Efficient MPI Broadcast Communication and File Distribution for Local-Area Clusters

Michael Brim and Joel Sommers
Project Report for CS 739: Distributed Systems

Abstract

Large clusters of commodity workstations have become a popular system architecture for parallel computing tasks. A common need in these clusters is the distribution of files to subsets of the nodes. Without an underlying distributed file system (as is often the case), this task is usually carried out with ad hoc tools, or better, solutions which use parallel programming models. For instance, collective communication operations in MPI can be effectively exploited for cluster file distribution. Distributions of MPI such as MPICH and LAM/MPI, however, implement the `MPI_Bcast` collective operation as serial `MPI_Send` operations, effectively limiting the scalability of this approach. Our work explores both the optimization of MPI collective operations through use of network broadcast mechanisms and the addition of a user-level library to facilitate file distribution across a local-area cluster. Inconsistent (but repeatable) experimental results across two test configurations do not allow us to make wide claims for our methods, but our results do indicate that our mechanisms are more scalable than existing file distribution methods, including those based on the LAM/MPI implementation of MPI.

1 Introduction

The Message Passing Interface (MPI) standard [6] was written in an effort to standardize the programming interface presented to developers across a wide variety of underlying parallel architectures. Since its writing, many implementations of the standard have been produced, including highly-tuned versions for proprietary massively-parallel processors (MPPs) such as the IBM SP and SGI Origin, as well as mostly hardware-independent implementations such as MPICH [1] and LAM/MPI [9], which have been ported to run on a vast array of machine types. In our work, we have focused on the latter type, and specifically the LAM/MPI implementation from Indiana University. Our arguments do not hold for tightly-coupled MPPs, which typically already provide efficient collective messaging through a proprietary message-passing layer.

Given the current trend towards using clusters of commodity systems connected by a high speed network in a rather loosely-coupled MPP as a cost effective alternative to the supercomputers of the past, most of the portable MPI implementations have been made to run on these new systems. However, none of these implementations have addressed the increasing scale of such systems in their treatment of collective operations. Current implementations make use of point-to-point messaging from the source rank[1] to all other ranks in a group communicator in a sequential progression. Although handling group operations in this

[1]Rank is the terminology used in MPI to denote a single process within a group communicator. As a programming construct, ranks are denoted by integers

manner leads to easier implementations and portability, it is obvious that such communication patterns are highly inefficient and by no means scalable.

In reaction to the inherent inefficiency present in current implementations, projects such as MagPIe [10] have examined the use of tree-based message distribution strategies to improve the performance of group operations. Not surprisingly, tree-based distribution shows much greater efficiency than sequential point-to-point communication by greatly reducing the computational and network overhead incurred by the source node and through overlapping of subsequent message transmission. It is our belief, however, that as the scale of clusters increases into the thousands and ten-thousands of nodes, tree-based distribution alone will not be sufficient. When cluster sizes reach those mentioned, it is easy to imagine that these systems will be organized as clusters of clusters, which readily adapts to tree-based distribution. However, we believe that even in this type of organization, much efficiency is lost at the lowest level, where communication between nodes will typically be occurring in a local-area network (LAN). Studies have shown that the use of reliable communication protocols such as TCP on LANs incurs much greater overhead than is necessary to provide reliable transmission, thus limiting the performance of parallel applications [13][11]. Therefore, our design includes the addition of a reliable multicast subsystem to the current LAM/MPI implementation for use in collective operations. We believe that the use of the subsystem will provide even greater efficiency in both computational and network utilization than sequential point-to-point and the current tree-based distribution mechanisms such as MagPIe.

One of the goals in producing the proposed communication subsystem is that it be efficient for data sizes ranging from very small (i.e., a few bytes) to very large (i.e., a few gigabytes). The reasoning behind this design decision stems from another goal we have for our project, that being to provide a high-efficiency mechanism for doing file distribution to groups of nodes in a cluster. The motivation behind this goal is similar to the goals of the MPI standard itself in that it is an attempt to provide programmers a consistent method for file distribution across many underlying hardware platforms by utilizing the communication systems already present in current MPI implementations. Note, however, that we are not proposing an alternative for high-performance I/O in MPI applications, as much work [15][14] has already focused on this subject. We advocate the addition of a library for "whole file" group distribution that makes use of our efficient group broadcast subsystem.

This paper proceeds by first elaborating the design and implementation of our system in Section 2. Included in this discussion is a detailed description in Section 2.2 of the multicast communication subsystem. The programming interface to file distribution is described in Section 2.3, along with the corresponding sequence of actions that can be expected by a programmer when using the interface. We describe the evaluation of our methodology and present results in Section 3. Section 4 follows with a discussion of work related to ours, and we conclude in Section 5.

2 System Design and Implementation

2.1 Design Goals

The reliable multicast communication subsystem is invisible to the programmer, as it is intended as a drop-in replacement for the current implementation of `MPI_Bcast` in LAM/MPI. LAM/MPI provides a fairly straightforward mechanism for implementing new low-level communication layers. The mechanism was originally intended to encourage various proprietary systems vendors

from 0 to one less than the total number of processes within the communicator.

to port LAM/MPI to their systems by allowing replacements to make use of any high efficiency message-passing layer provided by the system. In addition, any other functions whose implementation relies upon the calling of `MPI_Bcast` will benefit from our optimized subsystem when in use.

The file distribution interface is made available to programmers as part of a shared library for use in MPI applications. A stand-alone command-line tool has also been developed for use by non-MPI applications. There are many obvious examples of different systems that could benefit from the use of the file interface. The most straightforward use would be as part of a typical message passing program, wherein a programmer wishes for all nodes participating in the computation to have access to some data file, and as such could distribute the file to all ranks as the first step. The previous example might remind some of another obvious use with the same purpose, that being a file staging mechanism for batch queueing systems. An efficient file distribution utility has also been one focus of research [2] into the administration and ongoing maintenance of ever-growing clusters, where shared filesystems are not practical due to lack of scalability and server resource contention. In such systems, the mechanism could be used to synchronize updates to the system's configuration or install new nodes through propagation of system images [3].

2.2 Reliable Multicast Communication Subsystem

2.2.1 Overview

As noted in the MPICH FAQ [1] and also in the LAM/MPI reference, `MPI_Bcast` does not perform true multicast, but performs a point-to-point transfer for each message. This point-to-point communication is performed through TCP connections between each process in a communication group. The MPICH FAQ further points to problems in IP multicast implementations and complications arising from implementing a reliable multicast as reasons for using point-to-point communication for all messages.

Our multicast design assumes only a broadcast local network, such as Ethernet. Specifically, we do not rely on the presence of true IP multicast on the network. Instead, we use the broadcast capabilities of the UDP socket interface and construct an application-level notion of multicast. As noted previously, no modifications are required of application code to take advantage of our design.

LAM/MPI uses two separate communication interfaces: daemon-to-daemon and client-to-client. These interfaces characterize the degree to which LAM daemons are involved in MPI communications. The daemon-to-daemon interface specifies that all MPI communication is routed through the LAM daemon present on each cluster node. In contrast, the client-to-client interface enables more efficient communication by allowing MPI processes to communicate directly with each other. For our design, we implemented a new client-to-client (c2c) interface. The c2c is described by the request progression interface (RPI), which is subdivided into device-independent and device-dependent layers. We have modified the existing TCP-based RPI layer to include our UDP broadcast protocol. In addition, the existing implementation of collective operations in LAM/MPI explicitly performs point-to-point `MPI_Send` operations for each group member. We modified higher layers of LAM/MPI so that our RPI functions are used for `MPI_Bcast`.

The overall effect of our changes is that messages resulting from MPI group operations are actually sent as broadcast UDP packets. Each process listens to a specified port and thus receives all group traffic on the broadcast LAN. The port number used

version (8)	flags (8)	type (16)
rank (32)		
fragment sequence (32)		
message identifier (32)		

Figure 1: UDP Broadcast Header Layout (field sizes shown in bits)

is statically defined but may be overridden with an environment variable.

Since multiple groups may be in operation in the LAN cluster, we filter messages based on the context descriptor defined in each MPI communicator object. Each receiving process simply discards messages that are not intended for it. Process groups that are not logically related may simply use a different port to avoid the possible problem of one group having a negative performance impact on another group because of socket buffers becoming filled with unwanted messages.

2.2.2 Subsystem Implementation Details

The primary problem to solve in our communication architecture is reliable delivery of broadcast messages. We confine all reliability issues internal to our modified RPI. These low-level modifications are limited to three source files, and consist of approximately 1000 new lines of code. In this section we detail the basis and operation of our protocol.

Each process in LAM/MPI is statically linked with our modified library. At the same time the full-mesh of TCP connections is created for interprocess communication, a UDP socket is created and configured for broadcast capability. Messages sent via UDP are prepended with a header containing a version number, a message type, a rank identifier, a message identifier, a fragment sequence number, and a field for flags. In addition, the normal envelope header used in LAM/MPI is transmitted as header data. This envelope includes the LAM/MPI group context identifier. A diagram of the header data is shown in Figure 1.

In the header, the version identifier is currently 1, and is included to permit smooth evolution of the protocol. There is only one flag currently defined: the "more fragments" flag. The type field indicates whether the message is data, a positive acknowledgement or a negative acknowledgement. The rank field is used in congestion control and reliability, and the fragment sequence and message identifier elements are used for providing reliable, in-order delivery. Use and semantics of the header fields are explained below (with the exception of the version field.)

Since the length of the data passed in a `MPI_Bcast` may exceed the Ethernet frame size, we must fragment the application data at the sender and reassemble the fragments at each receiver. The two header elements, message identifier and fragment sequence number, provide a way to deliver and reassemble fragments into messages. The message identifier field is similar to the "identification" field in the header of IP packets, serving to identify fragments belonging to a given application message. The fragment sequence gives a byte-offset within a particular message. All fragments but the last in a message are marked with a more fragments bit in the flags field of the header. The fragment size is statically defined, but may be overridden with an environment variable to match the MTU for different broadcast LANs.

Our reliability technique, which we now detail, works similarly to Rizzo's `pgmcc` [12] and is influenced to a lesser extent by SRM [5]. `pgmcc` is a single-rate congestion control algorithm for the PGM multicast protocol. In `pgmcc`, a congestion control algorithm is employed between the sender and one designated member (the "acker") of the receiver pool. Positive acknowledgements are used for round-trip time calculations and for feedback to the the window-based congestion control algorithm. Negative acknowledgements, in addition to causing the sender to retransmit lost data, are used to determine the current acker. The intent is for group progress to proceed at the rate of the slowest receiver. A dampening factor is employed to avoid rapidly changing the current acker.

In our algorithm, we also employ both positive and negative acknowledgments. Receivers send negative acknowledgments if they receive an out-of-sequence fragment. Receivers send positive acknowledgments for the last fragment of every application message. In addition, one receiver (also called the "acker") sends an acknowledgment for every fragment successfully received in order. The acker is designated in the rank field of the message header. The result is reliable message transfer which is synchronous with the application code call to `MPI_Bcast`. It is synchronous because the sender does not exit its send loop until it has received a positive acknowledgment from each receiver. All acknowledgments are unicast to the sender rather than broadcast to the group.

Additionally, if a receiver does not receive a fragment within a configurable amount of time, it sends a negative acknowledgment. Likewise, if a sender does not receive final acknowledgments from receivers within a configurable amount of time, it retransmits the final fragment. Both of these timers are statically defined, but may be overridden by environment variables. By default, the receiver timeout is set to 100 milliseconds and the sender timeout is set to 50 milliseconds. These are rather conservative settings considering typical round-trip times commonly experienced on LANs, which are usually on the order of one millisecond or less for a 100Mbps Ethernet.

Retransmission uses a go-back-N policy: if a sender receives a negative acknowledgment, it begins transmitting data at the sequence number identified in the NACK. Like SRM, the sender makes an attempt to coalesce NACKs from multiple receivers by waiting a small amount of time (also configurable by an environment variable) before beginning transmission again. Retransmissions are broadcast to the entire group rather than specifically directed to those ranks which have sent NACKs.

As in `pgmcc`, the acker is not a fixed receiver rank, but changes depending on the most recent rank to send a NACK. The intent is to cause group progress to proceed at the rate of the slowest receiver. Currently, we do not employ hysteresis to avoid rapidly changing the acker. This has not proved to be a problem in practice (in fact, the acker has rarely changed), but we have only dealt with relatively small group sizes.

In our first implementation, we did not include any congestion control mechanism. Initial experiments revealed that the most common location of fragment loss was at the sender: sending rates were high enough to cause buffer overrun so that packets never made it onto the wire. In the second implementation, we used the congestion control mechanisms employed in `pgmcc`. We did not initially foresee congestion control to be an integral part of our modified RPI, but it has proved important.

Like `pgmcc` and TCP congestion control, we track two variables: a congestion window value (*cwnd*) and the number of packets in flight. These values are in terms of numbers of fragments, rather than in numbers of bytes as in TCP. While the number of packets in flight is less than *cwnd*, packets may be sent. When an ACK is received, the number of packets in flight is decremented and *cwnd* is updated based on its relationship to the value of *ssthresh*: if *cwnd* is less than *ssthresh*, it is incremented by one, but if it is greater than or equal to *ssthresh*, it increased by (1 / *cwnd*). This policy mimics the TCP slow-start and congestion-avoidance phases. When a NACK is received, *cwnd* is cut in half, and the next (*cwnd* / 2) ACKs are ignored in *cwnd* calculations. Our congestion control mechanism differs from `pgmcc` in that we do not estimate round-trip time (in pgmcc, RTT is in terms of packets rather than an actual amount of time). Also, as noted above, we do not measure individual receiver loss rates for deciding who the current acker should be. Initially *cwnd* is set to a 1 and *ssthresh* is set to 8 (each of which may be overridden by an environment variable.) As in pgmcc, *ssthresh* is a static value. We have not experimented with dynamically changing *ssthresh*.

In sum, the benefits to our methodology include avoidance of persistent buffering within our RPI because of our simple synchronous model of reliability, and the use of a congestion control mechanism to avoid excessive loss. The major drawback of our approach is related to the simplicity of our synchronous reliability approach, in that we require active communication from each member in the group for the final acknowledgment of each message. Though overall packet transfers are reduced from using point-to-point TCP connections, we are susceptible to an ACK implosion effect with very large group sizes. Addressing this problem is a subject for future work.

2.3 File Distribution Interface and Implementation

File distribution is available to MPI programs through the inclusion of a shared library written in C that is statically linked with the application. The library is completely independent of the underlying MPI implementation, and as such should be compatible with all current implementations that comply to version 1.0 of the MPI standard [6]. The file distribution interface presented to programmers is similar in style to all other MPI collective library calls, with the parameters consisting of a group communicator, a source filename, an optional destination filename, and a flag specifying the behavior desired if the destination file already exists. If no destination file is provided, the full pathname of the source file is assumed to be the destined file location. The two methods for invoking the file distribution mechanism are shown below. Both methods return zero upon success and one to indicate failure.

```
int Bcast_File1(MPI_Comm comm, char *srcfile, char *dstfile, int write-flag[2]);
int Bcast_File2(MPI_Comm comm, char *srcfile, int write-flag);
```

As is the case for all MPI collective operations, each process in the group makes the call to Bcast_File, and the resultant actions taken by each process are determined by its rank. Upon invocation, the source rank (rank 0) will first check to see if the passed file(s) was specified as a relative pathname(s). If so, it first obtains a full pathname for the source file (and destination if provided) and continues as follows. The source rank then uses MPI_Bcast to send a short status packet to the group that contains the following information: full pathname of the source file, current timestamp for the source file, current source file size in bytes, full pathname of destination file if different from source file, and the specified overwrite flag value. When the MPI_Bcast for the status packet has completed, the source rank proceeds to read the file and multicast fixed-size data buffers to the group using MPI_Bcast. The size of the buffer is set at compile time so that the source and destinations can agree on the amount of data being transferred per call to MPI_Bcast. Each non-source rank performs a lookup on the enclosed filename(s) to determine the appropriate course of action (see below cases). Note we assume that if a shared filesystem is present, we expect the mount points for the filesystem to be consistent across nodes in the cluster.

Case 1: Bcast_File invoked with source file and no destination

1. If the source file exists on a destination rank and has the same timestamp, the file is assumed to be located on a shared

[2] write-flag can assume the following values: 1=leave as is (default), 2=overwrite if newer, 3=overwrite

filesystem and no further action is performed.

2. If the source file exists and has a different timestamp or the file does not exist, the destination rank process will continue listening on the channel to receive the new file data and write it to the file.

Case 2: `Bcast_File` invoked with both source and destination files

1. If the source file exists and has the same timestamp, it is assumed that the file is on a filesystem shared with the source rank, which is responsible for writing the destination file.

2. If the source file exists and has a different timestamp or the file does not exist, it is assumed the destination file is not on a filesystem shared with the source rank. The destination rank is responsible for listening to the channel for new file data and writing the destination file according to the specified overwrite functionality.

One impact of our implementation has been glossed over in the above cases. Since the source rank invokes `MPI_Bcast` to send file data, all other ranks must do the same according to collective operation semantics. This requirement seems non-optimal in that even destinations who do not require updates will be forced to incur the overhead of receiving the file data. However, other approaches considered where the receiving group is reduced to only those destinations requiring update have the negative quality of necessitating additional negotiation protocols between the source and destinations that only serve to increase the overall latency and overhead incurred.

In addition to the shared library for collective file distribution, a command-line tool has been developed to allow non-MPI applications to make use of the file distribution mechanism. The command-line tool is simply a wrapper that invokes an MPI application whose sole purpose is to distribute the named file to a group of cluster nodes. The MPI application operates similarly to the process described above, but is also responsible for the required MPI initialization and finalization.

3 Performance Evaluation

3.1 Experimental Framework

To evaluate our modifications to LAM/MPI, we have compared the performance of an unmodified LAM/MPI to one which contains our modifications for optimized `MPI_Bcast` operations and file transfer. We focus on two sets of benchmarks: the first is a microbenchmark that compares the performance of a simple MPI program which calls `MPI_Bcast` for various data sizes and in different group sizes, and the second is an application-level benchmark that uses the file distribution library while varying file and group sizes.

In addition to the above benchmarks, the file distribution library performance using unmodified LAM/MPI is compared to the performance of the C3 cpush tool [2] using both scp and rcp. The purpose of this benchmark is to validate the assumption that a file distribution mechanism using MPI as its communication substrate is a viable alternative to current methods. The comparison is made for varying file and group sizes.

The metric we focus on for each benchmark is latency. Throughput is directly related to latency and transfer size, therefore we simplify our analysis by examining transfer latency only. All times reported have been measured using the time function provided by csh, and represent the arithmetic mean across three tests. We are unable to quantify goodput since we do not have access to trace packet-level events for the TCP (unmodified LAM/MPI) transfers to determine loss rates.

Our experiments are run on two different testbeds. The first testbed is composed of dedicated nodes selected from the departmental Linux cluster known as c2. Each cluster node contains dual Pentium III 933Mhz processors, 1GB of memory, and an Intel EtherExpress Pro 100 Ethernet adapter. The c2 cluster uses a Cisco Catalyst 4006 switch for physically connecting the nodes. The second testbed was constructed from various Linux workstations located in the instructional tux lab. Each workstation contains a single 800Mhz Pentium III, 256MB of memory, and a 3Com 3c905C Ethernet adapter. The instructional workstations are connected by multiple Intel 501T 100bT Ethernet switches. The Linux kernel version is 2.4.18 for each testbed.

3.2 Results

3.2.1 Multicast Library Microbenchmarks

Figure 2 shows results for the multicast library microbenchmarks run on the tux workstations. The benchmarks are run with two different block sizes (i.e., the size of the data presented to `MPI_Bcast`.) The left graph of Figure 2 shows results for a 1kB block size and the right graph shows results for a 32kB block size. Individual runs are labeled as "pt2pt" or "mcast", indicating whether the run used an unmodified LAM/MPI library or our modified broadcast-capable library, respectively. We used total transfer sizes of 100 bytes, 1kB, 10kB, 100kB, 1MB, 10MB, and 100MB. With sizes under 100kB, there is little difference between the two implementations, so results are only shown for data sizes from 100kB and greater. These transfer sizes are marked along the x-axis, which is presented in log scale with a base of 10. The y-axis marks latency of the transfer in seconds. The tests are run with group sizes of 8, 16, and 32 nodes.

In all cases, our library delivers lower latencies than are provided by the standard LAM/MPI implementation of `MPI_Bcast`. The results show, for both block transfer sizes, the incremental cost of larger group sizes when using serial, point-to-point transfers as with standard LAM/MPI. With our modified transport layer, the incremental cost of larger group sizes is greatly lessened. Also interesting to note is that in the 32kB block size case, the transfer latency for 32 nodes using our transport is roughly the same as for 8 nodes using an unmodified LAM/MPI.

3.2.2 File Distribution Library Benchmarks

One of the claims of our work is that people would benefit from a standard method for doing group file distribution using MPI. However, MPI is designed to be efficient at sending small amounts of data in the form of messages, while files tend to be large compared to messages. Thus we were eager to compare the performance of our MPI-based distribution technology with that of another current tool designed for file distribution. As such, the first set of measurements collected for evaluation of the file distribution library were for comparison with another current tool used for the same purpose, the C3 cpush tool. The cpush tool's design is quite simple. For every destination node specified in a configuration file, the source node forks a process to send the file data to that node. By default, cpush uses rsync to perform the file transfer, and an environment variable is used to determine whether rsync should make use of rsh or ssh to transfer its data. Using rsync has the advantage that if the destination file already exists, then only the differences between the source and destination files is transferred. Since the file distribution

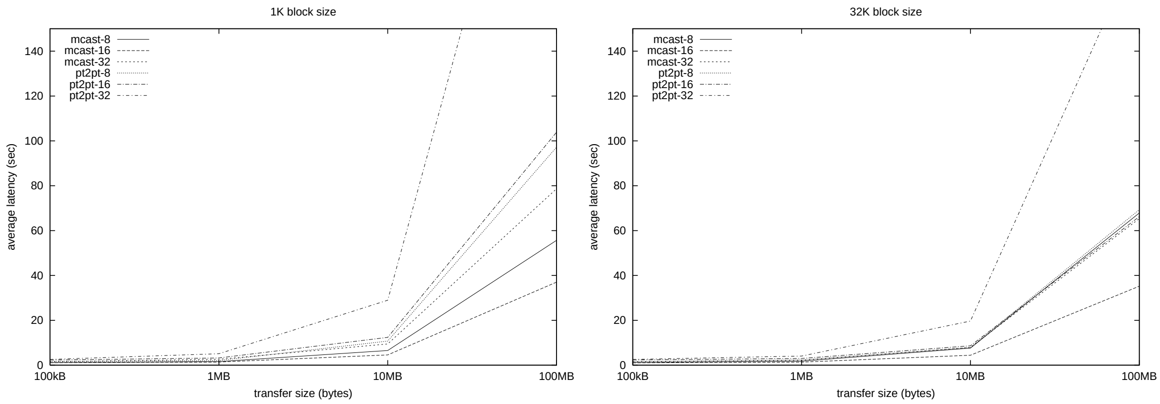

Figure 2: Tux Microbenchmark Results: Standard vs. Custom MPI_Bcast

library is focused towards whole-file distribution, the cpush tool was slightly modified for the purposes of generating a fair comparison. In all our tests, we assume that the destination file does not exist, and therefore rsync will only result in excess overhead as compared to just using rcp or scp to transfer the file. We therefore modified cpush to have each forked process directly call rcp or scp. The modified cpush was then compared to the command-line tool version of our library for varying file sizes and numbers of destinations run on the departmental Linux cluster.

The results from this comparison are summarized in Figures 3 and 4. Although the file sizes used are the same as the transfer sizes from the microbenchmarks, the results for files smaller than 1MB are not reported since the latencies are widely variable but still insignificant up to 1MB. Figure 3 shows the performance of cpush using both rcp and scp. One interesting result shown in this figure is that scp actually outperforms rcp for large files, which is contrary to the intuition that scp would have a disadvantage due to the overhead necessary to encrypt the file data. For smaller files, there is no clear performance winner. Figure 4 provides a comparison of cpush using scp to our file distribution tool (denoted as ”fbcast” in the graphs) using standard LAM/MPI. From the figure it is clear that the performance of our tool using MPI far exceeds that of cpush. For larger file sizes, cpush incurs linear increases in the time required for increasing numbers of nodes. In contrast, our MPI-based tool shows no such trend. In fact, the time needed to distribute to 32 nodes using our tool is consistently less than the time to send the same file to only 8 nodes using cpush. The implication of these results is that MPI-based file distribution is a viable and cost-effective alternative to current tools, even when making use of unmodified implementations of MPI.

Our next set of evaluations compared the performance of the file distribution library using the standard LAM/MPI `MPI_Bcast` versus that with our modified multicast enabled version. The results reported here are from tests performed using the tux instructional workstations. Similar tests were completed on the c2 cluster, where our file distribution library performed similarly when used with the multicast enabled `MPI_Bcast`. Unfortunately, however, the results for the standard MPI implementation performance were quite different from those obtained on the tux workstations. In the next section, we will attempt to justify these differences based upon analysis of the network performance of each testbed. The results from the tux experiments are given in Figure 5. In this figure, we show the performance of our distribution mechanism with the modified `MPI_Bcast` using a 32kB buffer for varying file sizes and numbers of destinations, as well as the performance when using the standard implementation for 32 destinations. Our results indicate that the performance improvements observed for the tests of the modified LAM/MPI

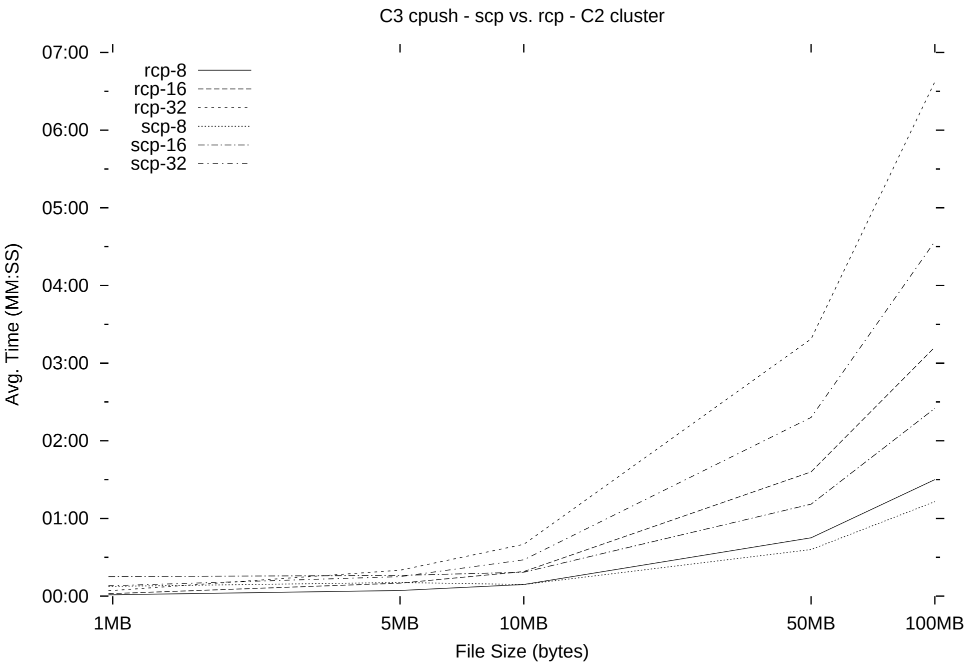

Figure 3: C2 Benchmark Results: C3 cpush - scp vs. rcp

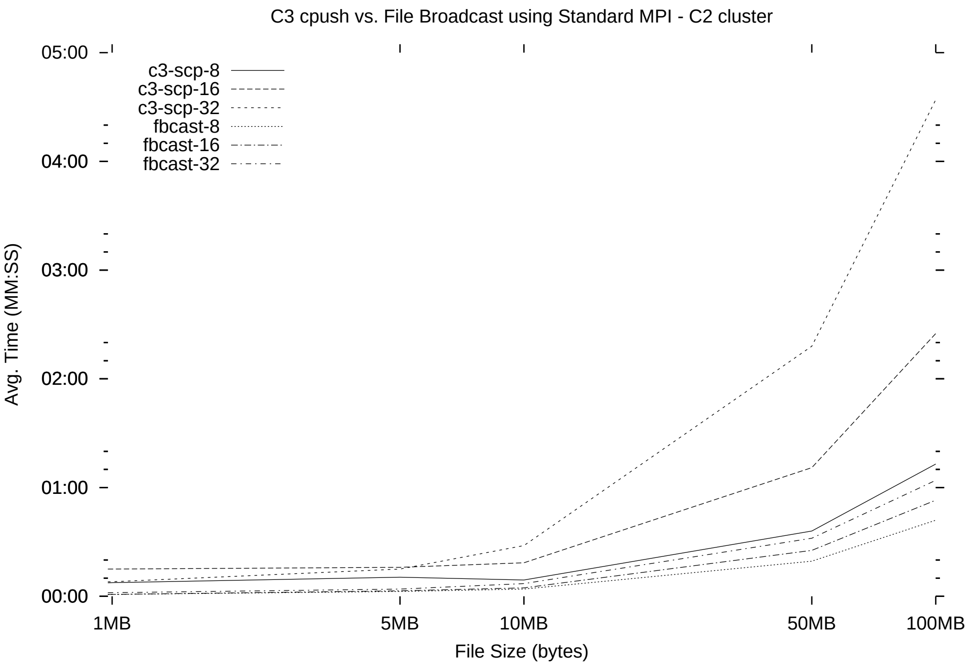

Figure 4: C2 Benchmark Results: C3 cpush vs. File Broadcast using Standard MPI

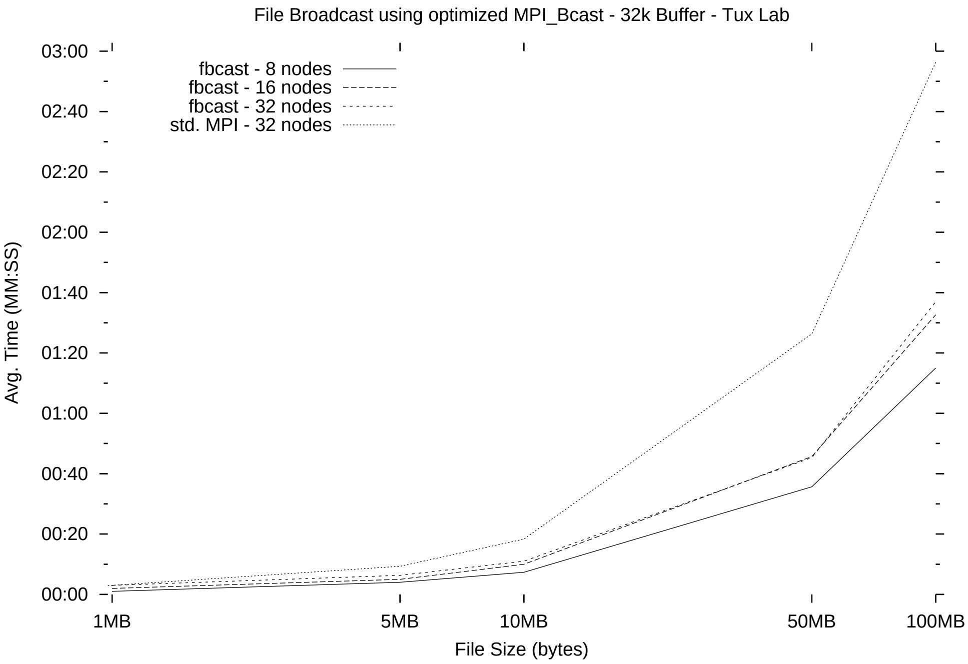

Figure 5: Tux Benchmark Results: File Broadcast - Optimized MPI_Bcast - 32kB Buffer

implementation given in the previous section are once again verified when considering its use by the file distribution library. There is a noticeable overhead for the file distribution library when comparing these results to those in the 32kB buffer graph of Figure 2. This overhead can be attributed to the fact that the microbenchmarks presented in the previous section do not account for actions necessary as part of normal file distribution by the library. Examples of extra activities include the need to send additional information such as the status packet, as well as the reading of data from the source file and the corresponding write on each destination. The figure also provides promising insight into the scalability of our library with the enhanced `MPI_Bcast`. The key observation to be made is the change in time due to increasing the number of destination nodes. As is shown, going from 16 to 32 destinations has little to no impact on the time required, suggesting that file distribution will scale with the enhanced `MPI_Bcast`. Referring again to Figure 4, we can see that increases in the number of nodes for the library using standard MPI do result in measurable distribution time increases. It is therefore our conclusion that as cluster sizes continue to grow, our enhanced `MPI_Bcast` will continue to improve relative to the standard implementation, and as a result, so will the performance of the file distribution library.

3.2.3 Network Performance of the Testbeds

The results presented in the previous two sections for comparing the standard implementation of `MPI_Bcast` versus our UDP-broadcast enabled version have been reported for only the tux testbed, due to discrepancies between the performance of the standard LAM/MPI implementation on each testbed. Specifically, the MPI performance reported when comparing the use of C3 cpush and the file distribution library using standard MPI on the c2 cluster for 32 nodes and that observed for the enhanced `MPI_Bcast` microbenchmarks on the tux workstations is quite different. For example, while the distribution of a 100MB file on the c2 cluster takes just over a minute, the transfer of the same amount of data takes nearly three minutes on the instructional

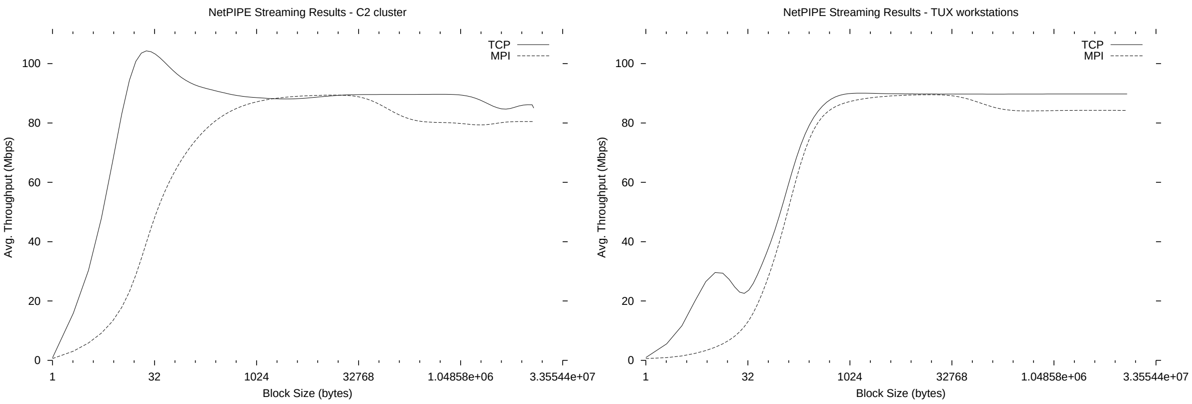

Figure 6: Network Performance of C2 & Tux Testbeds

workstations. Indeed, the results when comparing our modified LAM/MPI versus the standard implementation on the c2 cluster actually showed that our modifications had no positive effects on the performance of either `MPI_Bcast` or the file distribution library. Across the entire range of block sizes, file sizes, and numbers of nodes the results showed no difference or a small overhead penalty for the case of our modified implementation.

In an attempt to explain the differences in results for the two testbeds, we decided an analysis of the network performance on each testbed might provide useful information. Figure 6 shows the network performance for each testbed using both TCP and MPI. The results were generated using NetPIPE [7], a network protocol independent performance evaluator developed at Ames Laboratory. The graphs show the average throughput achieved for increasing block sizes when streaming the data from source to receiver. Under 1kB, the c2 cluster appears to exceed the 100Mbps limit of its underlying network, but the results for small block sizes are less accurate due to the precision of the timing mechanism used when compared to the block size. The same reasoning can be used to explain the local maxima shown for the tux workstations at around 8 to 16 bytes. The c2 testbed also exhibits poor consistency when the block size exceeds 1MB.

Unfortunately, the analysis has not provided any further explanation of the exact cause for the discrepancies between the two testbeds. Further in depth study of this phenomena is necessary to determine the exact cause for the observed differences in performance of the standard LAM/MPI implemenatation.

4 Related Work

A novel aspect of our work is that we have combined existing ideas from application-level multicast and have applied them in the realm of cluster computing. A long-standing deficiency of message-passing libraries is that group messages are actually delivered point-to-point. With an increasing shift towards clusters constructed from commodity PCs and broadcast-capable networks, delivering messages point-to-point incurs needless overhead.

Chen et al. [4] modified `MPI_Bcast` in MPICH to perform IP multicast, but their approach is undesirable because it does not offer reliable delivery of messages. Upon a call to `MPI_Bcast`, all receivers inform the sender of their ability to start receiving. The sender then sends out data, fragmenting messages as necessary to comply with Ethernet maximum frame sizes. There are no additional mechanisms described. They assume that the mere fact a receiver is ready to receive a message is assurance enough that the message will be reliably delivered. Additionally, their tests are based on low message sending rates. High traffic situations would likely reveal the inability of their protocol to deliver messages reliably.

The MagPIe [10] extensions to MPICH are designed to provide efficient group operations over a wide-area cluster. A tree-based algorithm is used to minimize local-area transmission latency. However, messages between tree nodes are transmitted point-to-point, as in an unmodified MPICH. We take the approach of optimizing local-area group operations, thus our modifications can be used in concert with MagPIe to improve overall group operation throughput on wide-area clusters.

Our application-level multicast design is influenced by the Tibco Rendezvous© product from Tibco Software, Inc. [8]. The similarity is in using a broadcast medium and message tags and separate ports to partition sender and recipient groups. Rendezvous implements a reliable ("certified") transport above its base messaging layer, but is rather heavyweight. In addition, Rendezvous does not include any notion of congestion control.

For multicast reliability, we draw upon Rizzo's `pgmcc` [12] and on Floyd et al.'s Scalable Reliable Multicast (SRM) protocol [5]. SRM defines an efficient receiver-based reliable multicast protocol. Though the protocol is intended for IP multicast, the algorithms described are applicable to our use in broadcast UDP. Specifically, we assimilate the ideas of receiver-based loss recovery, and waiting a random amount of time before sending a NACK. Our design diverges from SRM because we do not assume dynamic group membership. SRM is designed to deal with a highly dynamic sender and receiver pool.

Our protocol is based upon the reliability and congestion control algorithms in `pgmcc`. In `pgmcc`, there are both positive and negative acknowledgments. Positive acknowledgments provide feedback to the sender for congestion control. Negative acknowledgments indicate to the sender that a receiver is missing some data. The sender tracks negative acknowledgments in order to determine which receiver should be the acker. Effectively, the acker is the slowest receiver, and is the limiting factor to overall multicast throughput. We differ from `pgmcc` in that we do not perform round-trip time or loss measurements in determining the current acker.

5 Conclusions

We have explored efficient file distribution in a local-area workstation clusters along two lines: through the creation of a simple API and user-level library, and through a new implementation of the underlying transport of `MPI_Bcast`. Our results are suggestive, but not conclusive. They suggest that our collective communication will scale better with larger clusters than do current MPI and non-MPI methodologies. They are not conclusive in that for our test configurations, file distribution using our implementation of `MPI_Bcast` does not always yield lower transfer latencies than from an unmodified MPI distribution, even though our implementation performs similarly under both testbeds. Further experiments and investigations will be necessary to validate our methodology.

There are a number of directions which remain to be explored in the future. The most immediate direction is further investigation of the performance disparity experienced in our two test environments. Additionally, LAM/MPI is not the only widely used free implementation of MPI, so one task is to port our local-area broadcast mechanisms to the MPICH implementation. With faster local-area networks and larger send and receive buffers, there will be a growing need for more adaptive techniques within our congestion control algorithms. The *ssthresh* value, for instance, could vary dynamically similar to how it changes in TCP. Other improvements to the multicast subsystem are easily imagined.

References

[1] Argonne National Labs, et. al. MPICH—a portable implementation of MPI.

[2] Michael Brim, Ray Flanery, Al Geist, Brian Luethke, and Stephen Scott. Cluster command & control (C3) tool suite, 2001.

[3] IBM Linux Technology Center. System Installation Suite (SIS).

[4] H.A. Chen, Y.O. Carrasco, and A.W. Apon. MPI collective operations over IP multicast. In *Proceedings of the 3rd Workshop on Personal Computer-based Networks of Workstations (PC-NOW 2000)*, 2000.

[5] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6), December 1997.

[6] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[7] Iowa State University Research Foundation. Netpipe network protocol independent performance evaluator, 1999.

[8] Tibco Software Inc. Tibco Rendezvous.

[9] Indiana University. LAM/MPI.

[10] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.

[11] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, /1997.

[12] Luigi Rizzo. pgmcc: a TCP-friendly single-rate multicast congestion control scheme. In *Proceedings of SIGCOMM 2000*, pages 17–28, 2000.

[13] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local-area communication with fast sockets. pages 257–274, 1997.

[14] K. Stockinger and E. Schikuta. ViMPIOS: A truly portable MPI-IO implementation, 2000.

[15] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, 1999.