5.25	0	5.5	0.324	206
10	0	9	0	10	10	4.5	**10.0**	7.25	0	10.0	1.111	60
11	0	1	0	1	1	0.5	**1.0**	0.75	1	1.0	1.000	67
12	0	1	0	1	1	0.5	**1.0**	0.75	0	1.0	1.000	67

Table 5.2: POWER2 MA Bounds for LFK

To find out the performance that is achieved for these kernels, we have compiled and executed the kernels using different compiler options. Three experiments were done for three different compiler options:-

1. No Optimization (**No**); in this experiment the compiler did not used any optimization.
2. Basic Optimization (**-O**); here the compiler performed the basic optimization techniques like: common expression elimination, code motion, strength reduction, constant propagation, global register allocation, and instruction scheduling.
3. Advanced Optimization (**-O3**); here the compiler performed the optimization techniques of the -O level, and also performed more aggressive optimization techniques that might change the answers.

The results of these experiments are listed in table 5.3 and presented graphically in figure 5.4. The table shows the percentage of the measured performance relative to the MA performance bound.

We can draw the following conclusions from the results of these experiments:-

- The compiler generates bad object code that achieves only 5%-20% of the MA bound when no optimization is used. Hence using this compilation is only justifiable during the development time when correctness is the main issue.
- With the basic optimization the compiler generates code that achieves about 23%-79% of the MA bound.
- With the advanced optimization the compiler generates code that achieves about 25%-91% of the MA bound.
- Using advanced optimization should be done selectively since it might generate code that is worse than the one generated by the basic optimization, for Kernels 1, 2, 4, and 7 the advanced optimization did worse than the basic optimization.

LFK	No	-O	-O3
1	7.9	50.6	48.3
2	6.3	33.5	31.4
3	6.3	50.4	53.3
4	5.3	47.3	42.7
5	10.8	25.1	48.0
6	4.9	23.3	25.1
7	16.9	72.9	59.8
8	7.3	78.8	78.8
9	20.2	61.4	61.4
10	8.8	51.3	52.0
11	7.7	25.7	33
12	7.6	48.7	91.2
Avg.	9.2	47.4	52.1

Table 5.3: Livermore Kernels Percentage of Achieved Performance

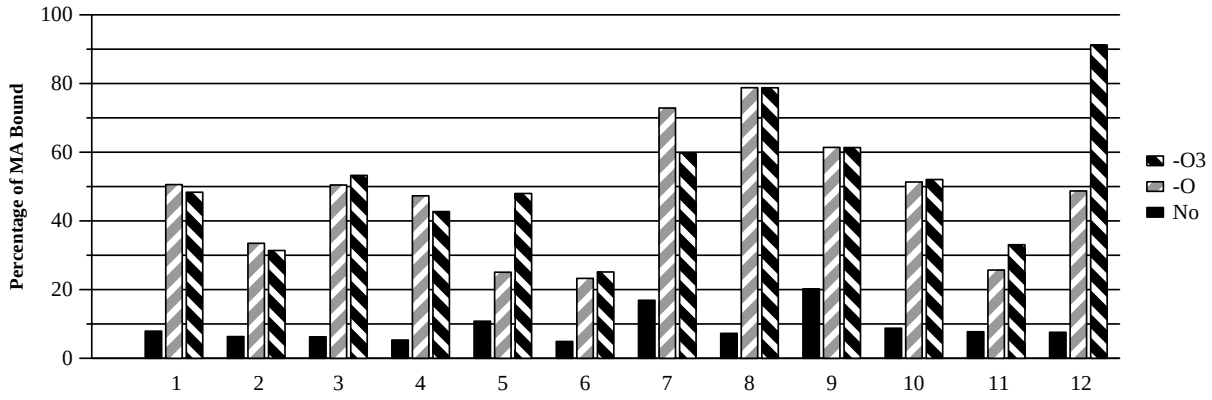

Figure 5.4: Livermore Kernels Performance

- Even with the advanced optimization, the average achieved performance is only 52% of the MA bound. This is a low percentage because Livermore kernels are compact loops, and usually they achieve better performance than real applications.

5.6 Why is it this low?

This section tries to identify the reasons for achieving only a fraction of the MA bound. First let us have another look at the FEMC. Figure 5.5 shows the MA bound time normalized to the measured execution time for routines A and B, it also shows the MA bound time when assuming that we have perfect cache and there are no cache misses (MA-PC).

This figure shows that the new MA with memory hierarchy model developed in this chapter does better job than previous models that did not took the memory hierarchy effects into consideration; this is obvious when we observe that the MA-PC model only explains

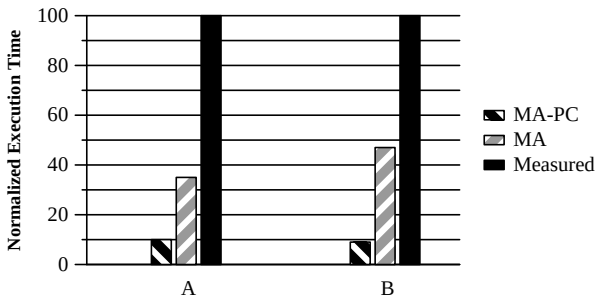

Figure 5.5: Performance of two FEMC routines

10% of the execution time for routine A, and 8% of the execution time for routine B. On the other hand, the new model explains 35% and 47% of the execution time.

After studying the source code and the object code generated by the compiler for the case-study programs used in this chapter, we have identified the following main reasons for getting only a fraction of the performance bound:-

Bad Schedule The compiler in many cases fails to generate code that exploits the POWER2 functional parallelism. This is the case for some of Livermore Kernels like 1, 3, and 12.

Cache Misses In cases where the working set does not fit in the cache, like FEMC routine A and routine B, the cache misses can largely lower the performance. Note that the model gives a lower bound for the essential misses, it takes into consideration the compulsory and capacity misses only. Other misses like the conflict misses are in many cases avoidable by proper array allocation in the memory space with respect to the cache boundaries.

Register Spill Sometimes, especially for complex loops, the compiler fails to generate a register allocation without spill, this was found in LFK 8.

Redundant Instructions In many cases the compiler generates redundant load instructions. This happens when it does not recognize that some elements loaded in one iteration are used in the next iteration.

5.7 What can we do to make it higher?

There are many techniques that were developed by different researchers to improve the performance of scientific applications. In this section we start by listing some of the techniques that are likely to give good results on the POWER2, then we show how the insights gained from the MA model can guide in performance tuning. Finally, we demonstrate an example of applying the MA model to a tuned kernel.

The following is a list of some of the performance improving techniques that gave good results on the POWER2:-

Loop Unrolling This is the technique that proved to be most useful in improving the performance of Livermore Kernels on POWER2. It yields codes that make better

exploitation of the POWER2 functional parallelism, and it reduces or eliminates the effect of control overheads.

Algorithmic Prefetching [21] This technique can be applied in POWER2 processor because it has two cache ports. This technique is useful in hiding load miss latency. In the POWER2 processor this can be done by using one cache port to make references for data that is needed in the next iteration, this port will probably stall due to cache misses. The second port, on the other hand, can keep accessing the prefetched data used by the current iteration.

Locality Improvement Techniques These are wide range of techniques that can be used to decrease the number of cache misses. Some of these techniques are Loop Interchange, Loop Blocking, and Loop Merging.

Software Pipelining This technique enables generating code schedules that make better usage of the POWER2 functional units.

The best thing about the MA bound is that it tells you how far you are from the optimum performance for the given application. This knowledge will put you in one of the following two situations:-

1. When the achieved performance is a small fraction of the MA bound, then you know that there is plenty of room for improvement. In this case it is a good idea to invest time in figuring out why this is happening, and work on making things better by using some of the techniques listed above.

2. When the achieved performance is only a little bit smaller than the MA bound, then you know that there is only little room for improving the application performance . In this case you either accept what you are getting, go on improving the code as in 1 above, or make use of the MA bound insights to change the application to get better MA bound.

When you accomplish the MA bound then the only thing you can do is to change the application in order to get new better MA bound. To do this you should examine the functional unit which has limited the MA bound, and try to come up with a modification for the application so that this unit has a reduced load without increasing the load of other functional units. This study has identified that usually one of the following determines the MA bound:-

1. The number of floating-point operations, as in LFK 1, 7, and 8. Here the only thing that can be done is to come up with a new algorithm that has less number of floating-point operations.

2. The number of memory references, as in the rest of the Livermore Kernels. Here one might consider changing the application to ensure that the access stride is one, this will enable the compiler to generate quadword load and store instructions, thus doing two memory accesses by one instruction in one cycle.

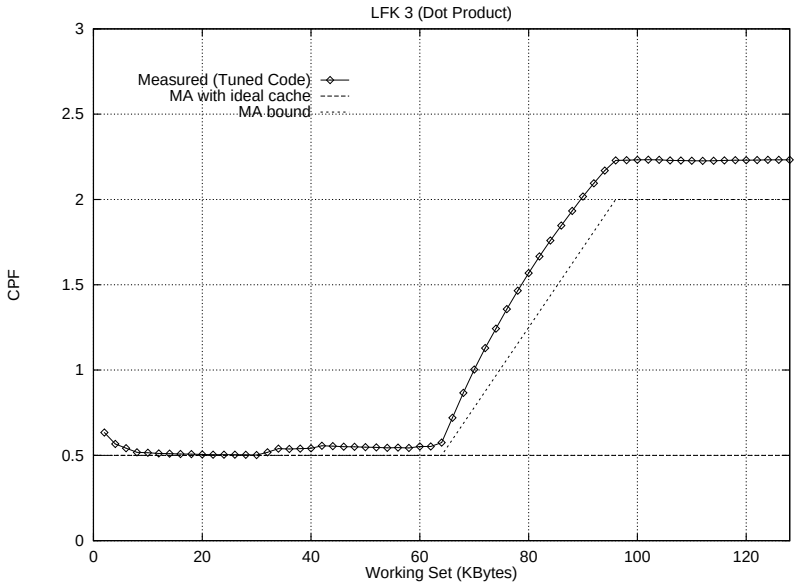

Figure 5.6: Performance of Tuned LFK 3

3. The number of cache misses, as in the FEMC routines. In this case the Locality Improvement Techniques can be used.

Near the end of this chapter, we would like to show one example of the power of the MA bound model developed in this chapter. The example is Livermore Kernel 3, this is a dot product kernel for two arrays. This kernel has achieved only 53.3% of the MA bound by using the advanced compiler optimization level. By unrolling the main loop of this kernel three times we were able to get 100% of the MA bound.

Figure 5.6 shows three curves as a function of the sum of the two arrays. The first curve is the measured CPF for the tuned code. The second curve is the MA CPF. And the third curve is the MA CPF assuming perfect cache with no cache misses. This figure illustrates that the MA model developed in this chapter is better than the older models for calculating an upper bound for the performance.

5.8 Conclusions

In this chapter we have developed a Machine-Application Bound model for the IBM POWER2 that models its micro-architecture and its memory hierarchy. We have presented experiments for finding the parameters needed to characterize the memory hierarchy performance, and developed a methodology for estimating the essential cache misses given the high-level source code of a scientific applications.

The test cases used in this chapter showed example applications that achieve a fraction of the MA bound performance. We have identified that bad instruction schedules, cache misses, register spill, and redundant instructions are the main problems that limit the performance.

We have outlined a methodology for performance improvement guided by the insights gained from the MA bound model. This methodology relies on finding the percentage of

achieved performance and identifying the functional unit that limited the MA bound, and using this information to suggest the performance improvement technique that is most likely to give good results.

Chapter 6

Conclusions

The performance models developed and presented in this report for message-passing multi-computers provide an estimation of an application execution time by inspecting its high-level source code. In the SP1 and the SP2, the execution time equals the sum of the computation time and the communication time.

We have identified the factors that affect the communication time on a multi-computer, and we have shown how this time is found by summing the times of the basic communication patterns. In this report, we have presented our techniques for timing the basic communication patterns, and our techniques for developing models that give the time of a communication pattern as a function of the message length and the number of processors. Throughout our development of the communication models, we modeled the communication time in two components; setup time and transfer time:-

$$T_{comm}(n,p) = t_{comm}(p) + \pi_{comm}(p)n$$

We have observed that splitting the communication time into these two components gives simple equations.

By studying the performance of several broadcast algorithms, we have demonstrated how the communication models are used to find the communication times. We have shown how these models are used to select the best algorithm and to develop an efficient broadcast algorithm that has optimum performance for a wide range of message lengths and number of processors.

We have also demonstrated how the communication models are used to do performance comparison between different computers that support the message-passing paradigm. Developing these communication performance models for different multi-computers reveals many aspects of their relative performance.

To model the computation performance of the processor nodes, we developed a Machine-Application Performance Bound model for the POWER2 processor. The MA Bound models gives a lower bound for an application execution time as a function of the design parameters of the POWER2 micro-architecture and memory hierarchy.

By analyzing some test cases, we have demonstrated how the MA Bound model is used to explain the achieved performance. And we have discussed how this model can guide the effort of tuning the computational performance of an application.

We have applied the computation and communication performance models to some of the FEMC routines [4], and we have seen that the modeled time ranges between 25% and 80% of the measured time. We have strong reasons to believe that the following are some of the factors that contribute to the difference between the modeled and measured times:-

1. load imbalance due to unbalanced domain decomposition and operating system interrupts,

2. compiler inefficiencies in generating optimum instruction schedules and in eliminating redundancies, and

3. unmodeled cache misses, since our approach for estimating the essential cache misses only gives a lower bound estimation for the essential cache misses, and does not estimate some cache misses in complex situations.

We plan to enhance our models to include the effects of load imbalance, and enhance our techniques for estimating the number of essential cache misses.

The MPI [22] message-passing library is widely accepted as the standard library for writing message-passing applications. It is becoming available for many multi-computer systems, and is becoming regarded as the message-passing library of choice for writing portable applications. We are looking forward to developing communication performance models for this library on the SP2, and to using these models to gain insights regarding its relative performance with respect to other available message-passing libraries.

Next, we plan to demonstrate the power of these models in performance tuning by carrying out case study of some of the NAS Parallel Benchmark Kernels. We hope that through techniques similar to the ones presented in chapters 3 and 5 will be able to develop efficient implementations of these kernels.

Bibliography

[1] W. H. Mangione-Smith, S. G. Abraham, and E. S. Davidson, "A performance comparison of the IBM RS/6000 and the Astronautics ZS-1," *IEEE Computer*, vol. 24, pp. 39–46, Jan. 1991.

[2] W. H. Mangione-Smith, T. P. Shih, S. G. Abraham, and E. S. Davidson, "Approaching a machine-application bound in delivered performance on scientific code," *IEEE Proceedings*, vol. 81, pp. 1166–1178, Aug. 1993.

[3] E. L. Boyd, W. Azeem, H. H. Lee, T. P. Shih, S. H. Hung, and E. S. Davidson, "A hierarchical approach to modeling and improving the performance of scientific applications on the KSR1," in *Inter. Conf. on Parallel Processing*, vol. 3, pp. 188–192, Aug. 1994.

[4] E. L. Boyd, G. A. Abandah, H.-H. Lee, and E. S. Davidson, "Modeling computation and communication performance of parallel scientific applications: A case study of the IBM SP2," in *Suprcomputing*, (Draft), Dec. 1995.

[5] D. Baily *et al.*, "The NAS parallel benchmarks," Tech. Rep. RNR-94-07, NASA Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035-1000, Mar. 1994.

[6] R. Hockney *et al.*, "Public international benchmarks for parallel computers," Tech. Rep. 1, PARKBENCH Committee, Feb. 1994.

[7] C. B. Stunkel *et al.*, "The SP2 communication subsystem," tech. rep., IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, Aug. 1994.

[8] S. W. White and S. Dhawan, "POWER2: next generation of the RISC System/6000 family," *IBM J. Research and Development*, vol. 38, pp. 493–502, Sept. 1994.

[9] C. B. Stunkel *et al.*, "Architecture and implementation of Vulcan," in *The 8th International Parallel Processing Symposium*, (Cancun, Mexico), Apr. 1994.

[10] IBM, *IBM AIX Parallel Environment, Parallel Programming Subroutine Reference*, 2.0 ed., June 1994.

[11] IBM, *IBM AIX PVMe User's Guide and Subroutine Reference*, 1.0 ed., Apr. 1994.

[12] A. Geist *et al.*, *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, Sept. 1994. ORNL/TM-12187.

[13] R. Hockney, "Performance parameters and benchmarking of supercomputers," *Parallel Computing*, vol. 17, pp. 1111–1130, Dec. 1991.

[14] Convex Computer Corporation, P.O. Box 833851, Richardson, TX 75083-3851, *Convex Exemplar Programming Guide*, x3.0.0.2 ed., June 1994.

[15] Q. F. Stout and B. Wagar, "Intensive hypercube communication," *Journal of Parallel and Distributed Computing*, vol. 10, pp. 167–181, 1990.

[16] J. I. Barreh, R. T. Golla, L. B. Arimilli, and P. J. Jordan, "POWER2 instruction cache unit," *IBM J. Research and Development*, vol. 38, pp. 537–544, Sept. 1994.

[17] D. J. Shippy and T. W. Griffith, "POWER2 fixed-point, data cache, and storage control units," *IBM J. Research and Development*, vol. 38, pp. 503–524, Sept. 1994.

[18] T. N. Hicks, R. E. Fry, and P. E. Harvey, "POWER2 floating-point unit: Architecture and implementation," *IBM J. Research and Development*, vol. 38, pp. 525–536, Sept. 1994.

[19] F. H. McMahon, "The Livermore Fortran kernels: A computer test of the numerical performance range," Tech. Rep. UCRL-53745, Lawrence Livermore Nat. Lab., Livermore, CA, Dec. 1986.

[20] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff, "Experimentally characterizing the behavior of multiprocessor memory systems: A case study," *IEEE Trans. Software Engineering*, vol. 16, pp. 216–223, Feb. 1990.

[21] R. C. Agarwal, F. G. Gustavson, and M. Zubair, "Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms," *IBM J. Research and Development*, vol. 38, pp. 563–576, Sept. 1994.

[22] E. Anderson *et al.*, "MPI: a message-passing interface standard," tech. rep., Message Passing Interface Forum, May 1994.