

# An Evaluation of Message Passing Implementations on Beowulf Workstations

P. H. Carns

W. B. Ligon III

S. P. McMillan

R. B. Ross

Parallel Architecture Research Lab

Clemson University

102 Riggs Hall

Clemson, SC 29634-0915

864-656-7223

pcarns@eng.clemson.edu walt@eng.clemson.edu spmcml@eng.clemson.edu rbross@eng.clemson.edu

*Abstract*— Beowulf workstations have become a popular choice for high-end computing in a number of application domains. One of the key building blocks of parallel applications on Beowulf workstations is a message passing library. While there are message passing library implementations available for use on Beowulf workstations, as of yet none have been specifically tailored to this new, unique architecture. Thus it is important to evaluate the existing packages in order to determine how these perform in this environment. This paper examines a set of four message passing libraries available for Beowulf workstations, focusing on their features, implementation, reliability, and performance. From this evaluation we identify the strengths and weaknesses of the packages and point out how implementations might be optimized to better suit the Beowulf environment.

## TABLE OF CONTENTS

- 1 INTRODUCTION
- 2 IMPLEMENTATION OF PACKAGES
- 3 PERFORMANCE
- 4 CONCLUSION

## 1 INTRODUCTION

As parallel processing has matured, two programming paradigms have been developed, shared memory and message passing. Particularly for distributed memory machines, message passing has become the most popular technique for implementing parallel applications. As this popularity increased, message passing interfaces and libraries developed and matured to match demand. Now it is common for commercial machines to include a machine-specific message passing implementation such as the NX library for the Paragon and the MPL and MPI libraries for the IBM SP-2.

As clusters of workstations were recognized as a viable platform, message passing libraries grew to support this new architecture as well. This led to libraries such as PVM [4], which was specifically designed for use in situations where the “virtual machine” often varied each time the software was

started. As the number of message passing libraries grew, standard API’s for message passing, such as MPI [13], were introduced to provide a common interface for programmers developing for multiple architectures, while allowing the architecture vendors the flexibility to implement the interface efficiently. This has helped to eliminate situations where programmers are forced to recode applications to move them to a new platform.

A new class of parallel machines, termed Beowulf workstations, has become a popular approach to providing high end computing resources. These machines consist of a Pile of PCs (PoPCs) built from off-the-shelf hardware components, a private high speed network such as switched fast ethernet, and freely available software tools and operating system. While message passing libraries exist that will operate on this platform, the characteristics of these machines differ significantly from previous clusters of workstations, providing thus far unexplored opportunities for software optimization specific to this environment.

While evaluations of message passing libraries and performance have already been performed on a variety of message passing implementations, most of them concentrate on how these libraries perform on MPP’s such as the Intel Paragon, the IBM SP-2, and the Thinking Machines CM-5 [1, 8, 3]. In order to better understand how message passing software performs on the Beowulf architecture, the many different interfaces and implementations should be qualitatively and quantitatively compared. In this paper we examine a number of message passing packages which can operate on Beowulf workstations, focusing on the features, implementation, reliability, and performance of these packages. Specifically, we evaluate two versions of the Message Passing Interface (MPI), LAM/MPI and MPICH [6], and the Parallel Virtual Machine (PVM). In addition, we will compare these to a new message passing implementation, which we will introduce, called Beowulf Network Messaging (BNM). By looking at the characteristics of these packages we hope to discover areas where improvement could be made with respect to operation in the Beowulf environment.

In the following introduction sections, we will provide back-

ground on the Beowulf workstation concept and how software available for this environment is maturing in addition to introducing the message passing packages. In Section 2 we qualitatively examine the software packages, discussing issues such as features and implementation details. In Section 3 we examine the performance of the packages both for spawning tasks and for some sample communication patterns. Conclusions are drawn in Section 4, recommendations are made on potential areas of improvement, and future areas of study are discussed.

### 1.1 *Beowulf*

The Beowulf-class parallel machine has evolved from early work in low cost computing. The first work in this area centered around clusters of workstations [2]. These clusters are often built using existing workstations which are used as interactive systems during the day, can be heterogeneous in composition, and rely on extra software to balance the load across the machines in the presence of interactive jobs. As it became obvious that workstations could be used for parallel processing, groups began to build dedicated machines from inexpensive, non-proprietary hardware. These “Pile-of-PCs” consist of a cluster of machines dedicated as nodes in a parallel processor, built entirely from commodity off the shelf parts, and employing a private system area network for communication [11]. The use of off-the-shelf parts results in systems that are tailored to meet the needs of the users, built using the most up-to-date technology at the time of purchase, and cost substantially less than previous parallel processing systems. The Beowulf workstation concept builds on the Pile-of-PCs concept by utilizing a freely available base of software. The free availability of most system software source encourages customization and performance improvements. Experiments have shown Beowulf workstations capable of providing high performance for applications in a number of problem domains.

One of the greatest strengths of commercial systems in general has always been the support, both in software and troubleshooting, that is made available to owners. Along this same vein the Beowulf community has banded together to build a software infrastructure and to assist one another with problems. Most of this software already existed, including the operating system, compiler, network file system, and most common utilities. However, it has become apparent that while this software is robust and fulfills users’ needs, there is room for improvement. Parallel file systems such as PVFS [9] provide better I/O performance and consistency for parallel applications using distributed data sets, processor-specific compiler enhancements and libraries can boost application performance, and kernel modifications can provide services such as global process ID’s, global signalling, and Distributed Shared Memory (DSM) which help build a more complete environment.

Along these same lines, the existing message passing libraries were built before the Beowulf environment had matured. Thus this software too could potentially be altered or rewritten to more effectively operate in the Beowulf environment. In the past the message passing libraries available for Beowulf have been used primarily by individuals on clusters of workstations. These individuals would most often configure and install the software personally, then start the necessary daemons on the appropriate machines when they wished to execute a parallel program. The set of machines used often varied between executions based on availability and load. In the Beowulf environment, on the other hand, message passing support should be considered system software. Ideally this software would be installed and configured by the administrator and any necessary daemons would be started along with other services when the machine is booted. Additionally many of these packages are written to support heterogeneous collections of machines. This again is not an issue in most Beowulf machines. Luckily most of these packages have options to disable encoding that would take place if heterogeneous collections were used.

### 1.2 *Message Passing Packages*

In this evaluation we will focus on implementations of three different interfaces, the Parallel Virtual Machine, Message Passing Interface, and Beowulf Network Messaging. Here we give an overview of these packages; details of the specific implementations will be discussed in section 2.

**1.2.1 *PVM***— The Parallel Virtual Machine (PVM) [4] was originally developed at Oak Ridge National Laboratory (ORNL) specifically to handle message passing on heterogeneous distributed computers. In addition to providing a message passing interface, PVM implements resource management, signal handling, and fault tolerance features that help build a user environment for parallel processing. As a result of these additional generic capabilities needed to pass data reliably in a heterogeneous environment, PVM is generally a less efficient message passing interface on Massively Parallel Processors (MPP’s) [5]. While PVM is the defacto standard for clusters of workstations, the need to implement additional features beyond the message passing interface hinders it from becoming ubiquitous on MPP’s.

PVM’s implementation and interface development occur mainly at ORNL. There are, however, commercial implementations of PVM available that are designed for efficient message passing on MPP architectures. One example is PVM<sub>e</sub> for the IBM SP-2 MPP [12]. These MPP implementations, along with competing implementations for PVM on clusters of workstations, are not common.

**1.2.2 *MPI***— The Message Passing Interface (MPI) Forum has been meeting since 1992 and is comprised of high performance computing professionals from over 40 organizations

[13]. Their goal is to develop a message passing interface that meets the needs of the majority of users in order to foster the use of a common interface on the ever-growing number of parallel machines. By separating the interface from the implementation, MPI provides a framework for MPP vendors to utilize in designing efficient commercial implementations.

A number of vendors have jumped on the MPI bandwagon, and vendor-supplied implementations are now available from IBM, Cray Research, SGI, Hewlett Packard, and others. In addition, a number of competing implementations have been created for clusters of workstations. Two popular choices on clusters, MPICH and LAM/MPI, will be evaluated in this study. These two implementations have been the subject of a previous study conducted on a cluster of DEC 3000/300 machines connected with FDDI [10].

The MPI Chameleon (MPICH) effort began in 1993 as an attempt to provide an immediate implementation of MPI that would track the standard as it matured [6]. It was developed at Argonne National Laboratory as a research project to provide features that make implementing MPI simple on many types of hardware. To do this, MPICH implements MPI over an architecture independent Abstract Device Interface (ADI). The ADI has a smaller interface than MPI, making it easier for vendors to implement, resulting in quicker development time without loss in efficiency. MPICH takes this one step further by implementing the ADI on top of what they call a “channel interface”, providing an even smaller interface for a vendor to implement. While the “channel interface” implementations will be extremely inefficient, it provides for a quick and dirty implementation that can be streamlined later by implementing the ADI piecemeal.

Local Area Multicomputer (LAM) originated at the Ohio Super computing Facility and is now maintained by the Laboratory of Scientific Computing at Notre Dame. LAM is a package that provides task scheduling, signal handling, and message delivery in a distributed environment, and is layered to allow implementation with any message passing interface. For example, PVM has been implemented over LAM, however only LAM’s MPI version will be evaluated in this study.

*1.2.3 BNM*— Beowulf Network Messaging (BNM) is currently under development at the Parallel Architecture Research Laboratory at Clemson University as a low level solution for task spawning and communication in the Beowulf environment. BNM provides only a minimal set of facilities, including remote task spawning, task ID management, and byte oriented message passing. The goal of the project is to provide a simple and efficient communications library for Beowulf that could be used as a building block for implementations of higher level interfaces such as MPI. At the moment BNM is in its infancy stage and the results of this study will have a direct impact on its development.

## 2 IMPLEMENTATION OF PACKAGES

While all of these packages provide a common core functionality, there are significant differences in the implementations that have an impact on both the ease of use and particularly the performance of applications using them. Three areas of particular interest are the software architecture and related tools, the approach used for spawning tasks, and the method of communication between tasks. Each of these will be discussed in turn here.

The versions of the packages we are using are as follows:

- PVM version 3.3.11
- LAM version 6.1
- MPICH version 1.1.1 (ch\_p4 interface)
- BNM version 1.0

### 2.1 Architecture

There are significant differences between the packages in terms of the architecture of the software and the tools provided. All packages provide a library of message passing primitives to which applications link. PVM and LAM/MPI provide an additional daemon that is started by the user before parallel applications are executed. BNM uses a similar daemon, but a single daemon on each node handles requests for all users and is started when the machine boots. MPICH, by default, attempts to use a system level daemon, but can be configured for either user level daemons or can option for the standard remote shell service.

In terms of debugging and monitoring tools, PVM and LAM/MPI are strongest. PVM includes a console allowing the user to check the status of PVM tasks and send signals to them. LAM provides a set of executable tools which provide similar functionality. MPICH provides very few runtime utilities, but it does, along with PVM and LAM, provide for log file generation and trace utilities. BNM, on the other hand, provides little or no support for monitoring or debugging.

There are two common techniques for starting parallel tasks using these implementations: the use of a command line executable that starts the parallel tasks and the use of library calls to spawn tasks on remote nodes. PVM, LAM/MPI, and BNM provide both mechanisms; PVM allows execution to be started from the console and allows parallel tasks to be started from within an application using `pvm_spawn()`; LAM/MPI provides `mpirun` to start tasks and additionally implements the MPI 2.0 `MPI_Spawn()` call which allows tasks to be started from within the application; BNM implements both `bnmrn` executable and a `bnm_spawn()` library call; and MPICH provides an `mpirun` executable for starting the parallel tasks, but does not support the `MPI_Spawn()` call.

Table 1: Message Passing Summary

Option	LAM	MPICH	PVM	BNM
Spawn method	User daemon	System, user, or rsh daemon	User daemon	System daemon
Startup command	mpirun	mpirun	pvm	bmrnrun
Spawn command	MPI_Spawn()	N/A	pvm_spawn()	bmrn_spawn()
UDP communication	default	No	default	No
UDP packet size	8K	N/A	4K (settable w/ pvm_setopt)	N/A
UDP Retransmission Timeout (observed)	500 - 1200 ms	N/A	10 ms	N/A
TCP communication	-c2c (mpirun option)	default	PvmDirectRoute (pvm_setopt)	default
TCP packet size	maximum	maximum	4K (settable w/ pvm_setopt)	maximum
Homogeneous mode	-O (mpirun option)	automatic	PvmDataRaw (pvm_initSend)	default

## 2.2 Task Spawning

There are three major factors that determine the time necessary for spawning tasks: the method used to start the process, the location of the executable and libraries, and the functions performed on startup. In all of our tests executables were stored on an NFS mounted file system, so this factor was held constant.

Starting tasks is accomplished via one of three techniques in these packages:

- direct use of remote shell to start each task
- use of remote shell to start a daemon which subsequently starts tasks
- use of a full-time daemon for starting tasks

Before a user begins running parallel applications under PVM, he or she first starts up PVM and defines the virtual machine. This process starts a user level PVM daemon on each of the nodes in the machine by using the remote shell facility. These daemons provide the user runtime spawning services for PVM tasks. One unique feature of the PVM daemon is its ability to start multiple tasks on the same node with only one communication from the parent; as we will see this leads to better performance from PVM when starting multiple tasks on the same node.

LAM/MPI uses a technique similar to PVM for spawning tasks. The user first starts up user level daemons on each node in the Local Area Multicomputer. These daemons then spawn MPI tasks for the user at runtime. BNM, like PVM and LAM, uses a daemon to start processes, however, this daemon provides a system service, so it is started when the machine boots, and only one such daemon is needed to serve multiple

users. BNM, LAM, and MPICH, unlike PVM, require multiple communications for multiple tasks spawned on the same node.

MPICH attempts to start remote processes by connecting to a default system level daemon, and if that daemon is unavailable, uses the remote shell facility. This default daemon can also be configured at the user level, but we were unable to get this daemon to startup remote processes properly during the testing. Therefore, the remote shell was used, which is extremely slow, particularly when the inetd server is used. Hence, MPICH is at a severe disadvantage when it comes to the speed of starting new tasks. However, we will see that other factors leads to comparable results with LAM/MPI.

The last factor in startup time is the amount of additional initialization and setup performed. For PVM (including PvmDirectRoute version) and BNM, this is minimal; network connections are set up but not established. LAM and MPICH require additional synchronization of the MPI\_COMM\_WORLD communicator from each task. MPICH, in order to synchronize, establishes TCP connections to the appropriate tasks and subsequently closes those connections before the initialization is complete. Additionally, LAM with the “-c2c” option selected, specifying direct task to task connectivity using TCP, requires all task to task connections be established before the spawn completes. This significantly effects the spawn time, as we will see in Section 3.

Other spawn environment options, not tested, are available on both the MPI and PVM versions. MPI allows the executable to be passed to the target node, not requiring the program to exist on that node. MPI and PVM have options to provide a current working directory and a search path to find the executable. They do so by setting the paths up in a script file before hand, during initialization of the virtual machine or local

area multicomputer daemons. BNM provides this capability by passing the environment every time a new message passing program executes. During testing, we used full pathnames for the executables, hence environments were not used, so we could eliminate this variable from our testing.

### 2.3 Message Passing

All of these packages use standard IP protocols for message passing between nodes. PVM by default passes messages in three steps. First the message is passed from the application to the local PVM daemon via a TCP stream socket (some implementations use UNIX stream sockets). The daemon then divides the message up into “packets” of approximately 4K bytes, which it passes to the PVM daemon on the remote machine using UDP. Each “packet” is acknowledged by the receiver individually, and lost packets are resent. The timeout and retransmission for lost packets is, from observation of network traffic, generally around 10 milliseconds on our network. PVM uses a simple round trip time estimator [4], does not seem to use Karn’s algorithm [14], and implements no delayed acknowledgment strategy.

Using the `PvmDirectRoute` option for PVM, TCP is used to communicate directly between application tasks. These connections are established when they are needed and they are left open, once connected, until application completion. When using TCP, PVM still breaks the messages into default packets of approximately 4K, but acknowledgements are not used (because they are not needed with a reliable protocol such as TCP). As we will see, this unnecessary packetization results in inefficient use of TCP’s maximum segment size. This packet size is modifiable with PVM daemon command line parameters and through the `pvm_setopt()` function.

LAM/MPI also uses UDP by default. Applications pass messages through UNIX stream sockets to the LAM daemon, which uses UDP to pass the message to the LAM daemon on the remote machine, which then passes the message to the application through a UNIX stream socket. LAM breaks messages into packets of approximately 8K when transferring across the network, and each packet is acknowledged individually. From observing network traffic, it seems LAM uses a slow start strategy, eventually (sends > 16K) allowing two outstanding unacknowledged packets which consistently resulted in lost IP fragments. The retransmission timeout was observed between 500 to 1200 milliseconds, and when combined with the large UDP packet size, can lead to poor performance over our Beowulf network.

When the “`-c2c`” option is selected, LAM/MPI switches to TCP connections for data transfer. As mentioned earlier, all these TCP connections are established when the tasks are spawned. These connections are made directly between the tasks, and messages are sent as a whole without being broken up into packets by the application. LAM’s TCP version uses

an eager send strategy with message sizes less than 16K and a rendezvous strategy when message sizes are greater than 16K instead of relying on TCP to handle buffering and flow control.

Both BNM and MPICH use TCP exclusively, directly connect between application tasks, and send messages without breaking up the packets at the application layer. They open connections when they are needed and hold them open until task completion.

In the next section we will see how these implementation details affect the overall performance of the message passing libraries both in spawning tasks and in passing messages.

## 3 PERFORMANCE

Our evaluation of the performance of these packages focuses on two key areas, start-up time for parallel tasks and message passing time for some common patterns. Spawn time for tasks on a Beowulf machine may or may not be important to the user, depending on the average run-time of the applications in use. For users running many iterations of short run-time applications this time can be critical. In any case, as more and more core functions are distributed across the parallel machine, the time to start a remote task will become more important.

The time to pass messages between tasks obviously has direct impact on the performance of applications, especially when the applications are more fine-grain. In our tests we attempt to cover some common logical configurations of processes that might be seen in applications either because of well-known algorithms or due to porting from other architectures.

We test one instance of MPI’s complex data type constructor utilities and compare those utilities with code that provides this service manually with user code. Finally, we will discuss reliability issues we ran into during these tests.

### 3.1 Test Setup

The Beowulf system used in these tests consists of the following:

- 17 single-processor 150 MHz Intel Pentium nodes
- 64 MB RAM per node
- 2 SMC Tulip-based ethernet cards per node using `tulip.c` v0.88
- 2 fast ethernet networks, one bus and one full-duplex switch
- Linux 2.0.34

One of the 17 nodes is used for interaction with the system, while the others are used solely for computation. The interactive, or “head”, node communicates with the other nodes over the bus network. All compute nodes communicate with each other over the full-duplex switch. On all tests, the “head” node is used to spawn off one process on each of the computation nodes and to time all tests using the `gettimeofday()` call.

All software was setup for a homogeneous cluster of workstations. PVM provides this with the `pvm_initsend(PvmDataRaw)` function, while LAM uses the “-O” option on the `mpirun` command line. MPICH detects this automatically and BNM provides no other functionality.

We also modified the 2.0.34 LINUX kernel to prevent TCP from resetting the slow start congestion window after not doing anything for a “long time” [7]. Without doing this, the tests we performed slow down considerably and prevent us from seeing TCP in action. Basically, the TCP versions never get out of slow start when the message size is bigger than the maximum segment size. For example, when passing messages around in a ring, the combination of the delayed acknowledgement and slow start prevents a node from retransmitting again for a “long time”. Since we want to see TCP implemented in full capacity, we removed this feature.

### 3.2 Starting Remote Tasks

In this test we used the head node to spawn off the processes onto the computation nodes with no computations performed. We started timing before spawning began and ended it after the spawning operations completed. For PVM, LAM, and BNM, we had the spawning task use the respective spawn library calls to perform this function and time the operation. With MPICH there is no spawn library call, so we timed the `mpirun` command.

As can be seen from Figure 1, MPICH takes the longest time to perform the initial one time startup of the message passing tasks. This makes sense considering MPICH uses the remote shell service (see Section 2.2). Also, LAM ,with the “-c2c” option set, starts to catch up as the number of processes spawned increases. We would expect to see this large increase as a result of the number of TCP connections that need to be established for a large number of spawned tasks. The other software versions performed well, with PVM providing the best task startup services, as we expected from our discussion in Section 2.2.

### 3.3 Message Passing Performance

The message passing tests consist of one master program executing on the head spawning off one process on each of the

computational nodes. The master’s job consists of spawning off the processes, doing the timing, and waiting for a one byte message from each of the processing nodes to signal completion of the test. We designed the tests such that the first computational node in the Beowulf system always gets assigned task 1, the second node gets task 2, and so on.

In performing these tests, we used nonblocking sends and receives on all occasions. Unless otherwise specified (See Section 3.3.3), message sizes are in bytes and we use simple data types in all communications. For instance, the `MPI_BYTE` datatype for MPI and the `pvm_pkbyte()` command for PVM are used for simple byte sends. BNM provides only byte oriented service and has no functionality to construct complex data types.

*3.3.1 Ring and Torus Tests*— In all the ring and torus tests, we performed 50 loops and varied the message size. The ring loop started with a send from task 1 to task 2 and so on. The ring loop ends with a send from task 16 to task 1 (see Figure 2). The torus loop test has a very similar setup as the ring loop, but data was passed only within a column or row of the torus matrix (4x4 torus - see Figure 3 and 4). For example, in the column test, data loops started at the head of each column, moved down to the end of the column and then finished back at the head. The row tests act the same way, except the data starts in the row heads.

As we can see from Figures 5, 6, and 7, the ring pattern take four times longer than the torus patterns which is expected. We are using a 16 node ring, and the torus is 4x4 with each column or row of the torus passing in a ring formation. The only real point of this, again, is to create different traffic patterns and determine why different algorithms perform better under different loads. As is obvious from the figures, PVM performs the best, while the others fall behind significantly. We need to discuss why that is the case, and then move on to discussing the small differences between the versions of MPI and BNM.

PVM implements its own reliable protocol layer over UDP (See Section 2.3), as does LAM, with PVM performing significantly better. This can be attributed to the smaller retransmission timeout values and the smaller packet size. The IP fragmentation that occurs is not as bad with PVM’s 4K packet size than it is with LAM at 8K. The smaller packets have less chance of losing a fragment on the network. In our tests, the 8K packet size caused LAM to perform erratically at larger message sizes and prevented us from getting consistent results over 20K where LAM will start to generate two complete outstanding packets (see Section 2.3).

The other versions using TCP have problems compared to PVM because of the retransmission strategy of the TCP protocol. This includes the retransmission timeouts, delayed acknowledgments, slow start, congestion avoidance, Nagle, and fast retransmit and recovery algorithms. All of these TCP algorithms are based upon testing in general networking traf-

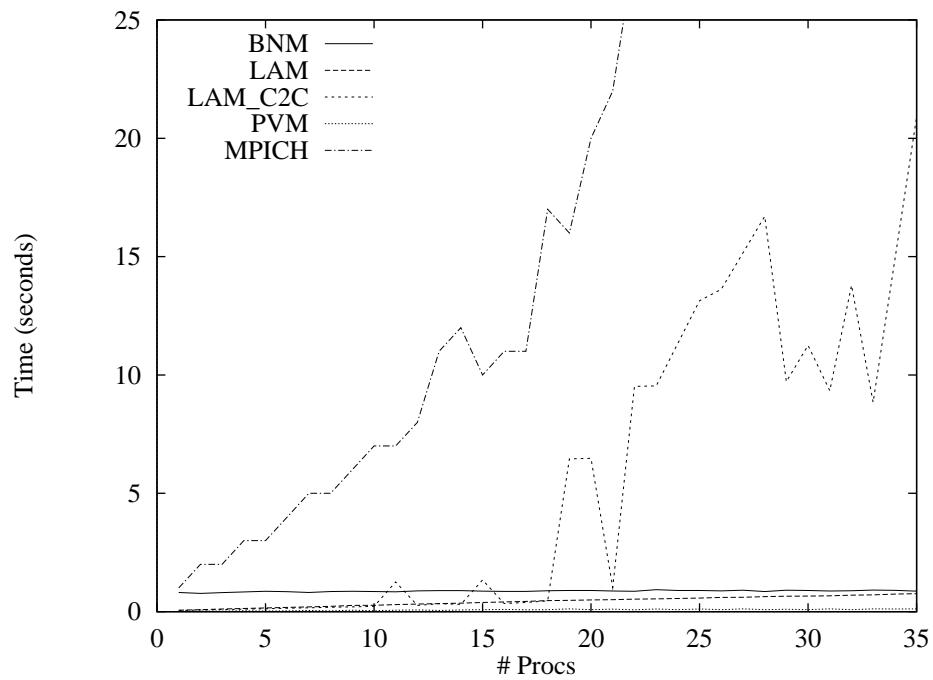


Figure 1: Spawning Tests

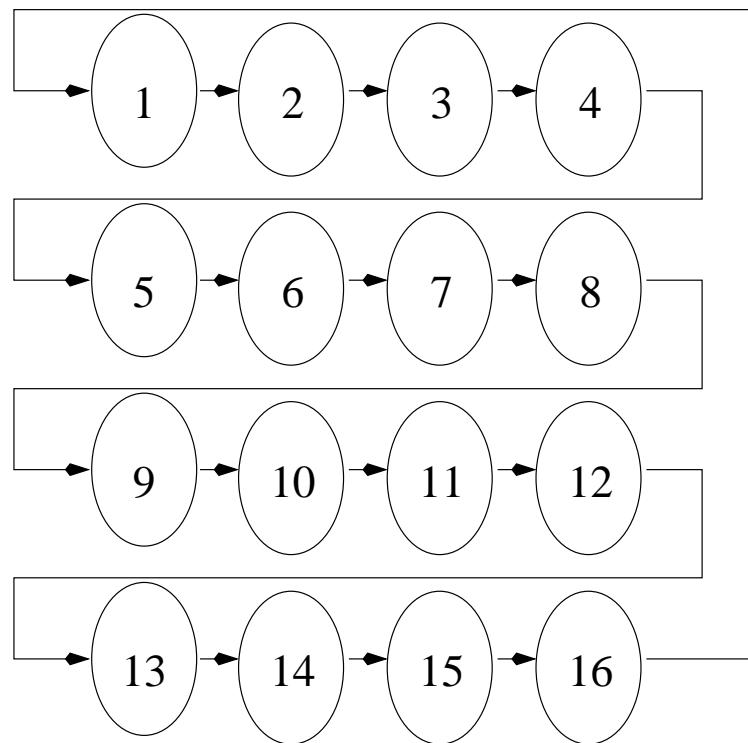


Figure 2: Ring Test Description

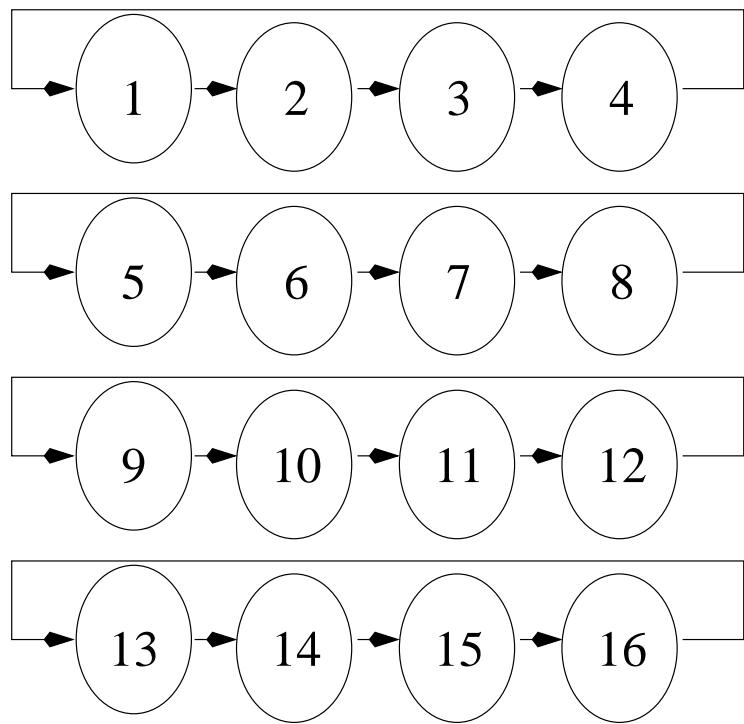


Figure 3: Row Torus Test Description

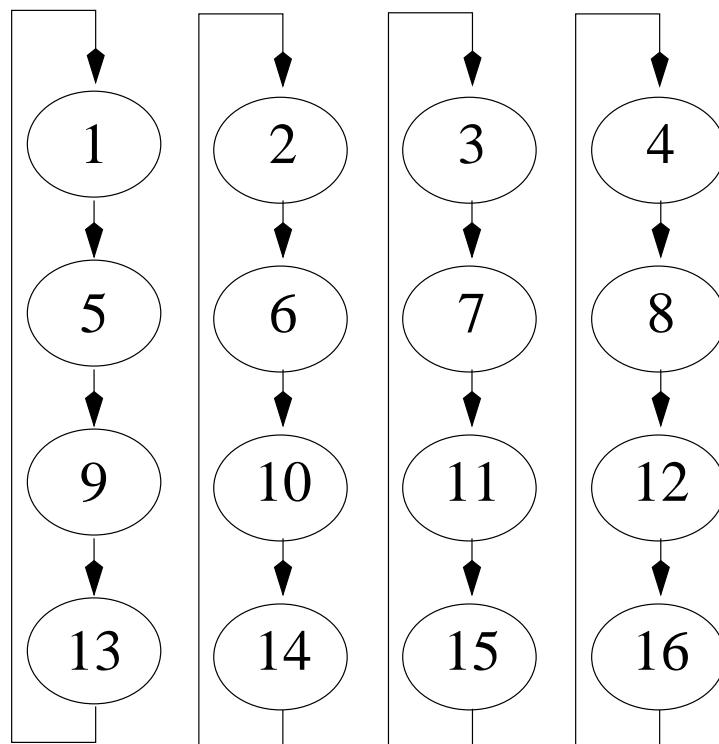


Figure 4: Column Torus Test Description

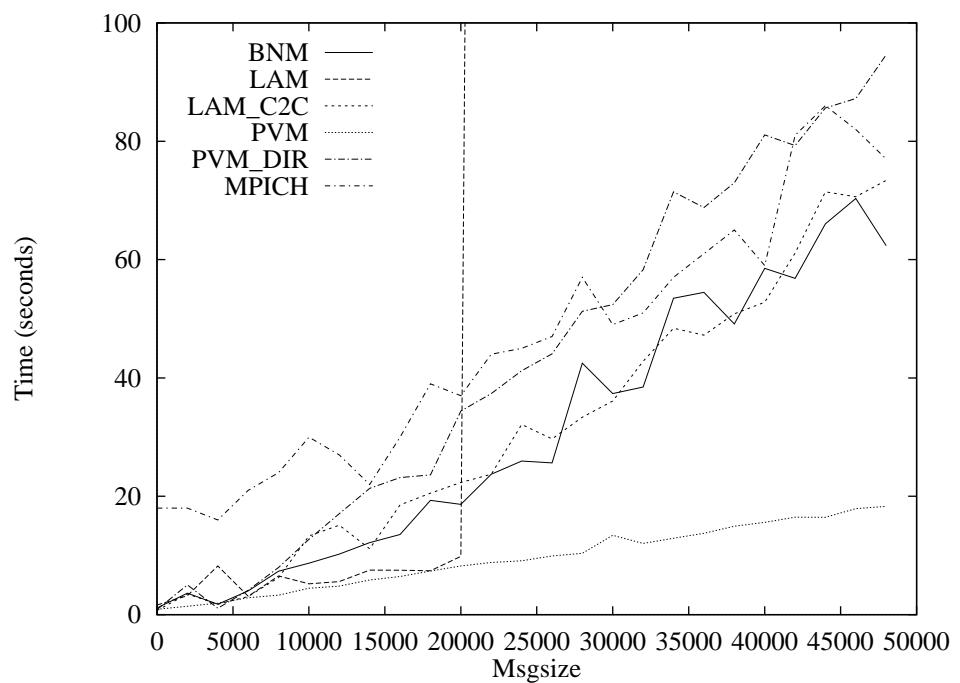


Figure 5: Ring Tests

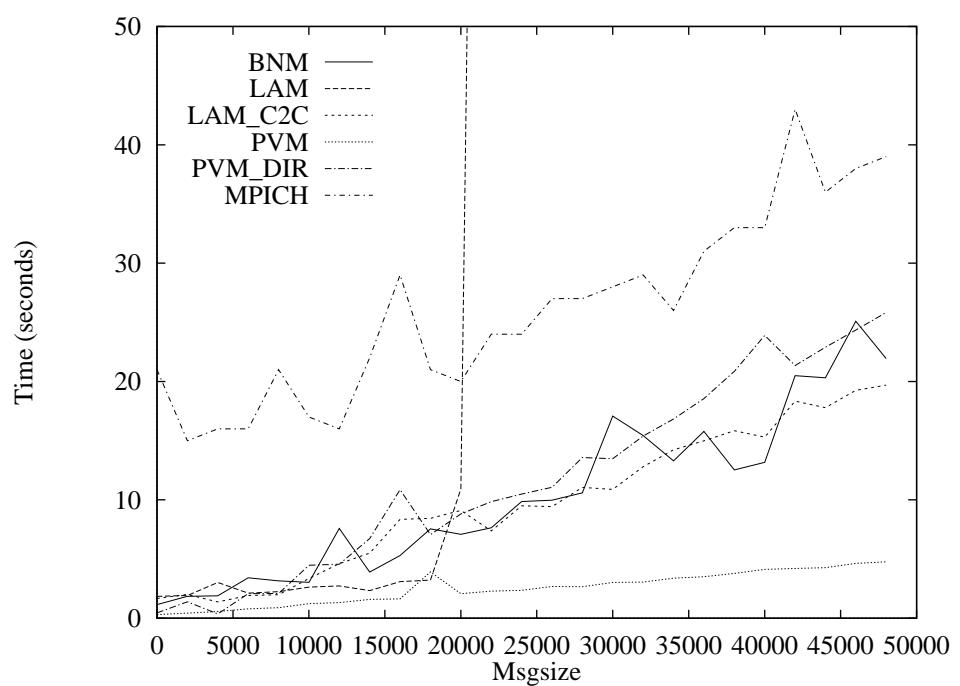


Figure 6: Row Torus Tests

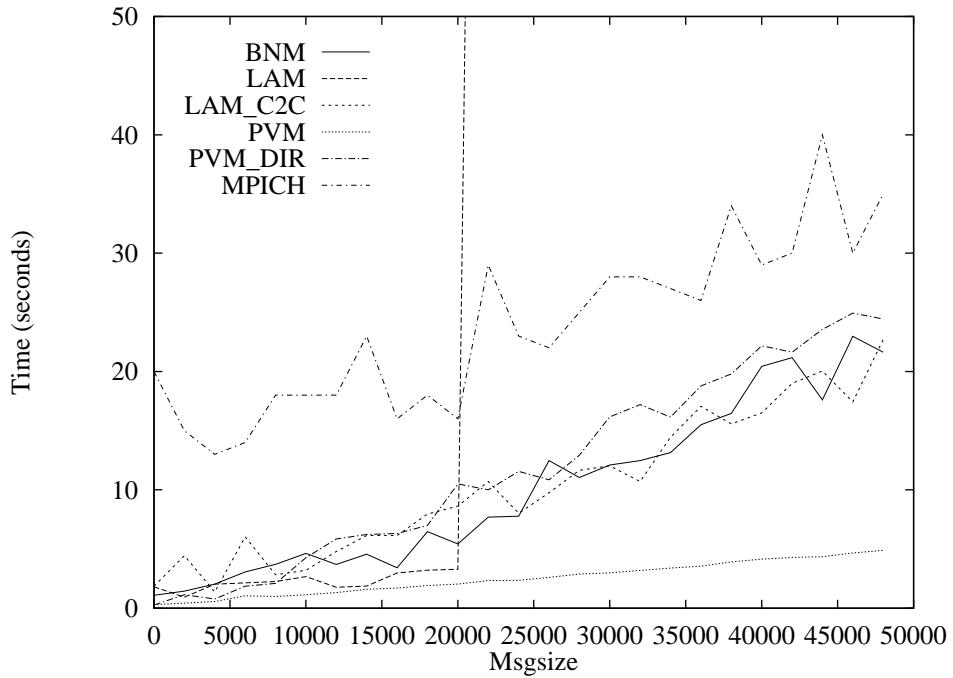


Figure 7: Column Torus Tests

fic environments, which will differ from traffic in a private network generating only message passing traffic. PVM’s direct route TCP implementation performs worse than the other TCP implementations because of the application layer packetization we described in Section 2.3.

**3.3.2 Process Bottleneck Tests**— In this test, a central bottleneck task, task 1, sends data to tasks 2 through 16. After receiving the data, tasks 2 through 16 send data back to task 1 (see Figure 8). We performed this test for five iterations of the sends and receives while we varied the message size.

From Figure 9, we see that the TCP versions of the message passing software packages generally perform consistently, while the UDP versions behave erratically and poorly as the message size increases. This would lead us to conclude TCP’s retransmission strategy behaves better, with heavy congestion, than what PVM and LAM implement over UDP. However, we see one striking example of unstable performance in LAM/MPI’s TCP version.

The only difference between LAM’s TCP version, compared to BNM, PVM, and MPICH, relates to the setup of communications. As we stated in section 2.3, LAM fully connects all tasks during task startup, while the others only connect when a communication is required between two tasks. What this creates, during these bottleneck tests, is an artificial round trip time (RTT) and retransmission timeout (RTO) calculated by TCP. Basically, when we start communications with LAM, data is sent immediately to all tasks and then promptly returned by all tasks. This obviously creates serious congestion on our switched network, as it is intended to, and usually

leads to lost TCP packets.

TCP starts with an RTO of three seconds, which on some occasions allows a task to receive data, send the corresponding acknowledgment packet with or without data, have that packet lost and retransmitted before the RTO of the original sending process expires. This leads to artificial RTT values and RTO calculations which, if conditions are appropriate, hit a “harmonic” leading to the RTO eventually clamping at its maximum value of two minutes. This problem is exacerbated by the back off strategy [7] used that doubles the RTO after every lost packet and does not reset itself until an acknowledgment is received from a non-retransmitted packet. For example, a packet being sent to the same processor could get lost every time we send the first time only, and because the back off count never gets reset, we will reach the maximum value of the RTO very quickly.

These RTT and RTO problems rarely occur with the other implementations using TCP. The other versions do not connect before communications, resulting in the first send of data from the bottlenecked process to be less congested. During these first set of sends, each task connection must be set up before sending data. Connecting tasks using TCP requires the process to block until the connection has been established, therefore, the only competing traffic in the network will be the connection handshake to the next processor. Because packets in this un-congested environment are not lost, we achieve a good initial RTT estimation and RTO calculation. To prove this point, we modified the bottleneck program for LAM, so that, on the first data send, the central task performs a send/receive combination to each task separately. After this

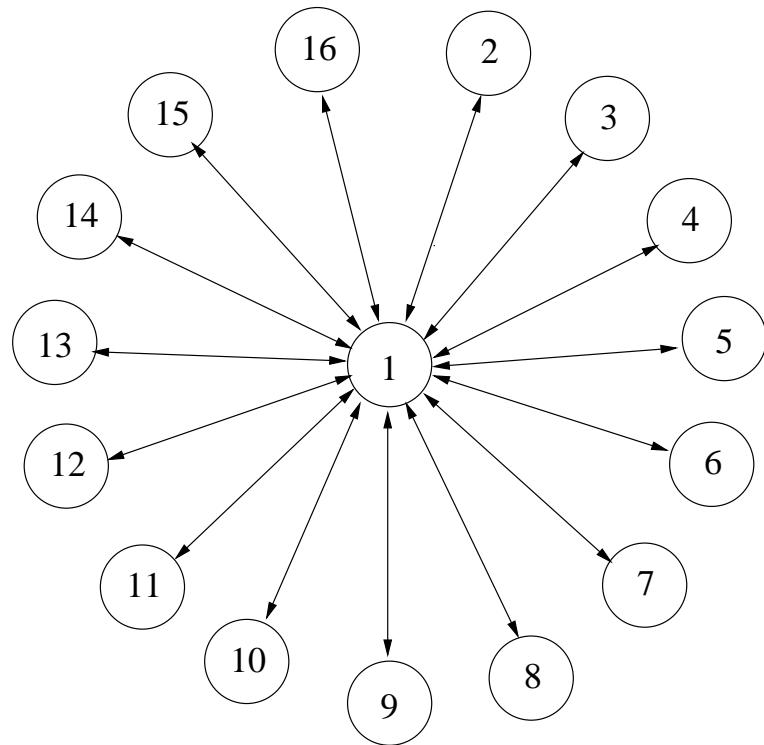


Figure 8: Bottleneck Test Description

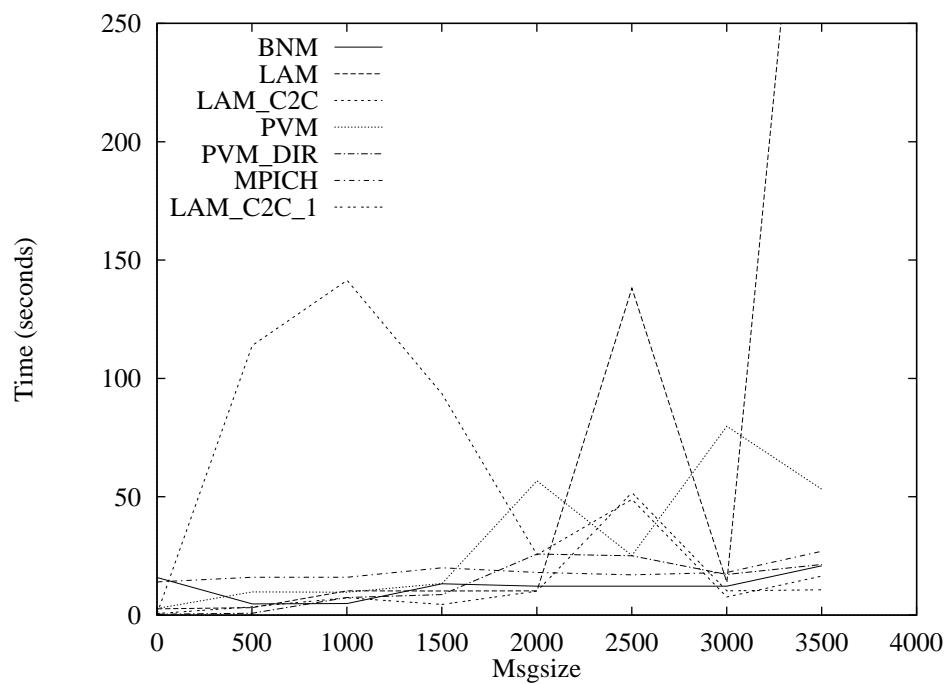


Figure 9: Bottleneck Tests

initial send, we started the identical bottlenecked algorithm as before. The results are shown in Figure 9 in the LAM\_C2C\_1 graph. The resulting performance measurements for LAM’s modified TCP tests were as stable as the other TCP packages.

We noticed one other interesting observation about message sizes outside of the range of Figure 9. When LAM (-c2c) changes its messaging strategy to rendezvous at 16K (see Section 2.3), it performs poorly relative to the other algorithms. The need for the bottlenecked processor to poll every time through the loop results in the network being used inefficiently, even though the resulting traffic will be un-congested.

**3.3.3 Matrix Transpose Tests**— Here, we have each computational node pass data around in the ring pattern discussed in Section 3.3.1, transposing a square matrix once before sending to the next node. We are trying to determine what, if any, pitfalls exist in using MPI’s derived data types. We utilize the MPI\_Datatype construct and the MPI\_vector and MPI\_hvector interfaces to perform a matrix transposition and then implemented a “raw” version of this matrix transposition manually for each version of software we are testing. Any versions in Figure 10 or 11 with an “R” in the name for LAM or MPICH are the raw versions that do not use the MPI special data constructs. Neither PVM or BNM have this functionality for transposing matrices. PVM provides generic data types and strided pattern capability for sends, but it is not as versatile as MPI.

From the results in the figures, we can see that it doesn’t make much difference if we use the raw versions or the versions using the special data types. The figures match up with the ring pattern test of the same message size patterns. Square the matrix size and multiply times four (integer matrices) to get the corresponding message size.

### 3.4 Reliability

LAM proved to have the most problems locking up or slowing down unpredictably during testing. One problem we could predict, was that LAM left the UNIX accept sockets open to the daemon after the LAM tasks completed. This, obviously, will eventually cause a program crash when we reach the LINUX file descriptor limit. It seems as if this problem only occurs when spawning with the MPI\_Spawn() command and not when using the mpirun command.

On a few occasion, PVM with PvmDirectRoute option deadlocked for unknown reasons. It was very difficult to reproduce. During the course of testing, we tried many different tests to see what impact they would have. One of the tests we did not discuss was doing ring tests on a large number of nodes. For example, we would start up 255 tasks on our 16 nodes system and then try to perform communications. While the UDP versions handled this fine (for small message sizes), the TCP versions all crashed consistently as a result of the 255 file descriptor limit of LINUX. This should rarely be a

problem on our system, but may be on larger node systems that want to communicate to a central process.

## 4 CONCLUSIONS

We can see from this study of message passing implementations on the Beowulf cluster that the general networking traffic algorithms of TCP do not always provide the best performance. The simple UDP implementation that PVM provides offers superior performance in some cases, while the TCP implementation of PVM shows us what not to do; application level packetization over TCP. We also observed that TCP’s RTT and RTO calculations can lead to artificial estimates and poor performance on a network whose RTT should be extremely small with little deviation.

We determined that the complex data types used by MPI do not cause a degradation in performance. In addition, we found that spawning tasks on the Beowulf cluster should be done with a specialized daemon with an option to spawn multiple tasks on the same node with one request.

From this paper, we discovered that modifications of the kernel TCP implementation can improve communication performance on our network. This has developed a focus for future work relating to modifying TCP’s transmission algorithms to boost performance on the Beowulf. We want to design a kernel patch that allows TCP parameters to be modifiable from the /proc file system and test performance while varying these algorithms. These tests, after obtaining an optimal TCP parameter configuration, should lead directly to an efficient version of MPICH over our system software and BNM. Additional focuses we have along these lines gear towards designing our own reliable protocol over ethernet, bypassing IP completely.

## References

- [1] Henri Casanova, Jack Dongarra, and Weicheng Jiang. The performance of PVM on MPP systems. Technical Report CS-95-301, University of Tennessee, August 1995.
- [2] Karen Castagnera, Doreen Cheng, Rod Fatoohi, Edward Hook, Bill Kramer, Craig Manning, John Musch, Charles Niggle, William Saphir, Douglas Sheppard, Merritt Smith, Ian Stockdale, Shaun Welch, Rita Williams, and David Yip. Clustered workstations and their potential role as high speed compute processors. Technical Report RNS-94-003, NAS Systems Division, NASA Ames Research Center, April 1994.
- [3] Jack J. Dongarra and Tom Dunigan. Message-passing performance of various computers. Technical report, University of Tennessee, January 1997.
- [4] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jiang, and V. Sunderam. *PVM: A Users’ Guide and*

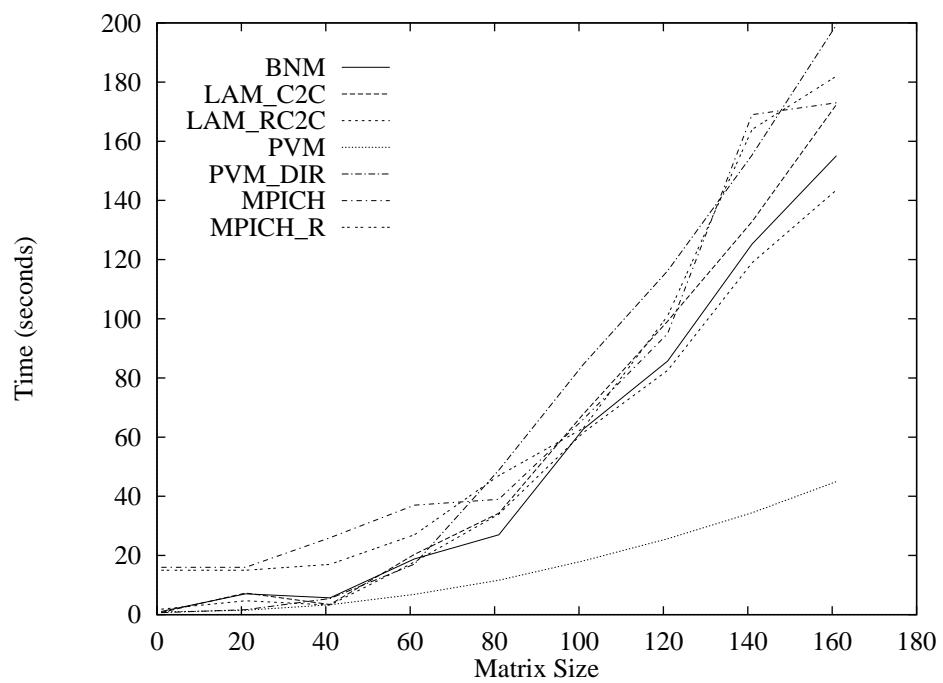


Figure 10: Transpose Tests

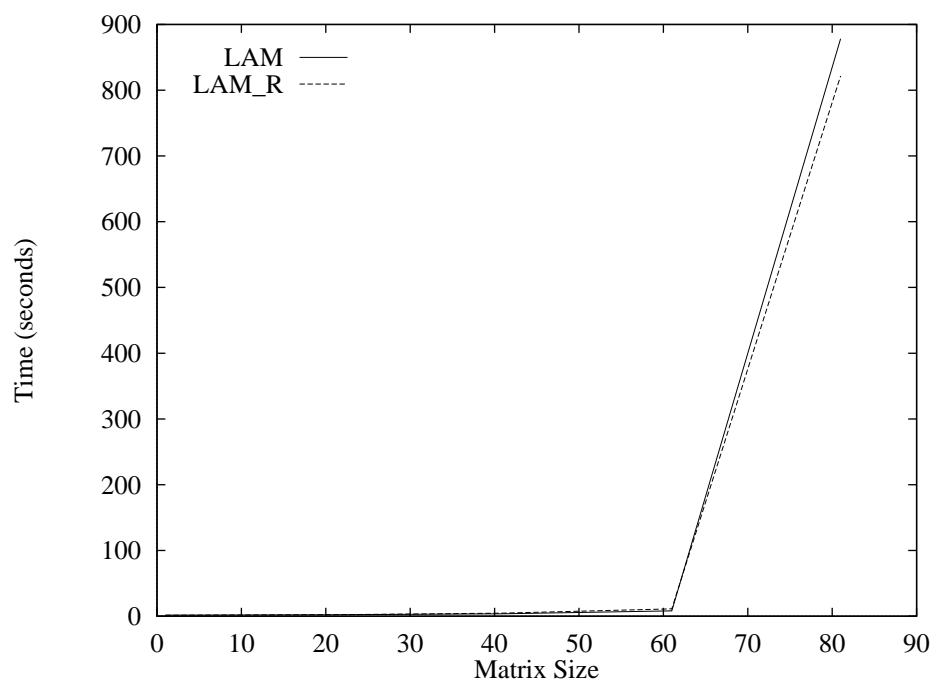


Figure 11: Transpose Tests

*Tutorial for Networked Parallel Computing.* MIT Press, 1994.

- [5] G.A Geist, J.A. Kohl, and P.M. Papadopoulos. PVM and MPI: a comparison of features. Technical Report DE-AC0596OR22464, Lockheed Martin Energy Research Corporation, May 1996.
- [6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
- [7] V. Jacobsen and M. Karels. Congestion control and avoidance. In *Proceedings of ACM SIGCOMM '88*, 1988.
- [8] Vijay Karamcheti and Andrew A. Chien. A comparison of architectural support for messaging in the TMC CM-5 and the cray T3D. In *Proceedings of the 1995 International Symposium on Computer Architecture*, June 1995.
- [9] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 471–480. IEEE Computer Society Press, August 1996.
- [10] Nick Nevin. The performance of LAM 6.0 and MPICH 1.0.12 on a workstation cluster. Technical Report OSC-TR-1996-4, Ohio Supercomputer Center, March 1996.
- [11] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. In *Proceedings of the 1997 IEEE Aerospace Conference*, 1997.
- [12] Bill Saphir and Sam Fineberg. Performance comparisons of MPL, MPI, PVMe. Technical report, NAS Parallel Systems, September 1998.
- [13] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.
- [14] W. Richard Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, 1994.

**Phil Carns** is a student at Clemson University working towards his B.S. in Computer Engineering. He is currently doing research as an honors project for the Calhoun Honors program. His interests are in message passing implementations and socket level interfaces.



**Walter Ligon** received his Ph. D. in Computer Science from the Georgia Institute of Technology in 1992. Since then he has been at Clemson University where he is an Assistant Professor in the Department of Electrical and Computer Engineering. His current research interests are in parallel and distributed systems, I/O for parallel systems, reconfigurable computing, and problem solving environments.



**Scott McMillan** received his Bachelors of Electrical Engineering from the Georgia Institute of Technology in 1992. Currently, he is working towards his M.S. in Computer Engineering at Clemson University. His interests include message passing implementations, parallel I/O, and systems level software.



**Robert Ross** received his B.S. in Computer Engineering from Clemson University in 1994. Currently he is working towards his Ph. D. in Computer Engineering at Clemson University. His interests include parallel file systems, scheduling algorithms, and visualization.

