A PERFORMANCE STUDY OF

LAM AND MPICH ON AN SMP CLUSTER

by

BRIAN PATRICK KEARNS

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2003

ABSTRACT

An abstract of the thesis of Brian Patrick Kearns for the Master of Science in Computer Science presented December 4, 2002.

Title: A Performance Study of LAM and MPICH on an SMP Cluster.

Many universities and research laboratories have developed low cost clusters, built from Commodity-Off-The-Shelf (COTS) components and running mostly free software. Research has shown that these types of systems are well-equipped to handle many problems requiring parallel processing. The primary components of clusters are hardware, networking, and system software. An important system software consideration for clusters is the choice of the message passing library.

MPI (Message Passing Interface) has arguably become the most widely used message passing library on clusters and other parallel architectures, due in part to its existence as a standard. As a standard, MPI is open for anyone to implement, as long as the rules of the standard are followed. For this reason, a number of proprietary and freely available implementations have been developed.

Of the freely available implementations, two have become increasingly popular: LAM (Local Area Multicomputer) and MPICH (MPI Chameleon). This thesis

compares the performance of LAM and MPICH in an effort to provide performance data and analysis of the current releases of each to the cluster computing community. Specifically, the accomplishments of this thesis are: comparative testing of the High Performance Linpack benchmark (HPL); comparative testing of su3_rmd, an MPI application used in physics research; and a series of bandwidth comparisons involving eight MPI point-to-point communication constructs. All research was performed on a partition of the Wyeast SMP Cluster in the High Performance Computing Laboratory at Portland State University.

We generate a vast amount of data, and show that LAM and MPICH perform similarly on many experiments, with LAM outperforming MPICH in the bandwidth tests and on a large problem size for su3_rmd. These findings, along with the findings of other research comparing the two libraries, suggest that LAM performs better than MPICH in the cluster environment. This conclusion may seem surprising, as MPICH has received more attention than LAM from MPI researchers. However, the two architectures are very different. LAM was originally designed for the cluster and networked workstation environments, while MPICH was designed to be portable across many different types of parallel architectures.

ACKNOWLEDGEMENTS

I gratefully acknowledge and thank my advisor, Dr. Karen Karavanic, for her help and guidance, and for allowing me to explore parallel programming on the Wyeast cluster.

This work was in part based on the MILC collaboration's public lattice gauge theory code. See http://media4.physics.indiana.edu/~sg/milc.htlm.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

This thesis compares the performance of two freely available implementations of the Message Passing Interface (MPI) on a cluster of dual-CPU SMPs built from readily available, low cost, Commodity-Off-The-Shelf (COTS) components and running mostly free software. These types of clusters have received a lot of attention during the last decade, serving as cost-effective alternatives to Massively Parallel Processors (MPPs) and Symmetric Multiprocessors (SMPs). They are distributed memory systems typically employing some sort of *message passing* to transfer data among the processes running on different processors. Laboratories, universities, and even some high schools now have access to parallel computing power once reserved for the private sector and a small number of affluent institutions.

The main components of clusters are hardware, system software, and networking. Hardware based decisions primarily involve aspects of individual node configuration. Networking considerations include the speed and topology of the network that will be used to connect the nodes of the cluster. System software includes, but is not limited to, the operating system and software necessary for achieving communication among the cluster nodes. This research focuses on system software performance, specifically the MPI message passing library layer. It compares the performance of LAM and MPICH on: *HPL*, the High Performance Linpack bench-

mark; *su3_rmd*, an application based on four-dimensional SU(3) lattice gauge theory; and a series of point-to-point *bandwidth comparisons*.

The demand for parallel processing is high. Application areas with computational needs that require parallel processing include biomedical research, fluid dynamics, global climate modeling, molecular modeling, nuclear test simulations, astronomic simulations, and many others. For example, one common manner by which to model the atmosphere is to divide it into a number of cells. Numerical computations can be performed within each cell to analyze and predict weather patterns and atmospheric changes. However, such computations are intractable with current serial computing capabilities. Wilkinson and Allen in their 1999 text [26], for example, note that an application forecasting the weather over the next 7 days would require over 100 days to achieve results on a single computer operating at a rate of 100 MFLOPS (100 Million Floating Point Operations per Second). Even today's uniprocessors capable of operating at over 500 MFLOPS fall well short of the 1.7 TFLOPS required to perform the forecasting operation in 10 minutes. Further, assuming we could imagine a uniprocessor operating in the TeraFlop range, it is nearly impossible to imagine memory access time not becoming a bottleneck.

Parallel applications have been and continue to be written for problems such as the one described above, and machines exist on which to run them. Highly specialized

massively parallel processors (MPPs), and symmetric multiprocessors (SMPs), have historically been used to run parallel applications, and are being developed and used today. These machines are extremely powerful computers consisting of multiple processors existing in a single box. MPPs are characterized by hundreds and even thousands of processors each having its own memory and running its own operating system. Some sort of message passing mechanism is required to transfer data among the processors. SMPs, on the other hand, are tightly coupled machines typically with fewer processors than MPPs, all of which share the same memory and run the same operating system. The advantage of using MPPs and SMPs is that custom systems can be designed to run parallel applications in an optimal manner according to the application domain. The disadvantages include extremely high cost (especially MPPs), poor scalability (especially SMPs), and difficult update and maintenance characteristics.

In the mid 1990's an effort began to see whether systems could be built from PCs that would rectify the above problems. Thomas Sterling and Donald Becker at The Center of Excellence in Space Data and Information Sciences (CESDIS) developed a system they called "Beowulf" in 1994 [21], a dedicated computing cluster consisting of sixteen nodes constructed from off-the-shelf PC's (DX4 processors) connected by a 10 Mbps Ethernet. Each PC, or node, had its own memory; the system

was truly distributed. As efforts in cluster research increased, the results showed that these types of systems were well-suited for many parallel applications.

Today, many universities and research laboratories are building and using clusters, and these systems have made a significant appearance (80 entries) on the most recent list of the world's 500 fastest computers [31]. PC clusters that are obtainable by universities and research centers are often characterized by: inexpensive, off-the-shelf hardware; some variant of the Unix/Linux operating system; Fast Ethernet, Gigabit Ethernet, or Myrinet networking; and freely available, open source software. The term *Beowulf Cluster* is often used to refer to these types of systems. Other names include *Pile-of-PCs (PoPCs),* and *Commodity-Off-The-Shelf (COTS)* clusters.

One of the latest trends in low cost, commodity cluster construction has been the development of clusters of SMPs. For example, Portland State University maintains an SMP cluster called Wyeast which is built from 48 dual-CPU Pentium III (866 Mhz) machines connected by a Fast Ethernet (100 Mbps). Most of these systems use a high level message passing facility for transferring data among the processes executing the parallel program. The underlying implementation of the message passing facility uses shared memory for intra-node data transfer, and uses a networking protocol such as TCP/IP or VIA for inter-node data transfer.

A very important software consideration in cluster design is the choice of the message passing facility. One attractive option is the use of message passing libraries. At one time, PVM (Parallel Virtual Machine) was the primary library used in distributed systems. It was the first truly portable message passing library, with a large number of users. Over the last decade, MPI (Message Passing Interface) has arguably become more widely used than PVM, due in part to its existence as a *standard*. As a standard, MPI is open for anyone to implement, as long as the rules of the specification (function bindings and behavior, and specific constants) are followed. As a result, several proprietary and open source implementations of MPI exist. Since low cost is one of the primary goals of cluster construction, many universities and research laboratories have opted for the freely available open source implementations. Of these, two have become increasingly popular: MPICH (MPI Chameleon) and LAM (Local Area Multicomputer).

MPICH has tracked the MPI standard from the beginning; has been the focus of many freely available MPI profiling tools; has been installed and tested on virtually every parallel system available (including clusters); and is the foundation for a large number of proprietary MPI implementations. LAM has gained a following among cluster users due to its usability, fast static process creation, and support for dynamic process creation. Preliminary research of this thesis found that many current and potential parallel programmers and cluster operators are interested in the perfor-

mance of LAM and MPICH on clusters, especially the performance of the most current versions.

This thesis aims to compare the performance of LAM and MPICH in an effort to provide performance data and analysis of the current releases of each to the cluster computing community. Although other research has been performed relating to performance comparisons of LAM and MPICH (see chapter 5), we feel that this research is unique because it compares the most recent releases of each library (LAM-6.5.6 and MPICH-1.2.4) of any studies we have found, it was conducted on an SMP cluster, and it provides a vast amount of data gathered from three major categories of experiments. Specifically, the accomplishments of this thesis are:

- Installation and comparative testing of the High Performance Linpack (HPL) benchmark;
- Installation and comparative analysis of a parallel physics application involving four-dimensional SU(3) lattice gauge theory; and,
- Bandwidth comparisons on eight point-to-point MPI communication primitives.

Chapter 2 of this thesis provides background information on cluster computing, message passing, MPI, LAM, and MPICH. The testing environment and installation of software used in this research is discussed in chapter 3. Experiments and

results are provided in chapter 4. Chapter 5 discusses other work related to MPI performance on clusters, and chapter 6 closes with future work and conclusions.

2 Background

In an effort to lay a framework for the later chapters, this chapter provides a brief background on clusters (section 2.1), Beowulf type clusters (section 2.2), two other parallel architectures (section 2.3), message passing (section 2.4), and MPI (section 2.5), including the LAM and MPICH implementations. Throughout this discussion, the words *process* and *processor* will be used. More than one process can run on a single processor, but the processes will not execute concurrently - rather they will be interleaved in a time-scheduled fashion by the operating system. If there is one process running on each processor, however, the processes can execute concurrently. A *node* can house one or more processors. Regardless of whether the systems to be mentioned have either one or more than one processor per node, the context of this discussion is that there is a one-to-one correspondence between processes and processors: there is one process running on each processor, and the processes can run concurrently.

2.1 What is a Cluster?

A cluster can be defined in very abstract terms to be a distributed computing system that consists of a collection of whole computers connected by a network that is used as a single computing resource [16]. Each computer, or *node*, contains at the very least a motherboard, one or more processors, main memory, cache memory, one or

more busses, one or more network interface cards (NICs), and optionally other components. Not all clusters have permanent storage (hard disk) on each node, although many do. Clusters are *parallel* systems because they are capable of executing more than one instructional flow of control at any given time (due to fact that there exists more than one processor in the system) if they are programmed to do so. For example, process A can be running on processor A at the same time as process B running on processor B. This is in contrast to, say, a multithreaded application or a Unix fork-exec program running on a single-CPU computer, where the operating system interleaves the ordering of the different threads of execution (*lightweight processes),* or heavyweight processes in the case of the fork-exec model, so that no two threads or processes are running at the same time.

Clusters are also *distributed* systems because each process (or group of processes, in the case of clusters of SMPs) runs on a processor (or group of processors) on a single node, and has direct access only to its local memory. Because there are multiple nodes in a cluster system, some sort of communication is required among processes running on different nodes in order for all processes to have access to all of the memory in the system. Therefore, another possible definition of clusters is that they are loosely coupled distributed systems that are primarily used for parallel processing. Not all distributed systems are used primarily for parallel processing. A specialized distributed system may be set up for a distributed database, for example,

where one of the primary reasons for the distributed environment is high availability (data can be replicated across the different computers).

There is not one single type of cluster. Individual workstations that are normally used in a serial manner by computer users have been networked at campuses and at companies to provide parallel computing capabilities during off-peak workstation use times. These are often referred to as “cycle harvesting” and “workstation farm” systems. Clusters have also been developed for commercial sale by companies such as DEC, Tandem, and IBM. In the mid 1990’s, research began to determine whether clusters could be built from personal computers constructed from readily available, Commodity-Off-The-Shelf (COTS) components. The research proved to be successful, and produced a type of cluster commonly referred to as a Beowulf.

2.2 Beowulf Clusters

The first Beowulf [21] was constructed in 1994 by Thomas Sterling and Donald Becker at The Center of Excellence in Space Data and Information Sciences (CES-DIS). It consisted of 16 PCs with DX4 processors, connected by a 10 Mbps Ethernet. Another Ethernet was later added to allow for channel bonding - a configuration in which the data is striped across two networks. Due to the success of this initial system, a lot of research has been conducted in the Beowulf realm over

the past eight years. Combined with better network technology and increasing processor speeds (with lower costs for both), this research has produced systems capable of executing large parallel applications at a much less expensive price than MPPs, and with better scaling capabilities than SMPs.

Today, Beowulf clusters are still built from PCs that are connected by a network. Individual nodes are constructed from inexpensive, COTS hardware components, and each node generally runs some variant of the Unix/Linux operating system, and other free or low cost system software. Examples of networks for Beowulfs include Fast Ethernet, Gigabit Ethernet, Myrinet, and FDDI. If a central switch is used, each PC, or node, has a direct connection to every other node in the cluster (with the switch acting as an intermediary). Commonly used processors for Beowulfs include those from the Intel Pentium family, various AMD processors, and the DEC Alpha. According to Sterling et al. [22], a Beowulf node typically has between one and four processors. The Wyeast cluster at Portland State University, for example, is a Beowulf type SMP cluster constructed from 48 nodes, each of which has 2 processors.

2.3 Other Types of Parallel Systems

Clusters are not the only types of systems used for parallel programming. Two other categories of parallel platforms, SMPs and MPPs, are often mentioned in the parallel programming literature, and are actively being developed and used today.

2.3.1 MPPs

Highly specialized MPPs (Massively Parallel Processors) have existed since the 1960's. Examples include the Intel Paragon and the Thinking Machines CM5. They consist of hundreds, even thousands, of processors in a single box, each having its own memory and running its own operating system. Message passing can be used for communication among the processes. MPPs can be fine-tuned for a specific application domain, or even for a specific application, and can therefore execute the application very quickly. A primary disadvantage of MPPs is that they are very expensive (can cost more than $10 million dollars [32]), and therefore are generally unavailable to many universities, research laboratories, and certainly most curious individuals. Differences between MPPs and clusters include: price (clusters are less expensive); system layout (the workings of an MPP exist in a single box, whereas clusters are composed of multiple boxes, or nodes, connected by a network); and power (a powerful MPP can be constructed for a specific application,

and that application would run faster than it would on a cluster; another way to put it is that the fastest MPPs are faster than the fastest clusters).

2.3.2 SMPs

SMPs (Symmetric Multiprocessors) have been around since the early 1970's. They are systems composed of multiple processors existing in a single box, all of which share the same memory and run the same operating system. Therefore, memory access is local to each processor. Examples of SMPs include the Intel Pentium Pro Quad and the Sun Enterprise. If message passing is used on an SMP, it is generally implemented by platform-specific, low level shared memory constructs. SMPs are typically less expensive than MPPs and generally do not scale to high numbers of processors. This means that as more processors are added to the system, the contention for memory on the bus becomes higher (due to the shared memory architecture). This leads to slower memory access times, and therefore slower overall execution time. The point at which adding more processors to an SMP becomes disadvantageous varies from machine to machine, though some define it to be in the range of 16 to 24 processors. This scalability problem has somewhat been alleviated by the NUMA (Non Uniform Memory Access) system - an SMP subtype first developed in 1997 in which the memory area is divided into different sections, resulting in lower contention for memory among the processors. Memory access is non-uni-

form, meaning that some memory access times take longer than others. However, the low *average* memory access time theoretically allows these systems to scale far higher than traditional SMPs.

An interesting convergence of clusters and SMPs has been the recent proliferation of affordable “hybrid systems”, or clusters of SMPs built from inexpensive components and running mostly free software. The Wyeast cluster at PSU, as mentioned earlier, is an example of such a system. It is a Beowulf-type cluster because it was designed from Beowulf principles, and can also be referred to as an SMP cluster because each node contains two processors. These systems are often programmed using a message passing programming model, implemented with message passing libraries. Message passing between processors on the same nodes is implemented by the library with low-level shared memory constructs, while message passing between processors on different nodes is implemented in terms of some sort of networking protocol (TCP/IP, for example).

2.4 Message Passing

As a process runs, it often needs to access data stored in main memory. In a shared memory system such as an SMP, each process has access to all of the memory in the system. However, because a cluster is a distributed system, each processor in a clus-

ter has direct access only to its local memory. In order for process 1 on node A to have access to the memory of process 2 on Node B, some sort of communication is required. One way to achieve this is with *message passing*. In the simplest sense, message passing refers to one running process sending information to another running process, which receives the information (see figure 1).

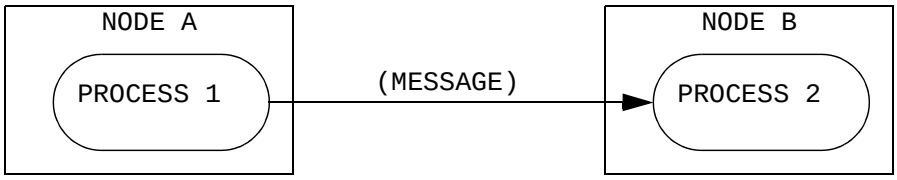


Figure 1: Point-to-Point Message Send and Receive

This figure depicts an example of point-to-point communication with message passing. Process 1 running on node A sends a message to process 2 running on node B, which receives the message. The message is not lost to process 1; it still has the data. Now process 2 has it as well. Process 1 can wait until process 2 receives the data before continuing, or it can return immediately from the send and execute its next instruction. The former is known as a blocking send, while the latter is a non-blocking send.

The message includes actual *data*, such as a large vector or a small integer, and also contains information commonly referred to as the *envelope*, which includes the source and destination addresses, the message identification tag, the size in bytes of the data being sent, and possibly other information.

The communication shown in figure 1 is an example of *point-to-point* communication. The message travels directly from one process to another process. After process 1 posts the send operation, it can either wait until process 2 posts its receive before continuing with its next instruction, or it can return immediately from the send and go on to the next instruction. These two scenarios are referred to as a *blocking send*, and a *non-blocking send*, respectively. The same situation holds true for process 2 as well; it can post a *blocking receive* or a *non-blocking receive*. Blocking guarantees that the send/receive will not complete until either the corresponding receive/send has been posted, or the message has been buffered by the system. This is advantageous because the programmer is assured that the memory allocated for the function call arguments is safe to use after the operation completes.

Blocking operations have two primary drawbacks, however. One is that they can lead to deadlock if the program is not written correctly and relies on the existence of system buffering for large message sizes. The other problem with blocking operations relates to performance. If process 1 posts a blocking send and has work it could be doing that does not depend on the arguments to the send call (call it “other work”), time is wasted because it is blocking - waiting for process 2 to post the receive. A non-blocking send would fix this problem. Process 1 returns immediately from the send, performs its other work, and then calls a *test* or *wait* operation to see if the receive has begun. If it has, process 1 can now modify the memory used

for the send arguments. The key is that process 1 was able to perform its other work without having to wait for process 2 to receive the message.

Point-to-point communication is not the only type of communication possible in a message passing environment. Synchronization routines, called *barriers*, exist to cause all of the processes to start or stop at the same time. Also, *collective communication* is possible, in which all processes (or possibly all processes within a certain group) communicate, and data is transmitted. Some collective operations also act implicitly as barriers. Examples of collective communication include *broadcast* (one process broadcasts data to all other processes), *all-to-all* (each process sends data to every other process), and *aggregate operations* (all processes send data to one specific process, which adds/averages/etc. the data).

Message passing in clusters is usually available in the form of *message passing libraries*. These libraries provide a collection of functions for programmers to call in languages such as C, C++, and Fortran. For many years, PVM (Parallel Virtual Machine) [7] was the primary message passing library used in distributed computing systems. PVM version 1 was developed at Oak Ridge National Laboratory in 1989 for use by researchers in the lab. The first official release was PVM version 2 in 1991. A complete rewrite occurred in 1992, producing PVM version 3, released in 1993. The primary goal of PVM was to develop a portable message passing envi-

ronment for heterogeneous distributed systems, providing a “parallel virtual machine” which gives the user the illusion of programming one single computer. One of the most important features of PVM is *dynamic process creation,* which refers to the ability for the program to spawn processes at run time.

PVM is still being used today. However, PVM is not an open standard in the sense that there is really only “one PVM”. As a result, a programmer using PVM is limited. Realizing the need for a message passing API that would be accepted as an open standard, researchers in the early 1990’s developed MPI (Message Passing Interface). MPI became very popular as a message passing standard among parallel programmers, and many freely available implementations have been developed. The popularity of MPI is evident by the attention it receives in many of the current research papers on clusters, message passing, and parallel profiling (see [1, 6, 18, 23, 24, 25] for example); and by the increasing number of proprietary and freely available implementations. In the text, High Performance Cluster Computing [2], R. Buyya asserts that MPI is the message passing library most widely used today at universities and research laboratories.

2.5 MPI

MPI stands for *Message Passing Interface.* It is a message passing standard that was developed in the early 1990's by the MPI Forum [20] - a group of researchers and professionals from more than forty organizations mainly from the United States and Europe who began meeting in 1992 and continues to meet today. The primary goal of the MPI Forum was to provide a *standard,* so that different research groups and companies could develop different implementations of the library, as long as they adhered to the rules of the standard (function bindings, specific constants, and function definitions). As a consequence, many different implementations of MPI exist today, but they all have the same interface. Therefore, MPI programs are portable among the various implementations. Aside from the possibility of a specific implementation not supporting a specific function or set of functions (which does occur - for example, LAM-6.5.6 does not support cancelling of sends in MPI-1.1), an MPI program should be able to be compiled with any implementation; the only difference at runtime should be performance.

MPI, like PVM, is truly portable across many platforms. One of the problems with early message passing environments was that they were platform-dependent. A program that ran on one parallel system would have to be re-written in order to run on another parallel system. This is not the case with MPI. An MPI program compiled

on a cluster running Linux, for example, can be re-compiled on a workstation network running Solaris Unix without being rewritten.

MPI is not a programming language. It is implemented as a collection of library routines with bindings for C, C++, and Fortran. Although it contains many functions, one commonly-heard phrase is “minimal MPI”, meaning that a programmer familiar with any of these languages can write meaningful programs with only six MPI functions. An example of a “Hello, World” program (not meaningful, but demonstrative) written in C using MPI calls is shown in figure 2 (next page). In this example, each process knows which part of the code to execute by following the branching statement.

Both MPI implementations discussed in this thesis provide environments in which to compile and run the programs. For compilation, wrappers are provided called *mpicc* (for C programs), and *mpif77* and *mpif90* (for Fortran programs), which essentially compile the program with the system’s C compiler or Fortran compiler and link in the appropriate MPI libraries. While convenient, these wrappers are not necessary, as one can manually link in the MPI libraries at the command line. To run the program, a script called *mpirun* is provided. This script enables the creation of the MPI processes on the various processors, and loads the executable program onto each node.

```
#include <stdio.h>
#include <string.h>
#include “mpi.h”
#define MAX_MSG 100

main(int argc, char* argv[])
{
   int my_rank;            /* rank of process */
   int num_procs;          /* number of processes */
   int source;             /* rank of sender */
   int dest;               /* rank of receiver */
   int tag = 0;            /* tag for messages */
   int size;               /* size of message */
   char message [MAX_MSG]; /* storage for message */
   MPI_Status status;      /* return status for receive */

   MPI_Init(&argc, &argv);                    /* start up MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);   /* local rank */
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs); /* total processes */

   /* All processes except process 0 send a hello message to process
   ** 0, which displays the messages.
   */
   if (my_rank != 0) {
      sprintf(message, “Hello World from process %d!”, my_rank);
      size = strlen(message) + 1;
      dest = 0;
      MPI_Send(message, size, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   }
   else {
      for (source = 1; source < num_procs; source++) {
         MPI_Recv(message, MAX_MSG, MPI_CHAR, source, tag,
                  MPI_COMM_WORLD);
         printf(“%s\n”, message);
      }
   }
   MPI_Finalize();
   exit(0);
}
```

Figure 2: A Simple MPI Program, “Hello, World”

Shown is a simple “Hello, World” program written in C with calls to the MPI library. Each process knows which part of the code to execute by following the branching statement. In this program, each process except process 0 creates a “hello” message and sends it to process 0. Process 0 receives and displays each message. If this program was named “my_hello”, to run this program with 4 processes, one could execute at the command line:
mpirun -np 4 my_hello.

MPI has been developed in stages, with each version of the standard made available in the *MPI Document*. This document is available at the MPI Forum web site [34]. The MPI-1.0 Document was released on May 5, 1994. This was followed by MPI-1.1 in 1995. In 1997, the MPI-1.2 and MPI-2 documents were released (the MPI-1.2 document is contained in the MPI-2 document). The current standard supported by most implementations is MPI-1.2. MPI-2 specifies several changes to the standard, one of the most notable being dynamic process creation and management. Prior to MPI-2, all MPI processes were specified before the program was run. With MPI-2, processes can create new processes at run time. Many MPI implementations currently support part of the MPI-2 standard. LAM, for example, provides support for dynamic process creation and management.

This research compares two freely available, open source implementations of MPI: MPICH and LAM. MPICH has tracked the MPI standard from the beginning; has been the focus of many freely available MPI profiling tools; has been ported to virtually every parallel system available (including clusters); and is the foundation for a large number of proprietary MPI implementations. LAM has not received as much attention as MPICH from vendors and tool designers, and has been ported primarily to homogeneous and heterogeneous clusters. However, it has gained a following among cluster users due to its usability, fast static process creation, and support for dynamic process creation.

2.5.1 MPICH

MPICH (MPI Chameleon) [9, 10] was first developed at Argonne National Laboratory in 1993 in an effort to provide an implementation that could be quickly ported to different systems and that would closely track the MPI standard definition as it evolved. It has been ported to many different systems, including: clusters running different Unix/Linux variants; Windows NT and Windows 2000 networks; MPPs such as Thinking Machines CM5, IBM SP, and Intel Paragon, and SMPs. MPICH is still being developed at Argonne today. The current release is MPICH-1.2.4. It supports all of MPI-1.2 and part of MPI-2, but does not support dynamic process creation. The MPICH developers are currently working on a release of MPICH that supports all of MPI-2.

MPI process creation in MPICH is achieved by remote shell invocation upon execution of the mpirun script. There is no virtual machine to boot up as with PVM. Message passing between processes on different nodes is achieved with TCP/IP, and message passing between processes on the same node is implemented by platform-specific shared memory constructs, using semaphores for process synchronization.

MPICH is built in a layered approach. All MPI functions and macros [layer 1] are implemented by functions and macros defined in the *Abstract Device Interface* (ADI) [layer 2]. Some of the ADI functions are implemented in terms of the target

machine, and others are implemented by another, lower layer, the *Channel Interface* (CI) [layer 3]. According to the original MPICH implementation document [10], the reasoning for the layered approach was to provide an implementation that could be ported quickly to different platforms, using as much shared code as possible, while maintaining a level of high performance.

The ADI is not specific to MPI. The documentation states that any higher level message passing library can be built on top of it. Different implementations of the ADI exist, allowing for portability across a wide variety of parallel platforms. Much of the code in the ADI is shared among the various MPICH ADI implementations. Code that is not shared is fine-tuned for specific implementations in an effort to achieve high performance.

The primary purpose of the Channel Interface is to define functions for low-level data transfer among processes. Because this interface is smaller than the ADI, MPICH can be quickly ported from system to system in a minimal manner using the Channel Interface, and then gradually expanded to implement the full ADI. The channel interface most commonly used for clusters is ch_p4, where p4 stands for Portable Programs for Parallel Processors, a small, highly portable parallel programming library.

2.5.2 LAM

LAM (Local Area Multicomputer) [30] was originally developed at the Ohio Supercomputing Center in 1989. The project was later moved to The Laboratory for Scientific Computing (LSC) at the University of Notre Dame. The LSC moved to Indiana University in the fall of 2001, and that is where the work on LAM is being done today. The current release of LAM is LAM-6.5.6. It supports all of MPI-1.2 except the cancellation of sends, and it supports part of MPI-2, including dynamic process creation and management.

The primary purpose of LAM was to provide an MPI library that would run efficiently on heterogeneous workstation networks and clusters. It has been tested on clusters and networked workstations running various Unix and Unix-like operating systems, including Solaris, OpenBSD, Linux, Mac OS X, IRIX, HP-UX, and AIX. As dedicated clusters have gained popularity, LAM has also gained popularity. Reasons for its popularity include the fact that it was originally developed for the cluster architecture, and the increasing amount of evidence that it has performed well on clusters (see [3, 13, 14, 27], for example).

LAM runs on each node as a user-level daemon. The nodes are listed in a boot file, and the user issues the *lamboot* command to get the daemons started. In this way it is similar to PVM. Because the daemons are running when the MPI program is

started, the MPI processes are created quickly. This is in contrast with MPICH, where no user-level daemons are running on the nodes before the program starts. As stated earlier, MPI process creation in MPICH is achieved with the use of a system-level daemon or by remote shell invocation. As a result, some research has indicated that process startup time is generally slower in MPICH [3].

LAM is implemented in two layers [29] which meet at an interface called the RPI (Request Progression Interface). The upper layer is the portable MPI API, and the lower layer consists of bindings and functions for two separate running modes, known as *lamd mode* and *c2c mode*. A user can specify which mode to use as an argument to the mpirun command (c2c is the default). Both implementations of the lower layer employ user-level LAM daemon processes for MPI process creation. However, the daemons may or may not have anything to do with the messages. The difference between lamd mode and c2c mode is that in lamd mode, the daemons also act as message intermediaries (an MPI process on node A sends a message to an MPI daemon on node A; the daemon on node A then sends the message to a daemon on node B, which finally sends the message to an MPI process on node B). In c2c mode, no such daemon activity occurs. LAM daemons are still used for MPI process creation, but they do not interfere with messages: a message from process 1 on node A will be sent directly to process 2 on node B.

A natural question to ask is: why have the daemons interfere with messages at all? The answer is that these daemons keep track of recoverable information about the messages that can be used for program monitoring and debugging. The trade-off is slower performance. Therefore, programs run in lamd mode will generally be slower than those run in c2c mode, but they will be able to be debugged and monitored directly with tools provided in the LAM environment, whereas the c2c programs will not. Interestingly, lamd mode uses UDP for message transfer, whereas c2c mode uses TCP. In fact, for this reason, some research [3] has found lamd programs to run faster than c2c programs in previous releases of LAM. It would be interesting to perform a full-blown study comparing the performance of lamd mode and c2c mode.

Preliminary testing using Linpack comparing the two modes in the research of this thesis showed that c2c mode was faster than lamd mode; therefore c2c mode for LAM was used in the comparisons with MPICH. One theory as to why the c2c programs ran faster than the lamd programs is that the absence of the daemon intermediaries outweighs the use of UDP in the current version of LAM.

3 Testing Environment and Software Installation

This chapter outlines the test environment and software that was installed to conduct the research of this thesis. All tests were performed on a dedicated group of nodes on the Wyeast Cluster located in the High Performance Computing Laboratory at Portland State University.

3.1 Wyeast Architecture

Wyeast consists of a head node and 48 dual-CPU SMP nodes (see figure 3, next page). Two 100 Mbps Fast Ethernets are installed on the cluster, using two central switches, and two Network Interface Cards (NICs) per compute node. The head node has three Gigabit NICs and connects to the outside world and to each of the two switches via Gigabit Ethernet. Each compute node has two 100 Mbps Fast Ethernet NICs which are used to connect to the switches. There is no common file system within the cluster.

It is not necessary to have two networks within the cluster. The benefits include availability and channel bonding potential. With channel bonding, the data is striped across two networks, leading to increased bandwidth. In the current configuration of Wyeast, the NICs on the designated MPI nodes are not set up for channel bonding. Therefore, only one of the Fast Ethernet networks is being used within the

cluster for all of the MPI test runs. Channel bonding research is currently being conducted, and MPI performance with and without channel bonding remains an open research area within the cluster. Tables 1, 2, and 3 (next page) present specifications on Wyeast's hardware, network, and software, respectively.

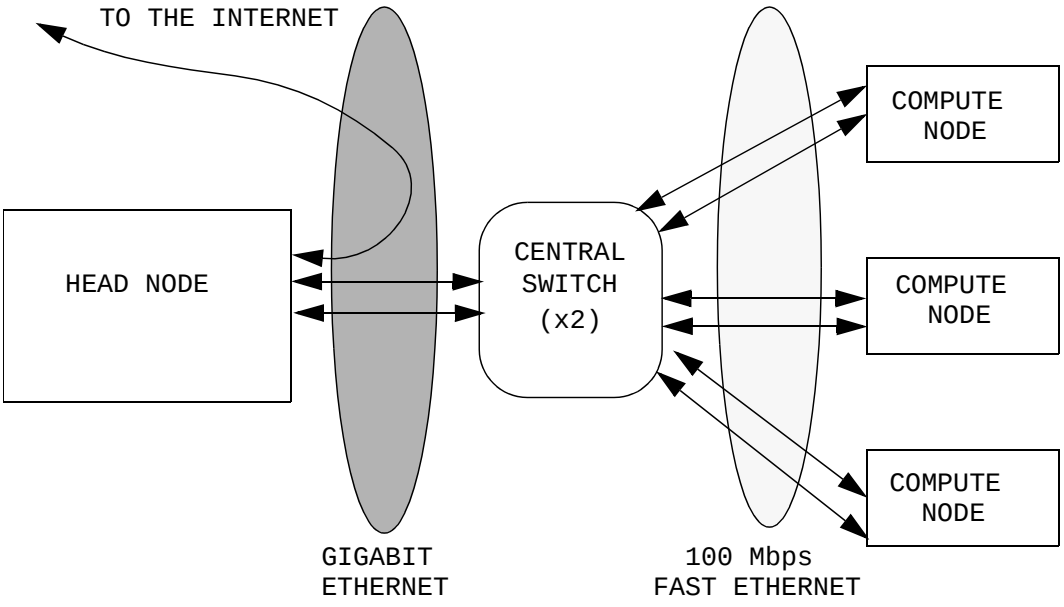


Figure 3: Abstract View of the Wyeast Cluster

This figure provides a high-level view of the Wyeast Cluster at Portland State University. Wyeast is an SMP cluster consisting of a head node and forty-eight compute nodes. The head node contains two 1Ghz processors and the compute nodes each contain two 866 Mhz Processors. The head node has a Gigabit Ethernet connection to the outside world and to each of the two switches. The compute nodes each have a 100Mbps Fast Ethernet connection to each of the two switches. Along with the head node, twelve compute nodes were used in this research.

Table 1: Wyeast Hardware Specification

Number of Nodes	48
Processors Per Node	2
Processor Type	Intel Pentium III
Processor Speed	866 Mhz
Motherboard	ECS d6vaa
Main memory	512 MB RAM per node
Secondary Storage	20 GB per node

Table 2: Wyeast Network Specification

Type	Fast Ethernet (100 mbps)
Topology	Double Switched Ethernet
Network Switch (x2)	Cicso 3548 with Gigabit Uplinks
Compute Node NIC (x2)	3Com 3c905c Fast Ethernet
Head Node NIC (x3)	SysKonnect Gigabit

Table 3: Wyeast Software Specification

Operating System	RedHat 7.2 (Linux)
Kernel Version	Linux 2.4.7
C Compiler	gcc 3.0
Fortran Compiler	g77 2.96
LAM	6.5.6
MPICH	1.2.4

3.2 Node Configuration

Along with the head node, twelve compute nodes were available for the MPI experiments, for a total of 26 processors. We developed scripts on the head node to facilitate many tasks, including copying programs and input files over to the compute nodes, and performing system-wide resource recovery. Because these scripts were needed frequently, we decided to use the head node as the launch node for mpirun.

Due to the absence of a common file system, the MPICH tests required the use of a *proc group file* for the mpirun command to work properly. This file is created automatically by mpirun under most circumstances, using information stored in an MPICH "machines file", and information entered on the mpirun command line. However, for several situations (heterogeneous systems, absence of a common file system, and others) MPICH requires the user to specifically create this file, and then direct mpirun to read it by using the *-p4pg filename* option to mpirun. Each time a new program is run or a different node configuration is used, this file must be changed. In addition, the local node (the node from which the mpirun command is launched) must be included in the proc group file. Therefore, with the local node being the head node in our case, the head node was required to be one of the compute nodes for the MPICH runs. For fairness in testing, it was decided to use the head node as a compute node for LAM as well, although LAM does not require this with our configuration.

3.3 Operating System and Compilers

Red Hat Linux 7.2 (Linux kernel 2.4.7) is the operating system for all nodes used in this research. Red Hat 7.0 was originally installed on all of the nodes in the cluster. We decided to stay with Red Hat, and update the MPI nodes to the most current release available at the time of the MPI tests. The C compiler used for compiling the libraries and compiling MPI programs is gcc-3.0. The Fortran compiler is g77-2.9.6. These compiler choices were made because they were already present on the cluster before the MPI research began; they successfully compiled both libraries; and both libraries subsequently compiled and successfully ran large MPI test programs. Time limitations prevented an investigation and possible subsequent installation of alternative compilers to determine which compilers are optimal for MPI on Wyeast.

3.4 LAM, MPICH, and MPE

LAM-6.5.6 was downloaded from: http://www.lam-mpi.org/download. Installation consisted of running a supplied configure script on the head node to generate a Makefile, executing the Makefile, and then copying the binaries over to the compute nodes. A similar approach was taken for MPICH-1.2.4, which was downloaded from: http://www.unix.mcs.anl.gov/mpi/mpich/download.html.

Configure options for LAM:

./configure --prefix=/usr/local/lam-6.5.6 --with-rpi=sysv --with-rsh=”ssh -x”

Configure Options For MPICH:

./configure --prefix=/usr/local/mpich-1.2.4 --with-comm=shared -rsh=ssh

LAM is available in three different versions of the Request Progression Interface. (Note that these are not the same as the RPI *modes* discussed earlier in section 2.5.2; each of the three RPI versions can be run in both modes). We installed the *sysv* RPI version of LAM, which is one of the two RPI’s suitable for SMP clusters. The other is *usysv.* Both use TCP/IP for message transfer between processes on different nodes, and shared memory for message transfer between processes on the same node. The difference is that the sysv RPI uses SYSV semaphores for synchronization, and usysv uses spin locks. The LAM documentation states that the sysv RPI likely performs better than usysv for SMP clusters. We were able to compile and install LAM using the usysv RPI, but the programs we compiled ran erroneously, so we decided to use the sysv RPI. A third RPI, *tcp*, uses TCP/IP exclusively for message transfer, and was not installed.

MPICH is available in many implementations because it has been ported to many different systems. We chose the *ch_p4* implementation, which is recommended for

SMP clusters. Another cluster implementation, *ch_p4mpd*, provides faster process startup but is only available for uniprocessor clusters. As with LAM's sysv implementation, the ch_p4 implementation of MPICH uses TCP/IP for inter-node message transfer, and shared memory (using semaphores for synchronization) for intra-node message transfer.

All LAM programs were run in *c2c* mode with the *-O* option to mpirun. In c2c mode, MPI messages bypass the LAM daemons. The -O option specifies to LAM that all nodes in the system are homogeneous, eliminating the need for data conversion. These choices were made based on results from preliminary testing of LAM programs on the cluster using Linpack. For MPICH programs, mpirun by default assumes that the system consists of homogeneous nodes. Therefore, no options were passed to mpirun aside from the -p4pg option, which was needed due to the absence of a common file system on Wyeast.

MPE (Multi-Processing Environment) [4] is a useful library for profiling parallel MPI programs. It was developed by Argonne for MPICH, and is included with the MPICH download, but is intended to be used with any MPI implementation. MPE has capabilities for producing log and trace files in various formats, and includes several viewers for log file analysis, including Jumpshot. Jumpshot is a graphical visualizer for the SLOG files created by using the MPE logging library. MPE is

installed automatically when selecting the default configure options to MPICH. In order to use MPE with LAM, the directions provided in MPE User's Guide [4] were followed for configuration and linkage.

4 Experiments and Performance Results

This chapter presents the experiments that were run and the results obtained. Three major categories of experiments were run: High Performance Linpack (HPL), a standard benchmark for parallel computers used by the top 500 list; su3_rmd, a program involving four dimensional SU(3) lattice gauge theory; and a series of bandwidth comparisons testing eight MPI point-to-point communication primitives.

To avoid confusion, the following definitions of commonly used abbreviations which will be used in the proceeding discussion and in the next chapter are provided: 1 Mbps refers to 1 Megabit per second; 1 MB/sec means 1 Megabyte per second; 1 MFLOPS is 1 million floating point operations per second, and 1 GFLOPS is 1 billion floating point operations per second. These definitions are given because researchers in networking often refer to bits, whereas the operating system literature mostly refers to bytes, and also because of unclear results found in the course of this research that stressed the importance of unit and abbreviation clarification. Note that Fast Ethernet has a theoretical maximum bandwidth of 100 Mbps, and Gigabit Ethernet has a theoretical maximum bandwidth of 1000 Mbps.

4.1 Linpack

The first set of experiments compare LAM and MPICH using the Linpack Benchmark [5]. Linpack is a program that solves a dense linear system of equations. It has become the benchmark used by the top 500 list [31], which ranks the world's 500 fastest computers. The implementation used in this research is HPL (High Performance Linpack) [35], because it is appropriate for distributed systems, is freely-available, and is the implementation used and recommended by the top 500 list. Parallelization is achieved by distributing the data among a two-dimensional grid (P by Q) of processors, the parameters of which are specified by the user. According to the HPL documentation, this ensures good load balancing and scalability characteristics. Along with the grid size, the values for a number of other parameters are left up to the user. These are placed in an input file called "HPL.dat" and can be changed from run to run. The only parameters that were varied for the Linpack tests of this research were grid size and problem size. All other parameters remained fixed, the values of which were decided upon by preliminary testing. A sample HPL.dat input file is provided in Appendix A.

Linpack was chosen to be used in this research for several reasons. We wanted to test the Wyeast cluster as soon as possible after the nodes were configured, and it therefore seemed suitable to compile and run a benchmark that is widely used, freely available, and has been accepted as a standard. Linpack provides results in terms of

both *performance* (timing results and GFLOPS performed) and *correctness* (the tests either pass or they fail). Many clusters have been tested using Linpack, and so it also gave us a reasonable gauge by which to compare the performance of Wyeast to other clusters, as well as ensuring that all elements of the parallel environment of Wyeast (MPI, operating system, networking, hardware) were functioning correctly. Finally, once the program was compiled and run with one of the MPI libraries, it could also be compiled and run with the other, leading to an environment in which the two could be compared running a large benchmark.

4.1.1 Description of Experiments

Linpack was compiled and executed for both LAM and MPICH on increasingly large process grid sizes (P by Q), corresponding to increasing numbers of processors. As an example of grid size, a 3x4 grid runs on 12 processors, and a 4x4 grid runs on 16 processors. For each grid size, tests were run with increasingly large problem sizes (corresponding to "N" in the HPL documentation). As an example of a problem size, N=15000 means that a matrix of 15000 by 15000 double precision (8 byte) elements will be used by the program.

The grid sizes tested (with number of processors in parenthesis) were: 1x1 (1), 1x2 (2), 2x2 (4), 2x4 (8), 2x5 (10), 3x4 (12), 4x4 (16), 4x5 (20), and 4x6 (24). The problem sizes tested were: 2000, 4000, 6000, 8000, 10000, 15000, 20000, and 25000.

The memory requirements of the program dictated that not all problem sizes could be run with all grid sizes. For example, a problem size of N = 15000 could not be run on a 2x2 process grid because the two nodes housing the 4 processors (1 GB total) did not have enough memory available to the program (1.8 GB required). Running the program with such a combination would lead to memory swapping and would therefore distort the overall execution time, and is not recommended in the HPL documentation. All allowable grid/problem size combinations were run five times. The final results given in the next section are mean results of the five runs for each combination.

4.1.2 Results and Analysis

LAM and MPICH performed similarly on the Linpack benchmark, with LAM performing slightly better in almost every case. Excluding the uniprocessor runs, LAM outperformed MPICH by 2.6% on average in terms of GFLOPS. The best result obtained was 9.18 GFLOPS. This occurred on run # 2 using LAM on a 4x6 process grid (24 processors) with a problem size of N = 25000. Tables 4 and 5 (pages 41

and 42) display the results of all combinations tested in terms of both GFLOPS and execution time. Each number reflects the mean result of five runs. We did not experience a high level of variation among the five repeated runs for each problem size / grid size combination. Appendix D contains the standard deviation numbers for all combinations tested.

Figure 4 (page 43) displays the results as a function of problem size for a fixed grid size of 4x6 (24 processors), the highest number of processors used. In figure 5 (page 43), the results for a fixed problem size of N=15k are displayed as a function of grid size. This is a good representative figure because it is a respectable problem size that was able to fit into almost all grid sizes. Further, the trends of the other problem size runs are similar, with the exception of the N=2k and N=4k problem sizes. Appendices B and C contains graphs of all results. In Appendix B, the results are shown as a function of grid size with respect to the various problem sizes. The graphs in Appendix C display the results as a function of problem size by keeping the various grid sizes fixed.

Table 4: Linpack Data

Grid Size	Problem Size	LAM GFLOPS	MPICH GFLOPS	LAM Seconds	MPICH Seconds
1x1	2,000	0.611	0.602	8.75	8.88
	4,000	0.637	0.637	66.99	67.10
	6,000	0.651	0.650	221.39	221.54
1x2	2,000	0.986	0.991	5.38	5.37
	4,000	1.089	1.096	39.28	38.94
	6,000	1.132	1.112	127.34	129.59
2x2	2,000	1.077	1.056	4.96	5.09
	4,000	1.464	1.454	29.25	29.45
	6,000	1.682	1.670	85.65	86.35
	8,000	1.781	1.777	191.81	192.17
	10,000	1.858	1.851	358.81	360.51
2x4	2,000	1.086	1.057	4.90	5.06
	4,000	1.987	1.914	21.53	22.24
	6,000	2.492	2.413	57.68	59.58
	8,000	2.838	2.789	119.42	122.25
	10,000	3.044	3.016	218.50	221.13
	15,000	3.410	3.017	660.43	725.37
2x5	2,000	0.991	0.955	5.39	5.55
	4,000	1.918	1.838	22.30	23.25
	6,000	2.690	2.581	52.98	55.83
	8,000	3.161	3.049	108.83	111.65
	10,000	3.524	3.467	189.35	192.28
	15,000	4.032	3.992	558.65	563.68

Table 4 displays the Linpack data for all problem sizes tested with 1, 2, 4, 8, and 10 processors. Each number reflects the mean result of 5 runs.

Table 5: Linpack Data, continued

Grid Size	Problem Size	LAM GFLOPS	MPICH GFLOPS	LAM Seconds	MPICH Seconds
3x4	2,000	1.059	1.036	5.10	5.16
	4,000	2.112	2.092	20.31	20.35
	6,000	2.855	2.809	51.02	51.22
	8,000	3.423	3.446	99.75	99.37
	10,000	3.848	3.786	173.96	177.15
	15,000	4.435	4.370	507.70	511.55
4x4	2,000	1.219	1.142	4.33	4.62
	4,000	2.593	2.414	16.78	17.66
	6,000	3.435	3.283	42.21	43.29
	8,000	4.186	4.147	81.66	82.79
	10,000	4.805	4.729	138.75	141.10
	15,000	5.715	5.560	395.52	403.66
	20,000	6.295	6.221	849.37	858.18
4x5	2,000	1.144	1.093	4.76	4.86
	4,000	2.636	2.555	16.30	16.73
	6,000	3.725	3.605	37.78	39.91
	8,000	4.774	4.654	71.05	73.84
	10,000	5.432	5.313	123.45	125.05
	15,000	6.496	6.371	345.63	355.04
	20,000	7.418	7.338	719.61	728.22
4x6	2,000	1.113	1.125	4.72	4.68
	4,000	2.590	2.514	16.46	16.93
	6,000	3.789	3.653	38.66	39.45
	8,000	4.998	4.843	68.14	70.98
	10,000	5.833	5.803	114.21	115.03
	15,000	7.236	7.076	311.77	314.68
	20,000	8.448	8.276	624.96	642.97
	25,000	9.077	8.980	1154.81	1161.37

Table 5 displays the Linpack data for all problem sizes tested with 12, 16, 20, and 24 processors. Each number reflects the mean result of 5 runs.

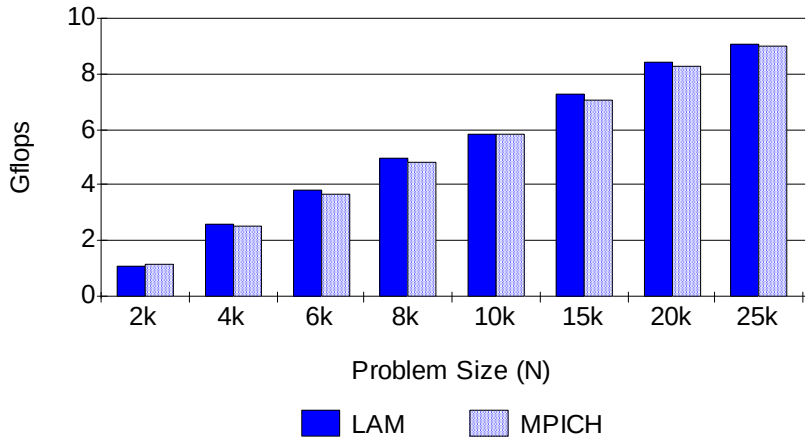


Figure 4: Linpack Results for Grid Size = 4x6

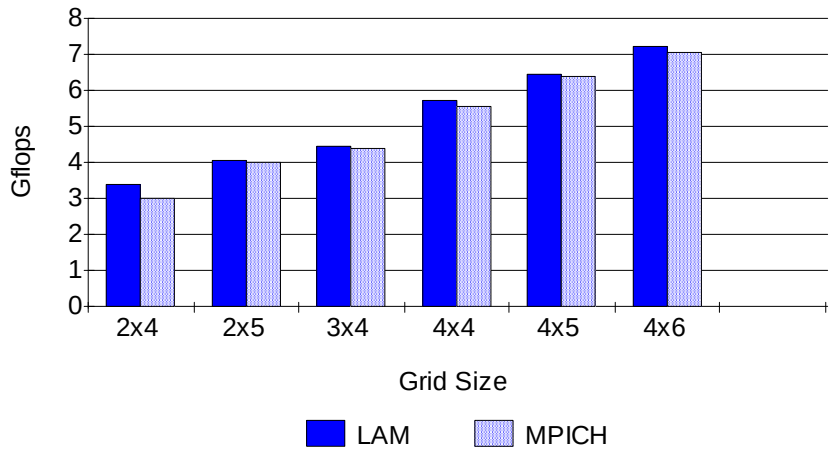


Figure 5: Linpack Results for N = 15k

Figures 4 and 5 display representative Linpack data in two different ways. In figure 4, results are shown as a function of problem size by keeping the grid size fixed. The 4x6 grid (24 processors) was the largest grid size tested, and the results displayed in figure 4 are representative of the results for the other grid sizes. In figure 5, the results are shown as a function of grid size for a fixed problem size of N=15k. This problem size was too large to fit into the 2x2 grid. All problem sizes except N=2k and N=4k resulted in patterns similar to the ones shown in figures 4 and 5.

In nearly every case, LAM and MPICH scaled for both increasing problem sizes and increasing grid sizes (processor counts). Two exceptions are for the N=2k and N= 4k problem sizes (see figures 6 and 7, next page). The conclusion here is that these problem sizes were likely too small to ensure that communication time would not become a bottleneck in overall execution time. The fact that a parallel program can be written for a problem does not always mean that it is advantageous to use the highest number of processors available. For some programs with small problem sizes, the communication time incurred by adding more processors outweighs the concurrency benefits. The Linpack 2k and 4k problem sizes appear to fall into this category.

For the other problem sizes, LAM and MPI scaled, but neither achieved linear speedup. Speedup is defined by equation: (Time on 1 processor) / (Time on N processors). Linear speedup is N for N processors. The largest problem size that fit into 1 node was N=6k; therefore, we were able to calculate the parallel speedup of LAM and MPICH with respect to serial execution for this problem size. The results are shown in figure 8 (page 46). While we were unable to get the uniprocessor time for the larger problem sizes due to memory requirements, the timing results for these problem sizes suggest that Linpack scales better for larger problem than for smaller problem sizes using both LAM and MPICH.

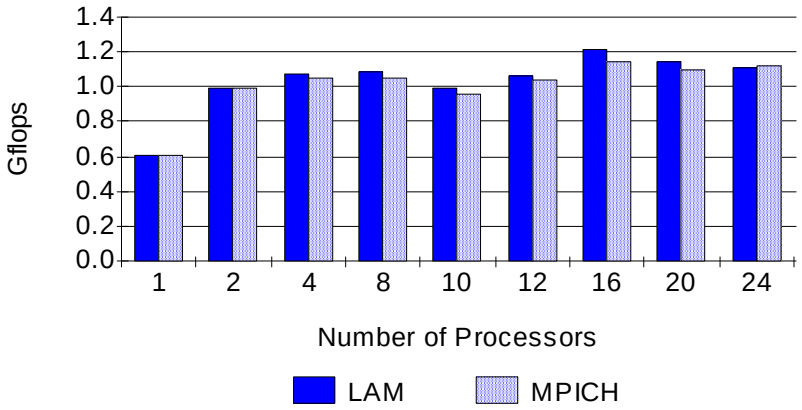


Figure 6: Linpack Results for N = 2k

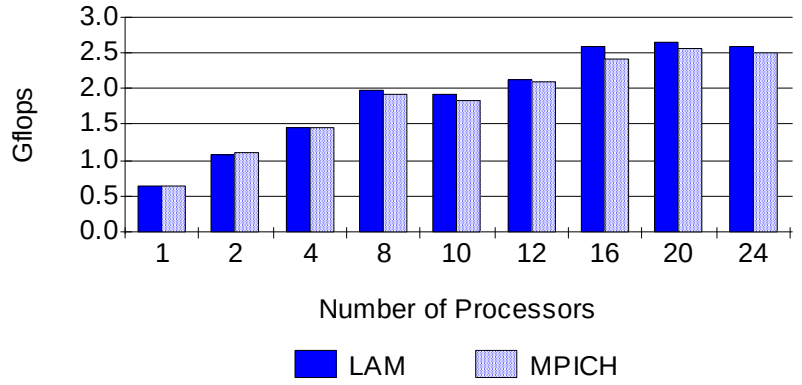


Figure 7: Linpack Results for N = 4k

Figures 6 and 7 display the performance of LAM and MPICH as a function of processor count for fixed problem sizes of N=2k and N=4k, respectively. Neither LAM nor MPICH scale consistently for these small problem sizes. The reason is likely that the communication cost incurred by adding processors begins to outweigh the benefits of parallel processing. These were the only problem sizes in which this characteristic was noticed.

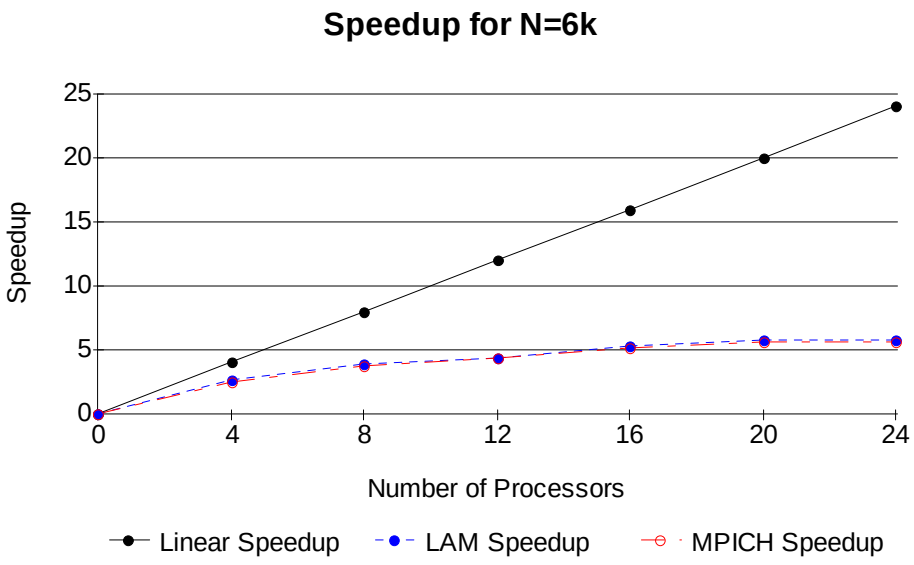


Figure 8: Speedup for N = 6k

Figure 8 displays the speedup of LAM and MPICH for a problem size of N=6k, the largest problem size that would fit in to the memory of 1 node. Speedup is defined by dividing the execution time using 1 processor by the execution time using N processors. Linear speedup is achieved when the time is N for N processors. Neither LAM nor MPICH achieve linear speedup for any of the problem sizes; however the speedup for both appears to be better for the larger problem sizes.

The Linpack results build confidence that the parallel environment is set up to properly compile and execute large MPI programs (all tests passed), they give us a way to compare the performance of the cluster with other clusters, and provide a legitimate starting point in comparing LAM and MPICH. If we were going to spend time

fine-tuning the Linpack input parameters and optimizing the parallel environment (MPI configure options, compiler choice, network options, etc.) with the goal of achieving the highest possible Linpack score, the results of this research indicate that the library to focus on would be LAM. This indication, however, is not over-whelming. The highest LAM score was 9.184 GFLOPS (4x6 grid; N = 25000). The highest MPICH score was 9.014 GFLOPS (4x6 grid; N = 25000).

4.2 su3_rmd

In an effort to strengthen the comparison of LAM and MPICH, we decided to test the performance of the libraries on another parallel program. Linpack is certainly a legitimate parallel program, but it is primarily used as a benchmark, and provides only one basis for comparison. The su3_rmd program is a simulation program used in physics research. It is the pure gauge variant of a variety of programs available from the MIMD Lattice Computation (MILC) Collaboration [36] involving a concept known as four-dimensional SU(3) lattice gauge theory. Parallelization is achieved by distributing lattice sites across nodes, and exchanging information among sites.

4.2.1 Description of Experiments

The overall problem size can be increased in su3_rmd by increasing the lattice dimensions (nx, ny, nz, nt) and increasing the number of trajectories (traj). The number of processors used must be a power of 2. We ran experiments with 5 problem sizes, shown in table 6 (next page). Since the number of processors used in su3_rmd must be a power of 2, we were able to use a maximum of 16 processors. For experiments 1, 2, and 3 we used 2, 4, 8, and 16 processors. Experiment 4 was not run with 2 processors because its problem size was too large to fit into the memory of 1 node. Similarly, experiment 5 was run only with 16 processors because its

problem size would not fit into the memory of 4 nodes, which is the number needed for an 8 processor run on our SMP cluster.

Table 6: Description of su3_rmd Experiments

Experiment	Problem Size	Processors Used
1	nx=ny=nz=4, nt=8, traj=50	2, 4, 8, 16
2	nx=ny=nz=6 nt=12 traj=75	2, 4, 8, 16
3	nx=ny=nz=8 nt=16 traj=100	2, 4, 8, 16
4	nx=ny=nz=12 nt=24 traj=150	4, 8, 16
5	nx=ny=nz=16 nt=32 traj=200	16

Table 6 displays the five problem sizes tested for su3_rmd. The number or processors used in su3_rmd must be a power of 2. Experiment 4 was not run with 2 processors because its problem size exceeded the memory of 1 node. Likewise, experiment 5 was run only with 16 processors because it would not fit into the memory of 4 nodes (8 processor run).

4.2.2 Results and Analysis

We measured wall clock time (total execution time) at the shell, and the program provides timing results for CPU time. LAM and MPICH performed similarly in terms of total execution time on all problem sizes tested except for the smallest and largest sizes. Table 7 (next page) presents the data for each combination tested in terms of wall clock time and CPU time versus the number of processors used. Figures 9 through 13 (pages 55 through 57) display these results graphically. In both the table and the graphs, the numbers shown are mean times of five runs for each combination. Standard deviation results of all repeated runs are provided in Appendix E.

One interesting observation from the timing results is that, in many cases, the CPU time for LAM is less than the CPU time for MPICH, but the total execution time is longer. For example, in experiment 4, with 8 processors, the total execution time for LAM is 759 seconds versus 752 seconds for MPICH, but the CPU time is 559 seconds versus 673 seconds. In the same experiment, using 16 processors, the execution time for LAM finally becomes less than that of MPICH, but only by 10 seconds. The difference in CPU time is 100 seconds. Table 8 (page 52) displays the results in terms of CPU time percentage versus communication/IO time percentage for all experiments.

Table 7: Results of su3_rmd in Seconds

Problem Size	Processors	LAM Wall Time	LAM CPU Time	MPICH Wall Time	MPICH CPU Time
Experiment 1 nx=ny=nz=4, nt=8, traj=50	2	7.46	6.48	6.94	6.68
	4	6.19	4.07	6.79	5.56
	8	4.32	2.13	6.41	5.04
	16	3.88	1.55	8.55	5.76
Experiment 2 nx=ny=nz=6, nt=12, traj=75	2	67.50	64.42	65.05	64.75
	4	37.68	31.73	38.08	36.52
	6	23.11	14.72	24.78	21.88
	16	15.61	7.85	22.08	18.31
Experiment 3 nx=ny=nz=8, nt=16, traj=100	2	294.47	293.23	288.28	287.82
	4	174.31	152.45	173.85	166.40
	8	97.95	69.88	94.77	88.19
	16	58.94	35.77	60.42	53.95
Experiment 4 nx=ny=nz=12 nt=24 traj=150	4	1284.57	1133.50	1284.34	1231.44
	8	759.38	559.25	752.49	672.81
	16	428.93	290.74	438.01	390.62
Experiment 5 nx=ny=nz=16 nt=32 traj=200	16	1683.32	1203.04	1830.65	1576.82

Table 7 displays the results of the su3_rmd tests. Results are displayed as execution time and CPU time in seconds. The only situation where the program did not scale occurred in the smallest problem size for MPICH. One reason for this is that process startup is slower in MPICH than in LAM, and with 16 processes this characteristic plays an important role for small problem sizes, in which startup time can be a large portion of overall execution time.

Table 8: su3_rmd CPU Time and Communication/IO Time

Experiment	LAM CPU	LAM comm/IO	MPICH CPU	MPICH comm/IO
1: 2 procs	86.9%	13.1%	96.3%	3.7%
4 procs	65.7%	34.3%	81.9%	18.1%
8 procs	49.3%	50.7%	78.6%	21.4%
16 procs	39.9%	60.1%	67.4%	32.6%
2: 2 procs	95.4%	4.6%	99.6%	0.4%
4 procs	84.2%	15.8%	95.9%	4.1%
8 procs	63.7%	36.3%	88.3%	11.7%
16 procs	50.3%	49.7%	82.9%	17.1%
3: 2 procs	99.6%	0.4%	99.8%	0.2%
4 procs	87.5%	12.5%	95.7%	4.3%
8 procs	71.3%	28.7%	93.1%	6.9%
16 procs	60.7%	39.3%	89.3%	10.7%
4: 4 procs	88.2%	11.8%	95.9%	4.1%
8 procs	73.6%	26.4%	89.4%	10.6%
16 procs	67.8%	32.2%	89.2%	10.8%
5: 16 procs	71.5%	28.5%	86.1%	13.9%

Table 8 displays the breakdown of the su3_rmd experiments in terms of percentage CPU time and percentage communication/IO time. The su3_rmd program provides timing results for CPU time. We timed overall execution time at the shell, and show the difference of execution time and CPU time here as percentage communication/IO time.

For the largest problem size tested (experiment 5), the total execution time for LAM is 1,683 seconds versus a total execution time of 1,831 seconds for MPICH. This indicates that LAM performs better than MPICH on larger problem sizes with a large number of processors. Unfortunately we were unable to test su3_rmd with a higher number of processors because the next allowable number for this program is 32, and we did not have 32 processors available for the MPI experiments. It would be interesting to see if the performance difference increases as both the problem size and the number of processors are increased.

For 16 processors on the smallest problem size, LAM is 2.2 times faster than MPICH. This is likely in part due to the fact that process startup time is faster in LAM than in MPICH. As discussed in section 2.5.2, LAM runs as a user level daemon on each node. These daemons are responsible for MPI process startup, and optionally for intercepting messages to collect trace data. Because the daemons are already running before the MPI program is executed, process startup time is fast. Conversely, process startup time in MPICH is slow because MPICH uses a remote shell invocation for each MPI process created (see Carns et. al. [3]). Startup time can become an important factor in the execution time of programs with small problem sizes as the number of processors is increased. This partially explains the

extreme performance difference of LAM and MPICH in experiment 1 using 16 processors.

We conducted a simple experiment to measure MPI process startup time in LAM and MPICH. In this experiment, all processes except process 0 sends a single integer (4 bytes) to process 0. Table 9 displays the results of the startup time experiment.

Table 9: Process Startup Time Results (seconds)

Processors	LAM	MPICH
2	0.51	0.23
4	0.53	0.63
8	0.54	1.42
12	0.55	2.24
16	0.57	3.04
20	0.58	3.93
24	0.60	4.79

Table 9 displays results from our process startup time experiments. Process startup time is high in MPICH due to the fact that MPICH uses remote shell invocation for MPI process creation. In contrast, LAM uses daemons on each node to start MPI processes. These daemons are already running when the LAM MPI program is executed.

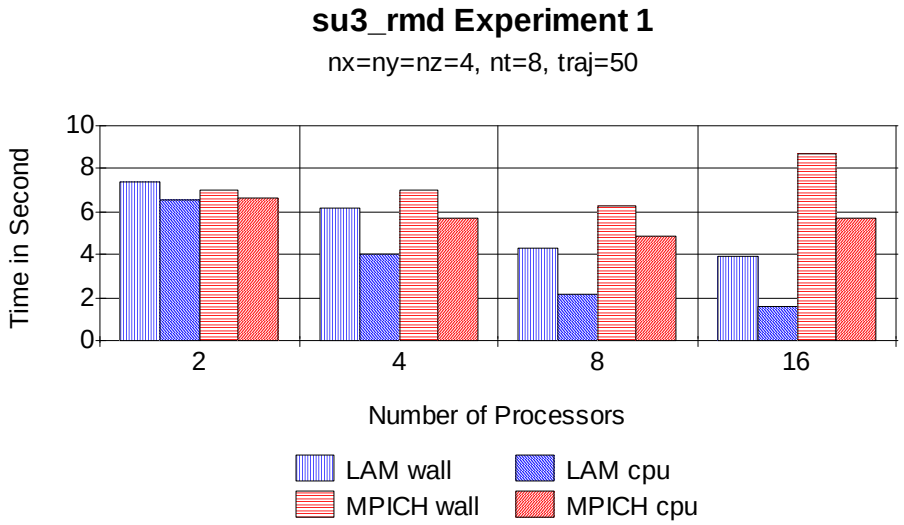


Figure 9: su3_rmd Results for Experiment 1

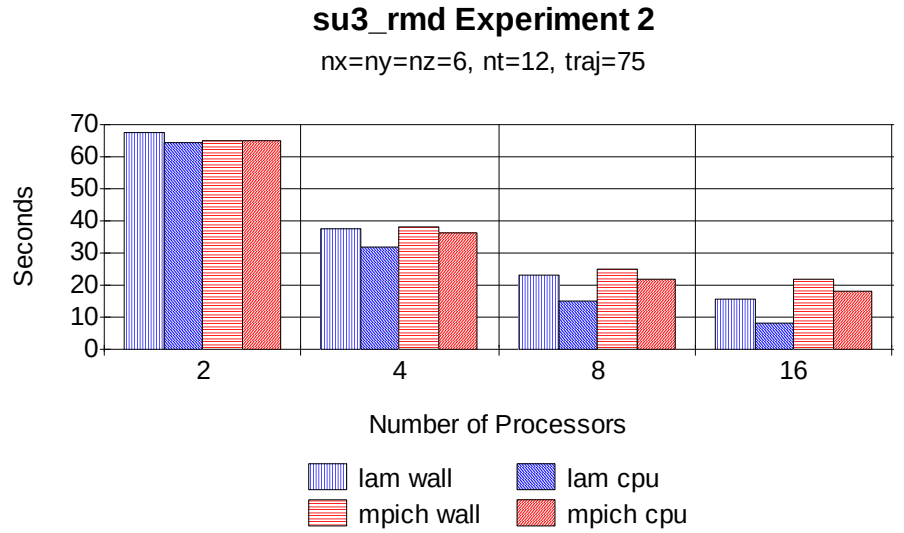


Figure 10: su3_rmd Results for Experiment 2

Figures 9 and 10 display the timing results for su3_rmd on the first two problem sizes. For the smallest size tested (shown in figure 9), it is evident that MPICH does not scale as processors are added. The reason for this is likely the fact that MPI process startup time is slow in MPICH.

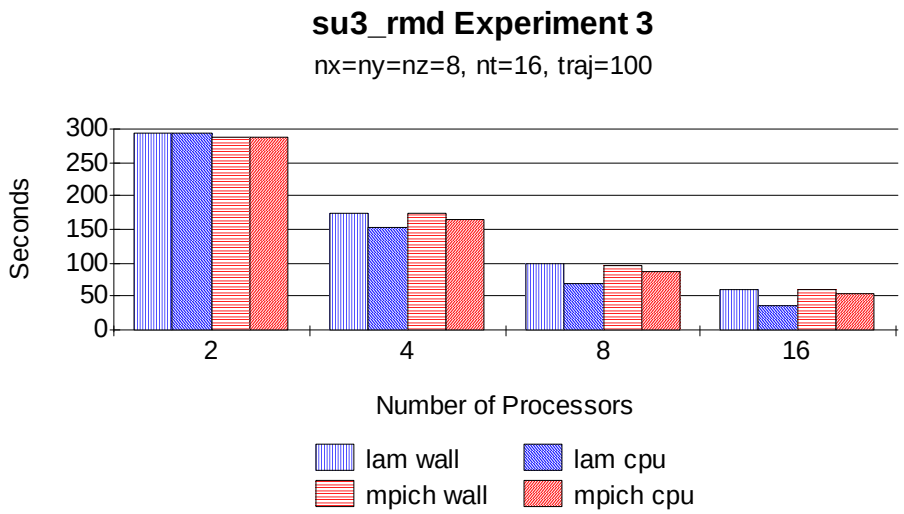


Figure 11: su3_rmd Results for Experiment 3

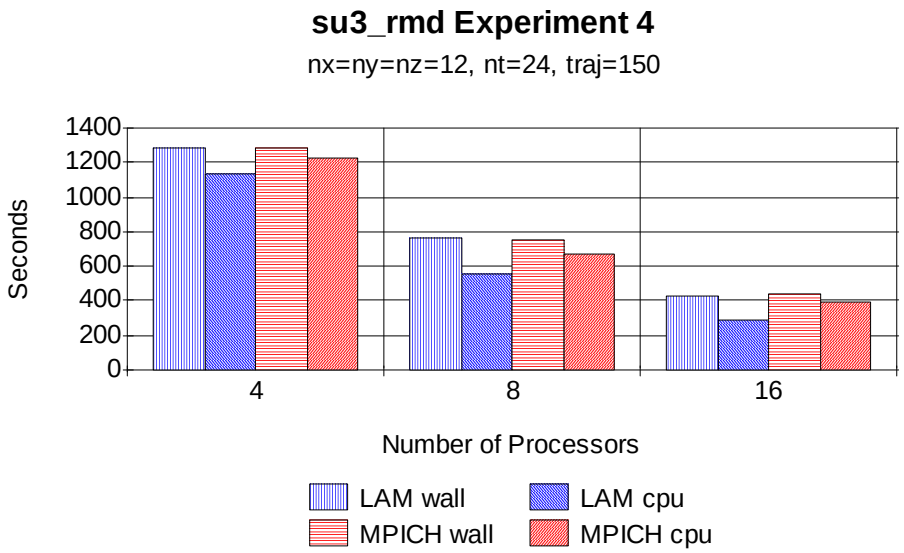


Figure 12: su3_rmd Results for Experiment 4

Figures 11 and 12 display the timing results for su3_rmd on larger problem sizes. One interesting observation is that, while the LAM programs use less CPU time, the overall execution time is nearly even for LAM and MPICH for these problem sizes.

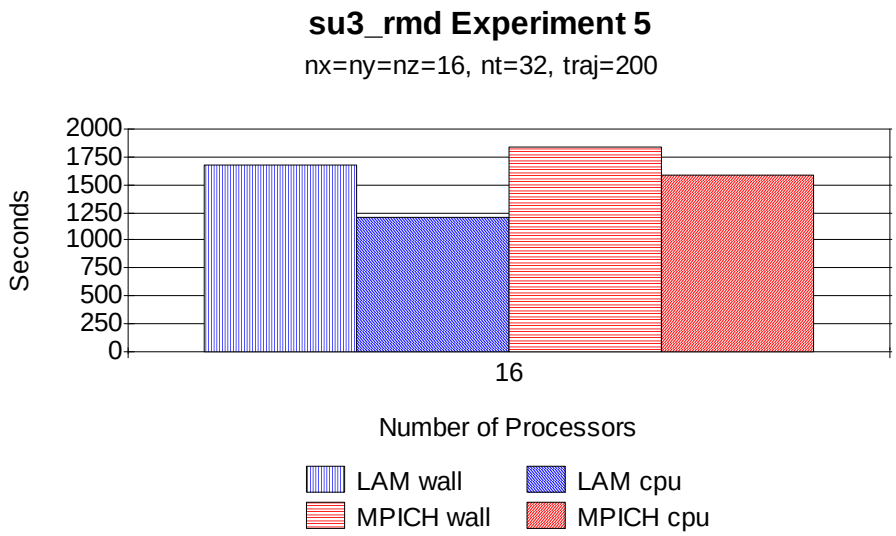


Figure 13: su3_rmd Results for Experiment 5

Figure 13 displays the timing results for su3_rmd on the largest problem size tested. This problem size was only tested with 16 processors because it would not fit onto the memory of the node configurations for fewer processors. Unlike experiments 1 through 4, LAM clearly outperforms MPICH in terms of both CPU time and overall execution time in this experiment.

We studied the performance of su3_rmd using the MPE library and Jumpshot, the visualization tool distributed with MPE (discussed in section 3.4). The tools fell short, however, in not providing information about the breakdown of CPU time, I/O time, and communication time. MPE and Jumpshot were helpful, however, for clearly logging and displaying every MPI function that su3_rmd uses, and also for providing an overall view of the communication pattern. For MPI point-to-point communication, the only construct used was MPI_Ssend/MPI_Irecv, a synchronous send with a non-blocking receive.

As can be seen from the MPE/Jumpshot histogram in figure 14 (next page), MPI_Ssend and MPI_Wait (which corresponds to MPI_Irecv) dominate communication time. Jumpshot uses a heuristic based on the number of function calls and the duration of each call to determine the relative importance of each MPI construct in the histogram display. We decided to take a closer look at MPI_Ssend/MPI_Irecv, along with seven other MPI point-to-point combinations. The results are presented in the next section. LAM performed better than MPICH in terms of bandwidth for Ssend/Irecv, providing a partial explanation as to why LAM performed better than MPICH on su3_rmd for the large problem size.

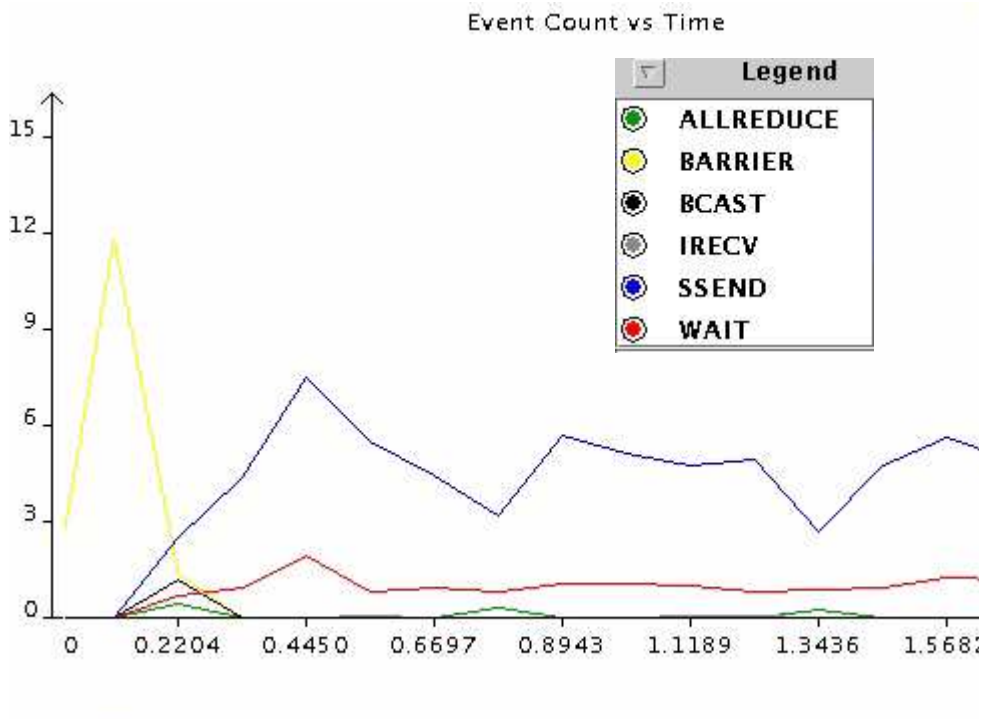


Figure 14: Histogram of MPI Constructs in su3_rmd

Shown in figure 14 is a screen shot of the Jumpshot graphical visualizer which is used to view the SLOG trace files generated by the MPE logging library. The SLOG file used for this screen shot contains information for an execution of su3_rmd using the problem size of experiment 2 with 16 processors. The histogram displays the relative importance of MPI functions as time elapses using a heuristic that involves the number of function calls made and the time spent in each call. Note that MPI_Barrier is only called at the beginning of the program. The histogram clearly shows that MPI_Ssend and the MPI_Wait corresponding to MPI_Irecv clearly dominate communication time in su3_rmd.

4.3 Bandwidth Comparisons

This section discusses the performance of LAM and MPICH on eight MPI point-to-point communication pairs using a traditional “ping-pong” bandwidth test program, in which a message is passed from process A to process B, which passes it back to process A. The source code can be obtained by viewing the on-line version of a paper entitled “MPI Performance Topics” [28].

The bandwidth program tests round-trip performance of the MPI point-to-point communication pairs shown in table 10 (next page). MPI provides both blocking and non-blocking sends and receives. Additionally, for sends, four communication modes are provided: standard, synchronous, buffered, and ready. Therefore, there are eight possible send operations and two possible receive operations. This results in sixteen point-to-point communication pairs possible in MPI. The bandwidth program tests eight of them, including many of the ones commonly found in MPI programs.

Detailed information regarding the various point-to-point communication options available in MPI can be found in the texts by Gropp and Lusk [12], and Pacheco [15]. A brief description of the operations in table 10 is presented here. Blocking operations do not return until the function arguments are safe to reuse. The argu-

ments are safe to reuse if the system has buffered the message and/or the matching operation has been posted.

Table 10: Point-To-Point MPI Bandwidth Tests

Pair	Description
MPI_Send / MPI_Recv	Blocking Send Blocking Receive
MPI_Send / MPI_Irecv	Blocking Send Non-blocking Receive
MPI_Isend / MPI_Recv	Non-blocking Send Blocking Receive
MPI_Isend / MPI_Irecv	Non-blocking Send Non-blocking Receive
MPI_Ssend / MPI_Recv	Synchronous Send Blocking Receive
MPI_Ssend / MPI_Irecv	Synchronous Send Non-blocking Receive
MPI_Issend / MPI_Recv	Non-blocking Synchronous Send Blocking Receive
MPI_Issend / MPI_Irecv	Non-blocking Synchronous Send Non-blocking Receive

Table 10 displays the MPI point-to-point communication combinations tested by the bandwidth program. The MPI_Ssend/MPI_Irecv combination was used in the su3_rmd program. See the text for a brief discussion of blocking, non-blocking, and synchronous communication.

Therefore, a blocking send may or may not block while waiting for a corresponding receive to be posted. It depends on whether or not the system provides temporary

buffering of messages. Most systems provide buffering up to a certain message size. For example, LAM provides buffering for messages up to 65 KB in size.

Note that blocking communication is different from synchronous communication in MPI terminology. Synchronous communication is more restrictive: it does not rely on system buffering but rather demands that the matching operation has been posted, and data transmission has begun. For example, a process executing a synchronous send will block until the corresponding receiving process has actually posted the receive and begun receiving the data into its argument buffer. For this reason, synchronous sends can be used to ensure the *safety* of MPI programs. If the program does not deadlock when all blocking sends are replaced with synchronous sends, one can be assured that the send and receive pairs have been coded in a correct order in terms of avoiding deadlock.

Non-blocking calls return immediately, and require the programmer to call a separate *wait* or *test* operation to determine if the corresponding operation has completed. When it has, the function arguments are safe to reuse. Non-blocking communication can be used to avoid deadlock and to improve performance by overlapping communication with computation.

The bandwidth program can be configured to test any range of message sizes, using any size for a step interval. For example, one can specify to test message sizes in the range [100, 1000], incrementing the message size every 100 bytes. The number of round-trip iterations can also be configured, and the program keeps track of the best, worst, and average times, using MPI_Wtime for each iteration. MPI_Wtime returns the time in seconds from some arbitrary time in the past. This arbitrary time is guaranteed not to change during the lifetime of a process, so a timing of some activity can be taken by surrounding the activity with timing calls and subtracting the difference.

We decided to perform bandwidth experiments to provide a fine-grained view of the point-to-point performance of LAM and MPICH. Most MPI programs are written using at least one of the eight point-to-point combinations shown in table 10, and testing the performance of the libraries on some of the commonly used combinations strengthened the overall comparison of LAM and MPICH in this research. Specifically, the su3_rmd application discussed in section 4.2 uses the MPI_Ssend/MPI_Irecv combination, and we decided to take a closer look at this pair. The specific bandwidth program used in this research was used because it tests this pair, along with the others listed in table 10. Further, it can be easily configured to reset the different parameters, and the timing results are provided in a clear, understandable form.

4.3.1 Description of Experiments

Each pair listed in table 10 was tested for three message ranges: “small” [300 to 4,800 bytes], “medium” [5,000 to 100,000 bytes], and “large” [200,000 to 2,000,000 bytes]. The message sizes were increased in increments of 300 bytes, 5,000 bytes, and 100,000 bytes, respectively. For each communication pair, for each message size tested, one hundred iterations were performed. The reported results are mean times. The tests were conducted using two processors, one on each of two nodes. Therefore, the tests reflect node-to-node performance over the network, rather than shared memory performance.

We noticed we were getting results that exceeded the theoretical maximum (12.5 MB/sec) for round trip bandwidth on a 100 Mbps Fast Ethernet for many of the tests involving non-blocking receives. A close examination of the bandwidth program revealed that the tests which involve non-blocking receives were not properly measuring round trip bandwidth, due to the ordering of MPI_Irecv and MPI_Wait. Process 1 correctly sends the message to process 2. However, process 2 does not wait for the non-blocking receive to complete before sending the message back to process 1. It does call MPI_Wait, but only after it sends the message back. At this point, it is too late - there is no reason to wait. The message has been sent back without ensuring that it has been correctly received in the first place, and so what is really happening is that two messages are being sent across the network at the same

time. This is possible because our switches are configured to run in full-duplex mode. Therefore, we were exceeding the theoretical maximum for round trip bandwidth on our 100 Mbps Fast Ethernet.

We modified the program to correctly wait before sending the message back in the cases involving non-blocking receives. When we did this, the results were under 12.5 MB/sec. In addition, there was less data skew. In the remainder of this thesis, we will refer to the *original program* and the *modified program* when discussing bandwidth results for non-blocking receives. We provide all data for both categories of results.

4.3.2 Results and Analysis

LAM performed better than MPICH on the Wyeast Cluster for the communication pairs tested between two nodes. This was expected for the Ssend/Irecv combination, because the su3_rmd application uses this pair for point-to-point communication, and LAM outperformed MPICH on the su3_rmd test for the large problem size. However, the overwhelming performance gains of LAM over MPICH for nearly all of the bandwidth experiments was not expected. Appendix F contains graphs displaying the results for every bandwidth experiment conducted (original program). The graphs represent the bandwidth of LAM and MPICH on the eight

communication pairs tested for each of the three message size ranges. They reflect the mean bandwidth results of one-hundred iterations for each combination tested. Raw data showing best, average, and worst times for each communication pair/message size combination is provided in Appendix G. Similarly, Appendix H contains graphs of results for pairs involving non-blocking receives for the modified program, and Appendix I contains the raw data for these results.

Figures 15 through 17 on the following pages display the results of LAM and MPICH on MPI_Send/MPI_Recv, the MPI blocking send and receive pair. This is the most basic of the point-to-point communication constructs in MPI, and one that is found in many MPI programs. LAM slightly outperformed MPICH on this combination for every message size. In figures 18 through 20, the results are presented for MPI_Ssend/MPI_Irecv (modified program) - a synchronous send with a non-blocking receive. This is the construct used by the su3_rmd application, discussed in section 4.2. The results clearly show that LAM performs better than MPICH for this construct, which is one reason why LAM performed better on su3_rmd on the large problem size.

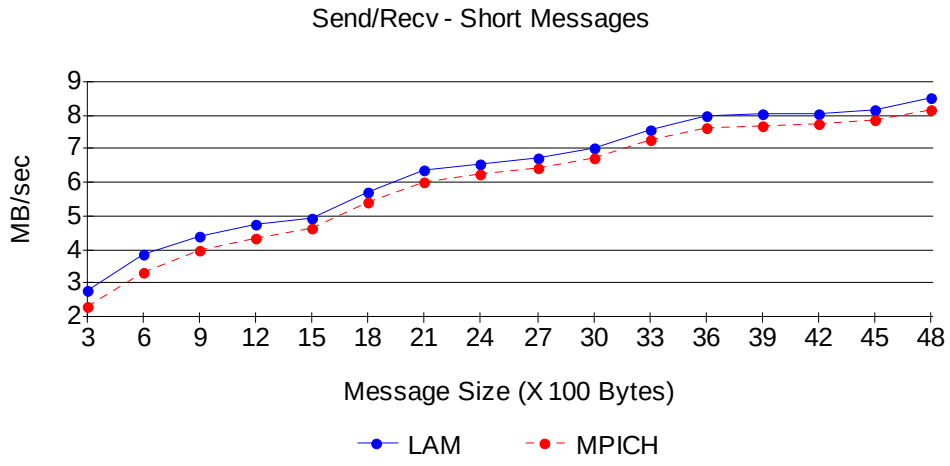


Figure 15: Bandwidth Results for Send/Recv - Short Messages

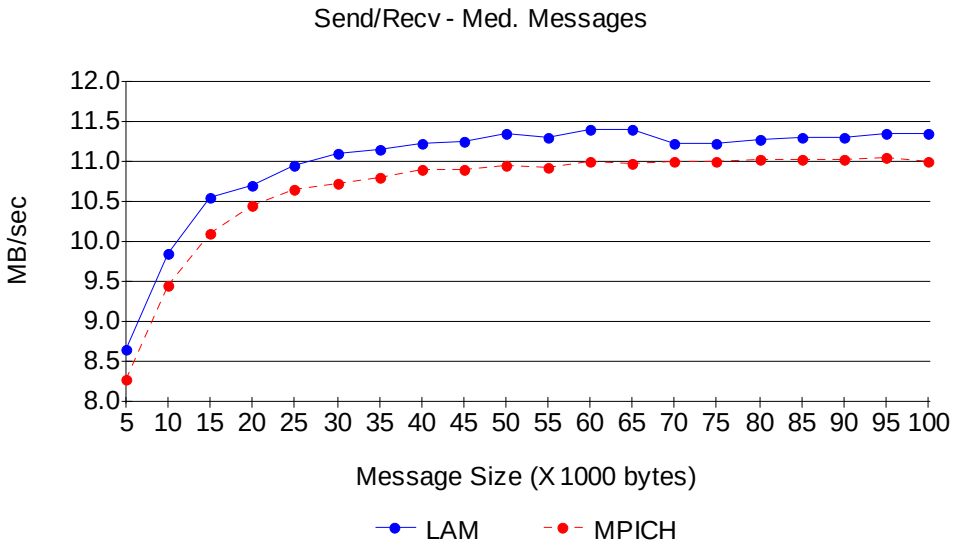


Figure 16: Bandwidth Results for Send/Recv - Medium Messages

Figures 15 and 16 display the bandwidth performance of LAM and MPICH on Send/Recv for short and medium range messages, respectively. Send/Recv is the basic blocking point-to-point blocking construct in MPI.

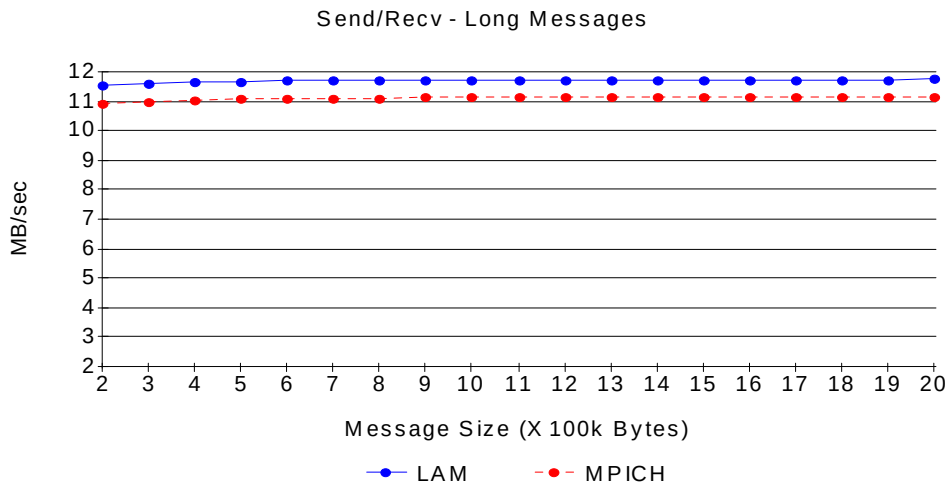


Figure 17: Bandwidth Results for Send/Recv - Long Messages

Figures 17 displays the bandwidth performance of LAM and MPICH on Send/Recv for long messages. Send/Recv is the basic blocking point-to-point blocking construct in MPI.

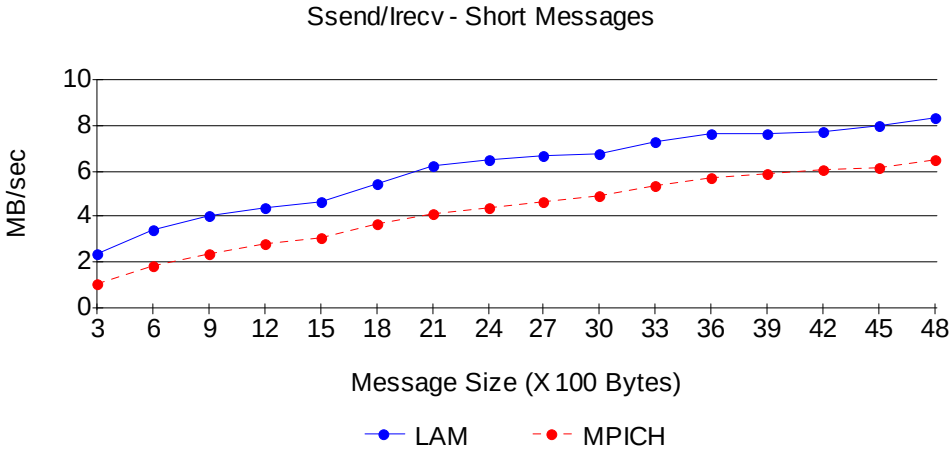


Figure 18: Bandwidth Results for Ssend/Irecv - Short Messages

Figure 18 displays the bandwidth performance of LAM and MPICH on Ssend/Irecv (modified program) for short range message sizes. Ssend/Irecv is a synchronous send with a non-blocking receive. This is the construct used in the su3_rmd application.

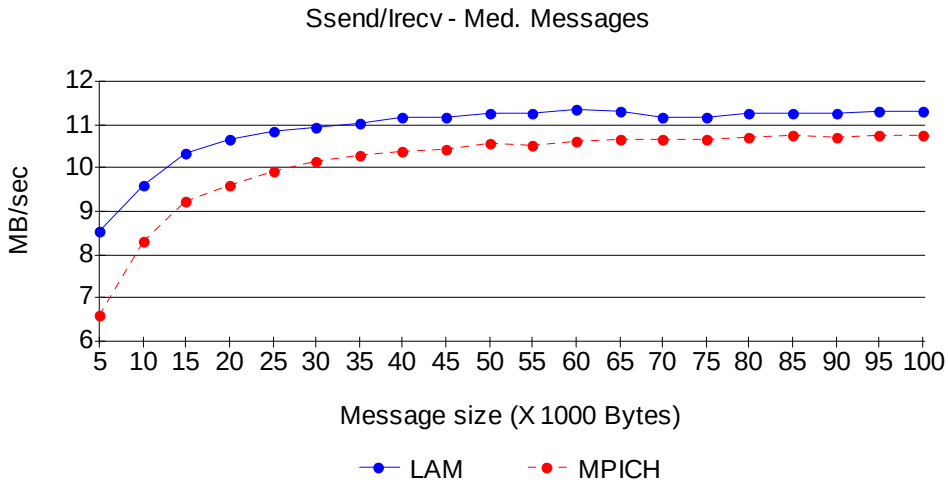


Figure 19: Bandwidth Results for Ssend/Irecv - Medium Messages

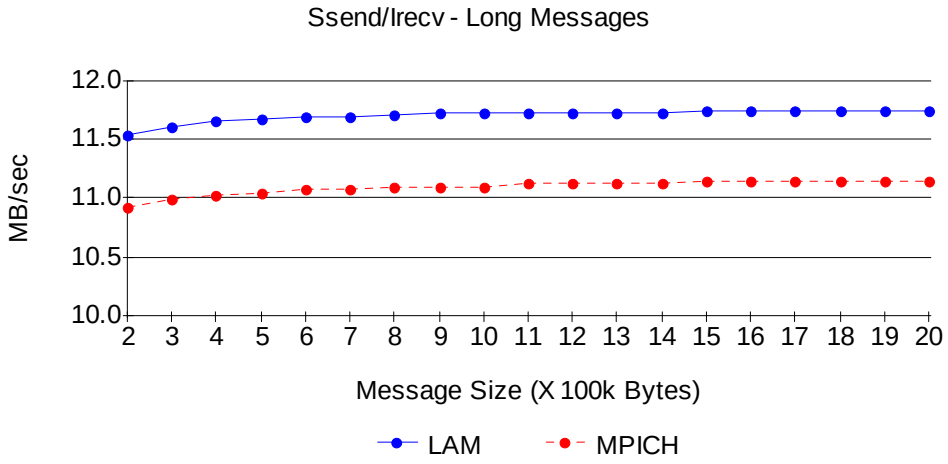


Figure 20: Bandwidth Results for Ssend/Irecv - Long Messages

Figures 19 and 20 display the bandwidth performance of LAM and MPICH on Ssend/Irecv (modified program) for medium and long range message sizes, respectively. Ssend/Irecv is a synchronous send with a non-blocking receive. This is the construct used in the su3_rmd application.

One interesting conclusion that can be made from the results is that MPICH narrows the gap in many cases when the message size exceeds 65 KB. This can clearly be seen in figures 16 and 19, and on most of the graphs for the medium size messages. The reason for this is that LAM switches from a *short* to *long* protocol at 65 KB by default. Although we used the default configuration, this setting can be changed by setting the environmental variable LAM_TCPSHORTMSGLEN. LAM's short protocol specifies that the message is sent in its entirety, using a system buffer if the matching receive has not yet been posted. With the long protocol, a "handshake" occurs. The sending process first sends the message envelope, which includes the source and destination addresses, message size, message tag, communicator, and possibly other information. A small amount of the actual data might also be sent. The sender then waits for the receiving process to send back an acknowledgement indicating that it has received the envelope. Upon acknowledgement from the receiver, the sending process then proceeds to send the actual data.

It may seem tempting to set the cutoff limit between short and long messages as high as possible in order to avoid the handshake, but there is a trade-off between the extra communication of the long protocol and the copying required in order to use the short protocol. The short protocol relies on system buffering. For messages up to the short limit, LAM buffers messages on the receiver's node in an address space

separate from the receive buffer argument if the receive has not yet been posted . Therefore, a blocking send can return without the matching receive being posted because the data has been buffered for the receiver, which must copy the data into its receive buffer. As message sizes increase, the time spent copying the buffer out-weighs the time spent during the handshake of the long protocol. The specific size is system dependent. Thus, an interesting future experiment for Wyeast would be to try different settings for the short limit and run bandwidth tests to determine the optimal setting.

MPICH implements three protocols for node-to-node communication: *short*, *eager*, and *rendezvous*, depending on the message size [11]. In each case, the message envelope is sent immediately. With short protocol, the data is sent along with the envelope. With eager protocol, the data is delivered whether or not the receiver acknowledges the receipt of the envelope, implying some sort of system buffering. In contrast, rendezvous protocol requires that the receiver requests the data by acknowledging receipt of the envelope. Eager protocol and rendezvous protocol seem analogous to LAM's short and long protocol, respectively. Although we were unable to determine the short/eager crossover size, we found the crossover size for eager/rendezvous to be 128 KB by performing fine-grained tests in the 100 KB to 200 KB range. For MPICH, the drop-off in bandwidth at the eager/rendezvous crossover size is not as severe as it is for LAM at the short/long crossover size.

4.4 Summary

We tested the performance of LAM and MPICH using three major categories of experiments: High Performance Linpack; su3_rmd, an MPI application used in physics research; and a series of round trip bandwidth comparisons. The performance of the libraries was most similar on Linpack, with LAM outperforming MPICH by 2.6% on average. LAM and MPICH also performed similarly on su3_rmd. However, for the largest problem size, the execution time for LAM was 1,683 seconds versus 1,831 seconds for MPICH. Using the MPE logging library and the Jumpshot visualization tool, we saw that the point-to-point communication construct used by su3_rmd was MPI_Ssend/MPI_Irecv, a synchronous send with a non-blocking receive. We took a close look at this combination, along with seven other MPI point-to-point combinations, using a series of bandwidth comparisons. LAM outperformed MPICH on all of the bandwidth comparisons, including MPI_Ssend/MPI_Irecv. These bandwidth results partially explain why LAM outperformed MPICH on the su3_rmd application on the large problem size.

The results show that LAM is at least as fast as MPICH, and generally faster, on the Wyeast cluster - especially for larger problem sizes. LAM was originally designed for the cluster and networked workstation environments, while MPICH was designed to be able to be quickly ported to many different types of parallel systems.

For this reason, MPICH relies on a heavily layered architecture, which might explain why MPICH generally ran slower than LAM on our tests.

5 Related Work

This chapter discusses previous studies related to the comparison of LAM and MPICH. Three of the studies were conducted on clusters, and one was conducted using two PCs. Each study is different in terms of platform, LAM and MPICH versions, and experiments. With a few exceptions, the common result is that LAM outperforms MPICH. It should be noted that none of the studies in this chapter compare the most recent version (at the time of this writing) of either library. However, several primary features of each have not changed dramatically during the courses of their respective evolutions. For example, LAM has always operated in a "virtual machine" environment with LAM daemon processes running on each node, whereas MPICH never has. Therefore, it is useful to examine the comparative performance of past versions of both libraries when performing a current study such as this to see if any of the performance trends have continued.

The studies are presented in order of latest to oldest version numbers of the libraries. Section 5.1 discusses a study that compares the bandwidth performance (Mbps) of LAM-6.5.1 and MPICH-1.2.1 on a 16 node Beowulf cluster with Fast Ethernet using the NAS-2.3 benchmark suite. Section 5.2 summarizes a comparison of LAM-6.3, MPICH-1.1.2, and MVICH (an MPI implementation based on MPICH using VIA technology for network protocol processing) on 2 PCs connected by Gigabit Ethernet using the NetPIPE-2.3 benchmark. In section 5.3, a study of the

performance (MFLOPS) of LAM6.3-b1 and MPICH-1.0.9 is discussed. The testing environment is a 16 node Beowulf Cluster with Fast Ethernet, and the program tested is a parallel N-body application conducted by researchers in the field of astronomy. Results are given in terms of MOPS (Millions of Operations Per Second). Section 5.4 discusses a bandwidth comparison of LAM-6.0 and MPICH-1.0.12 on an 8 node DEC workstation cluster connected by an FDDI network. This study was conducted using six "in house" benchmarks, and the results are given in terms of seconds versus message size (exact Mbps calculations not provided).

5.1 LAM-6.5.1 and MPICH-1.2.1

A 2001 report on the 16 node Beowulf cluster Grendel [1] at Uppsala University in Linkoping, Sweden provides a comparison of LAM-6.5.1 and MPICH-1.2.1 using the NAS Parallel Benchmark (NPB) 2.3 suite with a class B problem size. NAS-2.3 contains eight benchmarks based on computational fluid dynamics. Six of the benchmarks were used by the researchers. Grendel's nodes each contain 1 CPU (1 Ghz) and are connected by Fast Ethernet. The LAM tests were run in lamd mode, which employs user-level daemons as message intermediaries. In lamd mode, a message travels from an MPI process on node A to a LAM daemon process on node A; the daemon on node A then sends the message to a LAM daemon process on node B, which finally sends the message to an MPI process on node B. Also notable

about lamd mode is that UDP (as opposed to TCP in c2c mode) is used for data transfer. MPICH always uses TCP for data transfer between nodes and sends messages from MPI process to MPI process directly. Recall that our LAM results show that c2c mode performed better than lamd mode. This could be due to improvements of the c2c mode with the release of LAM-6.5.6.

The Grendel researchers found that MPI programs compiled with the LAM libraries provided 4.26% more MOPS than those compiled with MPICH. The report concluded that the LAM programs performed better because they exhibited faster communication over the network, primarily due to the use of UDP packets for data transfer. The study did not test the c2c mode, and cited research presented in the paper of Carns et al. [3] as one reason to use lamd mode. The NAS test was the extent of the comparison of LAM and MPICH in this report; due to the NAS results all further experiments used LAM (lamd mode) to test various aspects of the cluster.

5.2 LAM-6.3-b1 and MPICH-1.0.9

Astrolab is a cluster at the University of Washington which was used in a 1999 study [27] by the UW Department of Astronomy comparing LAM-6.3-b1 and MPICH-1.0.9. The cluster is composed of 16 single-CPU nodes (300 Mhz each) connected by Fast Ethernet. The program tested was PKDGRAV, a parallel N-body

program. The program was run on 2, 4, 8, 12, and 16 nodes using executables compiled with both libraries.

The performance of LAM and MPICH was very similar on the runs using up to 4 processors. As the number of processors increased, the performance of LAM became better than that of MPICH. With 12 processors, the MPICH program produced approximately 775 MFLOPS, while LAM achieved approximately 925 MFLOPS. After this point, MPICH experienced a steady decline, whereas the performance of LAM scaled at a near linear rate. Using 16 processors (the maximum tested), LAM achieved nearly 1200 MFLOPS, and MPICH achieved approximately 550 MFLOPS.

The conclusion of the Astrolab researchers was that LAM performed better than MPICH because in their opinion it is more suited for the Beowulf cluster architecture by design. They stated that LAM does not rely as heavily on the layered approach (Application Interface, Abstract Device Interface, Channel Interface) as MPICH, producing what they believe is a trade-off of speed on Beowulfs versus versatility in porting to other systems. As stated in the introduction, MPICH has been ported to nearly every parallel system imaginable (primarily due to its layered construction). For example, MPICH has been ported to the systems utilizing the VIA communication technology, whereas such ports have only recently begun with

LAM. Although there is no direct proof in the paper that the layered construction of MPICH is entirely the cause of its slower speed, the argument is believable and the reasoning is sound.

5.3 LAM-6.3 and MPICH-1.1.2

Ong and Farrell [14] conducted a study published in 2000 comparing LAM-6.3, MPICH-1.1.2, and MVICH (an MPI implementation utilizing VIA as the communication protocol). The tests were conducted using 2 PCs (450 Mhz each) connected by a Gigabit Ethernet using various NICs (Network Interface Cards). The primary finding of the study was that MVICH using VIA technology was superior to both LAM and MPICH, which rely on traditional TCP/IP for data transfer. The motivation of the study is very interesting, as the authors point out that the increasing availability of Gigabit Ethernet technology has shifted the bottleneck of message passing communication from the network media to network protocol processing. VIA (Virtual Interface Architecture) bypasses layers of the TCP/IP protocol stack, and reduces intermediate copies of data transfer during message transmission.

As far as the LAM and MPICH comparisons are concerned, LAM was reported to achieve superior bandwidth using NetPIPE-2.3 [19]. Three NICs were tested: Packet Engine GNIC-II, Alteon ACEnic, and SysKonnect SK-NET. LAM achieved

a maximum bandwidth of approximately 298 Mbps versus a maximum of 249 Mbps for MPICH. These numbers are reported for the SK-NET NIC. All LAM tests were conducted in fast client to client (c2c) mode, which uses daemons for MPI process creation but bypasses them for message transfer from MPI process to MPI process. It is worth repeating that in c2c mode, TCP (as opposed to UDP used in lamd mode) is used to transfer messages. Also, as stated earlier, MPICH does not use user-level daemons. Process startup is achieved by remote shell invocation, and TCP is always used for communication among nodes. Both LAM and MPICH show a drop in performance (moreso with LAM) at 128KB - the crossover size between "short" and "long" messages. The 128 KB size is the default setting in MPICH-1.1.2. LAM-6.3 has a default of 64 KB; the researchers changed this setting during the installation to 128 KB.

5.4 LAM-6.0 and MPICH-1.0.12

A 1996 report from the Ohio Supercomputer Center [13] compares the performance of LAM-6.0 and MPICH-1.0.12 on an 8 node DEC workstation cluster connected by a FDDI (Fiber-Distributed Data Interface) network. FDDI is capable of 100 Mbps bandwidth. The LAM tests were conducted using c2c mode. Six (presumably "in house") benchmarks were used in the comparison. Results were obtained by surrounding timing statements around a loop of communication and then dividing the

result by the number of iterations. Twenty of these observations (as the author called them) were performed for each benchmark and for each message size (message sizes ranged from 0 to 16,384 bytes), with the average of each set of twenty reported as final results. The LAM tests were run in c2c mode with the -O option to mpirun. This option specifies to LAM that the nodes of the parallel system are homogeneous, thus eliminating the need for data conversion. Note that our tests with LAM also used this option. It is unclear whether any other work (besides the DEC experiment) discussed in this chapter used or did not use the -O option (no references were made in any of the other published research results). MPICH requires no such mpirun option; it conveniently detects that the parallel system is homogeneous. In fact, in order to run MPICH in heterogeneous mode, a proc group file, as mentioned in chapter 3, is required.

Two tests, ping and ping-pong, were run with 2 processors (1 per node) and measured non-blocking, point-to-point communication. For message sizes up to 8,192 bytes in the ping test (one-way message transfer), LAM outperformed MPICH. For example, the time reported for LAM on a 2,000 byte message transfer is approximately 0.00051 seconds versus an MPICH time of approximately 0.00081 seconds. MPICH outperformed LAM in the ping test for message sizes greater than 8,192 bytes. The reported time for MPICH is approximately 0.0024 seconds versus approximately 0.0040 seconds for LAM for a message size of 10,200 bytes. The

8192 byte size discrepancy is also evident in the ping-pong (round trip message transfer) test, although the differences are not as dramatic, and the performance of both libraries is the same at 15,000 bytes, with LAM actually performing slightly faster between 15,000 and 16,384 bytes.

The author concluded that MPICH outperformed LAM for message sizes greater than 8,192 bytes because this is precisely the size at which LAM-6.0 distinguishes between “short” and “long” messages, and therefore switches from eager to rendezvous protocol. Recall that with eager protocol, a send is allowed to complete before a matching receive is posted, whereas a rendezvous protocol requires an acknowledgement from the receive before the send can complete. One common way in which to implement rendezvous protocol is to send the MPI message envelope first (containing information such as the source and destination addresses, ranks of the sender/receiver, and message tag), receive the acknowledgement, and then proceed by sending the actual data part of the message. In MPICH-1.0.12, the protocol change occurs at 16,384 bytes. Unfortunately, this was the longest message size tested for any of the experiments. It would be interesting to see the performance of both libraries for longer messages on the DEC cluster.

The four other benchmarks compared LAM and MPICH for MPI_Barrier, MPI_Broadcast, MPI_Gather, and MPI_Alltoall. MPI_Barrier is a synchronization

routine, and the other three are various forms of collective communication. Note that these are not the only forms of collective communication in MPI. The tests were conducted with 1 processor on each node of the cluster (8 total processors). The broadcast and all-to-all benchmarks tested message sizes in the range of 0 to 4,096 bytes. The range in the gather test was 4,000 to 16,384 bytes. No messages are sent with a barrier call. LAM outperformed MPICH for the barrier test (0.005185 seconds mean time versus 0.007268 seconds). For the broadcast test, LAM outperformed MPICH for all message sizes tested except those under 400 bytes. In the gather test, LAM outperformed MPICH for message sizes under approximately 9,500 bytes, with MPICH performing better between 9,500 bytes and 15,000 bytes. Between 15,000 bytes and 16,384 bytes, the performance of the libraries is approximately equal. LAM consistently outperformed MPICH (at least 3X faster for all sizes tested) in the all-to-all test.

5.5 Summary

This chapter has presented results of past research relating to the performance of LAM and MPICH. The studies are all unique in that they involve different platforms, different MPICH and LAM versions, and different test programs. While it was not possible to find a study comparing the same versions of LAM and MPICH that were compared in this thesis, the studies mentioned here are useful in under-

standing the performance of previous versions of the two libraries. Another reason the studies are useful is that they present different researchers' ideas as to why one library might perform better than the other on a given platform.

With a few exceptions, LAM outperformed MPICH in all of the studies. This common result does not imply that LAM is faster than MPICH in general. Rather, the studies (including this study) indicate that LAM runs faster on clusters. It is very probable that there are clusters running MPICH programs faster than they run LAM programs, especially considering the tuning potential and numerous configure options of both libraries. However, this thesis, along with the studies presented here, show that LAM is generally faster than MPICH on the clusters tested.

6 Conclusions and Future Work

We tested the performance of LAM and MPICH on three different categories of experiments: High Performance Linpack (HPL), the benchmark used by the top 500 list; su3_rmd, an MPI application used in physics research; and a series of bandwidth comparisons. LAM outperformed MPICH in nearly every Linpack test. Overall, LAM outperformed MPICH on Linpack by 2.6 % on average. The highest LAM score was 9.184 GFLOPS (4x6 grid; N = 25000). The highest MPICH score was 9.014 GFLOPS (4x6 grid; N = 25000). Both applications scaled as the number of processors was increased for all problem sizes except for the smallest two, N = 2k and N = 4k. This suggests that communication time becomes an unnecessary overhead for these problem sizes.

LAM and MPICH performed similarly on su3_rmd for most of the problem sizes tested. For the largest problem size, with the highest number of processors used, the execution time of LAM was 1,683 seconds versus an execution time of 1,831 seconds for MPICH, which is clearly significant. Because we experienced little variation among the repeated runs, the difference can be viewed as statistically significant. The MPI point-to-point construct used by su3_rmd is MPI_Ssend/ MPI_Irecv. We tested this combination, along with seven other point-to-point combinations. In every case, LAM outperformed MPICH. The performance difference

on MPI_Ssend/MPI_Irecv partially explains why LAM performed better than MPICH on su3_rmd for the large problem size.

One area of future research involves tuning and optimizing the LAM and MPICH libraries. The libraries were tested here without changing any of the configuration options, except that we used ssh (secure shell) instead of the default choice of rsh for both libraries. We did not change constants such as socket buffer sizes or protocol crossover sizes. The MPICH developers are in the process of developing a fully MPI-2 compliant implementation, and the frequency of LAM releases and the fact that LAM already provides dynamic process creation suggest that the LAM team will release a MPI-2 compliant implementation as well. It would be interesting to tune and optimize future MPI-2 compliant LAM and MPICH releases using programs involving MPI-2 functions.

Another area of future work involves detailed profiling of LAM and MPICH on su3_rmd and other "real life" applications. We attempted to profile using MPE and Jumpshot, but were unable to obtain aggregate function call times, and the trace files produced for LAM seemed to be erroneous. MPE and Jumpshot were valuable, however, for clearly displaying all MPI functions used by su3_rmd. It would be valuable to further analyze the performance of LAM and MPICH using a tool that provides trace file data and profiling information such as CPU time, communication time, and I/O time, as well as finer-grained information.

Running applications on the entire cluster remains to be done. With 48 nodes (96 processors), we would be able to more fully examine the scaling characteristics of the applications and of the cluster itself. Also, it would be interesting to examine a variety of MPI applications, each having a different communication pattern, to be able to determine empirically those kinds of MPI applications that scale well on the cluster architecture.

7 References

1. K. Andersson, D. Aronsson, and P. Karlsson. “An Evaluation of the System Performance of a Beowulf Cluster”. *National Supercomputer Centre in Linkoping Sweden, Internal Report No. 2001:4.* http://www.nsc.liu.se/support/articles/benchmarking.pdf. 2001.

2. R. Buyya (ed) *High Performance Cluster Computing: Programming and Applications, Volume 2*. Prentice Hall, NJ, USA, 1999.

3. P.H Carns, W.B Ligon III, S.P. McMillan, and R.B. Ross. “An Evaluation of Message Passing Implementations on Beowulf Workstations”. *Proceedings of the 1999 Extreme Linux Workshop.* June, 1999.

4. A. Chan, W. Gropp, and E. Lusk. “User’s Guide for MPE: Extensions for MPI Programs”. *Mathematics and Computer Science Division, Argonne National Laboratory.*

5. J. Dongarra, J. Bunch, C. Moller, and G.W. Stewart. “LINPACK User’s Guide”. *SIAM Publications.* Philadelphia, PA. 1979.

6. S. Browne, J. Dongarra, and K. London. “Review of Performance Analysis Tools for MPI Parallel Programs”. http://www.cs.utk.edu/~browne/perftools-review. December, 1997.

7. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A User’s Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

8. W. Gropp and E. Lusk. “Reproducible Measurements of MPI Performance Characteristics”. *Argonne National Laboratory, 1999 PVM MPI Meeting.* http://www.mcs.anl.gov/~gropp/bib/papers/1999/pvmmpi99/mpptest.pdf. 1999.

9. W. Gropp, E. Lusk, N. Doss, and A. Skellum. “A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard”. *Parallel Computing*, 22(6):789-828, September, 1996.

10. W. Gropp and E. Lusk. “User’s Guide for mpich, a Portable Implementation of MPI”. *Mathematics and Computer Science Division, Argonne National Laboratory.* 1996.

11. W. Gropp and E. Lusk. “MPICH Working Note: The Second-Generation ADI for the MPICH Implementation of MPI”. *Mathematics and Computer Science Division, Argonne National Laboratory.* 1996.

12. W. Gropp and E. Lusk. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* Second Edition, MIT Press, 1999.

13. N. Nevin, “The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster”. *Ohio Supercomputing Center, Technical Report OSC-TR-1196-4, Columbus, Ohio,* 1996.

14. H. Ong, and P. Farrell. “Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network”. *Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta, GA, October 10-14, 2000.*

15. P. Pacheco. *Parallel Programming With MPI.* Morgan Kaufman Publishers, Inc. San Francisco, CA. 1997.

16. G. F. Pfister. *In Search of Clusters (second edition).* Prentice Hall, NJ. 1998.

17. W. Saphir. “A Survey of MPI Implementations”. *Lawrence Berkeley National Laboratory, University of California, Berkeley, CA, Nov. 6, 1997.* Available from: http://www-library.lbl.gov/docs/LBNL/410/25/PDF/LBNL-41205.pdf.

18. S. Sistare and C. Jackson. “Ultra-High Performance Communication with MPI and the Sun Fire Link interconnect”. *Proceedings of the IEEE/ACM SC2002 Conference, 2002.*

19. Q. Snell, A Mikler, and J. Gustafson. “NetPIPE: Network Protocol Independent Performance Evaluator”. *Ames Laboratory, Scalable Computing Lab. Iowa State,* 1997.

20. M. Snir, S. Otto, S Huss-Lederman, D. Walker and J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1995.

21. T. Sterling, D. Becker, D. Savarese, J. Dorband, U. Ranawake, and C. Packer. “BEOWULF: A Parallel Workstation for Scientific Computation”. *Proceedings of the 1995 International Conference on Parallel Processing (IPCC),* Aug. 1995, Vol. 1, pp. 11-14.

22. T. Sterling, J. Salmon, D. Becker, and D. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters.* MIT Press, 1999.

23. J. Traff. “Implementing the MPI Process Topology”. *Proceedings of the IEEE/ ACM SC2002 Conference, 2002.*

24. J. Vetter. “Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies”. *ACM International Conference on Supercomputing, 2000* (Santa Fe, N.M.).

25. J. Vetter and A. Yoo. “An Empirical Performance Evaluation of Scalable Scientific Applications.” *Proceedings of the IEEE/ACM SC2002 Conference, 2002.*

26. B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, N.J. 1999.

27. “MPICH and LAM Performance on Astrolab”. http://www-hpcc.astro.washington.edu/faculty/trq/brandon/perform.html/. 1999.

28. “MPI Performance Topics”. http://www.llnl.gov/computing/tutorials/workshops/workshop/ mpi_performance/MAIN.html. July, 2002.

29. “Porting the LAM-MPI 6.3 Communication Layer”. Document authored by the “LAM Team”, available from the http://www.lam-mpi.org. March 8, 2000.

30. http://www.lam-mpi.org

31. http://www.top500.org/

32. http://www.netlib.org/pvm3/book/node8.html.

33. http://www-unix.mcs.anl.gov/mpi/mpich/

34. http://www.mpi-forum.org

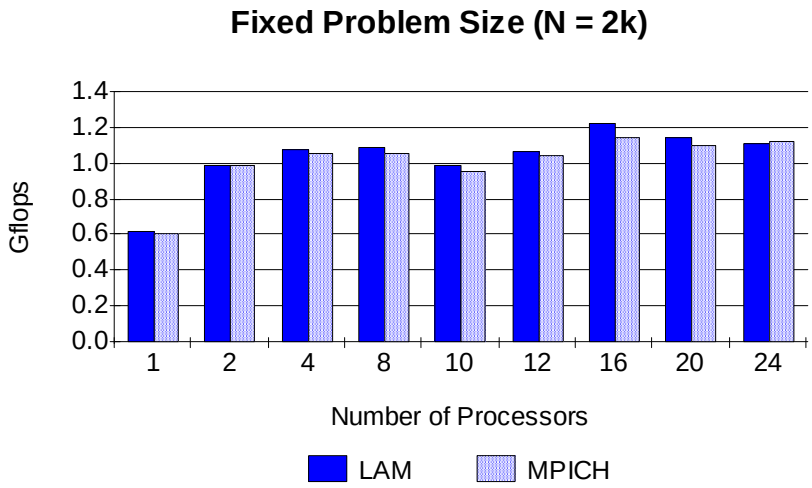
35. HPL Linpack Benchmark: http://www.netlib.org/benchmark/hpl/

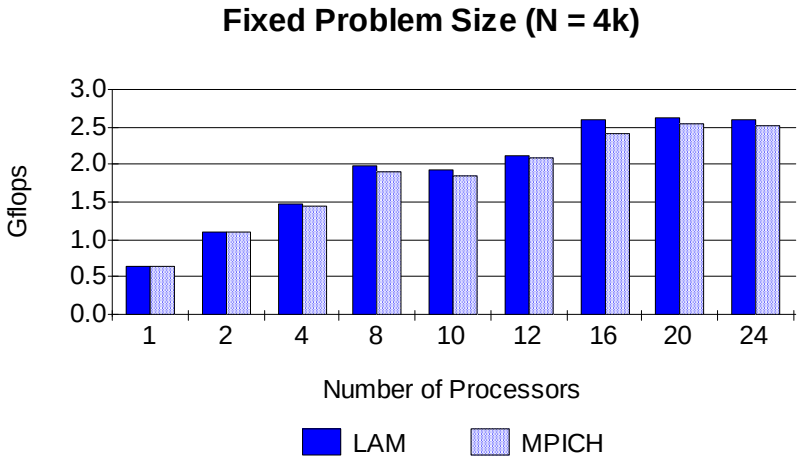
36. The su3_rmd program was obtained from the MIMD Lattice Computation (MILC) Collaboration: http://media4.physics.indiana.edu/~sg/milc.html

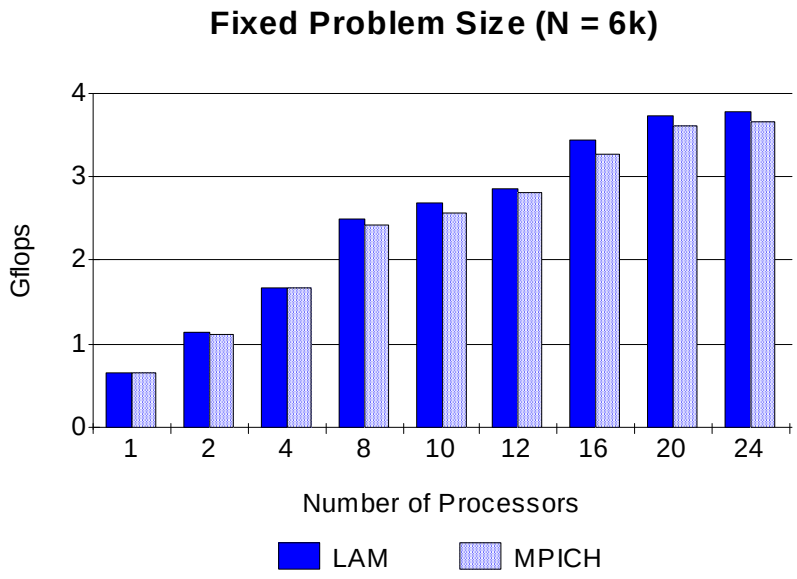
Appendix A: Sample HPL.dat Input File

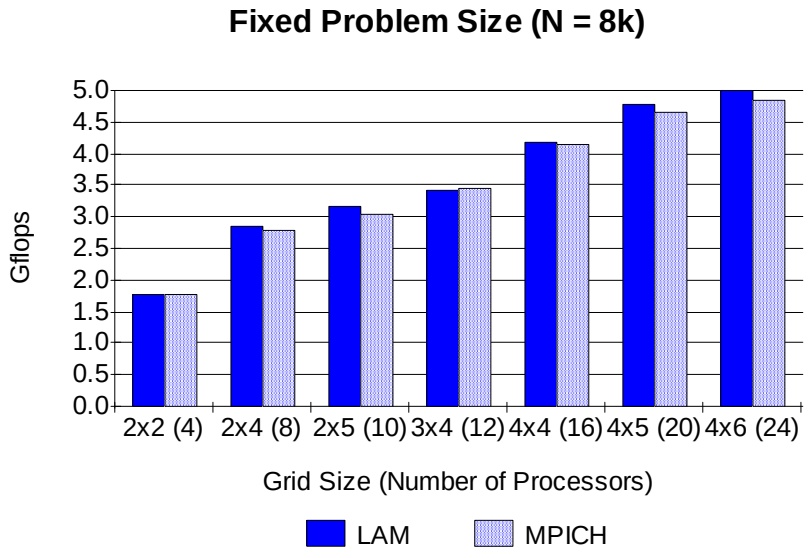
```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
1            device out (6=stdout,7=stderr,file)
6            # of problems sizes (N)
2000 4000 6000 8000 10000 15000         Ns
1            # of NBs
150          NBs
1            # of process grids (P x Q)
4            Ps
6            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINS (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
2            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
80           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

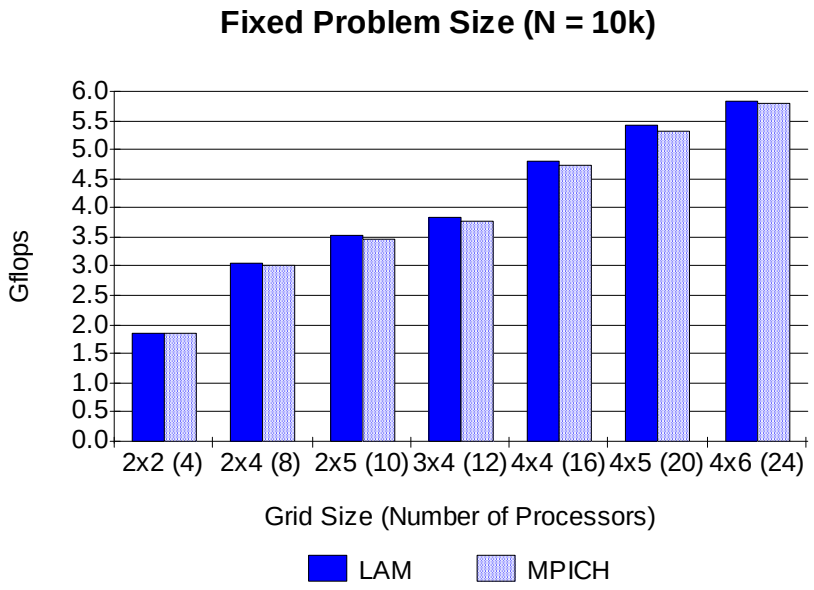
Appendix B: Linpack Results for Fixed Problem Sizes

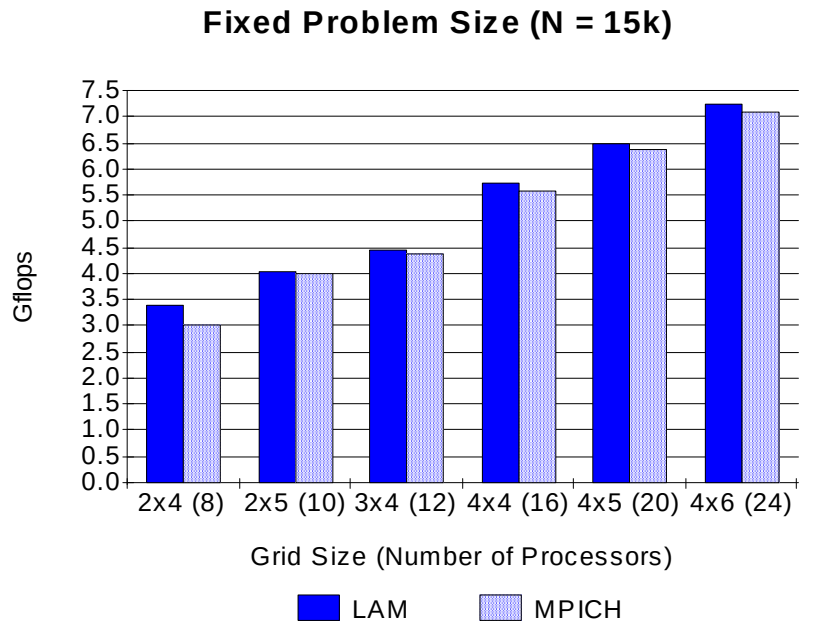


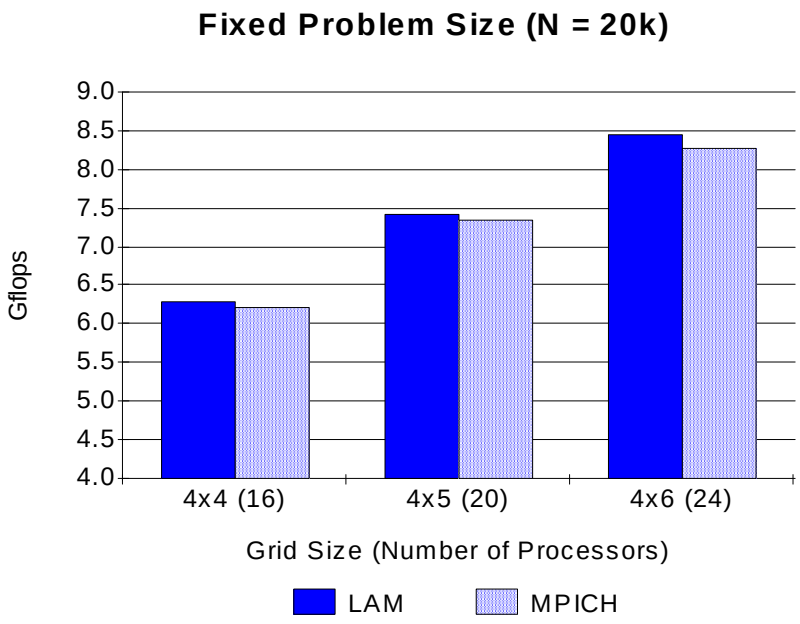


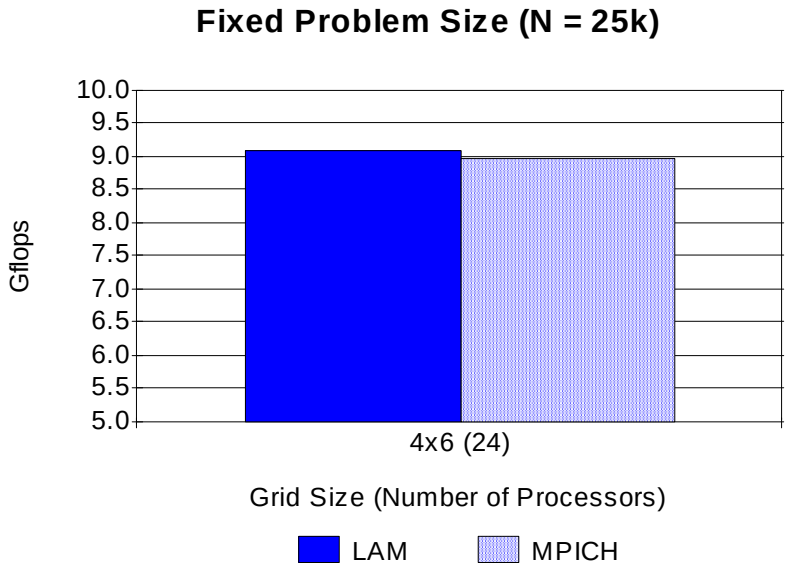

Fixed Problem Size (N = 6k)
Gflops
0
1
2
3
4
1
2
4
8
10
12
16
20
24
Number of Processors
LAM
MPICH


Fixed Problem Size (N = 8k)
Gflops
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
2x2 (4)
2x4 (8)
2x5 (10)
3x4 (12)
4x4 (16)
4x5 (20)
4x6 (24)
Grid Size (Number of Processors)
LAM
MPICH

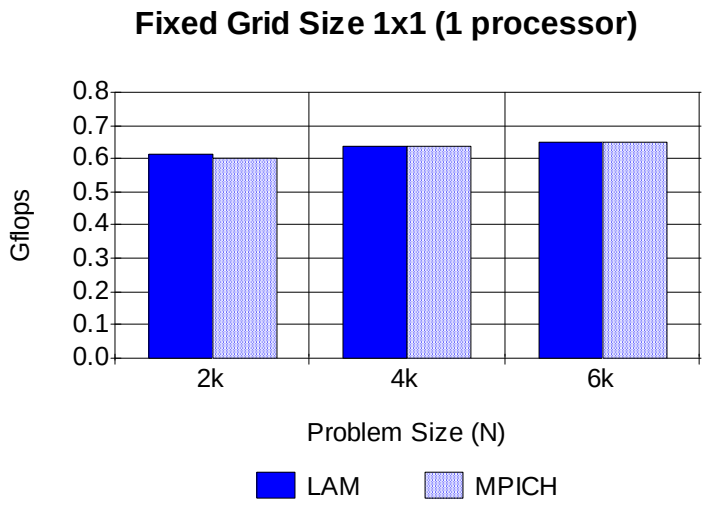

Fixed Problem Size (N = 10k)
Gflops
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2x2 (4)
2x4 (8)
2x5 (10)
3x4 (12)
4x4 (16)
4x5 (20)
4x6 (24)
Grid Size (Number of Processors)
LAM
MPICH

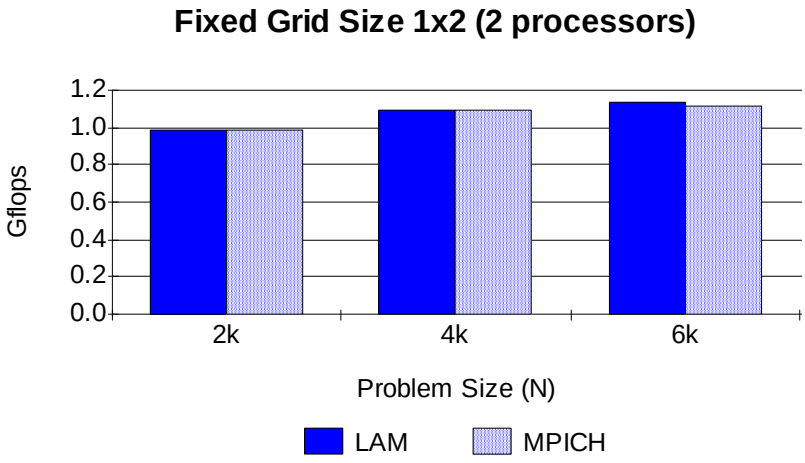

Fixed Problem Size (N = 15k)
Gflops
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2x4 (8)
2x5 (10)
3x4 (12)
4x4 (16)
4x5 (20)
4x6 (24)
Grid Size (Number of Processors)
LAM
MPICH

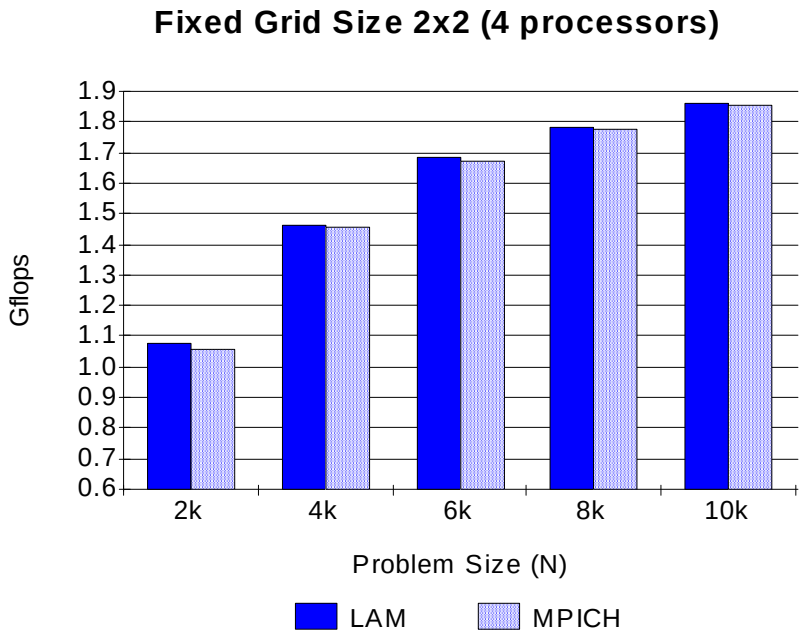

Fixed Problem Size (N = 20k)
9.0
8.5
8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
Gflops
4x4 (16)
4x5 (20)
4x6 (24)
Grid Size (Number of Processors)
LAM
MPICH

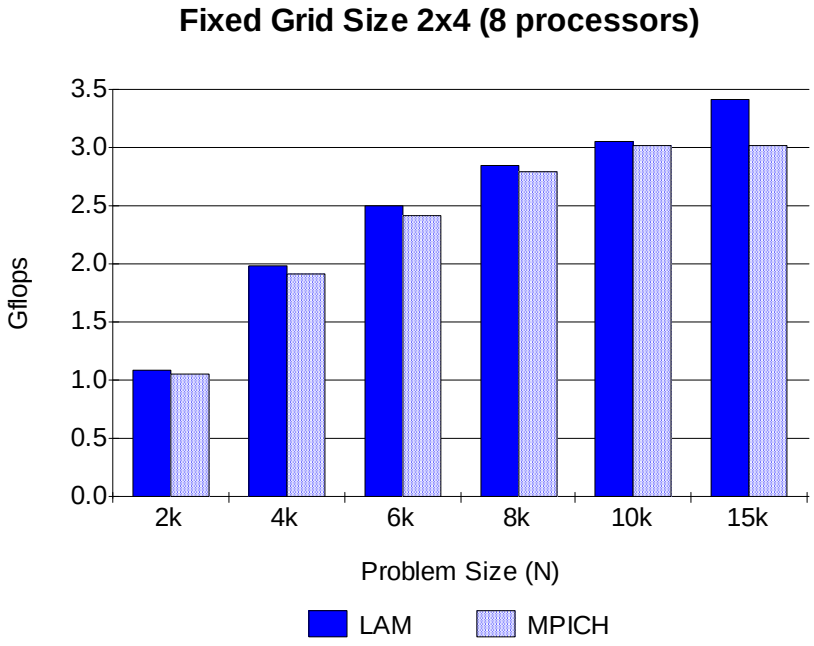

Fixed Problem Size (N = 25k)
10.0
9.5
9.0
8.5
8.0
7.5
7.0
6.5
6.0
5.5
5.0
Gflops
4x6 (24)
Grid Size (Number of Processors)
LAM
MPICH

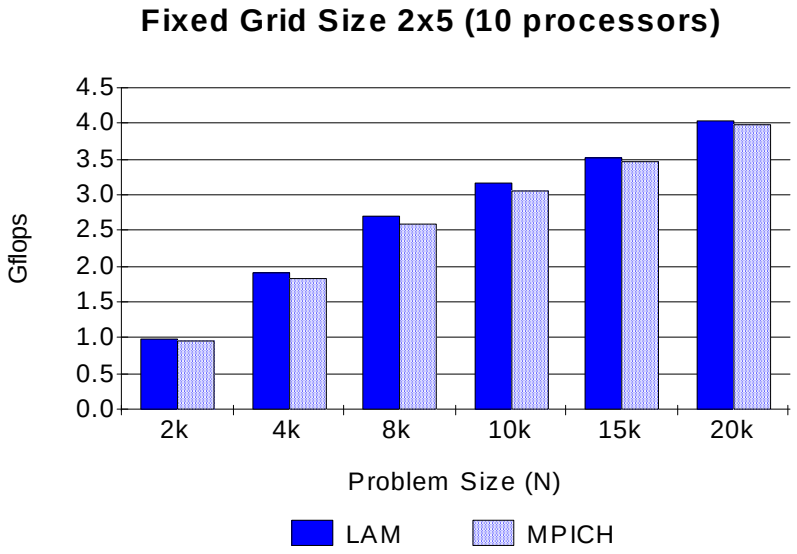
Appendix C: Linpack Results for Fixed Grid Sizes

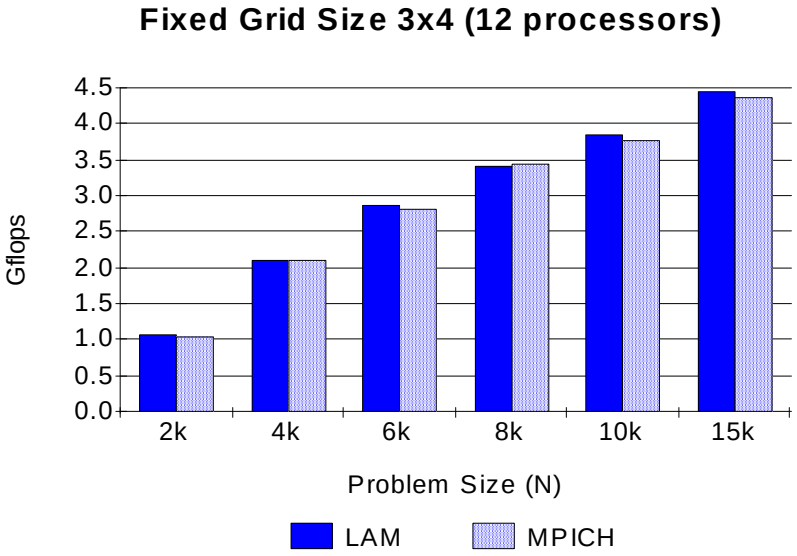


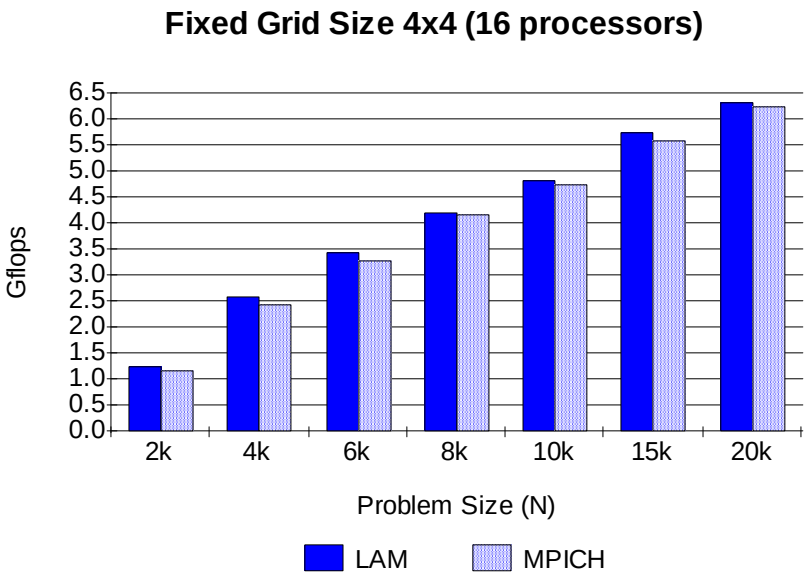


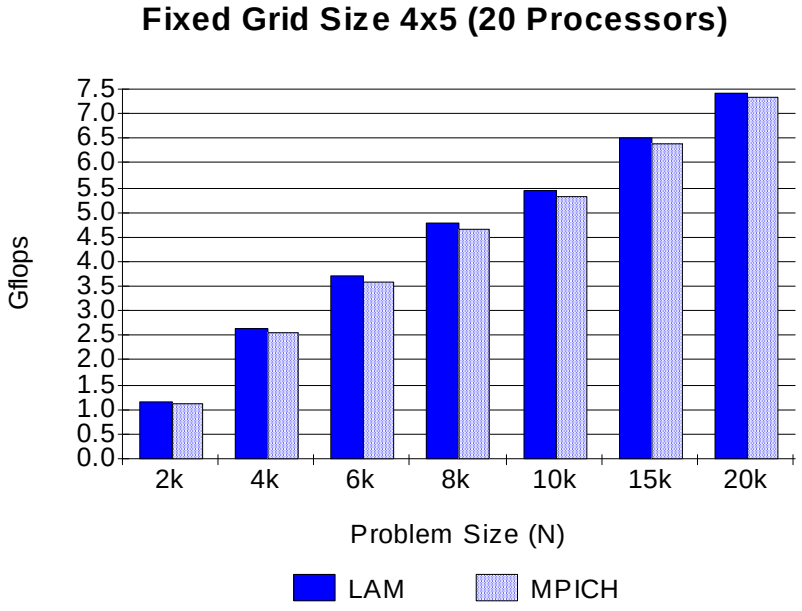

Fixed Grid Size 2x2 (4 processors)
Gflops
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
2k
4k
6k
8k
10k
Problem Size (N)
LAM
MPICH

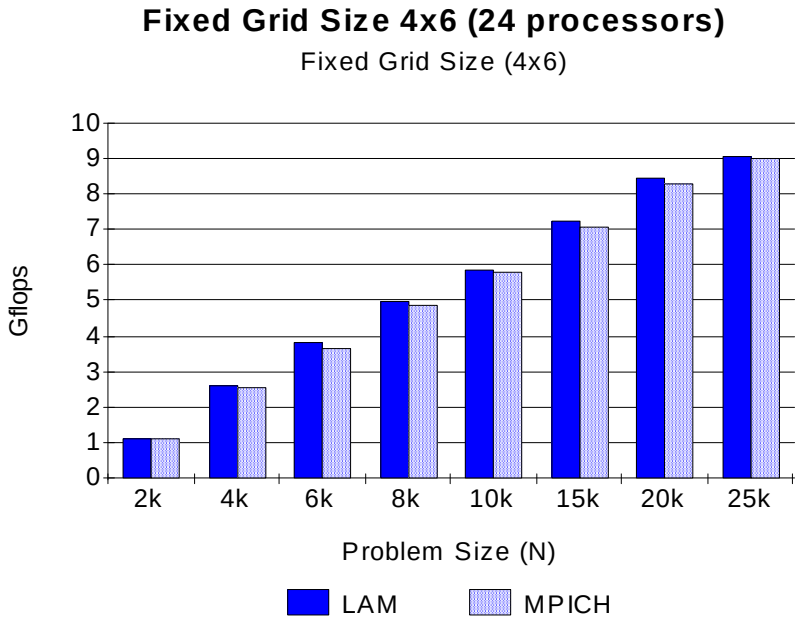

Fixed Grid Size 2x4 (8 processors)
Gflops
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2k
4k
6k
8k
10k
15k
Problem Size (N)
LAM
MPICH


Fixed Grid Size 2x5 (10 processors)
Gflops
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2k
4k
8k
10k
15k
20k
Problem Size (N)
LAM
MPICH


Fixed Grid Size 3x4 (12 processors)
Gflops
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2k
4k
6k
8k
10k
15k
Problem Size (N)
LAM
MPICH


Fixed Grid Size 4x4 (16 processors)
Gflops
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2k
4k
6k
8k
10k
15k
20k
Problem Size (N)
LAM
MPICH


Fixed Grid Size 4x5 (20 Processors)
Gflops
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2k
4k
6k
8k
10k
15k
20k
Problem Size (N)
LAM
MPICH


Fixed Grid Size 4x6 (24 processors)
Fixed Grid Size (4x6)
Gflops
10
9
8
7
6
5
4
3
2
1
0
2k
4k
6k
8k
10k
15k
20k
25k
Problem Size (N)
LAM
MPICH

Appendix D: Standard Deviation for Linpack Repeated Runs

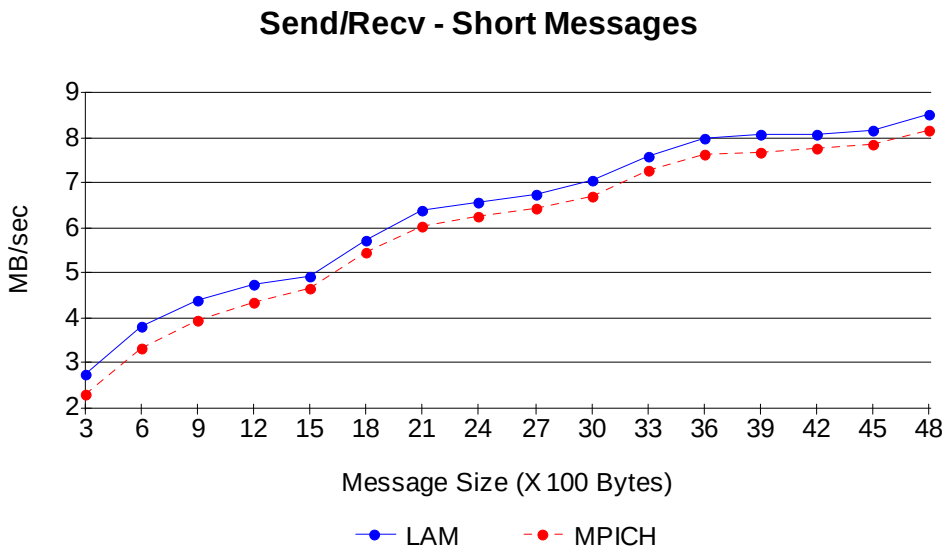
Grid Size	Problem Size	LAM	MPICH
1x1	2,000	0.00067	0.00010
1x1	4,000	0.00039	0.00024
1x1	6,000	0.00037	0.00019
1x2	2,000	0.00745	0.00422
1x2	4,000	0.00303	0.00040
1x2	6,000	0.00098	0.00426
2x2	2,000	0.01503	0.00818
2x2	4,000	0.00759	0.00402
2x2	6,000	0.00133	0.00102
2x2	8,000	0.00089	0.00075
2x2	10,000	0.00407	0.00387
2x4	2,000	0.01127	0.00150
2x4	4,000	0.01858	0.01977
2x4	6,000	0.01559	0.00811
2x4	8,000	0.02128	0.01376
2x4	10,000	0.00821	0.01076
2x4	15,000	0.00934	0.08515
2x5	2,000	0.01243	0.01273
2x5	4,000	0.00997	0.00492
2x5	6,000	0.02681	0.01374
2x5	8,000	0.02042	0.01269
2x5	10,000	0.01124	0.00595
2x5	15,000	0.00299	0.02693
3x4	2,000	0.01028	0.00174
3x4	4,000	0.02926	0.01991
3x4	6,000	0.03099	0.00582
3x4	8,000	0.03187	0.01266
3x4	10,000	0.01525	0.01105
3x4	15,000	0.04227	0.04408

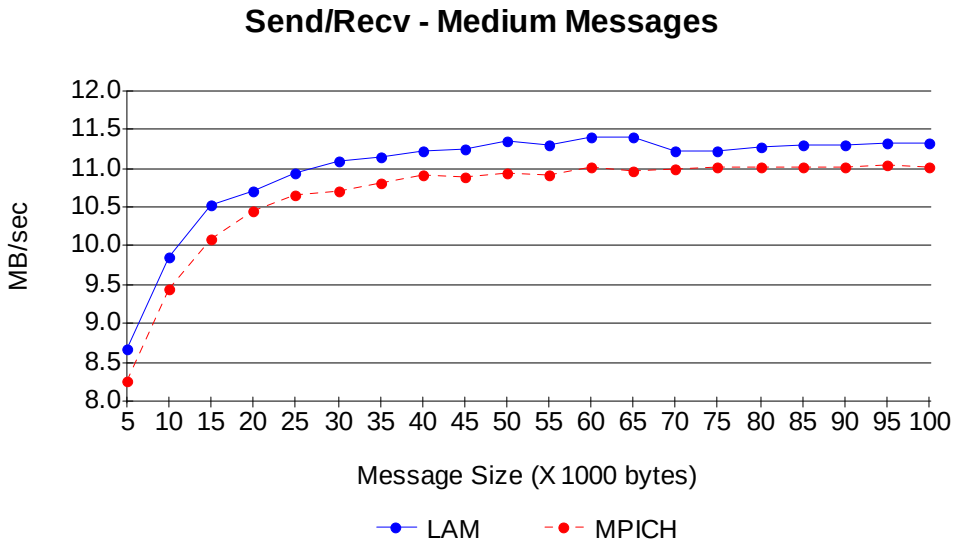
Grid Size	Problem Size	LAM	MPICH
4x4	2,000	0.03306	0.04468
4x4	4,000	0.04882	0.02836
4x4	6,000	0.04596	0.07677
4x4	8,000	0.02209	0.02657
4x4	10,000	0.07670	0.03000
4x4	15,000	0.02822	0.01927
4x5	2,000	0.03929	0.02527
4x5	4,000	0.06208	0.01474
4x5	6,000	0.09246	0.03542
4x5	8,000	0.05180	0.06016
4x5	10,000	0.07972	0.08071
4x5	15,000	0.03672	0.06141
4x5	20,000	0.02682	0.07099
4x6	2,000	0.04023	0.02713
4x6	4,000	0.02420	0.03581
4x6	6,000	0.07464	0.04133
4x6	8,000	0.03081	0.05022
4x6	10,000	0.05695	0.06211
4x6	15,000	0.05854	0.06190
4x6	20,000	0.08662	0.03569
4x6	25,000	0.06681	0.02211

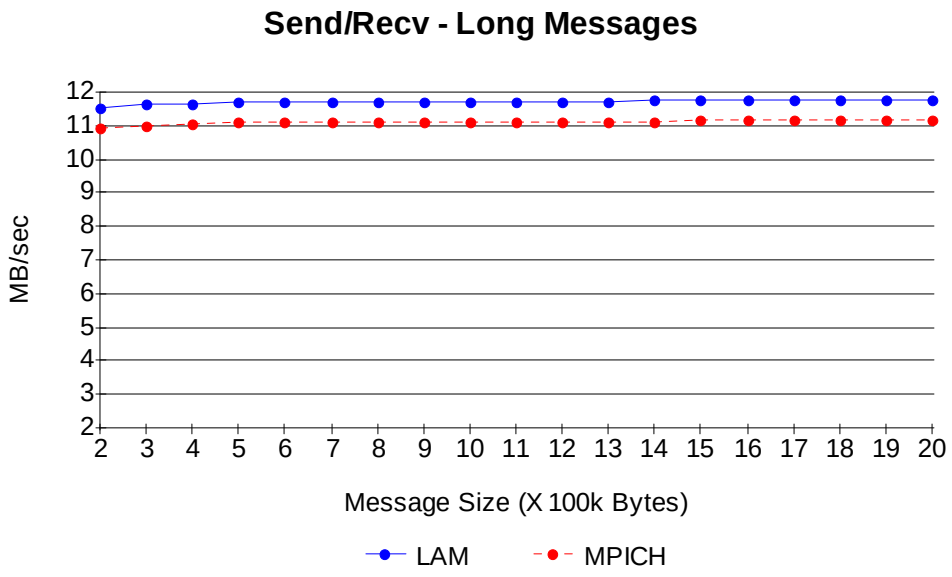
Appendix E: Standard Deviation for su3_rmd Repeated Runs

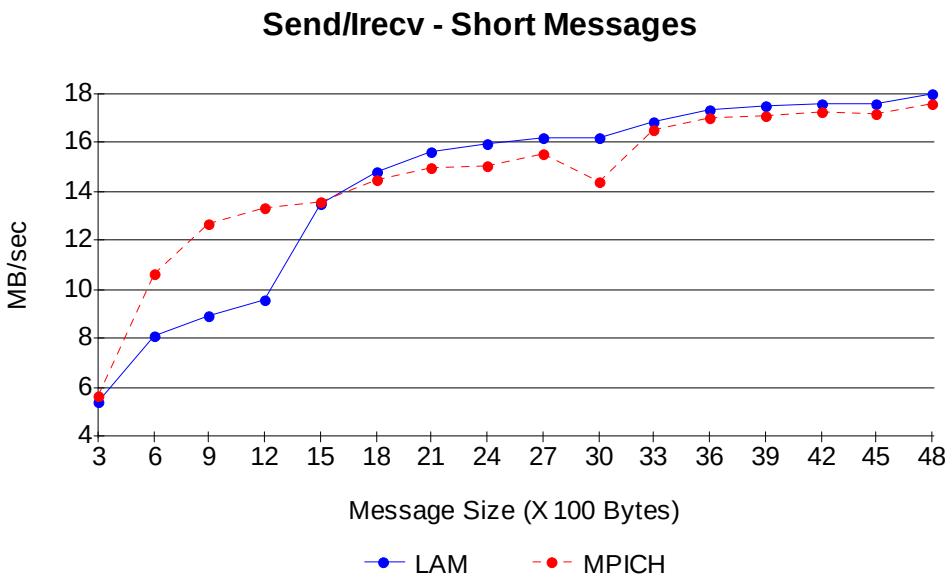
Problem Size	Processors	LAM Wall	LAM CPU	MPICH WALL	MPICH CPU
Experiment 1 nx=ny=nz=4, nt=6, traj=50	2	0.09287	0.10419	0.03929	0.00800
	4	0.07950	0.05083	0.25492	0.25492
	8	0.13691	0.10315	0.35704	0.35704
	16	0.07782	0.16228	0.38370	0.38370
Experiment 2 nx=ny=nz=6, nt=12, traj=75	2	1.39872	0.19906	0.08405	0.09091
	4	0.33223	0.62933	0.18203	0.20675
	8	0.24848	0.10778	0.11016	0.09988
	16	0.34971	0.11825	1.08901	1.18567
Experiment 3 nx=ny=nz=8, nt=16, traj=100	2	0.09209	0.09330	0.17577	0.16017
	4	0.29842	0.66479	0.41761	0.42544
	8	0.52518	0.45579	0.34681	0.43961
	16	0.41639	0.30688	0.35491	0.44512
Experiment 4 nx=ny=nz=12, nt=24, traj=150	4	0.83731	1.13228	0.88773	0.66028
	8	0.45913	2.56203	1.17123	0.88490
	16	0.29600	0.39520	1.39421	0.52378
Experiment 5 nx=ny=nz=16, nt=32, traj=200	16	1.61788	0.61788	0.63312	1.06683

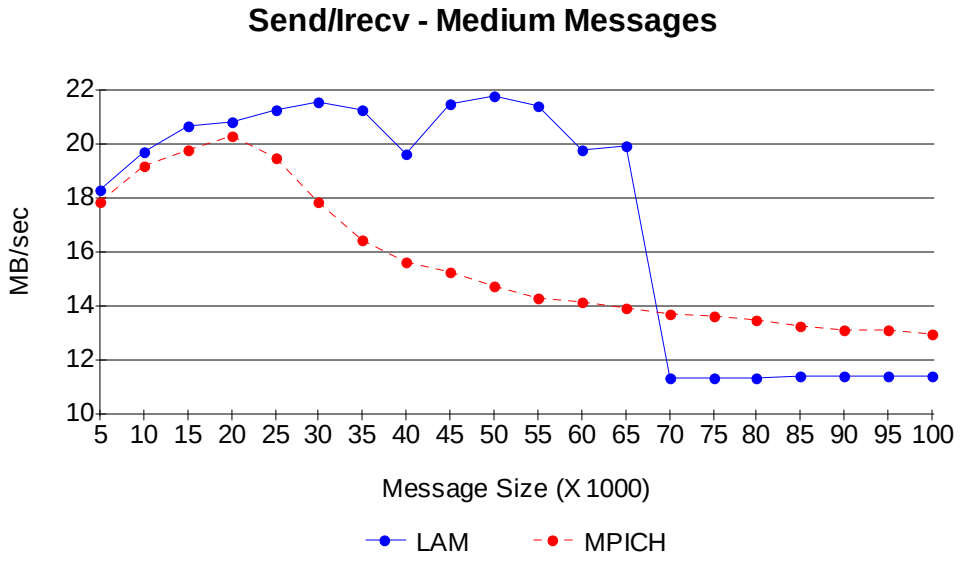
Appendix F: Bandwidth Comparison Graphs

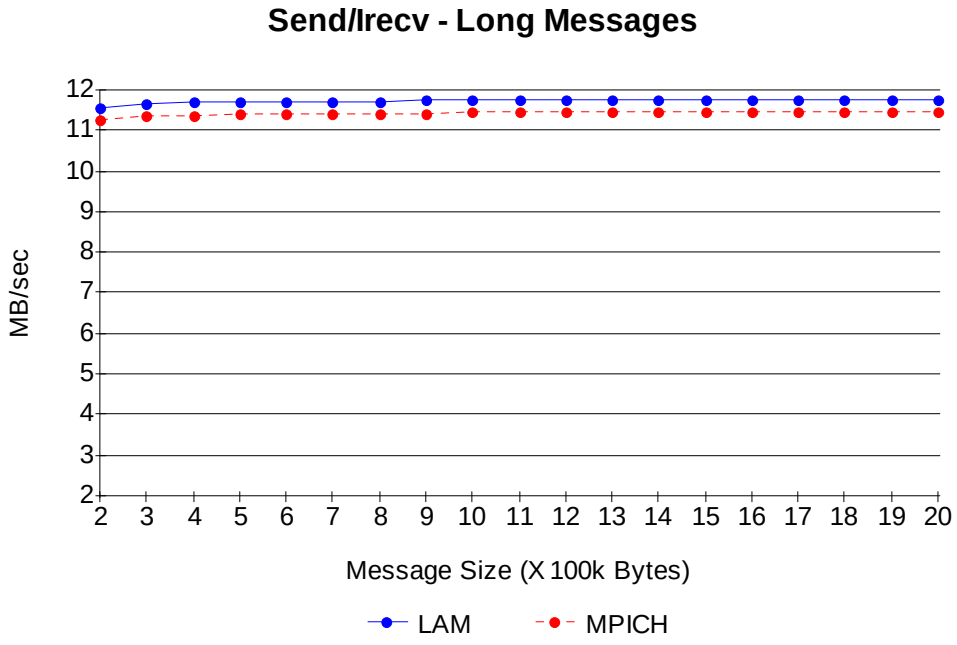


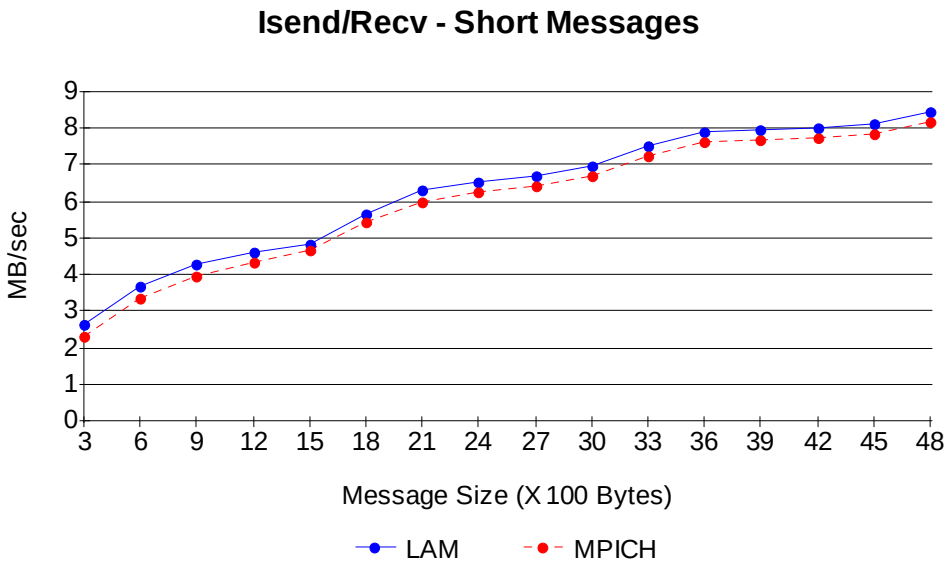


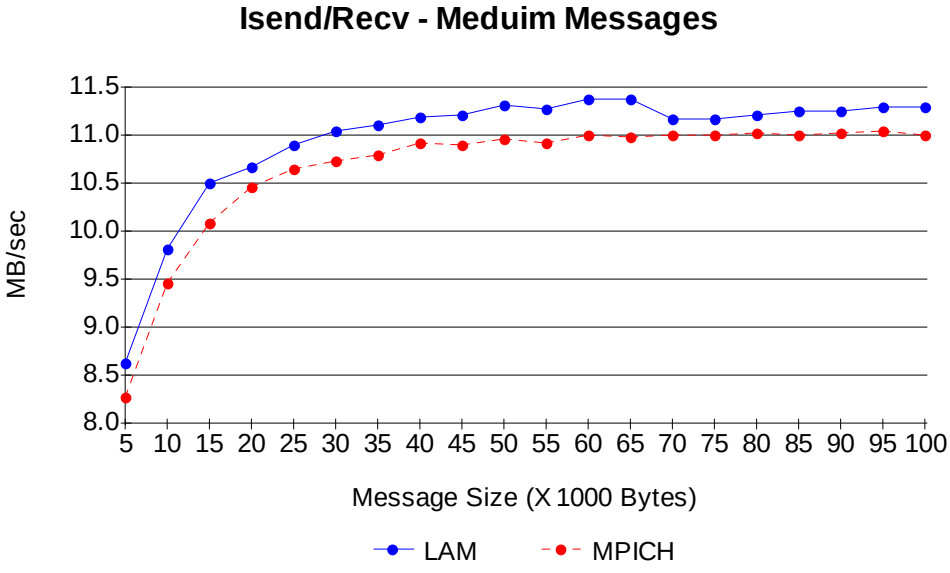

Send/Recv - Long Messages
MB/sec
12
11
10
9
8
7
6
5
4
3
2
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 100k Bytes)
LAM
MPICH

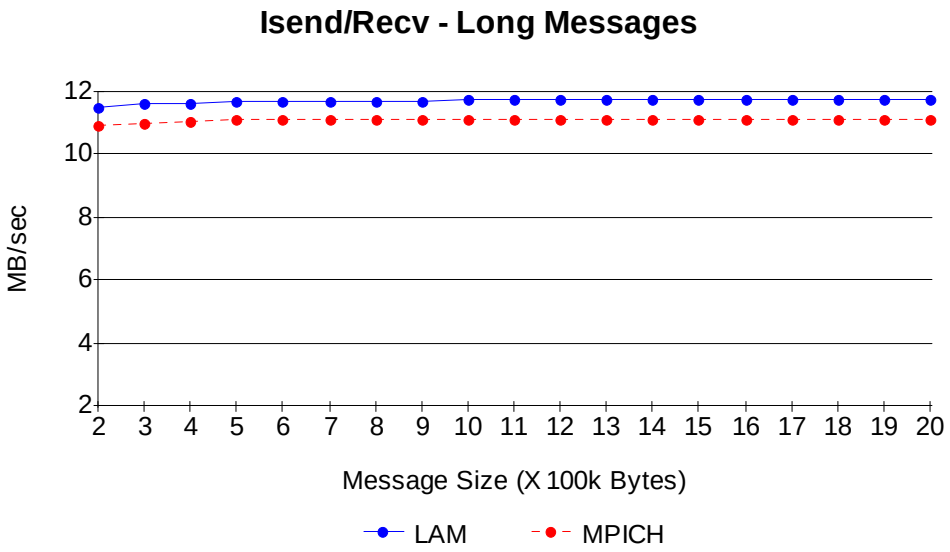

Send/Irecv - Short Messages
MB/sec
18
16
14
12
10
8
6
4
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH

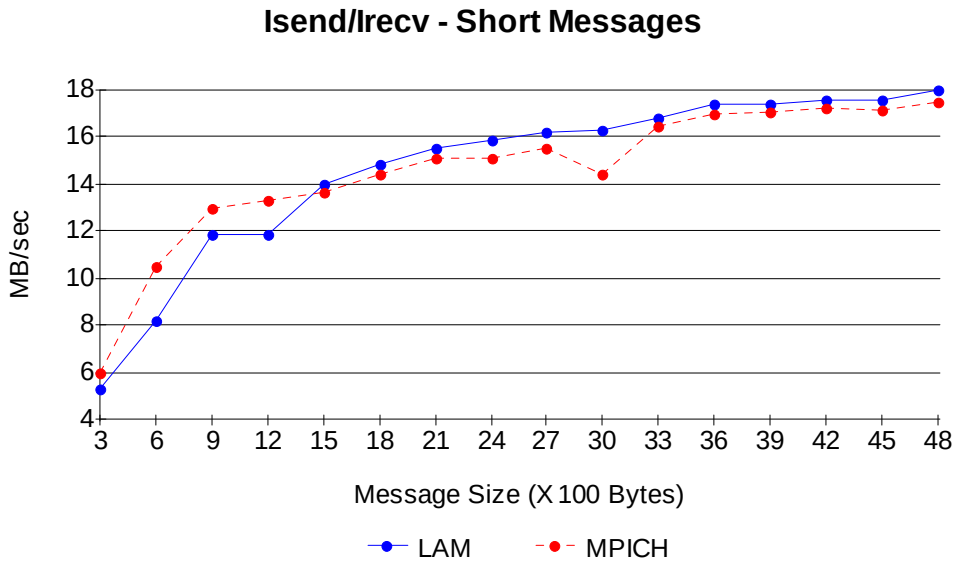

Send/Irecv - Medium Messages
MB/sec
22
20
18
16
14
12
10
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message Size (X 1000)
LAM
MPICH

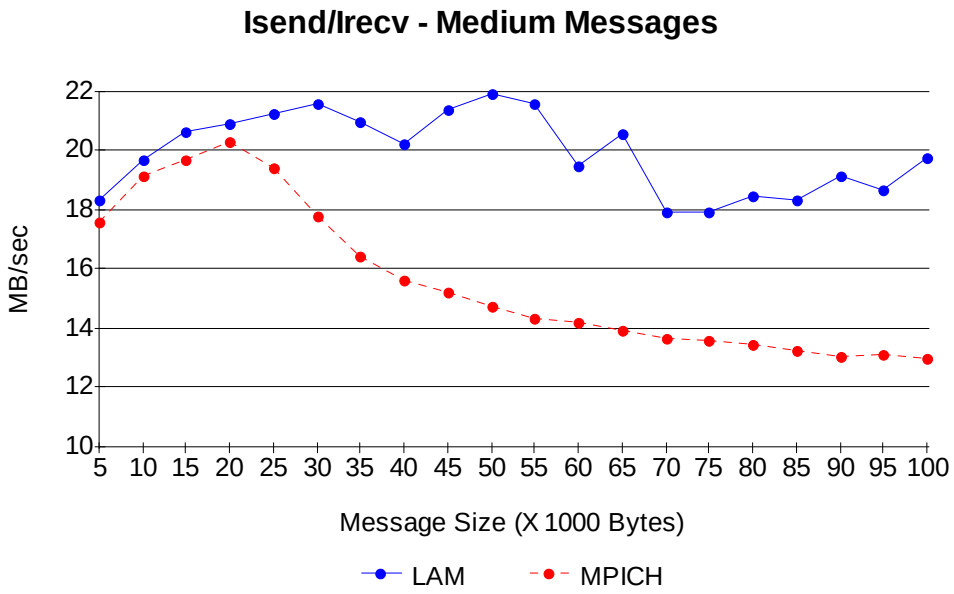

Send/Irecv - Long Messages
MB/sec
12
11
10
9
8
7
6
5
4
3
2
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 100k Bytes)
LAM
MPICH

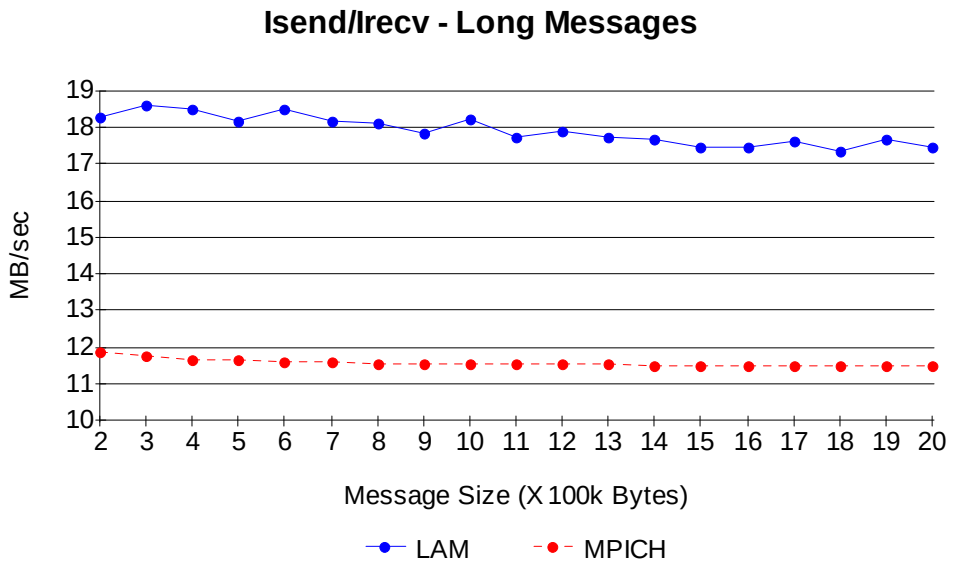

Isend/Recv - Short Messages
MB/sec
0
1
2
3
4
5
6
7
8
9
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH

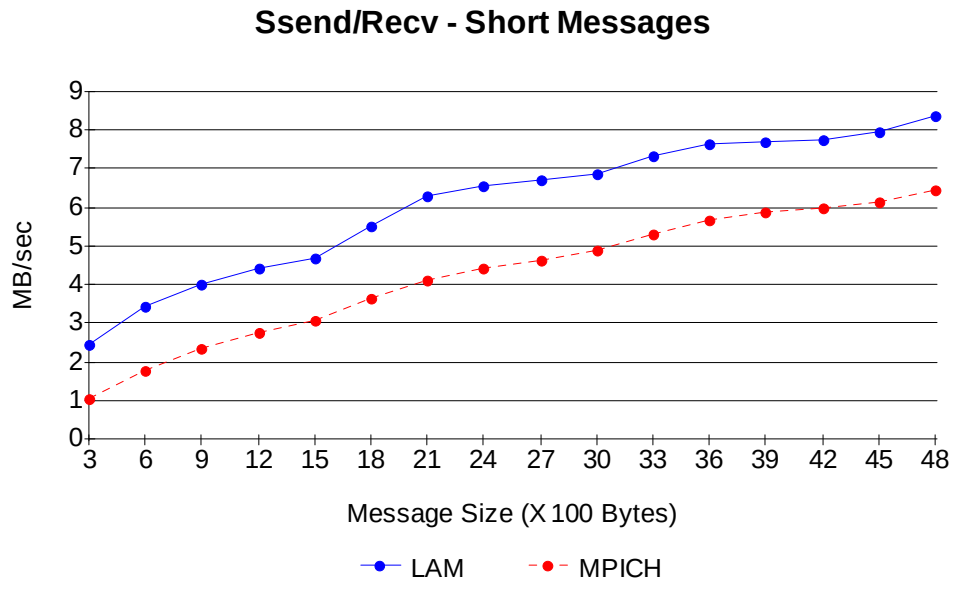

Isend/Recv - Meduim Messages
MB/sec
8.0
8.5
9.0
9.5
10.0
10.5
11.0
11.5
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message Size (X 1000 Bytes)
LAM
MPICH

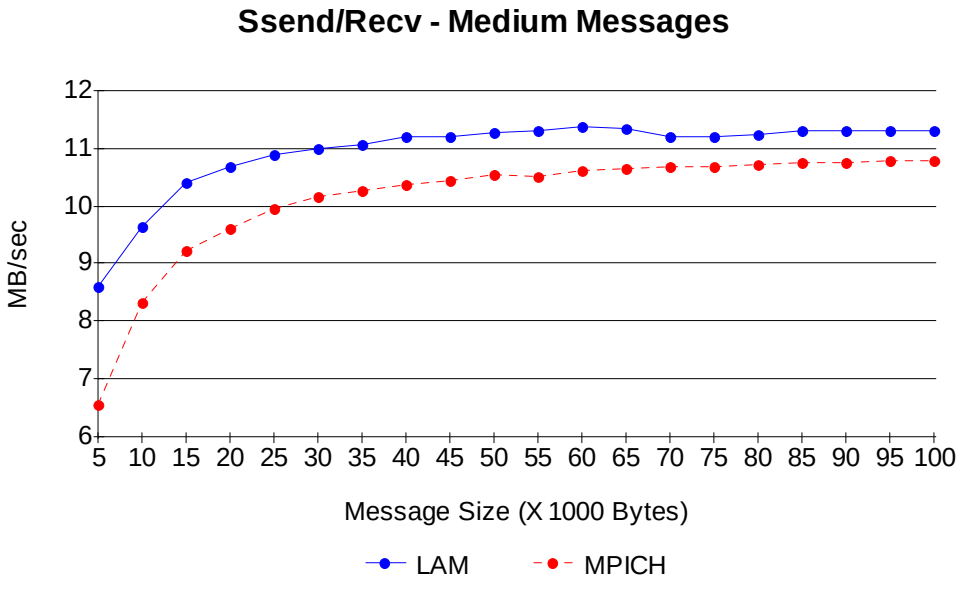

Isend/Recv - Long Messages
MB/sec
12
10
8
6
4
2
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 100k Bytes)
LAM
MPICH

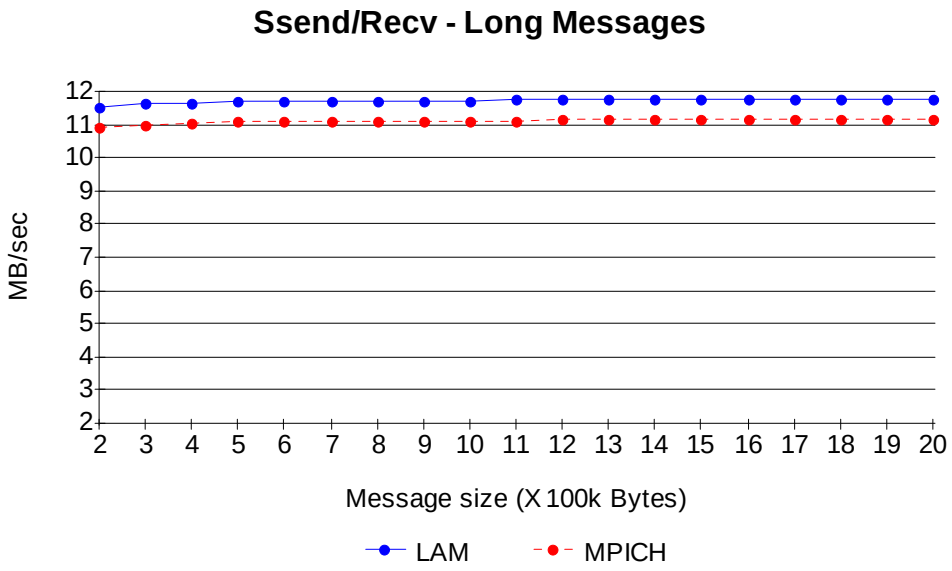

Isend/Irecv - Short Messages
MB/sec
18
16
14
12
10
8
6
4
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH

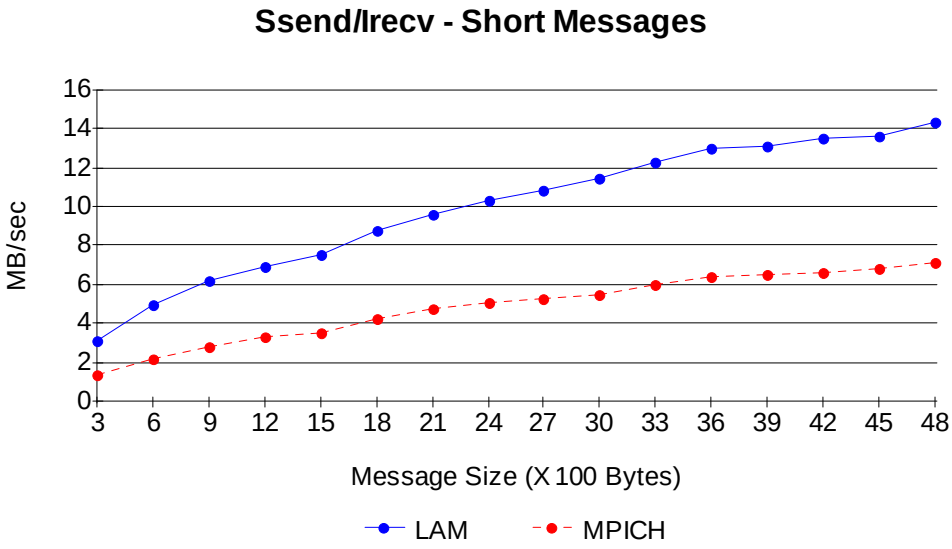

Isend/Irecv - Medium Messages
MB/sec
22
20
18
16
14
12
10
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message Size (X 1000 Bytes)
LAM
MPICH

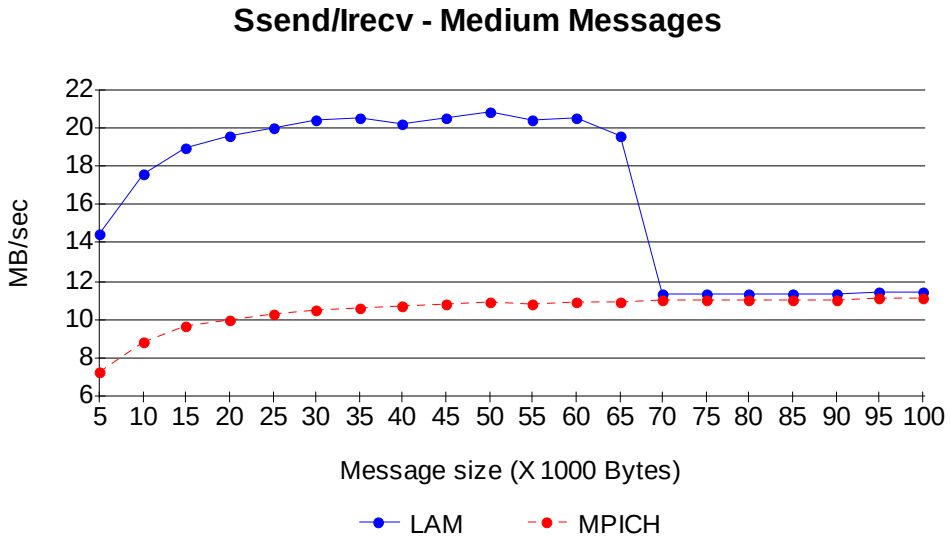

Isend/Irecv - Long Messages
MB/sec
19
18
17
16
15
14
13
12
11
10
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 100k Bytes)
LAM
MPICH

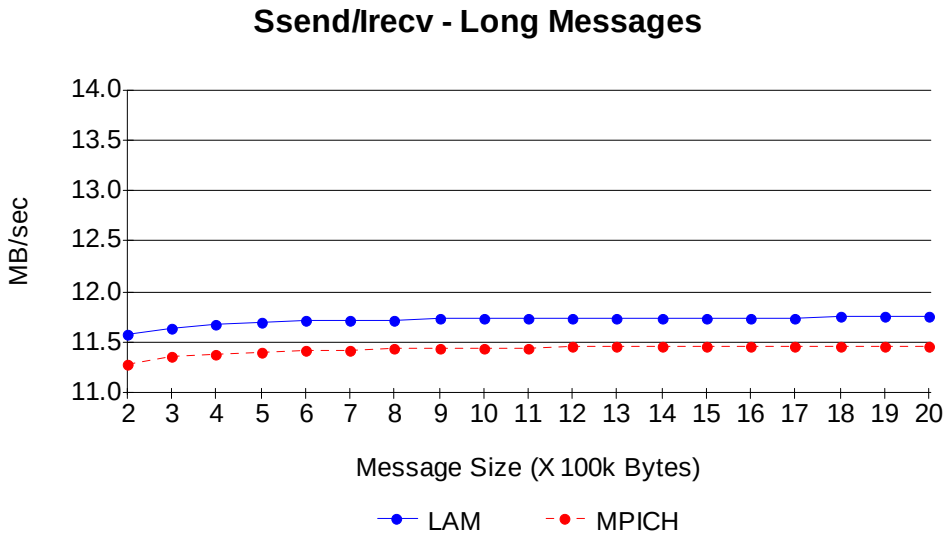

Ssend/Recv - Short Messages
MB/sec
0
1
2
3
4
5
6
7
8
9
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH

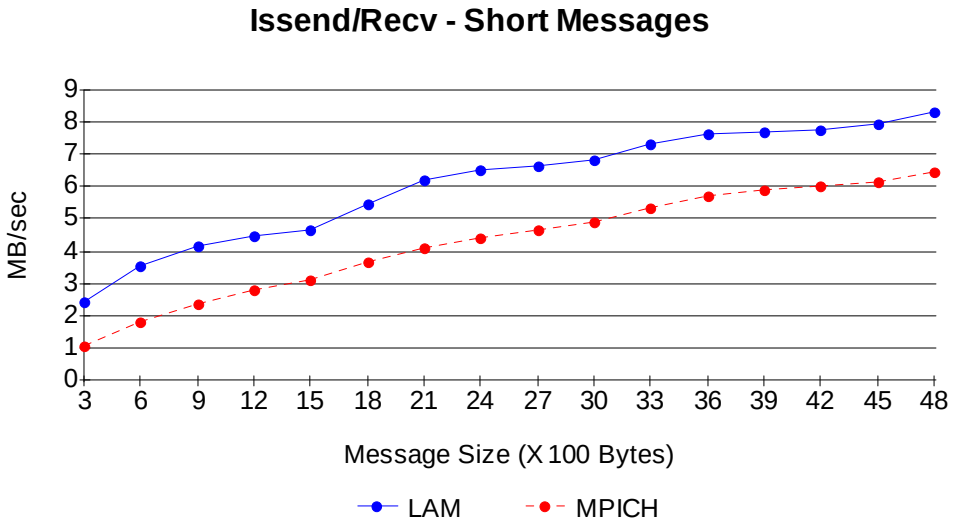

Ssend/Recv - Medium Messages
MB/sec
6
7
8
9
10
11
12
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message Size (X 1000 Bytes)
LAM
MPICH

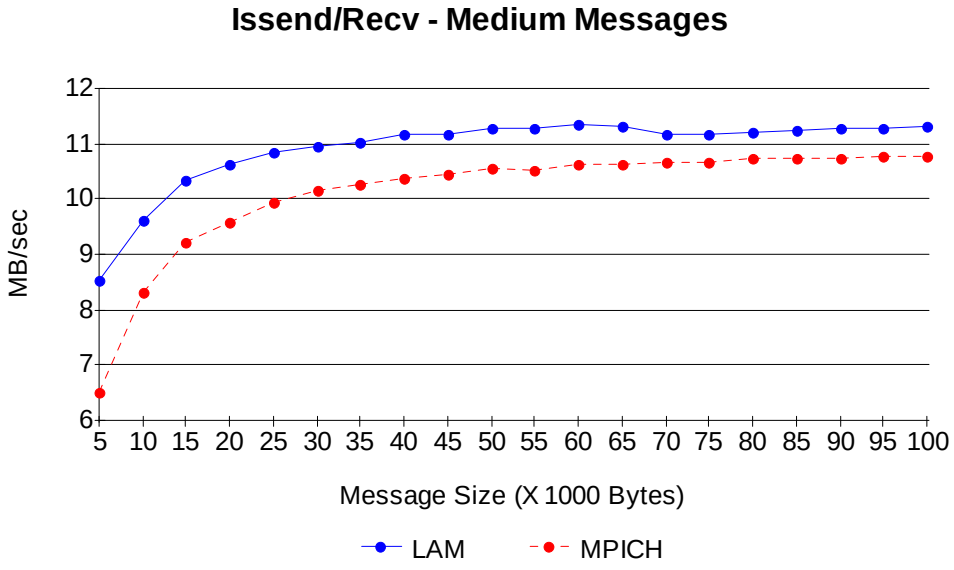

Ssend/Recv - Long Messages
MB/sec
12
11
10
9
8
7
6
5
4
3
2
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message size (X 100k Bytes)
LAM
MPICH

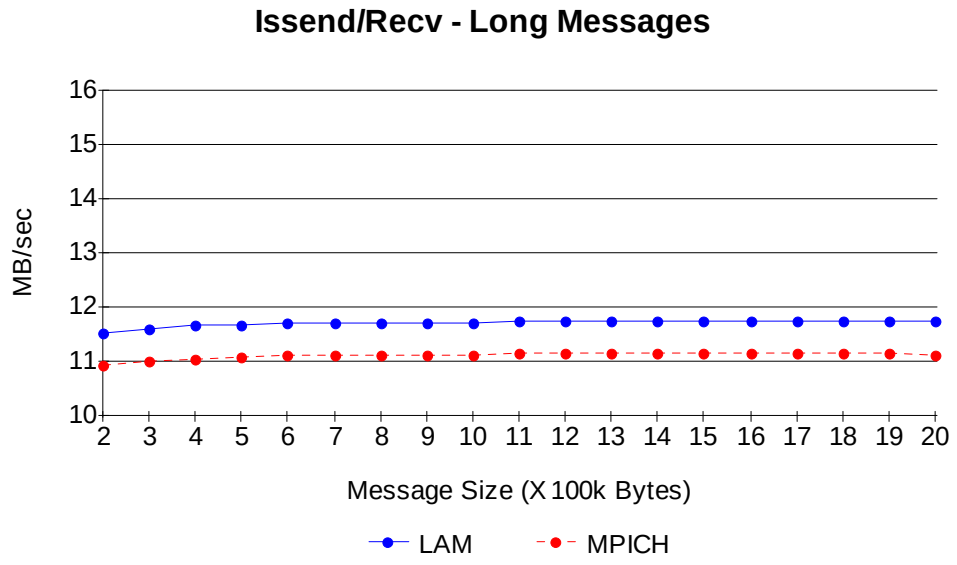

Ssend/Irecv - Short Messages
MB/sec
16
14
12
10
8
6
4
2
0
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH

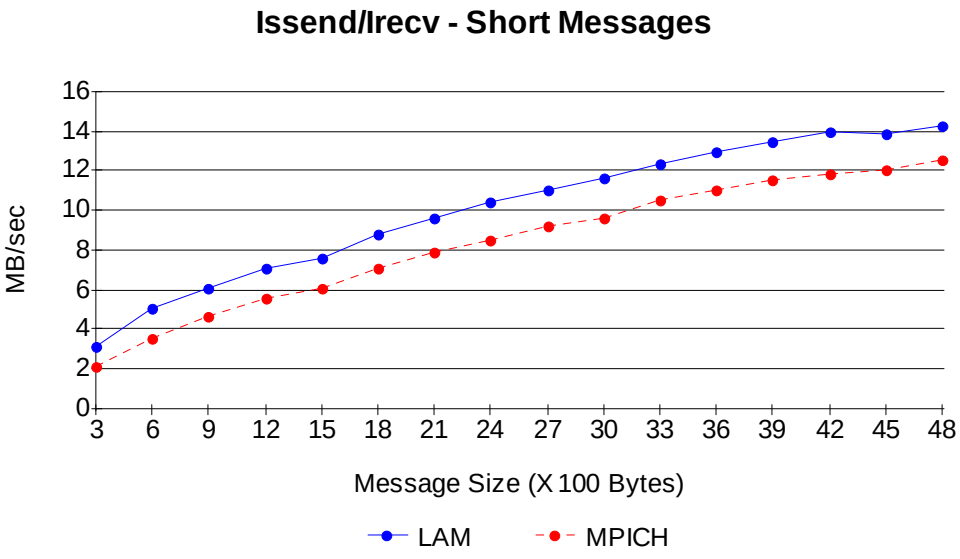

Ssend/Irecv - Medium Messages
MB/sec
22
20
18
16
14
12
10
8
6
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message size (X 1000 Bytes)
LAM
MPICH

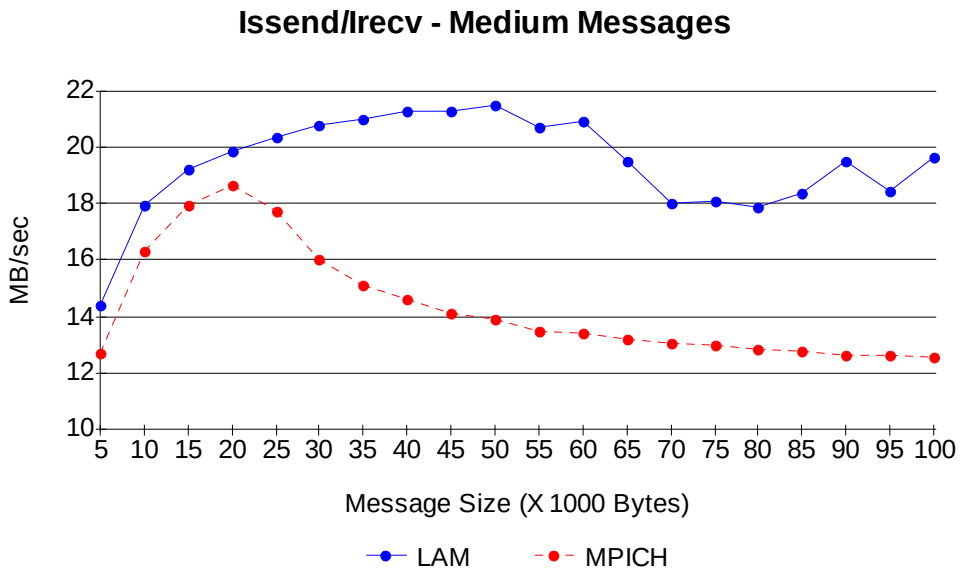

Ssend/Irecv - Long Messages
MB/sec
14.0
13.5
13.0
12.5
12.0
11.5
11.0
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 100k Bytes)
LAM
MPICH

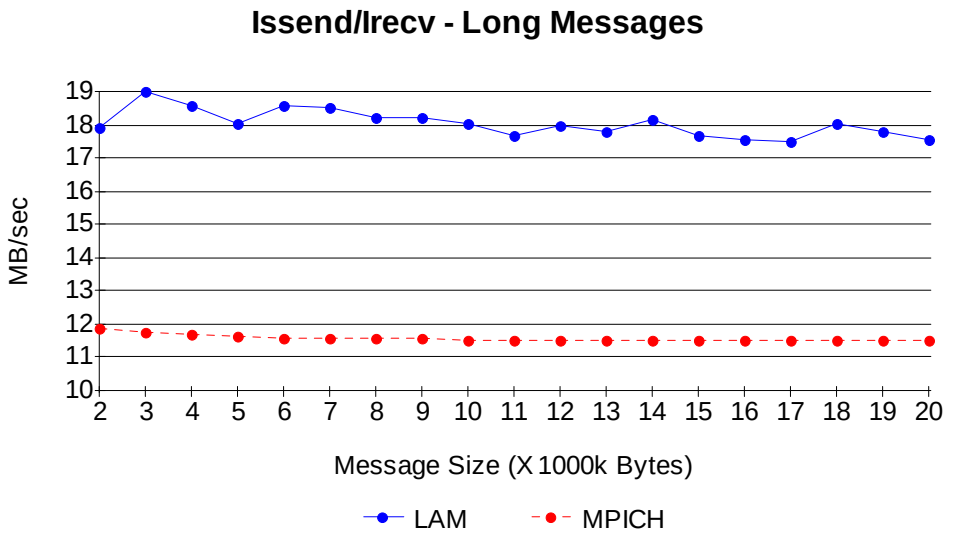

Issend/Recv - Short Messages
MB/sec
0
1
2
3
4
5
6
7
8
9
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH


Issend/Recv - Medium Messages
MB/sec
6
7
8
9
10
11
12
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message Size (X 1000 Bytes)
LAM
MPICH


Issend/Recv - Long Messages
MB/sec
16
15
14
13
12
11
10
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 100k Bytes)
LAM
MPICH


Issend/Irecv - Short Messages
MB/sec
16
14
12
10
8
6
4
2
0
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH


Issend/Irecv - Medium Messages
MB/sec
22
20
18
16
14
12
10
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message Size (X 1000 Bytes)
LAM
MPICH


Issend/Irecv - Long Messages
MB/sec
19
18
17
16
15
14
13
12
11
10
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 1000k Bytes)
LAM
MPICH

Appendix G: Bandwidth Raw Data

Short Messages

Construct	Message Size	LAM Best	LAM Avg	LAM Worst	MPICH Best	MPICH Avg	MPICH Worst
Send/Recv	300	2.79	2.76	2.63	2.35	2.30	2.08
	600	3.86	3.83	3.73	3.41	3.34	3.08
	900	4.42	4.39	4.28	4.00	3.95	3.71
	1,200	4.75	4.74	4.60	4.37	4.33	3.89
	1,500	4.98	4.93	4.79	4.71	4.64	4.50
	1,800	5.80	5.72	5.61	5.55	5.44	5.33
	2,100	6.46	6.39	6.25	6.10	6.01	5.80
	2,400	6.62	6.56	6.43	6.37	6.24	6.02
	2,700	6.78	6.72	6.57	6.51	6.42	6.24
	3,000	7.14	7.04	6.77	6.80	6.70	6.53
	3,300	7.69	7.59	7.44	7.33	7.26	7.10
	3,600	8.04	7.98	7.84	7.68	7.62	7.49
	3,900	8.08	8.05	7.89	7.72	7.68	7.54
	4,200	8.10	8.05	7.85	7.80	7.74	7.59
	4,500	8.22	8.16	8.05	7.92	7.85	7.57
	4,800	8.59	8.53	8.43	8.26	8.17	7.70
Send/Irecv	300	8.00	5.40	3.73	12.24	5.66	2.79
	600	14.12	8.13	4.86	21.82	10.62	3.69
	900	12.33	8.93	6.23	25.00	12.68	4.51
	1,200	12.31	9.60	7.29	24.49	13.37	4.88
	1,500	24.58	13.51	5.72	24.19	13.61	5.49
	1,800	38.72	14.81	6.37	29.03	14.45	6.23
	2,100	33.59	15.59	7.17	29.37	15.01	7.28
	2,400	28.24	15.92	7.96	22.75	15.09	8.19
	2,700	26.21	16.20	8.68	23.28	15.56	8.49
	3,000	24.00	16.21	9.01	23.44	14.37	8.53
	3,300	28.94	16.86	9.85	27.97	16.56	9.19
	3,600	28.58	17.36	10.51	26.18	17.03	10.27
	3,900	25.74	17.53	11.10	24.45	17.11	10.86
	4,200	25.45	17.60	11.31	24.28	17.23	10.87
	4,500	25.94	17.62	11.26	25.50	17.16	10.88
	4,800	28.23	17.96	11.52	25.81	17.57	11.37

Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Isend/Recv	300	2.67	2.64	2.44	2.36	2.31	2.11
	600	3.73	3.70	3.52	3.41	3.35	3.20
	900	4.32	4.28	4.10	4.00	3.95	3.74
	1,200	4.66	4.63	4.49	4.40	4.34	4.20
	1,500	4.93	4.84	4.65	4.74	4.66	4.55
	1,800	5.75	5.66	5.15	5.56	5.44	5.26
	2,100	6.42	6.30	5.95	6.11	6.00	5.32
	2,400	6.64	6.55	6.37	6.36	6.24	5.99
	2,700	6.79	6.71	6.54	6.51	6.41	6.25
	3,000	7.09	6.99	6.84	6.79	6.70	6.56
	3,300	7.61	7.52	7.28	7.37	7.26	7.07
	3,600	7.97	7.92	7.78	7.71	7.63	7.47
	3,900	8.00	7.95	7.72	7.72	7.68	7.55
	4,200	8.08	8.03	7.90	7.76	7.72	7.59
	4,500	8.19	8.10	7.94	7.92	7.86	7.73
	4,800	8.56	8.46	6.83	8.28	8.18	7.99
Isend/Irecv	300	7.15	5.24	3.85	11.54	5.98	2.78
	600	15.61	8.22	4.67	20.00	10.45	3.70
	900	22.77	11.89	5.03	24.66	12.99	4.51
	1,200	24.00	11.89	5.58	24.00	13.34	5.03
	1,500	24.39	13.95	5.62	23.44	13.67	5.55
	1,800	33.93	14.81	6.59	28.13	14.45	6.41
	2,100	33.33	15.49	7.33	28.19	15.06	7.25
	2,400	26.23	15.85	8.23	23.65	15.13	7.96
	2,700	25.96	16.21	8.64	23.58	15.56	8.40
	3,000	26.66	16.33	8.97	26.20	14.44	8.24
	3,300	29.20	16.77	9.61	23.91	16.50	10.03
	3,600	28.57	17.36	10.36	23.92	16.97	10.79
	3,900	25.57	17.43	11.24	24.00	17.09	11.03
	4,200	25.76	17.58	11.14	23.86	17.19	11.13
	4,500	25.94	17.58	11.14	24.06	17.12	11.19
	4,800	27.66	17.97	11.62	23.65	17.53	11.97

Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Ssend/Recv	300	2.54	2.42	2.24	1.08	1.05	0.97
	600	3.48	3.43	3.33	1.81	1.77	1.66
	900	4.06	4.03	3.92	2.37	2.33	2.21
	1,200	4.43	4.40	4.02	2.82	2.76	2.05
	1,500	4.76	4.68	4.55	3.15	3.08	2.91
	1,800	5.59	5.53	5.31	3.73	3.65	3.42
	2,100	6.34	6.27	6.12	4.18	4.10	3.74
	2,400	6.61	6.55	6.44	4.48	4.40	4.26
	2,700	6.76	6.70	6.58	4.70	4.64	4.49
	3,000	6.95	6.86	6.74	4.98	4.91	4.75
	3,300	7.47	7.34	7.14	5.41	5.33	5.17
	3,600	7.75	7.65	7.51	5.75	5.67	5.47
	3,900	7.77	7.68	7.54	5.93	5.86	5.60
	4,200	7.84	7.75	7.64	6.06	5.98	5.72
	4,500	8.09	7.98	7.80	6.22	6.13	5.92
	4,800	8.48	8.37	8.23	6.52	6.43	6.27
Ssend/Irecv	300	3.19	3.09	2.87	1.42	1.33	1.19
	600	5.04	4.91	4.62	2.30	2.21	2.04
	900	6.27	6.17	5.96	2.92	2.80	2.54
	1,200	7.12	6.96	6.70	3.37	3.26	3.07
	1,500	7.64	7.50	7.25	3.65	3.56	3.33
	1,800	9.00	8.77	8.43	4.34	4.21	3.95
	2,100	9.86	9.62	9.31	4.86	4.72	4.44
	2,400	10.50	10.30	10.02	5.16	5.04	4.84
	2,700	11.07	10.86	10.31	5.39	5.26	5.07
	3,000	11.63	11.48	11.15	5.69	5.52	5.20
	3,300	12.52	12.28	11.85	6.13	6.01	5.65
	3,600	13.14	12.99	12.61	6.50	6.36	6.11
	3,900	13.47	13.16	12.79	6.67	6.55	6.20
	4,200	13.70	13.53	13.23	6.76	6.65	6.39
	4,500	13.95	13.64	13.29	6.88	6.77	6.39
	4,800	14.46	14.32	12.40	7.20	7.10	6.95

Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Issend/Recv	300	2.47	2.42	2.30	1.09	1.06	0.96
	600	3.56	3.54	3.44	1.85	1.80	1.70
	900	4.16	4.14	4.07	2.41	2.37	2.24
	1,200	4.52	4.48	4.38	2.84	2.80	2.62
	1,500	4.70	4.64	4.48	3.15	3.10	2.92
	1,800	5.56	5.48	5.34	3.76	3.67	3.48
	2,100	6.29	6.21	6.07	4.20	4.11	3.66
	2,400	6.56	6.50	6.38	4.49	4.41	4.28
	2,700	6.72	6.66	6.58	4.74	4.65	4.50
	3,000	6.92	6.84	6.68	5.02	4.93	4.70
	3,300	7.42	7.31	7.17	5.43	5.36	5.14
	3,600	7.74	7.62	7.44	5.78	5.69	5.39
	3,900	7.80	7.67	7.53	5.95	5.88	5.64
	4,200	7.84	7.73	7.59	6.06	6.00	5.77
	4,500	8.06	7.96	7.81	6.23	6.15	5.94
	4,800	8.44	8.32	8.14	6.52	6.45	6.20
Issend/Irecv	300	3.43	3.13	2.79	2.39	2.10	1.67
	600	5.33	5.06	4.49	3.99	3.57	2.92
	900	6.64	6.06	5.84	5.17	4.67	3.91
	1,200	7.50	7.12	6.72	6.03	5.55	4.89
	1,500	7.92	7.58	7.37	6.83	6.04	3.54
	1,800	9.12	8.78	8.53	8.11	7.13	3.93
	2,100	9.77	9.64	9.19	9.35	7.94	4.49
	2,400	10.55	10.39	10.02	9.82	8.49	4.81
	2,700	11.34	10.99	10.67	10.04	9.20	8.08
	3,000	12.17	11.65	11.21	10.56	9.67	8.00
	3,300	12.67	12.38	11.93	11.62	10.53	5.86
	3,600	13.14	13.01	12.57	12.41	11.00	6.15
	3,900	13.76	13.51	12.96	12.62	11.53	6.23
	4,200	14.12	13.96	13.46	12.65	11.83	6.50
	4,500	14.29	13.83	13.41	12.93	12.02	10.65
	4,800	14.52	14.25	13.78	13.83	12.54	6.79

Medium Messages

Construct	Message Size	LAM Best	LAM Avg	LAM Worst	MPICH Best	MPICH Avg	MPICH Worst
Send/Recv	5,000	8.73	8.66	7.39	8.39	8.27	7.49
	10,000	9.89	9.86	9.62	9.51	9.45	9.12
	15,000	10.58	10.54	10.46	10.14	10.10	9.93
	20,000	10.74	10.71	10.67	10.50	10.46	10.25
	25,000	10.97	10.95	10.88	10.69	10.65	10.58
	30,000	11.10	11.09	11.02	10.75	10.72	10.65
	35,000	11.16	11.14	11.11	10.83	10.80	10.76
	40,000	11.24	11.22	11.20	10.94	10.91	10.78
	45,000	11.26	11.24	11.22	10.91	10.89	10.84
	50,000	11.36	11.35	11.33	10.98	10.95	10.78
	55,000	11.33	11.31	11.29	10.95	10.92	10.89
	60,000	11.41	11.40	11.38	11.03	11.01	10.97
	65,000	11.42	11.41	11.39	10.99	10.97	10.88
	70,000	11.24	11.22	11.20	11.02	11.00	10.98
	75,000	11.24	11.23	11.21	11.03	11.01	10.93
	80,000	11.28	11.27	11.25	11.05	11.03	11.00
	85,000	11.31	11.30	11.29	11.04	11.02	11.01
	90,000	11.32	11.31	11.30	11.04	11.03	10.99
	95,000	11.35	11.34	11.33	11.06	11.05	11.01
	100,000	11.35	11.34	11.27	11.03	11.01	10.90
Send/Irecv	5,000	44.62	18.29	9.89	43.86	17.88	9.50
	10,000	23.84	19.72	16.04	23.64	19.21	14.87
	15,000	24.33	20.64	17.74	23.53	19.76	16.46
	20,000	23.05	20.85	17.56	21.31	20.31	19.37
	25,000	23.53	21.25	19.06	20.33	19.47	18.47
	30,000	22.26	21.54	20.01	20.15	17.83	17.13
	35,000	22.91	21.27	11.43	18.05	16.47	16.27
	40,000	22.98	19.64	11.49	17.06	15.64	15.28
	45,000	22.91	21.45	11.44	16.07	15.25	14.87
	50,000	22.65	21.76	11.51	15.70	14.77	14.40
	55,000	22.74	21.42	11.51	14.42	14.31	13.46
	60,000	23.39	19.77	11.46	14.83	14.15	13.77
	65,000	22.53	19.96	11.96	15.08	13.91	12.79
	70,000	11.33	11.30	9.80	13.80	13.67	13.29
	75,000	11.33	11.31	11.29	13.84	13.60	13.44
	80,000	11.36	11.35	11.33	13.52	13.46	12.96
	85,000	11.39	11.38	11.36	13.68	13.29	12.92
	90,000	11.39	11.38	11.37	13.13	13.09	12.86
	95,000	11.42	11.41	11.39	13.28	13.14	13.10
	100,000	11.42	11.41	11.33	13.63	12.97	12.28

Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Isend/Recv	5,000	8.70	8.63	8.33	8.38	8.28	7.61
	10,000	9.84	9.81	9.72	9.51	9.46	9.27
	15,000	10.53	10.49	10.41	10.14	10.09	9.93
	20,000	10.70	10.66	10.59	10.51	10.45	10.37
	25,000	10.91	10.89	10.85	10.67	10.65	10.61
	30,000	11.07	11.05	11.01	10.75	10.72	10.69
	35,000	11.12	11.10	11.06	10.83	10.80	10.65
	40,000	11.21	11.19	11.16	10.94	10.91	10.86
	45,000	11.22	11.20	11.17	10.91	10.89	10.84
	50,000	11.33	11.32	11.30	10.97	10.95	10.86
	55,000	11.30	11.28	10.99	10.93	10.91	10.62
	60,000	11.38	11.37	11.33	11.02	11.00	10.84
	65,000	11.39	11.38	11.23	10.99	10.97	10.94
	70,000	11.19	11.17	11.15	11.01	11.00	10.97
	75,000	11.19	11.17	11.13	11.03	11.01	10.99
	80,000	11.22	11.21	11.20	11.04	11.02	11.00
	85,000	11.26	11.25	11.23	11.03	11.01	10.93
	90,000	11.27	11.26	11.24	11.04	11.02	10.96
	95,000	11.30	11.29	11.27	11.06	11.04	11.02
	100,000	11.31	11.29	11.25	11.02	11.01	10.97
Isend/Irecv	5,000	48.55	18.34	9.76	23.92	17.62	3.89
	10,000	23.56	19.71	16.08	23.56	19.16	14.93
	15,000	24.19	20.65	17.64	22.44	19.72	17.49
	20,000	23.16	20.92	18.64	22.79	20.31	17.48
	25,000	23.73	21.27	19.13	20.33	19.45	18.59
	30,000	23.38	21.56	19.91	18.61	17.82	17.35
	35,000	23.22	20.97	11.43	18.06	16.46	16.04
	40,000	23.18	20.21	11.49	17.03	15.62	15.31
	45,000	22.77	21.38	11.83	15.85	15.25	15.06
	50,000	22.79	21.93	21.19	15.81	14.77	14.46
	55,000	22.86	21.58	11.52	14.40	14.31	13.44
	60,000	23.38	19.49	11.46	14.79	14.17	13.77
	65,000	23.49	20.56	11.99	15.05	13.90	12.82
	70,000	22.26	17.90	8.19	14.15	13.67	13.53
	75,000	22.12	17.92	8.74	13.87	13.60	13.43
	80,000	22.13	18.48	9.15	13.53	13.46	12.95
	85,000	22.05	18.34	9.30	13.75	13.28	12.90
	90,000	22.15	19.13	9.58	13.13	13.08	12.89
	95,000	22.31	18.65	10.04	13.34	13.14	13.09
	100,000	22.07	19.79	10.15	13.63	12.97	12.28

Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Ssend/Recv	5,000	8.66	8.59	8.21	6.70	6.56	2.91
	10,000	9.73	9.65	9.51	8.37	8.32	8.13
	15,000	10.46	10.39	10.31	9.29	9.24	9.03
	20,000	10.70	10.67	10.63	9.65	9.59	9.46
	25,000	10.91	10.88	10.82	10.01	9.94	9.81
	30,000	11.00	10.98	10.93	10.19	10.16	10.05
	35,000	11.09	11.06	11.03	10.30	10.25	10.15
	40,000	11.22	11.20	11.17	10.43	10.38	10.32
	45,000	11.21	11.19	11.16	10.48	10.43	10.36
	50,000	11.29	11.28	11.24	10.59	10.56	10.49
	55,000	11.31	11.29	11.26	10.56	10.52	10.47
	60,000	11.38	11.37	11.34	10.65	10.61	10.54
	65,000	11.35	11.35	11.32	10.67	10.64	10.57
	70,000	11.22	11.21	11.18	10.71	10.68	10.61
	75,000	11.23	11.21	11.19	10.73	10.69	10.58
	80,000	11.26	11.25	11.23	10.75	10.72	10.68
	85,000	11.30	11.29	11.27	10.78	10.75	10.69
	90,000	11.30	11.29	11.27	10.78	10.75	10.70
	95,000	11.33	11.32	11.31	10.80	10.77	10.74
	100,000	11.34	11.32	11.26	10.80	10.77	10.71
Ssend/Irecv	5,000	14.68	14.52	13.42	7.36	7.22	6.62
	10,000	17.76	17.58	17.23	8.94	8.86	8.61
	15,000	19.19	19.00	18.60	9.76	9.68	9.49
	20,000	19.78	19.64	19.34	10.08	9.99	9.89
	25,000	20.26	20.06	17.04	10.37	10.31	9.79
	30,000	20.53	20.42	20.25	10.57	10.50	10.37
	35,000	20.71	20.57	19.79	10.64	10.59	10.49
	40,000	20.87	20.21	11.83	10.76	10.70	10.61
	45,000	20.88	20.52	11.78	10.84	10.76	10.69
	50,000	21.02	20.89	20.60	10.92	10.88	10.79
	55,000	20.92	20.40	18.33	10.88	10.84	10.75
	60,000	21.05	20.55	19.93	10.97	10.93	10.84
	65,000	20.68	19.64	11.74	10.99	10.95	10.87
	70,000	11.32	11.31	11.29	11.04	10.99	10.90
	75,000	11.32	11.30	11.29	11.05	10.99	10.93
	80,000	11.36	11.35	11.32	11.08	11.03	10.97
	85,000	11.38	11.37	11.35	11.10	11.06	10.99
	90,000	11.39	11.38	11.36	11.11	11.06	11.00
	95,000	11.41	11.40	11.39	11.12	11.08	11.03
	100,000	11.42	11.40	11.34	11.11	11.08	11.01

Construct	Message Size	LAM Best	LAM Avg	LAM Worst	MPICH Best	MPICH Avg	MPICH Worst
Issend/Recv	5,000	8.62	8.54	8.06	6.70	6.51	3.54
	10,000	9.69	9.60	9.47	8.37	8.31	8.12
	15,000	10.40	10.35	10.28	9.29	9.23	8.52
	20,000	10.68	10.64	10.57	9.64	9.58	9.27
	25,000	10.88	10.85	10.79	9.99	9.94	9.84
	30,000	10.98	10.96	10.93	10.19	10.15	10.05
	35,000	11.06	11.04	11.02	10.29	10.25	10.18
	40,000	11.20	11.17	11.15	10.41	10.37	10.07
	45,000	11.18	11.16	11.14	10.48	10.44	10.34
	50,000	11.28	11.26	11.23	10.58	10.55	10.45
	55,000	11.28	11.27	11.25	10.55	10.52	10.46
	60,000	11.36	11.35	11.33	10.65	10.61	10.55
	65,000	11.34	11.33	11.31	10.68	10.64	10.57
	70,000	11.19	11.17	11.15	10.69	10.67	10.61
	75,000	11.19	11.17	11.15	10.71	10.68	10.60
	80,000	11.23	11.22	11.20	10.76	10.72	10.65
	85,000	11.26	11.25	11.23	10.78	10.75	10.70
	90,000	11.27	11.26	11.25	10.78	10.74	10.70
	95,000	11.30	11.29	11.28	10.80	10.77	10.70
	100,000	11.31	11.30	11.28	10.79	10.76	10.71
Issend/Irecv	5,000	14.64	14.43	13.40	14.14	12.72	3.76
	10,000	18.12	17.94	17.54	16.95	16.33	15.23
	15,000	19.48	19.23	19.01	18.82	17.97	9.44
	20,000	20.06	19.87	19.64	19.18	18.67	17.31
	25,000	20.56	20.38	20.21	18.88	17.76	9.73
	30,000	20.99	20.81	20.68	18.67	16.00	13.25
	35,000	21.17	21.02	20.33	17.48	15.13	13.42
	40,000	21.37	21.18	21.00	16.26	14.65	13.12
	45,000	21.43	21.27	21.09	15.37	14.11	12.61
	50,000	21.66	21.49	21.28	15.79	13.89	12.27
	55,000	21.71	20.75	11.96	13.84	13.50	12.89
	60,000	21.80	20.96	11.90	14.27	13.41	12.94
	65,000	21.63	19.51	12.09	13.43	13.16	12.83
	70,000	22.17	18.02	8.39	13.71	13.06	12.78
	75,000	22.11	18.10	8.57	13.87	12.95	12.14
	80,000	22.15	17.89	9.16	13.04	12.83	12.48
	85,000	22.20	18.40	9.45	13.55	12.74	11.99
	90,000	22.17	19.51	9.77	12.81	12.64	12.35
	95,000	22.20	18.43	10.04	13.06	12.60	12.34
	100,000	22.12	19.68	10.35	12.62	12.53	12.25

Long Messages

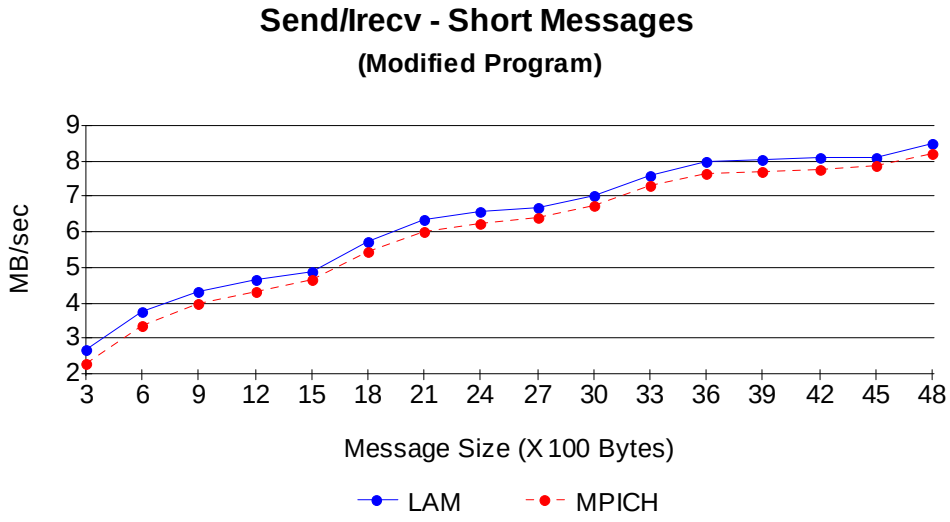
Construct	Message Size	LAM Best	LAM Avg	LAM Worst	MPICH Best	MPICH Avg	MPICH Worst
Send/Recv	200k	11.55	11.55	11.54	10.93	10.91	10.85
	300k	11.62	11.62	11.62	11.01	11.00	10.99
	400k	11.66	11.66	11.65	11.05	11.05	10.94
	500k	11.68	11.68	11.67	11.08	11.08	11.06
	600k	11.70	11.70	11.68	11.10	11.10	11.08
	700k	11.71	11.71	11.70	11.11	11.11	11.10
	800k	11.72	11.72	11.71	11.12	11.11	11.10
	900k	11.72	11.72	11.72	11.12	11.12	11.11
	1,000k	11.73	11.73	11.72	11.13	11.12	11.11
	1,100k	11.73	11.73	11.73	11.13	11.13	11.12
	1,200k	11.73	11.73	11.73	11.13	11.13	11.12
	1,300k	11.73	11.73	11.73	11.14	11.13	11.13
	1,400k	11.74	11.74	11.74	11.14	11.13	11.13
	1,500k	11.74	11.74	11.74	11.14	11.14	11.13
	1,600k	11.74	11.74	11.74	11.14	11.14	11.13
	1,700k	11.74	11.74	11.74	11.14	11.14	11.13
	1,800k	11.74	11.74	11.74	11.14	11.14	11.13
	1,900k	11.75	11.74	11.70	11.14	11.14	11.13
	2,000k	11.75	11.75	11.75	11.15	11.14	10.88
Send/Irecv	200k	11.59	11.58	11.57	11.29	11.27	10.73
	300k	11.65	11.64	11.64	11.36	11.35	11.33
	400k	11.68	11.68	11.67	11.38	11.37	11.34
	500k	11.70	11.70	11.69	11.40	11.40	11.37
	600k	11.71	11.71	11.71	11.42	11.41	11.39
	700k	11.72	11.72	11.69	11.43	11.42	11.40
	800k	11.73	11.72	11.72	11.43	11.43	11.41
	900k	11.73	11.73	11.73	11.44	11.43	11.43
	1,000k	11.73	11.73	11.73	11.45	11.44	11.43
	1,100k	11.74	11.74	11.73	11.45	11.44	11.44
	1,200k	11.74	11.74	11.74	11.45	11.45	11.44
	1,300k	11.74	11.74	11.74	11.45	11.45	11.44
	1,400k	11.74	11.74	11.74	11.46	11.45	11.37
	1,500k	11.74	11.74	11.74	11.46	11.45	11.42
	1,600k	11.74	11.74	11.74	11.46	11.45	11.44
	1,700k	11.75	11.75	11.74	11.46	11.45	11.45
	1,800k	11.75	11.75	11.74	11.46	11.46	11.45
	1,900k	11.75	11.75	11.75	11.46	11.46	11.45
	2,000k	11.75	11.75	11.75	11.46	11.46	11.46

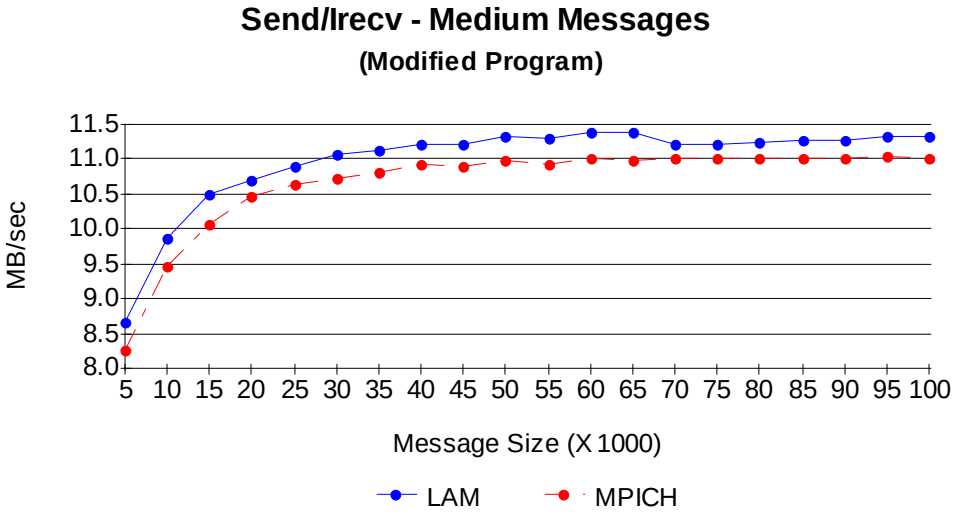
Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Isend/Recv	200k	11.53	11.52	11.51	10.92	10.91	10.86
	300k	11.61	11.60	11.60	11.01	11.00	10.96
	400k	11.65	11.64	11.64	11.05	11.04	11.03
	500k	11.67	11.67	11.67	11.09	11.08	11.07
	600k	11.69	11.69	11.69	11.10	11.10	11.08
	700k	11.70	11.70	11.69	11.11	11.10	11.09
	800k	11.71	11.71	11.71	11.12	11.11	11.10
	900k	11.72	11.71	11.71	11.12	11.12	11.11
	1,000k	11.72	11.72	11.72	11.13	11.12	11.11
	1,100k	11.72	11.72	11.70	11.13	11.12	11.12
	1,200k	11.73	11.73	11.73	11.13	11.13	11.12
	1,300k	11.73	11.73	11.73	11.14	11.13	11.13
	1,400k	11.73	11.73	11.73	11.14	11.13	11.13
	1,500k	11.74	11.73	11.73	11.14	11.14	11.13
	1,600k	11.74	11.74	11.74	11.14	11.14	11.13
	1,700k	11.74	11.74	11.73	11.14	11.14	11.13
	1,800k	11.74	11.74	11.74	11.14	11.14	11.13
	1,900k	11.74	11.74	11.74	11.15	11.14	11.13
	2,000k	11.74	11.74	11.74	11.15	11.14	11.14
Isend/Irecv	200k	22.64	18.27	9.84	12.19	11.89	11.62
	300k	22.74	18.61	10.76	11.79	11.74	11.68
	400k	22.80	18.53	10.44	11.69	11.66	11.61
	500k	22.84	18.20	10.65	11.72	11.63	11.60
	600k	22.84	18.48	10.81	11.62	11.60	11.57
	700k	22.85	18.16	10.94	11.64	11.58	11.51
	800k	22.82	18.13	11.09	11.58	11.56	11.54
	900k	22.88	17.84	7.10	11.56	11.55	11.52
	1,000k	22.95	18.21	11.18	11.58	11.53	11.48
	1,100k	22.92	17.75	11.23	11.53	11.52	11.51
	1,200k	23.08	17.89	10.16	11.53	11.51	11.50
	1,300k	22.93	17.75	11.30	11.55	11.51	11.50
	1,400k	22.93	17.71	11.34	11.51	11.50	11.49
	1,500k	22.96	17.49	11.31	11.53	11.50	11.47
	1,600k	23.05	17.46	11.40	11.51	11.50	11.48
	1,700k	23.01	17.62	11.42	11.50	11.49	11.48
	1,800k	23.00	17.33	11.43	11.52	11.49	11.46
	1,900k	22.94	17.71	11.45	11.49	11.49	11.48
	2,000k	22.90	17.49	11.47	11.55	11.48	11.41

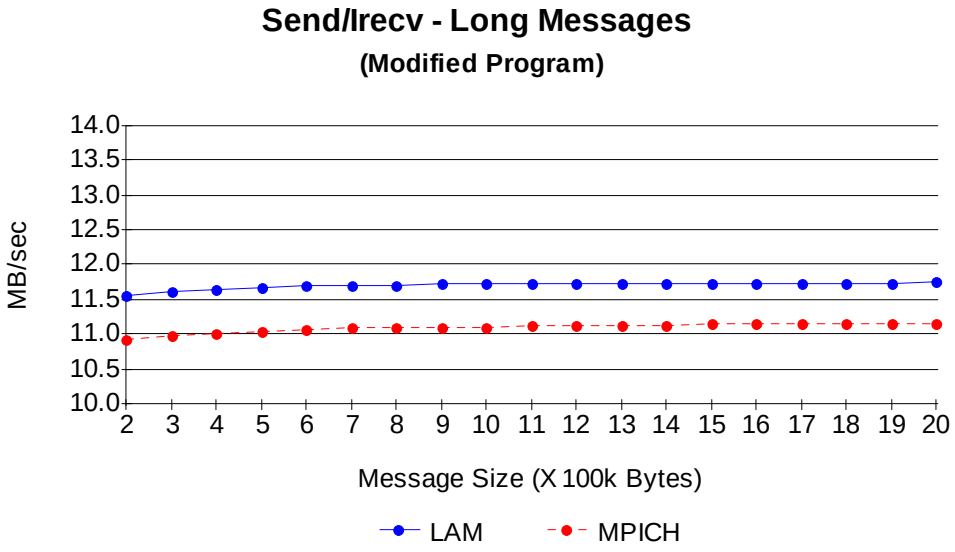
Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Ssend/Recv	200k	11.54	11.54	11.51	10.93	10.91	10.89
	300k	11.62	11.61	11.61	11.01	11.00	10.99
	400k	11.66	11.65	11.65	11.05	11.04	11.01
	500k	11.68	11.68	11.67	11.08	11.08	11.06
	600k	11.70	11.69	11.69	11.10	11.10	11.07
	700k	11.71	11.70	11.70	11.11	11.10	11.09
	800k	11.72	11.71	11.71	11.12	11.11	11.10
	900k	11.72	11.72	11.72	11.12	11.12	11.11
	1,000k	11.73	11.72	11.72	11.13	11.12	11.11
	1,100k	11.73	11.73	11.73	11.13	11.12	11.11
	1,200k	11.73	11.73	11.73	11.13	11.13	11.12
	1,300k	11.73	11.73	11.73	11.14	11.13	11.13
	1,400k	11.74	11.74	11.74	11.14	11.14	11.13
	1,500k	11.74	11.74	11.74	11.14	11.14	11.13
	1,600k	11.74	11.74	11.74	11.14	11.14	11.13
	1,700k	11.74	11.74	11.74	11.14	11.14	11.13
	1,800k	11.74	11.74	11.74	11.14	11.14	11.13
	1,900k	11.74	11.74	11.74	11.14	11.14	11.14
	2,000k	11.75	11.75	11.75	11.15	11.15	11.14
Ssend/Irecv	200k	11.58	11.58	11.57	11.30	11.27	11.03
	300k	11.65	11.64	11.64	11.36	11.35	11.33
	400k	11.68	11.67	11.65	11.38	11.37	11.34
	500k	11.70	11.69	11.69	11.41	11.39	11.39
	600k	11.71	11.71	11.71	11.42	11.41	11.39
	700k	11.72	11.72	11.71	11.43	11.42	11.41
	800k	11.72	11.72	11.72	11.44	11.43	11.41
	900k	11.73	11.73	11.73	11.44	11.43	11.43
	1,000k	11.73	11.73	11.73	11.44	11.44	11.43
	1,100k	11.74	11.73	11.73	11.45	11.44	11.43
	1,200k	11.74	11.74	11.68	11.45	11.45	11.44
	1,300k	11.74	11.74	11.74	11.45	11.45	11.44
	1,400k	11.74	11.74	11.74	11.45	11.45	11.45
	1,500k	11.74	11.74	11.74	11.46	11.45	11.45
	1,600k	11.74	11.74	11.74	11.46	11.45	11.42
	1,700k	11.75	11.74	11.74	11.46	11.45	11.45
	1,800k	11.75	11.75	11.74	11.46	11.46	11.45
	1,900k	11.75	11.75	11.75	11.46	11.46	11.45
	2,000k	11.75	11.75	11.75	11.46	11.46	11.46

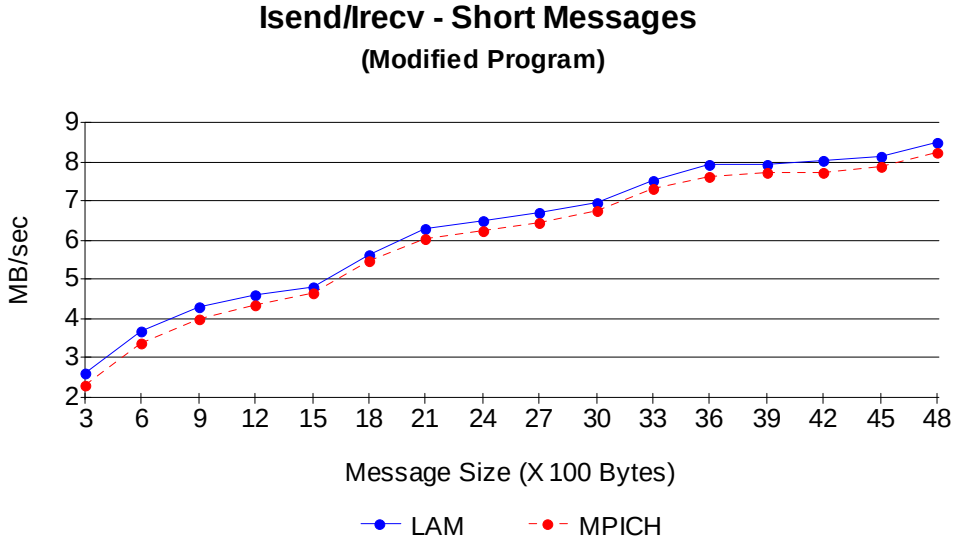
Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Issend/Recv	200k	11.53	11.52	11.42	10.93	10.91	10.86
	300k	11.61	11.60	11.60	11.01	11.00	10.96
	400k	11.65	11.65	11.64	11.05	11.04	11.02
	500k	11.67	11.67	11.67	11.08	11.08	11.07
	600k	11.69	11.69	11.69	11.10	11.10	11.08
	700k	11.70	11.70	11.70	11.11	11.10	11.09
	800k	11.71	11.71	11.71	11.12	11.11	11.10
	900k	11.72	11.71	11.71	11.12	11.12	11.11
	1,000k	11.72	11.72	11.72	11.13	11.12	11.11
	1,100k	11.72	11.72	11.65	11.13	11.13	11.12
	1,200k	11.73	11.73	11.73	11.13	11.13	11.12
	1,300k	11.73	11.73	11.73	11.14	11.13	11.13
	1,400k	11.73	11.73	11.73	11.14	11.13	11.13
	1,500k	11.74	11.73	11.73	11.14	11.14	11.13
	1,600k	11.74	11.74	11.73	11.14	11.14	11.13
	1,700k	11.74	11.74	11.74	11.14	11.14	11.04
	1,800k	11.74	11.74	11.74	11.14	11.14	11.13
	1,900k	11.74	11.74	11.74	11.14	11.14	11.09
	2,000k	11.74	11.74	11.74	11.15	11.12	8.44
Issend/Irecv	200k	22.70	17.94	9.59	12.21	11.90	11.61
	300k	22.72	18.98	10.71	11.79	11.74	11.67
	400k	22.80	18.59	10.56	11.69	11.67	11.62
	500k	22.77	18.05	10.65	11.72	11.63	11.60
	600k	22.90	18.55	10.92	11.62	11.60	11.57
	700k	22.86	18.49	10.93	11.65	11.58	11.51
	800k	22.86	18.19	9.74	11.58	11.56	11.54
	900k	22.84	18.19	11.11	11.56	11.55	11.53
	1,000k	22.86	18.06	11.21	11.58	11.53	11.48
	1,100k	22.92	17.68	11.23	11.54	11.52	11.50
	1,200k	22.94	18.00	11.27	11.53	11.51	11.50
	1,300k	22.91	17.78	11.30	11.55	11.51	11.50
	1,400k	22.88	18.18	11.34	11.51	11.50	11.49
	1,500k	22.94	17.69	11.37	11.54	11.50	11.47
	1,600k	22.97	17.54	11.39	11.51	11.50	11.48
	1,700k	22.96	17.47	11.42	11.50	11.49	11.48
	1,800k	22.90	18.03	11.43	11.52	11.49	11.46
	1,900k	22.95	17.77	11.46	11.49	11.49	11.48
	2,000k	22.98	17.58	11.43	11.49	11.49	11.48

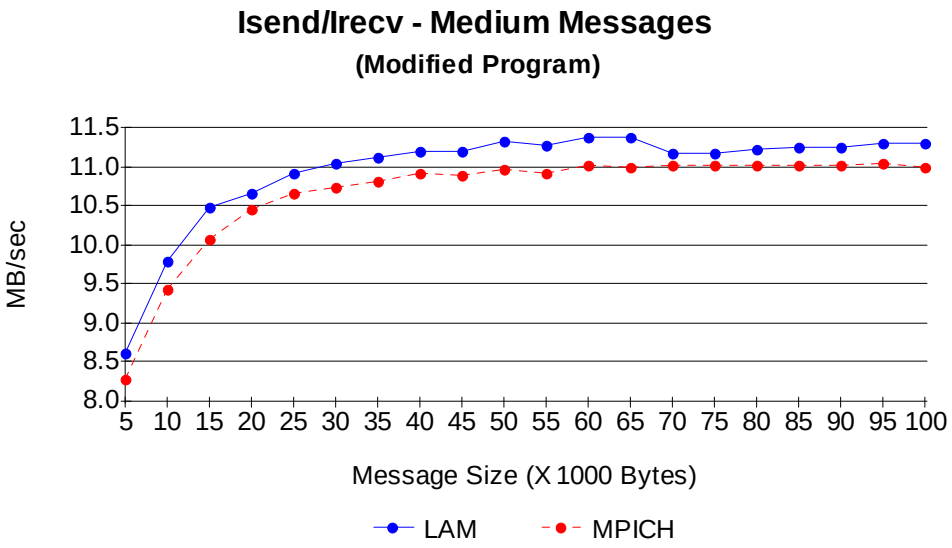
Appendix H: Bandwidth Graphs - Modified Non-Blocking Receives

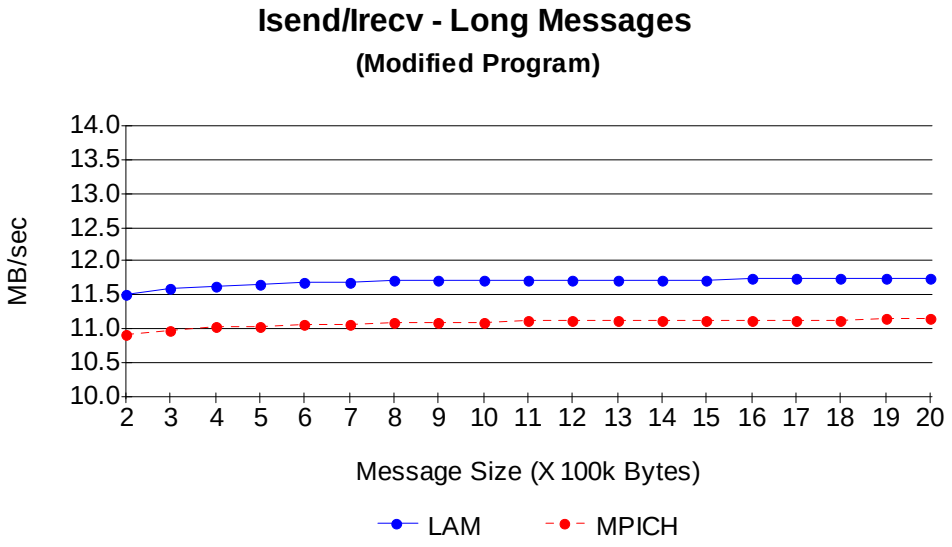


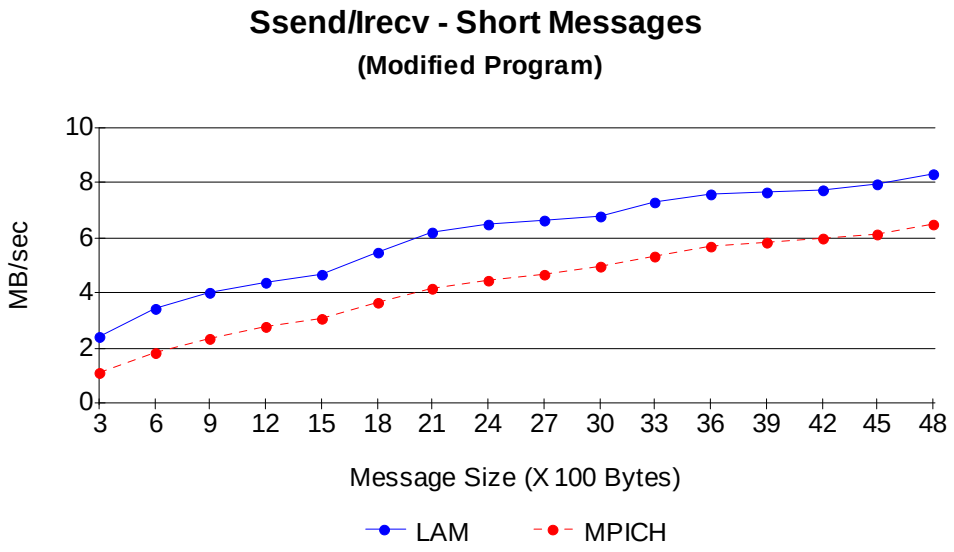


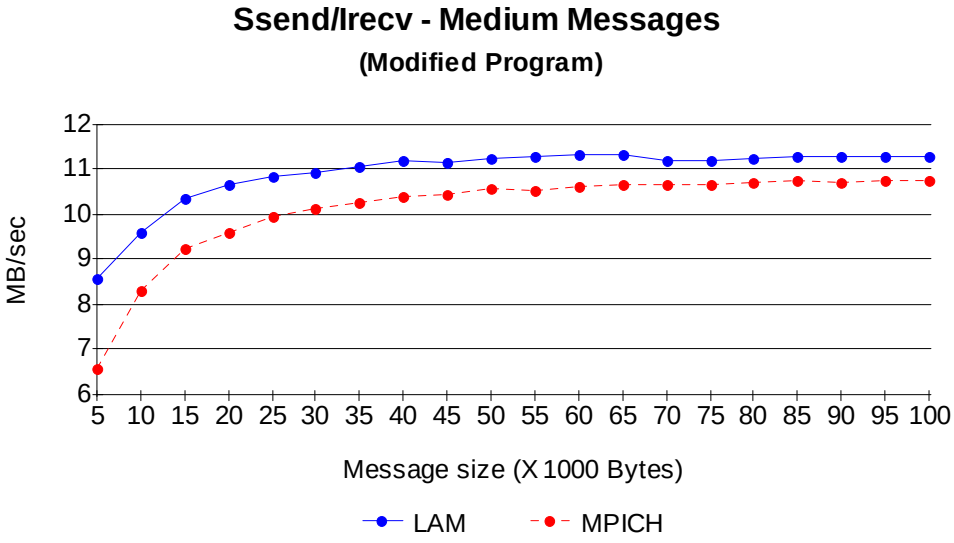

Send/Irecv - Long Messages
(Modified Program)
MB/sec
14.0
13.5
13.0
12.5
12.0
11.5
11.0
10.5
10.0
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 100k Bytes)
LAM
MPICH

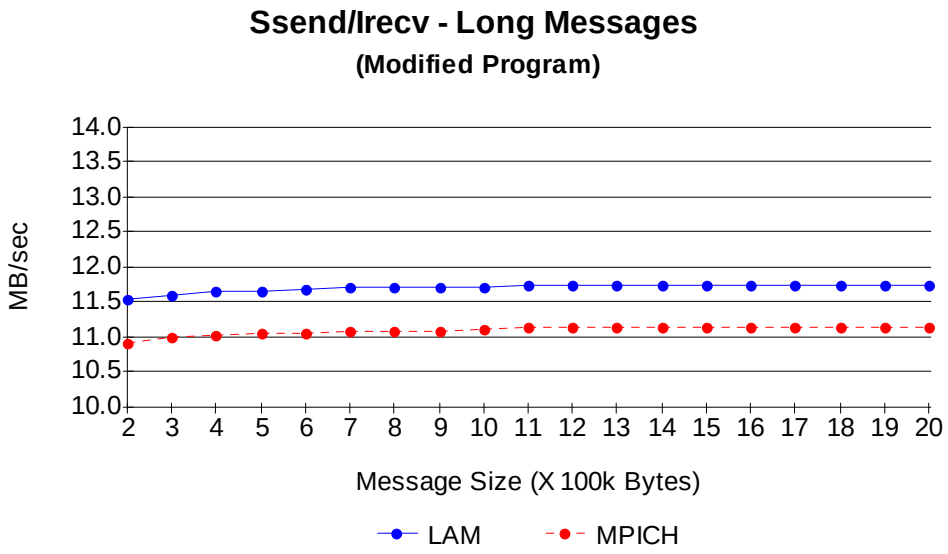

Isend/Irecv - Short Messages
(Modified Program)
MB/sec
9
8
7
6
5
4
3
2
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH

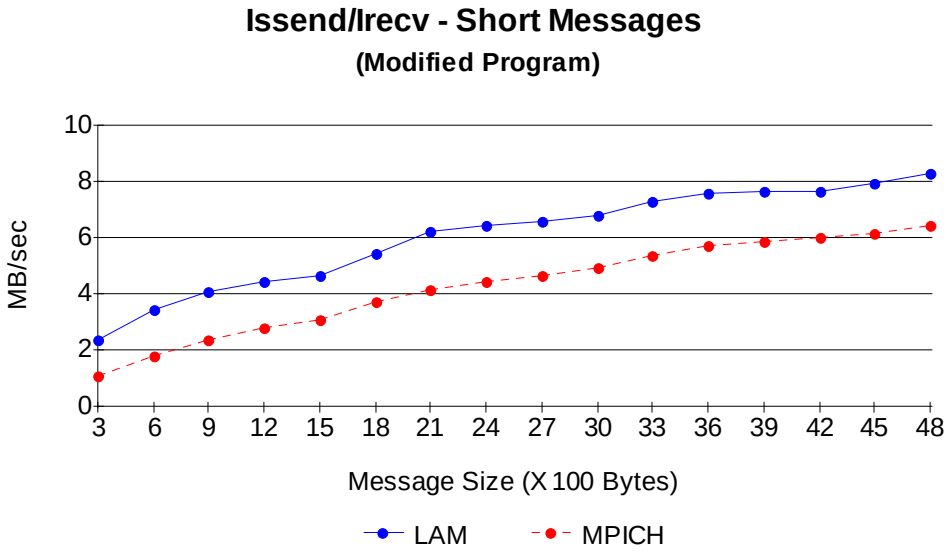

Isend/Irecv - Medium Messages
(Modified Program)
MB/sec
11.5
11.0
10.5
10.0
9.5
9.0
8.5
8.0
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message Size (X 1000 Bytes)
LAM
MPICH

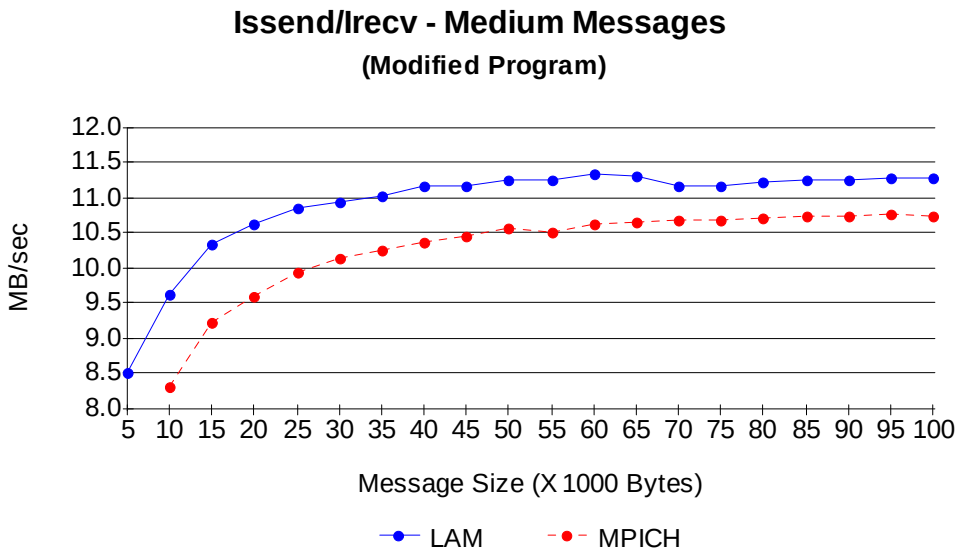

Isend/Irecv - Long Messages
(Modified Program)
MB/sec
14.0
13.5
13.0
12.5
12.0
11.5
11.0
10.5
10.0
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 100k Bytes)
LAM
MPICH

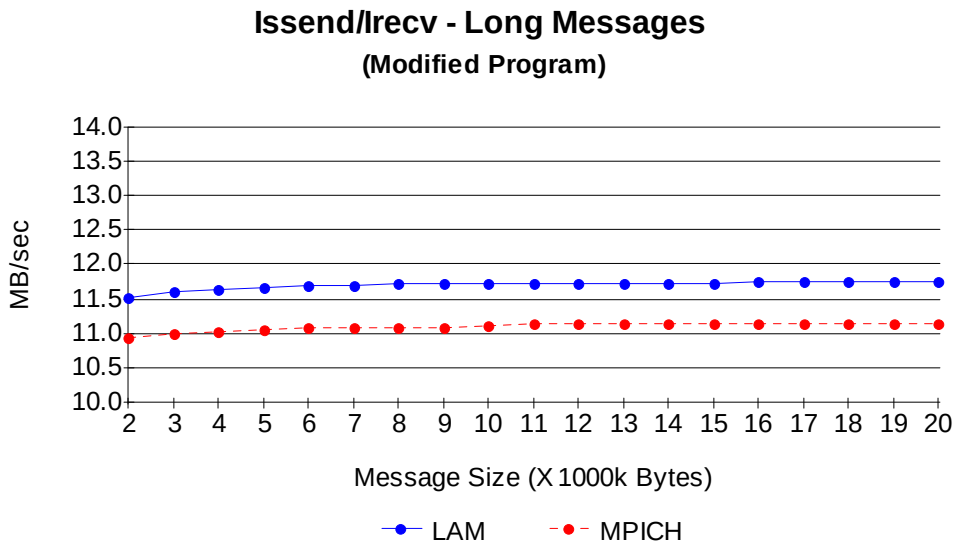

Ssend/Irecv - Short Messages
(Modified Program)
MB/sec
0
2
4
6
8
10
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH


Ssend/Irecv - Medium Messages
(Modified Program)
MB/sec
6
7
8
9
10
11
12
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message size (X 1000 Bytes)
LAM
MPICH


Ssend/Irecv - Long Messages
(Modified Program)
MB/sec
14.0
13.5
13.0
12.5
12.0
11.5
11.0
10.5
10.0
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 100k Bytes)
LAM
MPICH


Issend/Irecv - Short Messages
(Modified Program)
MB/sec
10
8
6
4
2
0
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
Message Size (X 100 Bytes)
LAM
MPICH


Issend/Irecv - Medium Messages
(Modified Program)
MB/sec
12.0
11.5
11.0
10.5
10.0
9.5
9.0
8.5
8.0
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Message Size (X 1000 Bytes)
LAM
MPICH


Issend/Irecv - Long Messages
(Modified Program)
MB/sec
14.0
13.5
13.0
12.5
12.0
11.5
11.0
10.5
10.0
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Message Size (X 1000k Bytes)
LAM
MPICH

Appendix I: Bandwidth Raw Data for Modified Non-Blocking Receives

Short Messages

Construct	Message Size	LAM Best	LAM Avg	LAM Worst	MPICH Best	MPICH Avg	MPICH Worst
Send/Irecv	300	2.70	2.66	2.07	2.37	2.31	2.16
	600	3.77	3.74	3.58	3.40	3.37	3.17
	900	4.34	4.29	4.03	4.03	3.97	3.81
	1,200	4.69	4.64	4.23	4.39	4.34	4.16
	1,500	4.96	4.90	4.68	4.74	4.67	4.57
	1,800	5.77	5.70	5.56	5.58	5.46	5.25
	2,100	6.46	6.36	5.83	6.13	6.02	5.78
	2,400	6.65	6.55	6.41	6.37	6.26	6.05
	2,700	6.80	6.71	6.51	6.51	6.43	6.30
	3,000	7.13	7.03	6.82	6.81	6.73	6.55
	3,300	7.67	7.57	7.36	7.38	7.29	7.12
	3,600	8.02	7.97	7.85	7.73	7.64	6.74
	3,900	8.09	8.02	7.81	7.75	7.70	7.59
	4,200	8.12	8.10	8.05	7.78	7.74	7.57
	4,500	8.23	8.12	7.98	7.95	7.89	7.73
	4,800	8.60	8.52	8.33	8.28	8.21	8.02
Isend/Irecv	300	2.63	2.61	2.49	2.38	2.31	2.14
	600	3.75	3.68	3.39	3.41	3.36	3.18
	900	4.33	4.31	4.22	4.00	3.96	3.81
	1,200	4.67	4.63	4.52	4.39	4.32	3.92
	1,500	4.85	4.80	4.62	4.72	4.66	4.51
	1,800	5.73	5.64	5.45	5.58	5.45	5.28
	2,100	6.40	6.31	6.16	6.12	6.02	5.73
	2,400	6.59	6.51	6.27	6.35	6.24	6.07
	2,700	6.74	6.68	6.52	6.55	6.43	6.24
	3,000	7.08	6.97	6.67	6.81	6.73	6.59
	3,300	7.60	7.51	7.35	7.37	7.29	7.14
	3,600	7.95	7.91	7.77	7.73	7.66	7.48
	3,900	8.00	7.94	7.73	7.77	7.71	7.55
	4,200	8.05	8.02	7.91	7.81	7.76	7.57
	4,500	8.18	8.14	8.05	7.94	7.88	7.73
	4,800	8.54	8.47	8.36	8.30	8.20	7.97

Construct	Message Size	LAM Best	LAM Avg	LAM Worst	MPICH Best	MPICH Avg	MPICH Worst
Ssend/Irecv	300	2.49	2.38	2.23	1.09	1.06	0.95
	600	3.49	3.46	3.35	1.85	1.81	1.70
	900	4.08	4.01	3.89	2.41	2.37	2.24
	1,200	4.40	4.36	4.29	2.85	2.79	2.65
	1,500	4.72	4.65	4.53	3.16	3.10	2.94
	1,800	5.55	5.48	5.35	3.76	3.67	3.50
	2,100	6.29	6.21	6.03	4.23	4.14	3.97
	2,400	6.57	6.51	6.34	4.49	4.42	4.29
	2,700	6.72	6.64	6.49	4.73	4.65	4.49
	3,000	6.90	6.78	6.54	5.00	4.94	4.78
	3,300	7.38	7.29	7.10	5.42	5.36	5.13
	3,600	7.71	7.60	7.45	5.75	5.68	5.47
	3,900	7.73	7.65	7.49	5.95	5.87	5.63
	4,200	7.79	7.71	7.59	6.07	6.01	5.77
	4,500	8.01	7.94	7.83	6.22	6.16	5.93
	4,800	8.44	8.31	8.14	6.55	6.46	6.28
Issend/Irecv	300	2.42	2.34	2.21	1.09	1.06	0.86
	600	3.46	3.42	3.28	1.84	1.80	1.65
	900	4.14	4.05	3.97	2.41	2.36	2.22
	1,200	4.49	4.42	4.35	2.82	2.78	2.63
	1,500	4.70	4.61	4.06	3.16	3.10	2.94
	1,800	5.52	5.45	5.32	3.74	3.68	3.50
	2,100	6.26	6.20	6.09	4.23	4.14	3.98
	2,400	6.53	6.45	6.31	4.49	4.42	4.29
	2,700	6.63	6.58	6.48	4.72	4.66	4.52
	3,000	6.84	6.76	6.59	5.01	4.93	4.78
	3,300	7.34	7.27	7.11	5.44	5.36	5.20
	3,600	7.65	7.57	7.29	5.77	5.69	5.47
	3,900	7.71	7.61	7.51	5.95	5.88	5.62
	4,200	7.69	7.66	7.53	6.08	6.01	5.77
	4,500	7.99	7.92	7.80	6.23	6.15	5.96
	4,800	8.42	8.32	8.12	6.54	6.46	6.21

Medium Messages

Construct	Message Size	LAM Best	LAM Avg	LAM Worst	MPICH Best	MPICH Avg	MPICH Worst
Send/Irecv	5,000	8.74	8.67	8.40	8.38	8.26	7.56
	10,000	9.89	9.86	9.79	9.50	9.45	9.23
	15,000	10.56	10.51	10.45	10.11	10.06	9.94
	20,000	10.72	10.69	10.66	10.50	10.46	10.35
	25,000	10.94	10.91	10.86	10.67	10.65	10.52
	30,000	11.09	11.07	11.05	10.74	10.72	10.56
	35,000	11.14	11.12	11.07	10.84	10.81	10.65
	40,000	11.23	11.21	11.18	10.95	10.92	10.89
	45,000	11.24	11.22	11.20	10.92	10.90	10.88
	50,000	11.35	11.34	11.31	10.99	10.97	10.93
	55,000	11.31	11.30	11.28	10.95	10.93	10.91
	60,000	11.40	11.38	11.36	11.05	11.02	11.00
	65,000	11.40	11.39	11.37	11.00	10.98	10.97
	70,000	11.22	11.20	11.17	11.03	11.01	10.97
	75,000	11.22	11.21	11.18	11.03	11.01	10.99
	80,000	11.26	11.25	11.22	11.03	11.02	11.00
	85,000	11.30	11.28	11.24	11.04	11.02	10.82
	90,000	11.30	11.28	11.26	11.03	11.01	10.98
	95,000	11.33	11.32	11.30	11.05	11.04	10.98
	100,000	11.33	11.32	11.30	11.02	11.01	10.99
Isend/Irecv	5,000	8.70	8.62	8.31	8.42	8.27	7.56
	10,000	9.84	9.80	9.73	9.48	9.43	9.25
	15,000	10.53	10.48	10.40	10.12	10.06	9.93
	20,000	10.69	10.65	10.60	10.49	10.45	10.39
	25,000	10.93	10.90	10.83	10.67	10.65	10.60
	30,000	11.07	11.05	11.01	10.75	10.73	10.70
	35,000	11.13	11.11	11.07	10.84	10.82	10.70
	40,000	11.21	11.19	11.17	10.94	10.92	10.88
	45,000	11.22	11.20	11.18	10.91	10.89	10.86
	50,000	11.33	11.32	11.29	10.98	10.96	10.94
	55,000	11.29	11.28	11.26	10.95	10.92	10.90
	60,000	11.38	11.36	11.35	11.04	11.02	11.00
	65,000	11.38	11.37	11.36	11.00	10.98	10.95
	70,000	11.18	11.16	11.14	11.02	11.01	10.99
	75,000	11.19	11.17	11.15	11.03	11.01	10.98
	80,000	11.23	11.21	10.97	11.03	11.01	10.75
	85,000	11.27	11.25	11.18	11.03	11.01	10.98
	90,000	11.26	11.25	11.22	11.02	11.01	10.94
	95,000	11.30	11.29	11.27	11.05	11.03	10.98
	100,000	11.31	11.29	11.27	11.01	11.00	10.99

Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Ssend/Irecv	5,000	8.64	8.56	8.15	6.68	6.59	5.97
	10,000	9.68	9.62	9.55	8.38	8.31	8.07
	15,000	10.41	10.36	10.30	9.29	9.23	9.09
	20,000	10.68	10.65	10.61	9.65	9.60	9.53
	25,000	10.88	10.86	10.82	10.00	9.94	9.35
	30,000	10.98	10.95	10.80	10.19	10.15	10.00
	35,000	11.07	11.05	10.99	10.31	10.27	10.17
	40,000	11.20	11.18	11.14	10.43	10.39	10.35
	45,000	11.18	11.17	11.11	10.47	10.44	10.36
	50,000	11.28	11.26	11.24	10.58	10.56	10.49
	55,000	11.29	11.27	11.25	10.56	10.52	10.46
	60,000	11.37	11.35	11.33	10.67	10.63	10.55
	65,000	11.34	11.33	11.31	10.68	10.65	10.57
	70,000	11.20	11.19	11.17	10.71	10.68	10.61
	75,000	11.21	11.19	11.15	10.71	10.67	10.64
	80,000	11.25	11.24	11.21	10.75	10.71	10.64
	85,000	11.28	11.27	11.26	10.77	10.74	10.68
	90,000	11.29	11.27	11.26	10.76	10.72	10.69
	95,000	11.32	11.31	11.30	10.78	10.75	10.71
	100,000	11.32	11.31	11.25	10.78	10.75	10.65
Issend/Irecv	5,000	8.60	8.52	8.14	6.70	6.56	2.94
	10,000	9.68	9.63	9.52	8.38	8.31	8.13
	15,000	10.40	10.35	10.24	9.30	9.23	9.05
	20,000	10.67	10.64	10.40	9.65	9.60	9.49
	25,000	10.87	10.85	10.81	10.01	9.95	9.85
	30,000	10.97	10.95	10.90	10.19	10.15	10.06
	35,000	11.06	11.04	10.98	10.31	10.26	10.12
	40,000	11.20	11.17	11.09	10.43	10.38	10.18
	45,000	11.18	11.16	11.10	10.48	10.45	10.38
	50,000	11.27	11.26	11.22	10.59	10.56	10.47
	55,000	11.28	11.26	11.24	10.56	10.52	10.43
	60,000	11.36	11.35	11.32	10.67	10.63	10.55
	65,000	11.33	11.32	11.30	10.68	10.65	10.58
	70,000	11.18	11.17	11.13	10.72	10.68	10.63
	75,000	11.19	11.17	11.15	10.71	10.68	10.63
	80,000	11.23	11.22	11.19	10.74	10.70	10.66
	85,000	11.26	11.25	11.18	10.77	10.74	10.69
	90,000	11.27	11.26	11.24	10.76	10.73	10.69
	95,000	11.30	11.29	11.28	10.79	10.76	10.72
	100,000	11.31	11.29	11.28	10.78	10.75	10.70

Long Messages

Construct	Message Size	LAM Best	LAM Avg	LAM Worst	MPICH Best	MPICH Avg	MPICH Worst
Send/Irecv	200k	11.55	11.54	11.54	10.94	10.92	10.90
	300k	11.62	11.62	11.61	11.00	10.99	10.97
	400k	11.66	11.65	11.65	11.03	11.02	10.99
	500k	11.68	11.68	11.67	11.06	11.05	11.04
	600k	11.70	11.69	11.69	11.08	11.07	11.06
	700k	11.71	11.70	11.67	11.08	11.08	11.06
	800k	11.71	11.71	11.71	11.09	11.09	11.07
	900k	11.72	11.72	11.72	11.10	11.09	11.09
	1,000k	11.73	11.72	11.72	11.10	11.10	11.09
	1,100k	11.73	11.73	11.72	11.14	11.13	11.13
	1,200k	11.73	11.73	11.73	11.14	11.13	11.12
	1,300k	11.73	11.73	11.73	11.14	11.13	11.12
	1,400k	11.74	11.74	11.73	11.14	11.13	11.13
	1,500k	11.74	11.74	11.73	11.14	11.14	11.13
	1,600k	11.74	11.74	11.73	11.14	11.14	11.13
	1,700k	11.74	11.74	11.74	11.14	11.14	11.14
	1,800k	11.74	11.74	11.74	11.14	11.14	11.14
	1,900k	11.74	11.74	11.74	11.15	11.15	11.14
	2,000k	11.75	11.75	11.74	11.15	11.15	11.14
Isend/Irecv	200k	11.53	11.52	11.51	10.94	10.92	10.90
	300k	11.61	11.60	11.60	11.00	10.99	10.95
	400k	11.65	11.64	11.46	11.03	11.03	10.99
	500k	11.67	11.67	11.67	11.06	11.05	11.03
	600k	11.69	11.69	11.68	11.08	11.07	11.06
	700k	11.70	11.70	11.70	11.08	11.08	11.07
	800k	11.71	11.71	11.70	11.09	11.09	11.07
	900k	11.71	11.71	11.71	11.10	11.09	11.09
	1,000k	11.72	11.72	11.71	11.10	11.10	11.09
	1,100k	11.72	11.72	11.72	11.13	11.13	11.13
	1,200k	11.73	11.73	11.72	11.14	11.13	11.12
	1,300k	11.73	11.73	11.72	11.13	11.13	11.13
	1,400k	11.73	11.73	11.73	11.14	11.13	11.13
	1,500k	11.73	11.73	11.73	11.14	11.14	11.05
	1,600k	11.74	11.74	11.72	11.14	11.14	11.13
	1,700k	11.74	11.74	11.74	11.14	11.14	11.14
	1,800k	11.74	11.74	11.74	11.14	11.14	11.14
	1,900k	11.74	11.74	11.74	11.15	11.15	11.14
	2,000k	11.74	11.74	11.74	11.15	11.15	11.14

Construct	Message Size	LAM			MPICH		
		Best	Avg	Worst	Best	Avg	Worst
Ssend/Irecv	200k	11.54	11.53	11.53	10.94	10.92	10.87
	300k	11.61	11.61	11.60	11.00	10.99	10.97
	400k	11.65	11.65	11.60	11.03	11.02	10.99
	500k	11.68	11.67	11.67	11.06	11.05	11.04
	600k	11.69	11.69	11.69	11.08	11.07	11.05
	700k	11.70	11.70	11.70	11.08	11.08	11.06
	800k	11.71	11.71	11.71	11.09	11.09	11.08
	900k	11.72	11.72	11.71	11.10	11.09	11.09
	1,000k	11.72	11.72	11.72	11.10	11.10	11.09
	1,100k	11.73	11.72	11.72	11.13	11.13	11.13
	1,200k	11.73	11.73	11.73	11.14	11.13	11.12
	1,300k	11.73	11.73	11.73	11.14	11.13	11.12
	1,400k	11.74	11.73	11.73	11.14	11.13	11.13
	1,500k	11.74	11.74	11.73	11.14	11.14	11.13
	1,600k	11.74	11.74	11.74	11.14	11.14	11.14
	1,700k	11.74	11.74	11.74	11.14	11.14	11.13
	1,800k	11.74	11.74	11.74	11.14	11.14	11.13
	1,900k	11.74	11.74	11.74	11.15	11.15	11.14
	2,000k	11.75	11.74	11.74	11.15	11.15	11.14
Issend/Irecv	200k	11.53	11.52	11.51	10.95	10.92	10.90
	300k	11.61	11.60	11.60	11.00	10.99	10.96
	400k	11.65	11.64	11.64	11.04	11.02	11.01
	500k	11.67	11.67	11.67	11.06	11.05	11.04
	600k	11.69	11.69	11.68	11.08	11.07	11.05
	700k	11.70	11.70	11.70	11.08	11.08	11.07
	800k	11.71	11.71	11.71	11.09	11.09	11.08
	900k	11.71	11.71	11.71	11.10	11.09	11.08
	1,000k	11.72	11.72	11.72	11.10	11.10	11.09
	1,100k	11.72	11.72	11.72	11.13	11.13	11.11
	1,200k	11.73	11.73	11.72	11.13	11.13	11.12
	1,300k	11.73	11.73	11.73	11.13	11.13	11.13
	1,400k	11.73	11.73	11.73	11.14	11.13	11.13
	1,500k	11.73	11.73	11.73	11.14	11.14	11.13
	1,600k	11.74	11.74	11.73	11.14	11.14	11.13
	1,700k	11.74	11.74	11.74	11.14	11.14	11.11
	1,800k	11.74	11.74	11.74	11.14	11.14	11.13
	1,900k	11.74	11.74	11.74	11.15	11.15	11.14
	2,000k	11.74	11.74	11.74	11.15	11.15	11.14