LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Study of Performance and Optimization of MPI Over 100BaseT Switched Ethernet Networks

Computation Structures Group Memo 412
August 5, 1998

Michael J. Broxton

This research was conducted at the MIT Laboratory for Computer Science. Michael J. Broxton is a high school student in the 1998 Research Science Institute at MIT, organized by the Center for Excellence in Education. Resources and equipments were made available by Professor Arvind and the Computational Structures Group with funding provided in part by the Defense Advanced Research Projects Agency of the Department of Defense under the Ft. Huachuca contract DABT63-95-C-0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Study of Performance and Optimization of MPI Over 100BaseT Switched Ethernet Networks

Michael J. Broxton

under the direction of
Mr. James C. Hoe
Massachusetts Institute of Technology

Research Science Institute
August 5, 1998

Abstract

This study is a benchmark of MPICH, a freely available distribution of the MPI parallel processing library. It was conducted on a 100BaseT switched Ethernet network. Results showed that, while performance on the 100BaseT network did not rival that of commercially available parallel processing technologies, the price to performance ratio is reasonable. This study also focuses on analyzing performance bottlenecks and their causes. Optimizations that can be made to MPI programs based on observed performance are also discussed. These optimizations include limits to message size and message posting frequency.

1 Introduction

While supercomputers provide the advantage of parallel processing, they are difficult to maintain and very expensive. The recent advent of parallel computing over local area networks has made new possibilities available to smaller companies that cannot afford more expensive solutions. In 1995, the Message Passing Interface (MPI) standard was developed. MPI defines a basic set of parallel processing functions which can easily be integrated into C or Fortran [5]. These routines provide message passing between homogeneous or heterogeneous machines on an Ethernet network [5].

This study focuses on benchmarking the performance of MPI. The test environment is a network of PowerPC machines running AIX on a 100BaseT network. The tests are loosely based on the LogP set of mini-benchmarks [1]. These mini-benchmarks yield four basic variables that are inherent in any basic network communication: the latency, the overhead, the gap, and the number of processors [2]. LogP is typically used to benchmark multi-processor machines, but is used here with some modification to study MPI. The basic communication primitives in MPI are used to generate simple and easily manipulated messages. Performance changes are observed by modifying how messages are sent from MPI. By changing these properties, each of the LogP variables can be calculated.

The reason for benchmarking the latency of MPI is to determine whether 100BaseT is a viable alternative to vendor-specific parallel processor networks. The results of this benchmark show how fast MPI performs on a 100BaseT network. This paper focuses on two main areas. First an analysis of the performance of the network will be made. Then the implications of the benchmarks for optimization of MPI programs will be discussed.

2 Methods and Theory

2.1 The LogP Model

LogP is a widely used method of testing the performance of multi-processor machines. LogP measures the interactions that occur at the lowest level of processor communication. MPI contains a set of "primitive" communication functions which have very simple, predictable algorithms that provide this low level interaction. Calls to these simple functions are made and timed repeatedly. Often, more than one variable will be measured during this benchmark [2], making accurate measurements of a network's performance difficult. More advanced comparisons are necessary to interpret the data correctly.

For the purpose of this study, the LogP model was expanded to include one extra variable. This extra variable allows differentiation between the sending and receiving overhead. The five variables used in this study are:

- Latency (L) - The amount of time spent on the network or being processed by network hardware.
- Sending Overhead (o_s) - The amount of time the processor spends calculating and sending the instruction on the sending node.
- Receiving Overhead (o_s) - Processing time spent on the receiving processor, or node.
- Gap(g) - The time it takes a message to go from one processor to the other. This time includes processor overhead on both nodes. $G = o_s + L + o_r$
- Processors(P) - The number of processors.

For any given communication exchange between two processors, a LogP equation can be written. For example, if processor 0 sends processor 1 a message, and processor 0 then receives a reply, two exchanges have taken place. The measure of each of these particular types of interactions is one Round Trip Time (RTT). RTT times both nodes and the network connecting them[4]. The equation $2(o_s + L + o_r) = RTT$ reflects the total time for the two messages to be transmitted. The o_s and the o_r is time spent at the node, and L is time spent on the network. If a small enough packet size is selected and sent repeatedly, the time between consecutive sends will reflect the o_s of the system [1]. Timing each RTT cycle will reveal a curve that rises to a limit at which the receiving processor is always processing packets. However, this limit cannot be used as the o_r measurement, because latency may be part of the measured time.

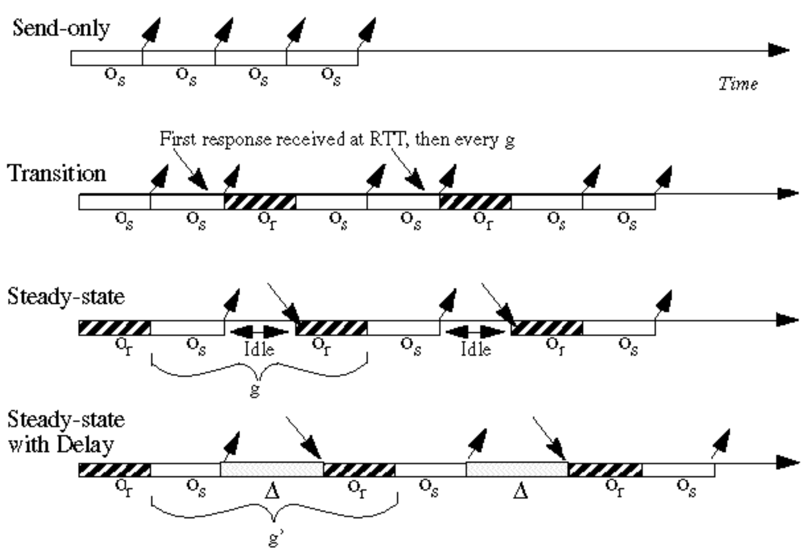

Figure 1: *Illustration of LogP method. Demonstrates introduction of an artificial idle time to create a constant gap. From this constant gap, the o_r can be extracted. [1]*

To differentiate between latency and receiving overhead, a round trip experiment is conducted. First, the sending processors send small (1 byte) packets to another node. The receiving node is instructed to send a reply as soon as it receives its first packet. After a brief transitionary period, an equilibrium is reached where the reply will come back as quickly as the send and an idle time will appear between each communication. The gap is then measured as the time between the send and the reply. A constant gap is formed by setting an artificial idle time that is greater than any natural idle time. If the o_s and the artificial idle time are subtracted from artificially controlled gap, the result is the o_r. Figure 1 gives a graphical representation of this concept.

2.2 Algorithms and Approach

To time the interactions between the processors, a clock accurate to one microsecond was written using the PowerPC time base registry. The clock records the value of the tick held in the registry before each new send or receive call is posted. The returned values are stored in arrays. Storing data in arrays rather than writing directly to an output file reduces overhead on the machines, which interferes with the timing. These values are later subtracted from each other and converted into microseconds so that each value shows the amount of time that elapsed between new postings.

This study focuses on MPI's ability to send blocking and non-blocking communications. A blocking communication will initiate a send or receive request, and not exit until that request has been satisfied [5]. The other node must have completed its call before the local program continues execution. MPI also allows a non-blocking call to be issued. A non-blocking send does not require a matching receive [5]. The non-blocking request will still exist, but program execution can continue. The latency tests are timed using the average of 100 interactions (send and receive pairs or round trips) conducted 100 times.

By averaging together 100 pairs, false data points from other network traffic and machine overhead are reduced. Three different conditions are simulated, each yielding a different result that contributes to the LogP problem:

- Blocking communications going from a source node to destination node with no reply. This is a one-way communication. This test shows the latency for one communication. Since this call will not exit until the receive on the other node has completed, it reflects both remote and local overhead.
- Non-blocking one-way send. Non-blocking communications exit even if a matching receive has not been posted. This test measures only the o_s and the latency of the network. By comparing this value to the value returned by the first test, the o_r can be determined.
- Blocking round-trip communication. The source node receives a reply message from the destination before starting a new cycle. This test provides the round trip time (RTT).

2.3 Analysis and Optimization

The optimizations suggested in this paper are based on understanding the MPI buffer. A buffer is a local data store kept in the RAM of the local processor. It is set at a fixed size, and entries are simply queued into the buffer as they wait for the network to send them. The buffer will behave differently depending on the method used to pass a message in MPI. Blocking communications do not use the buffer because they are never placed in a queue. If characteristics such as the message size of non-blocking callsare manipulated, the graphed results will show how the buffer behaves. Based on this data, optimization schemes can be devised that take advantage of the buffer design.

3 Results and Discussion

3.1 Test Environment

The tests were run on a cluster of computers running AIX 4.2 with 100 MHz PowerPC 604 chips. The computers were networked on a switched 100BaseT Network. Two machines were chosen from the cluster and consistently used throughout the tests. MPICH, a popular MPI implementation, was used as the MPI library. Only the MPI_Send(), MPI_Isend(), MPI_Recv(), and MPI_Irecv() were timed. The messages were 16 byte characters.

3.2 Performance

The one-way graphs in figure 2 show that first packets that are sent show a brief period during which the receiver and the sender are establishing an equilibrium. This period is shown in the graphs by the initial climb and dip. After this feature, the two nodes synchronize their communications and no further changes appear in the graph. The average and standard deviation are taken from this starting point.

The results in Tables 1 and 2 show that the non-blocking send gains a very small increase in performance over the blocking call. This increase in also observed in the receiving end, which suggests that overall, the non-blocking communication is slightly faster, but when used in one-way communication, does not provide a notable performance increase. There is a far more noticeable decrease in the RTT. This is most likely due to the fact that node 0 has time to prepare to receive the reply packet.

The difference between the blocking and non-blocking send results shown in Table 1

	One-Way	Round-Trip
Blocking	145.7 dev: 18.3	1356.5 dev: 5.9
Non-blocking	140.5 dev: 17.6	1282.5 dev: 5.9

Table 1: *Latency for Sending Processor. All times in microseconds. There is only a small change between the blocking and non-blocking calls. Round trip time shows a much larger difference.*

	One-Way	Round-Trip
Blocking	145.2 dev: 3.8	1356.4 dev: 5.8
Non-blocking	140.5 dev: 3.5	1282.5 dev: 5.5

Table 2: *Latency for Receiving Processor. All times in microseconds. The receive data very closely resembles the send data, which indicates that the receiving node is able to process messages immedietly upon receiving them.*

suggests that the blocking send reflects the overhead of the receiver whereas the non-blocking send, which is about 5-10 micro-seconds faster, does not include time spent waiting for MPI_Recv() to be posted. This means that most of the time reflected in the measured time is time spent on the network or on the sending computer. The o_s is not likely to be much larger than the o_r. Thus, the bottleneck experienced when using MPI is caused by the network. Given these results, a comparison can now be made between 100BaseT and commercial parallel processing technologies.

The RTT is a good basis for comparison to other parallel processor technologies. The RTT recorded for MPI in this study was 1282 microseconds. The Intel Paragon parallel processor technology has an average RTT of about 20 microseconds[1]. Clearly, MPI over 100BaseT does not rival the commercially offered parallel architectures in performance. The cost of a 50 node MPI system, however, would only be about $100,000 in comparison to the much more expensive Paragon. Linux, a free variant of UNIX which is supported by

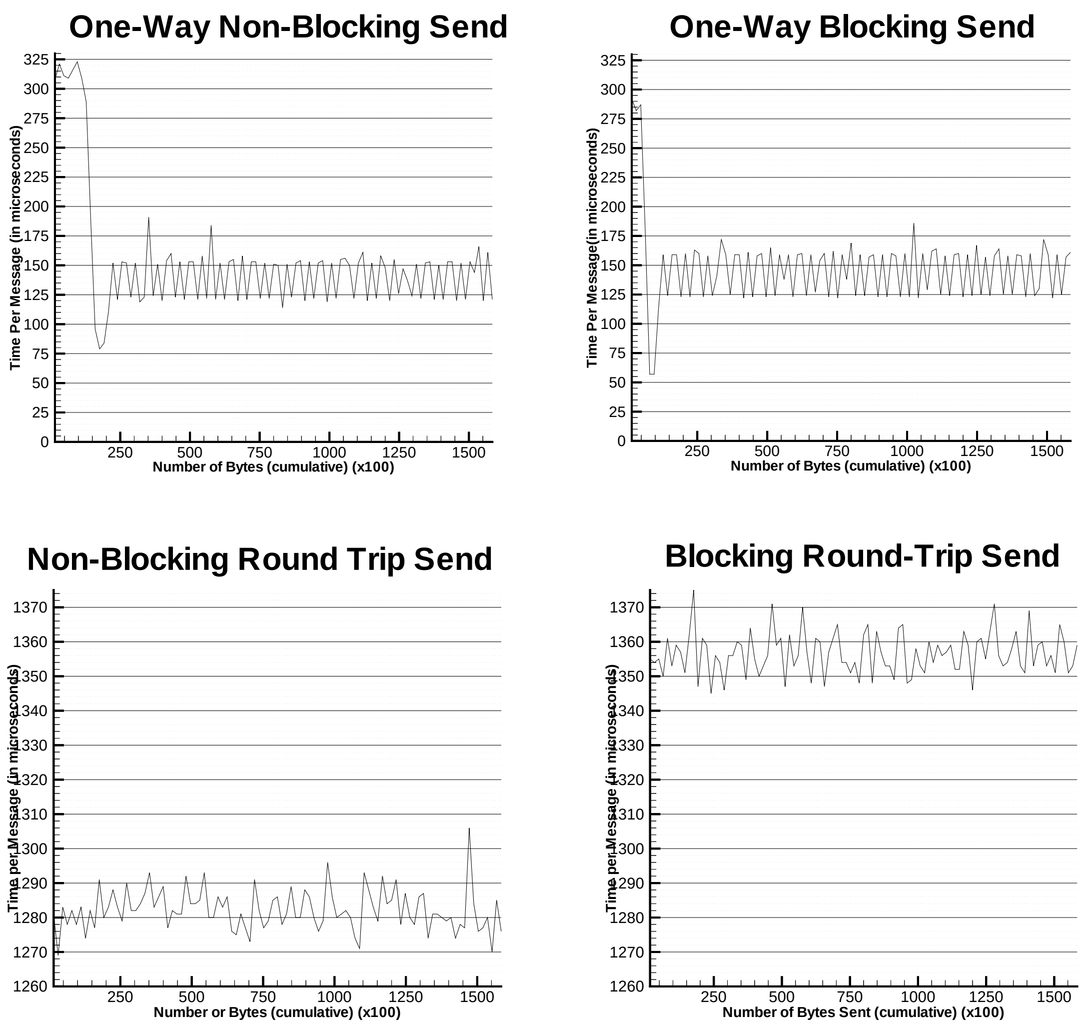

Figure 2: *Time per message for sending node. All byte values should be multiplied by 100 to reflect their actual values.*

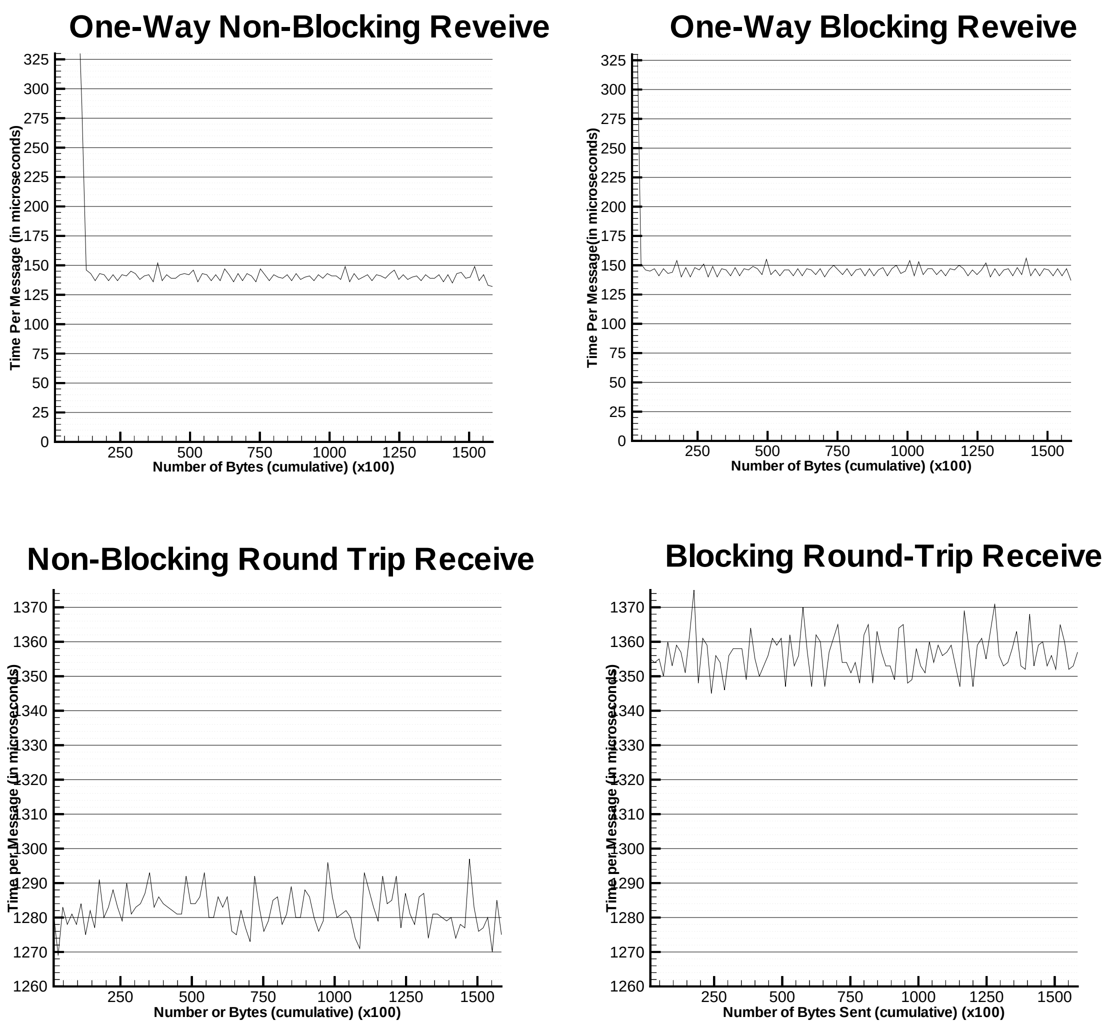

Figure 3: *Time per message for receiving node. All byte values should be multiplied by 100 to reflect their actual values.*

MPICH, could be installed on these systems and MPICH is itself freely available. Despite its slower message passing interface, this computing environment would have considerable processing power for a relatively low price.

One such system has recently been assembled and tested in Los Alamos National Lab[3]. Scientists at LANL have assembled a "mail-order supercomputer" called Avalon. The Avalon computer is made up of DEC Alpha workstations networked using Ethernet. This computer has been rated amongst the 500 most powerful in the world, and cost only $150,000. Avalon's operating system and software are both free. Avalon exemplifies the viability of parallel computing over a network.

3.3 Optimization

During the course of the benchmark, several tests were conducted that resulted in unexpected results. In Figure 4, a line with a constant slope is shown for the first 10 KB. At 10 KB, an increase in the number of packets transmitted per unit time is observed. This indicates that at 10KB, a brief performance increase is experienced. After 50KB, the line stabilizes to the original slope. The slope of the 10KB-50KB increase is twice that of the original.

When running large parallel programs, this can be exploited to yield a large performance increase. Although the possibility was not proven in this study, it is likely that this change in performance is caused by a buffer filling up. The increase is always at 10KB, which suggests that MPI has an initial 10KB buffer. After this buffer is full, MPI changes its network posting method to clear the buffer. Programs could take advantage of this property by avoiding sending large amounts of information (greater than 54KB) at one time.

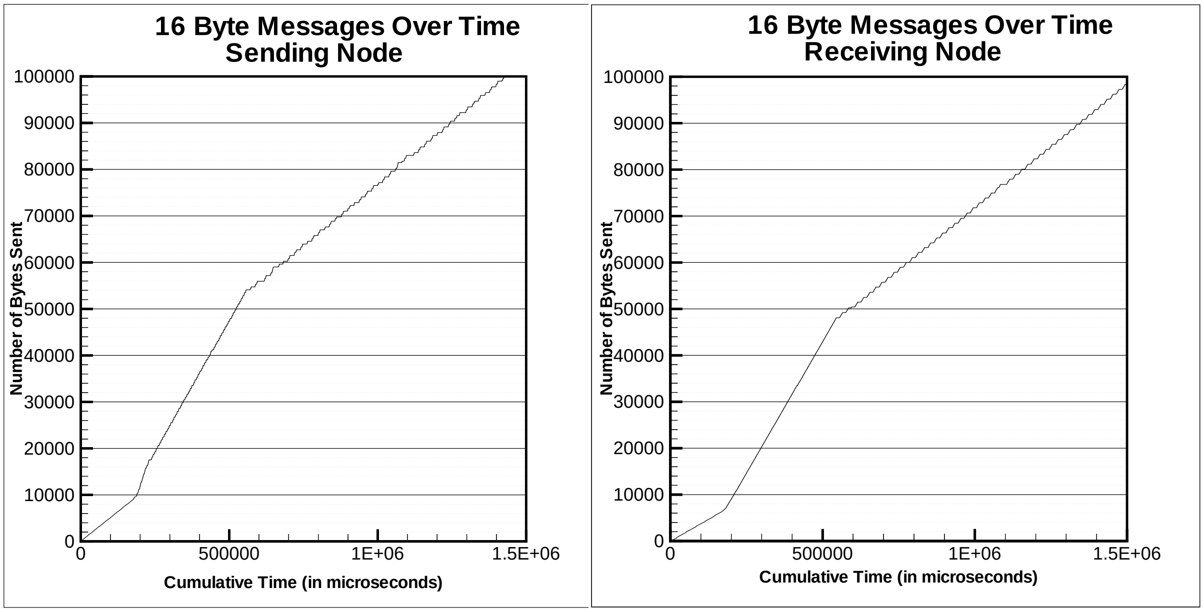

Figure 4: *Bytes transmitted over elapsed time as measured by sending and receiving nodes.*

Message size can also be a major factor in the speed of MPI. When messages that are any power of two in size are sent, a constant latency is observed as is shown in the 16 byte graph in Figure 5. If however, message size is changed to a non-binary value, a bottleneck will occur after a certain number of packets have been transmitted. At this point, latency triples. Figure 5 shows a 5 byte message that exhibits this behavior. By using a power of two for packet size, MPI programs that send many similar packets in bursts are greatly optimized.

Further work can be done on the study of optimizations. The idea of limited buffer is theoretical, and was not tested in this study. A study of the MPI source code and more tests that vary packet size and frequency would provide a solid analysis of the buffer. A more comprehensive comparison can also be used for the latency tests. Using the same code on other MPI-based platforms would provide a more accurate picture of how powerful 100BaseT

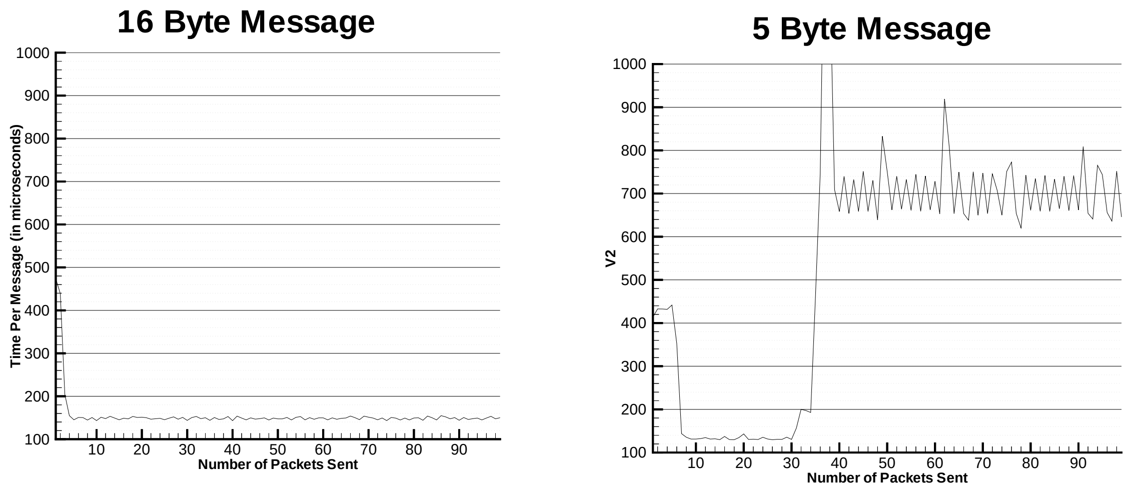

Figure 5: *The 16 byte graph shows no change in the latency because it is a power of two. The latency of the 5 byte message triples after a certain number of packets has been sent. Large gains in performance can be made by using powers of two for message sizes.*

MPI systems are. Other MPI distributions could also be tested. These other distributions will exhibit different characteristics and require different optimizations. Comparing MPICH with other MPI distributions would also show what characteristics are inherent in the design of the MPI standard, and which characteristics are a function of the individual distribution.

4 Conclusion

This study has been focused on two areas. First, the latency of MPI was tested on a 100BaseT switched Ethernet network. It was determined that this network was a slower medium for the network-based parallel processor than commercially available parallel processor technology. While it may not offer the same speed, the MPI supercomputer's cost to performance ratio is low enough to warrant interest. It was also determined that buffer size of the MPICH implementation can be exploited to increase the performance of a communication intensive program. The size of the message also has effects on the performance of the data communication.

5 Acknowledgments

I would like to thank the people at the Research Science Institute and the Center for Excellence in Education for making this summer possible. I would also like to give special thanks to James C. Hoe, my mentor. James is a Graduate Student at MIT. He was very patient and willing to take time out of his busy schedule whenever I had questions. I would also like to thank James' supervisor, Professor Arvind for making his laboratory available for me to use.

References

[1] D.E. Culler, Lok Tin Liu, Richard P. Martin, Chad Yoshikawa.: "LogP Performance Assessment of Fast Network Interfaces", 22 November 1995.

[2] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken: LogP: Towards a Realistic Model of Parallel Computation. In ıIn Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 262-273, 1993

[3] Los Alamos National Laboratory News Release: "Los Alamos mail-order supercomputer among world's fastest" available on-line as: http://www.lanl.gov/external/news/releases/archive/98-089.html

[4] R.P. Martin, A.M. Vahdat, D.E. Culler, T.E. Anderson: "Effects of communication Latency, Overhead, and Bandwidth in a Cluster Architecture"

[5] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. June 12, 1995

[6] Peter S. Pacheco: A User's Guid to MPI. March 30, 1998

A Source Code

```
#include <iostream.h>
#include <fstream.h>
#include "mpi.h"


// Constant Declarations
const NUMSEND=100;
const NUMAVG=100;
const MSG_LEN=16;
const BUFFER_SIZE=1000000;

// Function Prototypes
long long Get_timer();
void Remote_Node(int myrank,int tag);
void Local_Node(int myrank,int tag);
int Write_Output(long long send_input[NUMSEND], long long recv_input[NUMSEND]);

void main(int argc, char** argv) {
        int myrank,p,c,source,dest,tag=50,tempint;
        char message[MSG_LEN];
        MPI_Status status;
// Timer related Declarations
        long long time,diff;


// MPI Initialization Calls
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &p);

        if (myrank != 0) {
                Remote_Node(myrank,tag);
        }
        else {
                Local_Node(myrank,tag);
        }
        MPI_Finalize();
}

//---------------------------Remote_Node()------------------------------------
// This code executes on the remote node.  This node waits for the master ----
// to start sending, then it either recieves or replies back for round    ----
// trip testing.  It converts ticks to microseconds before sending results----
// to the master node.                                                    ----
//----------------------------------------------------------------------------
void Remote_Node(int myrank,int tag) {

        MPI_Status status;
        MPI_Request request;
        int source,dest,c1,c2,tempint;
        char message[MSG_LEN];
        long long tempstorage[NUMSEND];

// Variable Inits
        source=0;
        dest=0;

// Uncomment the MPI_Send line to do RTT testing
        for (c1=0;c1<NUMSEND;c1++) {
                for(c2=0;c2<NUMAVG;c2++) {
                        MPI_Recv(message,MSG_LEN,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
        //              MPI_Send(message,MSG_LEN,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
```

```
                }
                tempstorage[c1]=Get_timer();
        }

//  Transmit Data to a node that can write a file....
        for (c1=1;c1<NUMSEND;c1++) {
                tempint = (int)(((tempstorage[c1]-tempstorage[c1-1])/NUMAVG)/16.66);
                MPI_Send(&tempint,1,MPI_INT,dest,tag,MPI_COMM_WORLD);
        }
}
//---------------------------Local_Node()-----------------------------------
// This is the code written on the master node (node 0)  The main loop -------
// simply issues send and recieve requests and records them to an array. -----
// The rest of the code is dedicated to converting the tick values to    -----
// microseconds.  This node also recieves results from the other node's ------
// timing routine because remote nodes cannot open output files.        ------
//--------------------------------------------------------------------------
void Local_Node(int myrank,int tag) {

        MPI_Status status;
        MPI_Request request;
        char mybuffer[BUFFER_SIZE];
        int source,dest,c1,c2,tempint;
        long long backup_storage[NUMSEND];
        long long recv_storage[NUMSEND],send_storage[NUMSEND];
        char message[MSG_LEN];

// Variable Inits
        source=1;
        dest=1;

// Allocate Buffer Space....
// Note that this buffer will only be used if a MPI_Bsend() is used.
        MPI_Buffer_attach(mybuffer,BUFFER_SIZE);

        cout << "Starting Tests...";

// Uncomment the MPI_Recv call here to do round trip timing.   Change the MPI_Send call
// here as well to MPI_Isend or MPI_Bsend...
        for (c1=0;c1<NUMSEND;c1++) {
                for(c2=0;c2<NUMAVG;c2++) {
                        MPI_Send(message,MSG_LEN,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
        //              MPI_Recv(message,MSG_LEN,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
                }
                backup_storage[c1]=Get_timer();
        }
        cout << "Done.\n";
// Convert Ticks to microseconds.
        for(c1=1;c1<NUMSEND;c1++) {
                send_storage[c1] = (int)(((backup_storage[c1]-backup_storage[c1-1])/NUMAVG)/16.66);
        }
        cout << "Storing data...";
// Recieve results from the remote node (the remote node has already converted
// to microseconds)
        for (c1=1;c1<NUMSEND;c1++) {
                MPI_Recv(&tempint,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
                recv_storage[c1] = tempint;
        }
        Write_Output(send_storage,recv_storage);
        cout << "Done.\n";
}


//----------------------------Get_Timer()-----------------------------------
//---- Get_Timer() returns a number (in ticks) that is currently recorded in -
//---- the register of the processor.  The long long int datatype which is ---
//---- specific to gcc and g++ is used to handle this large number. ----------
```

```
//--------------------------------------------------------------------------
long long Get_timer() {
        register long vUpper;
        register long vLower;

        asm volatile("mftb %1\n" "mftbu %0":"=r"(vUpper),"=r"(vLower):);

        return ((((long long)vUpper)<<32) | vLower);
} // End Get_timer()

//---------------------------Write_Output()---------------------------------
// This function writes the values of the two arrays (send and recieve) ------
// to a file.  To change the name of the output, change the lines below ------
// The two input arrays already are converted into milliseconds earlier ------
// in the program.                                                      ------
//--------------------------------------------------------------------------
int Write_Output(long long send_input[NUMSEND], long long recv_input[NUMSEND]) {
        int tempint,c;
        long long internal_store[NUMSEND];
        long long sum_send,sum_recv;

// Output files:
        ofstream outFile_Recv("dat/bench6.2/recv.dat",ios::out);
        ofstream outFile_Send("dat/bench6.2/send.dat",ios::out);

        if ((!outFile_Recv) || (!outFile_Send)) {
                cout << "Error opening outfile!  Aborting....\n";
                return 0;
                }
        else {
                sum_send = 0;
                sum_recv = 0;
                for (c=1;c<NUMSEND;c++) {
// Uncomment the following two lines to write data-files that reflect cumulative time
        //              sum_send += send_input[c];
        //              sum_recv += recv_input[c];
                        outFile_Send << c*MSG_LEN << "  " << send_input[c] << endl;
                        outFile_Recv << c*MSG_LEN << "  " << recv_input[c] << endl;
                }
                return 1;
        }
}
```