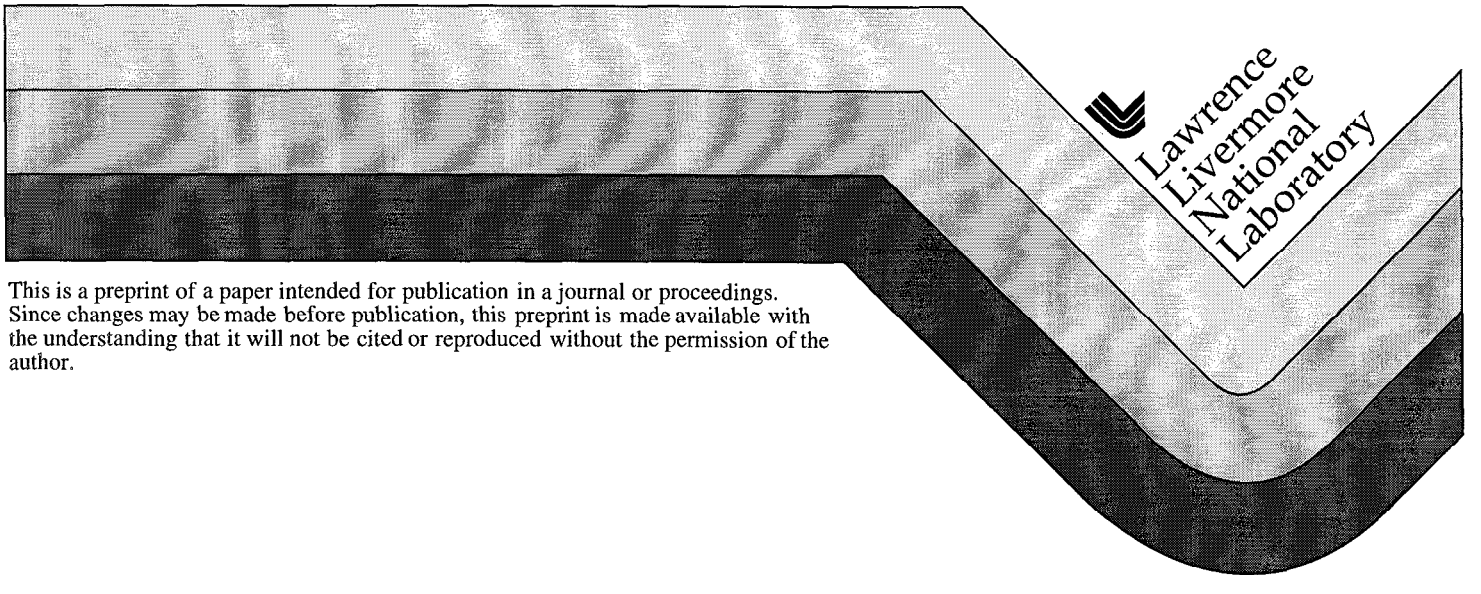
UCRL-JC-133177
PREPRINT

The Quandary of Benchmarking Broadcasts

B.R. de Supinski
N.T. Karonis

This paper was prepared for submittal to the
8th International Symposium on High Performance Distributed Computing
Redondo Beach, CA
August 3-6, 1999

February 5, 1999



The Quandary of Benchmarking Broadcasts

Bronis R. de Supinski
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
bronis@llnl.gov

Nicholas T. Karonis
High-Performance Computing Laboratory
Department of Computer Science
Northern Illinois University
DeKalb, IL 60115
karonis@niu.edu

Abstract: A message passing library's implementation of broadcast communication can significantly affect the performance of applications built with that library. In order to choose between similar implementations or to evaluate available libraries, accurate measurements of broadcast performance are required. As we demonstrate, existing methods for measuring broadcast performance are either inaccurate or inadequate. Fortunately, we have designed an accurate method for measuring broadcast performance.

Measuring broadcast performance is not simple. Simply sending one broadcast after another allows them to proceed through the network concurrently, thus resulting in accurate per broadcast timings. Existing methods either fail to eliminate this pipelining effect or eliminate it by introducing overheads that are as difficult to measure as the performance of the broadcast itself. Our method introduces a measurable overhead to eliminate the pipelining effect.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract no. W-7405-Eng-48.

MPI collective communication operations allow communication involving several tasks to be specified with a single set of function calls. Benchmarking these collective communications is important. Accurate measurements allow implementers to evaluate different algorithmic choices. Users could use the benchmarks to choose between different available implementations. In addition, accurate and complete measurements could guide use of a given implementation to improve application performance. In short, the parallel processing community needs accurate, succinct and complete measurements of the performance of collective communications.

Benchmarking collective communications is a hard problem. Since collective communications inherently involve multiple tasks, complete characterization of their behavior could require an overwhelming amount of data. Worse, accurate measurement of their behavior is difficult due to the possibility of concurrency between successive collective communications. Some benchmarks use knowledge of the communication algorithm to predict the timing of events and, thus, reduce concurrency between the collective communications that they measure. However, accurate event timing predictions are often impossible since network delays and local processing overheads are stochastic. Further, reasonable predictions are not possible if source code of the implementation is unavailable to the benchmarker.

We focus on measuring the performance of broadcast communication. First, we discuss the performance properties of collective communications, based on a model derived from the LogP communication model. Then, we demonstrate that several methods previously used to measure broadcast performance not only fail to measure several important properties but can inaccurately measure the properties that they do measure. Finally, we present our accurate method for measuring broadcast performance and discuss how to extend it other MPI collective operations.

Section 1: Modeling Collective Communications

A method to benchmark implementations of collective communications needs to measure several properties. Almost all collective communication benchmarks attempt to measure the time required to complete the communication, from its first send until its last receive. Although this is an important quantity, these methods overlook several other important properties, such as local processing overheads and the potential to overlap computation with communication. In this section, we use a model of collective communications based on the logP model to characterize the important performance properties of collective communications

In the LogP model, four parameters capture point-to-point communication [4, 5]. The send overhead, $\mathbf{o_s}$, is the time during which a processor is sending a message, while the receive overhead, $\mathbf{o_r}$, is the portion of the time that a processor is receiving a message that cannot be overlap with the message transmission. **L**, is the (wire) latency, the time that a message actually spends in transit from its source to its destination; the more conventional definition of message latency is equal to $\mathbf{o_s + L + o_r}$. The final parameter, the gap, **g**, measures the ability to overlap computation and communication while fully utilizing the communication system and is equal to the minimum interval between consecutive message sends or receives.

We extend the LogP model to capture collective communications more accurately. Our extensions apply to both asynchronous (the collective communications in MPI that take a root argument) and synchronous collective communications [3]. The per processor overhead is the time, $\mathbf{o_i}$, spent sending and receiving messages by each processor, **i**, that participates in the collective communication. The per processor gap is the minimum interval of time, $\mathbf{g_i}$, between consecutive occurrences of the same collective communication at processor **i**. Finally, each message used

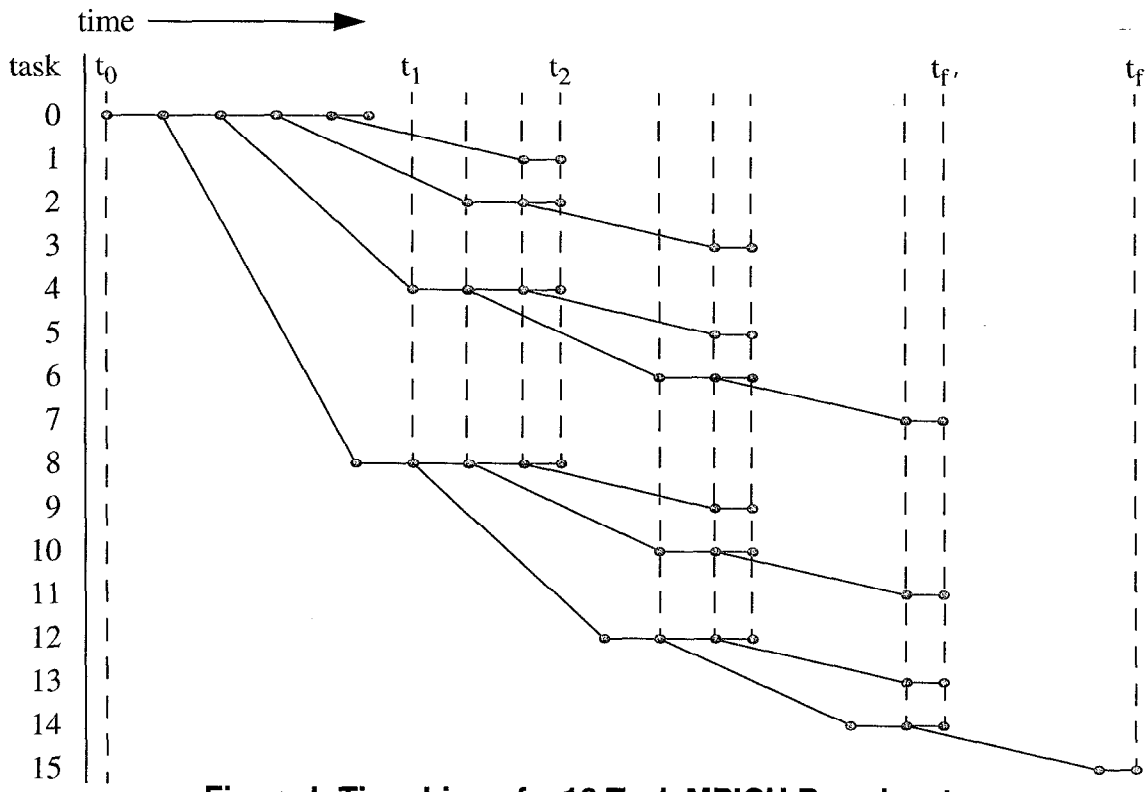


Figure I: Time Line of a 16 Task MPICH Broadcast

in the collective communication has some wire latency, **L**, that is the time the message actually spends in transit. Figure I shows the time line of a 16 task broadcast for the binomial tree algorithm used in MPICH that we use to illustrate our extended model. In our figures, we assume that **L** and the time spent sending or receiving each message are constant. In general, these quantities are not constant due to network contention and optimizations like message forwarding.

Most collective communication benchmarks attempt to measure the total time that it takes to complete the communication, which is its *operation latency*, **OL**. In Figure I, $\mathbf{OL} = \mathbf{t_f} - \mathbf{t_0}$, the difference between the time at which the last processor finishes the operation and the time at which the first processor begins the operation. **OL** is the important latency for collective communications; a method that measures **OL** without measuring **L** would be sufficient. Several benchmark methods attempt to measure **OL**.

Several factors make measuring **OL** difficult. The pipelining effect, the potential for overlapping consecutive communications, causes many of the inaccuracies [3]. In addition, the first processor to begin the operation or the last processor to finish the operation is difficult to identify in general, even with algorithmic knowledge. For example, consider a 15 task broadcast in MPICH. Our model predicts that the last processor to finish the operation will be one of tasks 7, 11 and 13. The correct choice varies with each communication due to stochastic delays and overheads. This uncertainty makes it impossible to use an extension of the ping-pong test generally used to measure overall point-to-point latency ($\mathbf{o_s} + \mathbf{L} + \mathbf{o_r}$). To overcome this difficulty, our method measures the operation latency, $\mathbf{OL_i}$, to each destination, **i**, of the broadcast. The largest of these measurements can be used as a reasonable estimate of **OL**.

Important aspects of the algorithm's performance are omitted if we use only **OL** as our benchmark, even if we measure **OL** accurately. **OL** measures how long it takes to accomplish the

```
Root (task 0):
   t1 = current wallclock
   for x = 1 to some large M
      MPI_BCAST
   t2 = current wallclock

   Report (t2 - t1)/M

All other tasks:
   for x = 1 to some large M
      MPI_BCAST
```

Send Latency Benchmark

```
Root (task 0):
   t1 = current wallclock
   for x = 1 to some large M
      MPI_BCAST
      MPI_BARRIER
   t2 = current wallclock

   Report (t2 - t1)/M

All other tasks:
   for x = 1 to some large M
      MPI_BCAST
      MPI_BARRIER
```

Broadcast Barrier Benchmark

```
Task 0:
   size = MPI_COMM_SIZE
   t1 = current wallclock
   for x = 1 to some large M
      for root = 0 to size - 1
         MPI_BCAST
   t2 = current wallclock

   Report (t2 - t1)/(M * N)

All other tasks:
   size = MPI_COMM_SIZE
   for x = 1 to some large M
      for root = 0 to size - 1
         MPI_BCAST
```

Broadcast Round Benchmark

```
Root (task 0):
   size = MPI_COMM_SIZE
   t1 = current wallclock
   for x = 1 to some large M
      MPI_BCAST
      for y = 0 to size - 1
         MPI_RECV any ACK
   t2 = current wallclock

   Report (t2 - t1)/M

All other tasks:
   for x = 1 to some large M
      MPI_BCAST
      MPI_SEND ACK to root
```

Broadcast Acknowledge Benchmark

Figure II: Broadcast Benchmark Methods

entire communication; each per processor overhead ($\mathbf{o_i}$) measures how long that task cannot be working on something else. As a result, the collective communication user is often as interested in the $\mathbf{o_i}$ values as **OL**. Users must know how long each processor is busy ($\mathbf{o_i}$) and how long it takes the data to reach that processor ($\mathbf{OL_i}$) in order to properly load balance an application. We present a method to measure $\mathbf{OL_i}$ in this paper. The per processor overheads can be measured with a method similar to that used to measure the overhead of point-to-point communications [5].

Section 2: Previously Proposed Broadcast Benchmark Methods

In this section, we experimentally evaluate four previously proposed broadcast benchmark methods, which are shown in Figure II. Our experiments use the MPICH binomial tree implementation and two linear broadcast implementations with which we replaced it. Our results from testing these implementations with each of the proposed benchmark methods demonstrate that all of

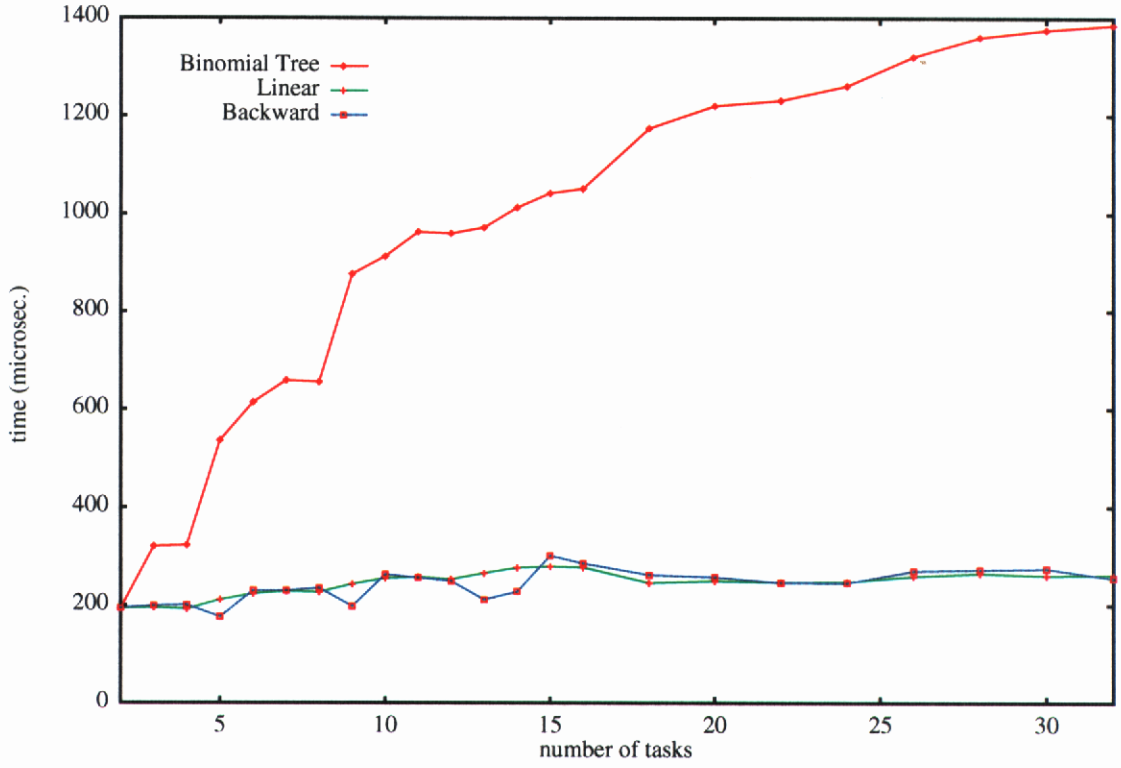


Figure III: Send Latency Benchmark

the methods are insufficient: three of them are inaccurate and the other is incomplete.

In a linear broadcast, the root sends to some task and returns. All other tasks wait to receive from their preceding task and then return after sending the data on down the line (except the last to receive the data, which simply returns). Our linear broadcast algorithms are two simple variations of this algorithm. In our *linear* implementation, the preceding task of task i is task (i -1) mod group size; in our *backward* implementation, the preceding task of task i is task (i +1) mod group size. Intuitively, it is clear that these two implementations are essentially identical. Algorithmically, **OL** is a linear function of communicator size for these implementations, while it is a logarithmic function of the communicator size for the binomial tree implementation.

We ran our tests in the batch partition of the technology refresh SP2 machine "blue" at Lawrence Livermore National Laboratory. This machine is composed of 332 Mhz 604e 4-way SMP nodes. At the time of our tests, the batch partition had 149 nodes and the operating system was AIX 4.2.1. We compiled the various versions of MPICH for all tests with the -g option and used the default optimization level. Our tests were run with either 16 tasks on 4 or 32 tasks on 8 nodes, with MPI tasks assigned cyclically to nodes and used IP for all MPI communication. Tests with **n** tasks, with **n** less than the total number of tasks in the job, used the first **n** tasks. Thus, all tests involved internode communication. Our test job was the only job running on those nodes, although other jobs were concurrently using the network.

For all of our measurements, M = 150. Each data point of our graphs is the mean of several (between 8 and 30) reported measurements; a test was stopped when the standard deviation of the measurements was less than 3% of their mean. We found that tests that did not achieve the cut-off point corresponded with higher measurements. Since we ran our tests in a production environment, it was not feasible to obtain exclusive access to the machine and the inability to achieve the cut-off point indicated a heavily loaded network (one particular code makes extensive use of

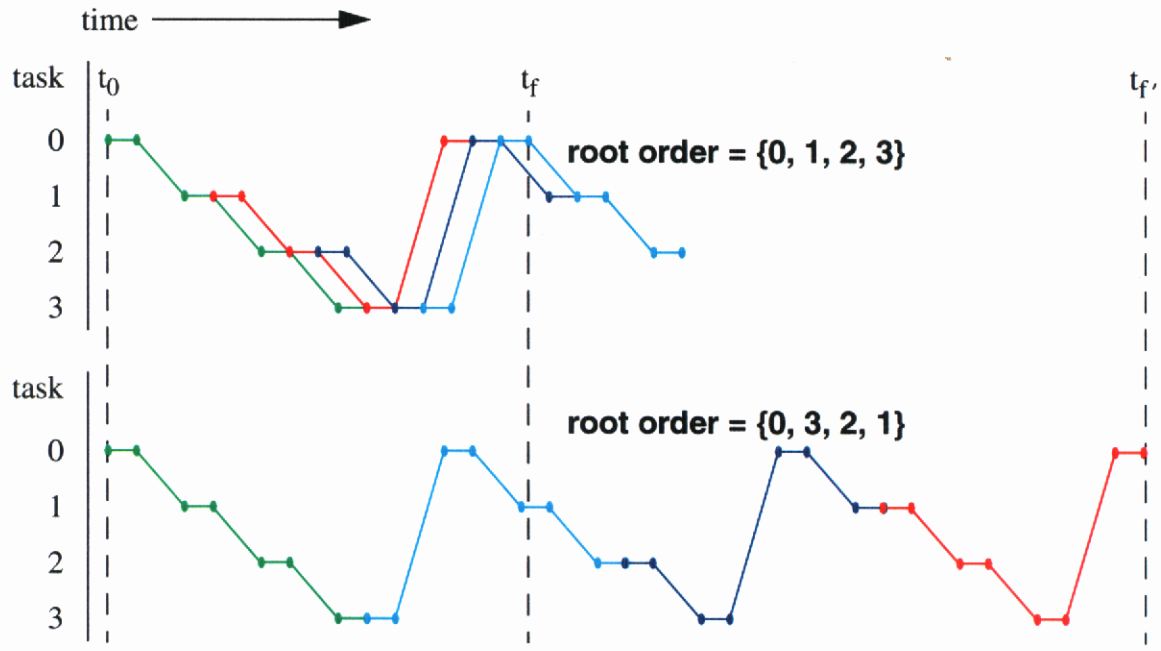


Figure IV: Order Dependence of Pipelining Effect

MPI_Alltoallv and, thus causes considerable network congestion). Except where noted, we repeated all tests until we obtained one that achieved the cut-off point. Thus, our measurements correspond to those that would be obtained with a lightly loaded network.

The send latency benchmark [1, 2, also MPICH performance test suite] actually measures $\mathbf{g_0}$, the minimum interval between broadcasts at the root. There are several problems with this test. Figure III shows the results obtained for our three broadcast implementations with a message size of 256 bytes. Similar results are obtained for larger (64k) messages. As our experimental results demonstrate, $\mathbf{g_0}$ is essentially constant for a linear broadcast. Thus, this benchmark could erroneously lead one to conclude that the linear broadcasts scale well.

Several researchers use the *broadcast rounds* benchmark method, which measures the time to complete some large number of broadcast rounds [1, 3, 6]. A broadcast round consists of one broadcast by each possible root. Unfortunately, the amount of pipelining is highly dependent on the order used to cycle over the tasks, as Figure IV shows. Broadcast rounds accurately measure **OL** if the last node to receive the current broadcast is the root of the next broadcast, as is the case for a linear broadcast if the roots are cycled in the opposite order of the broadcast. However, the pipelining effect is significant if the roots are cycled in the same order. In general, the broadcast rounds benchmark does not provide an accurate measurement with any root order since the last task to receive the broadcast is stochastically determined for most algorithms. Figure V shows our experimental results for a 256 byte broadcast, using the root order {0, 1,..., size-1}. Results for a 64k broadcast are similar. Algorithmic analysis indicates that the pipelining effect is also significant with the binomial tree implementation. Our results demonstrate that the broadcast rounds method is inaccurate, although it does provide a reliable lower bound of **OL**.

We note that the broadcast rounds benchmark does not scale well. Even if the number of rounds (M) is reduced, the method has a tendency to flood the network. This may explain why we were unable to obtain data points that achieved our cut-off point of standard deviation < 3% of mean for the binomial tree implementation with 12 or more tasks or for the linear implementation with more than 20 tasks. We view this poor scaling as an additional drawback of the method.

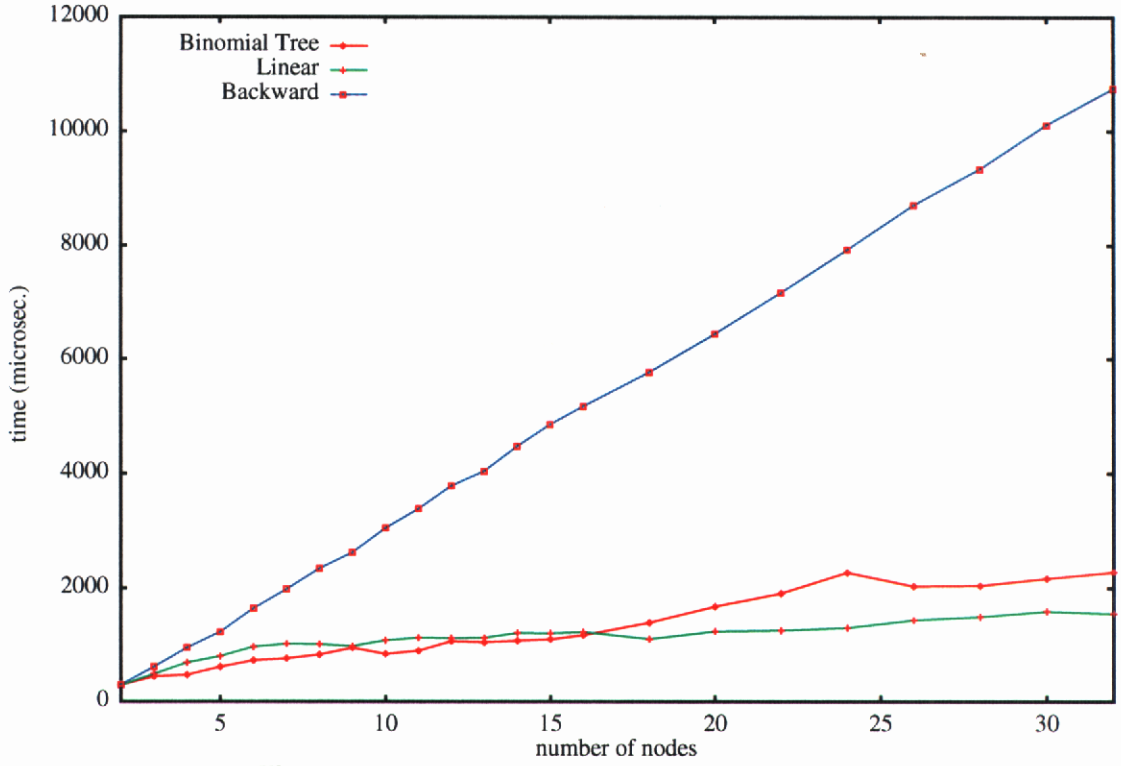


Figure V: Broadcast Rounds Benchmark

The *broadcast barrier* method measures the time required for some large number of broadcast-barrier pairs [9]. This method eliminates the pipelining effect since the barriers ensure that two broadcasts are never in progress concurrently. Thus, it provides a reliable upper bound on **OL**. Unfortunately, measuring barrier latency, like **OL** for a broadcast, is difficult. In addition, even if an accurate measurement of the barrier cost were available, we could not simply subtract it from the broadcast measure since a broadcast can overlap with the barrier before or after it.

We used the linear barrier shown in Figure VI to test the broadcast-barrier method. In our linear barrier, communication starts with task 0 and travels twice around the task ring. The barrier finishes the second time that the communication reaches task **n** - 2, when all tasks are guaranteed that all other tasks have reached the barrier. Figure VII shows two sets of results: in the upper graphs, the linear barrier uses messages that are the same size as the broadcast messages; in the lower graphs, the barriers use the zero length messages. All graphs also show the results obtained

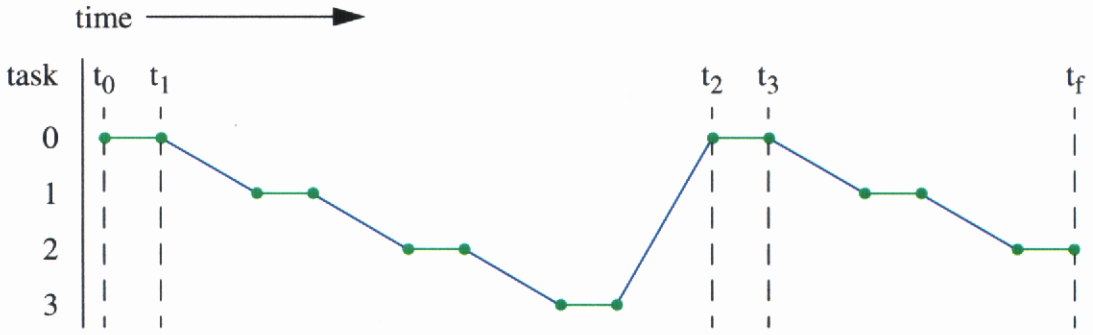
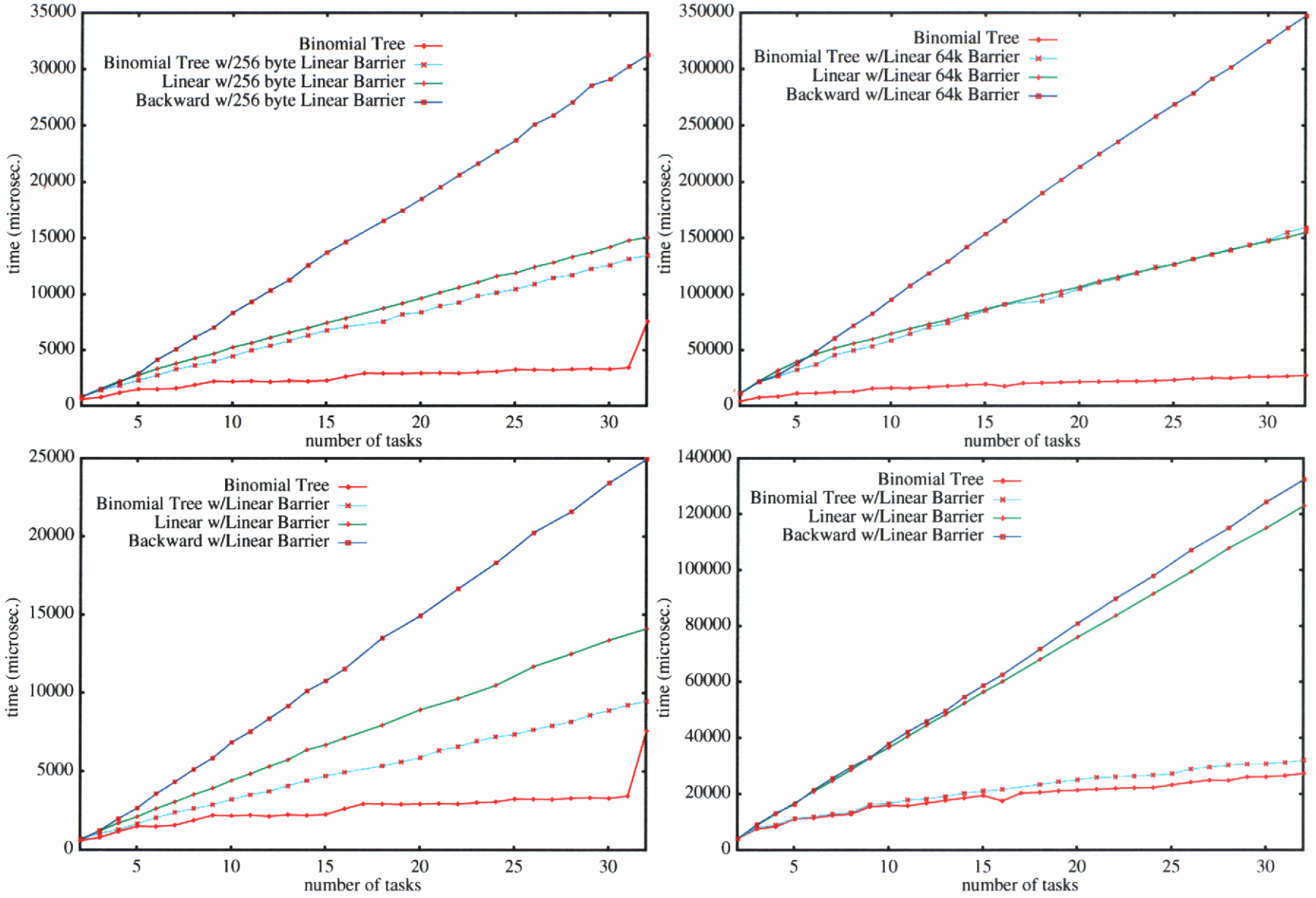


Figure VI: Inefficient Barrier Implementation



a) 256 byte broadcast

b) 64k broadcast

Figure VII: Broadcast Barrier Benchmark

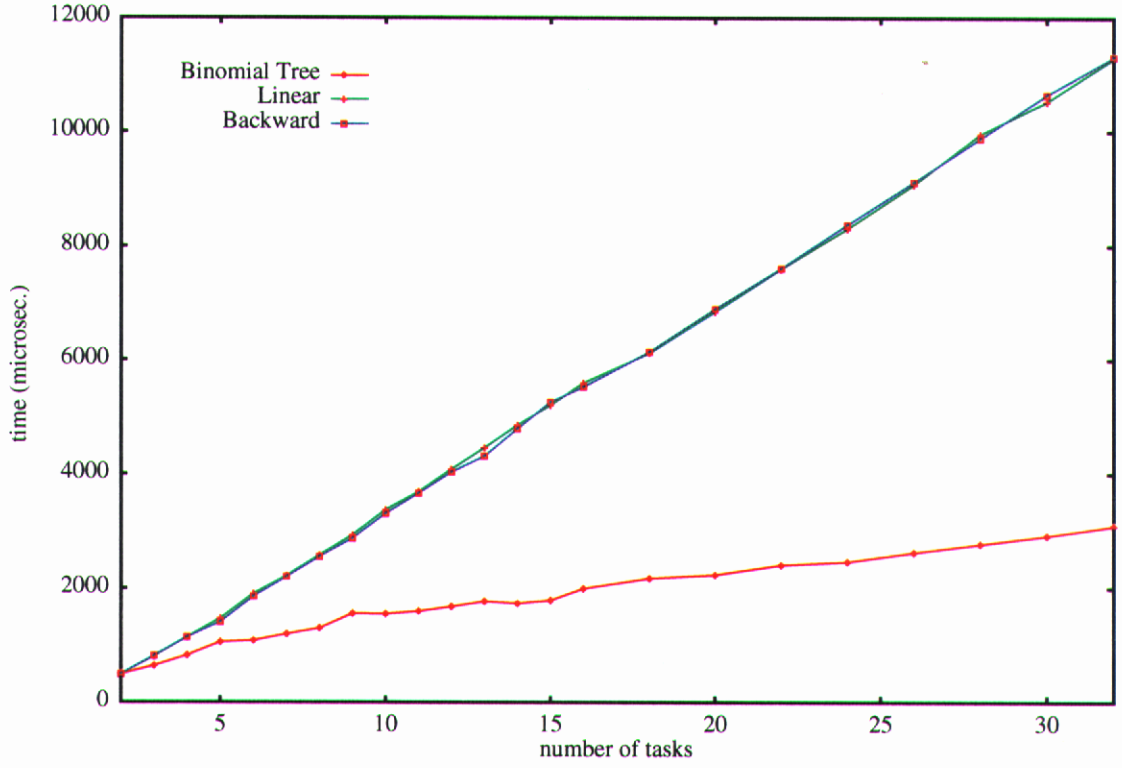


Figure VIII: Broadcast Acknowledge Benchmark

with MPICH's standard binomial tree algorithm and hypercube barrier implementation. Our results demonstrate several problems for this method. The cost of the barrier can dominate the measurement. Since the barriers can overlap with the broadcasts, we cannot simply subtract some measure of the broadcast performance. Finally, the barrier implementation may not be under the control of the benchmarker.

The *broadcast acknowledge* method is a variation of the broadcast barrier method [8]. In this method, the barriers are replaced by acknowledgments explicitly sent to the root. Like the broadcast barrier method, the broadcast acknowledge method provides a reliable upper bound on **OL** since the root does not begin the next broadcast until it has received an acknowledgment from every other task. Unlike the broadcast barrier method, the broadcast acknowledge method always evaluates different broadcast implementations with the same (pseudo) barrier. The results for a 256 byte broadcast shown in Figure VIII demonstrate that this method has potential. However, a broadcasts can still proceed concurrently with the (pseudo) barriers that surround it. Since the amount of overlap varies with the broadcast implementation, it is not possible to accurately correct for the overhead introduced by the acknowledgments. Also, this overhead increases linearly with communicator size and will probably dominate the measurement with large communicators.

Section 3: An Accurate Broadcast Benchmark Method

Our method accurately measures broadcast performance because we do not attempt to measure **OL**. Instead, we observe that a method can be designed that accurately measures $\mathbf{OL_i}$, the operation latency for an individual task **i**. We can use the maximum of these measurements as a reasonable estimate of **OL** in order to provide a succinct measure of performance as the number of tasks increases and for comparison purposes to other broadcast benchmark methods.

```
t1 = current wallclock
for x = 1 to some large M
   MPI_SEND to i
   MPI_RECV from i
t2 = current wallclock
RTLi = (t2 - t1)/M

MPI_BARRIER
MPI_BCAST
MPI_RECV ACK from i

t1 = current wallclock
for x = 1 to some large M
   MPI_BCAST
   MPI_RECV ACK from i
t2 = current wallclock
Ei = (t2 - t1)/M

Report OLi = Ei - (RTLi/2)
```

Root (task 0)

```
for x = 1 to some large M
   MPI_RECV from root
   MPI_SEND to root

for x = 1 to some large M + 1
   MPI_BCAST
   MPI_SEND ACK to root
```

Current task i

```
for x = 1 to some large M + 1
   MPI_BCAST
```

All other tasks

Figure IX: OL_i Benchmark Method

Figure IX shows our method for measuring $\mathbf{OL_i}$, which we repeat for each possible i. Our method works for a simple reason - it eliminates the pipelining effect only along the broadcast path from the root to the task **i**. Broadcasts that are concurrently in progress along other paths do not affect our measurement. Our method relies on the assumptions that the acknowledgment to

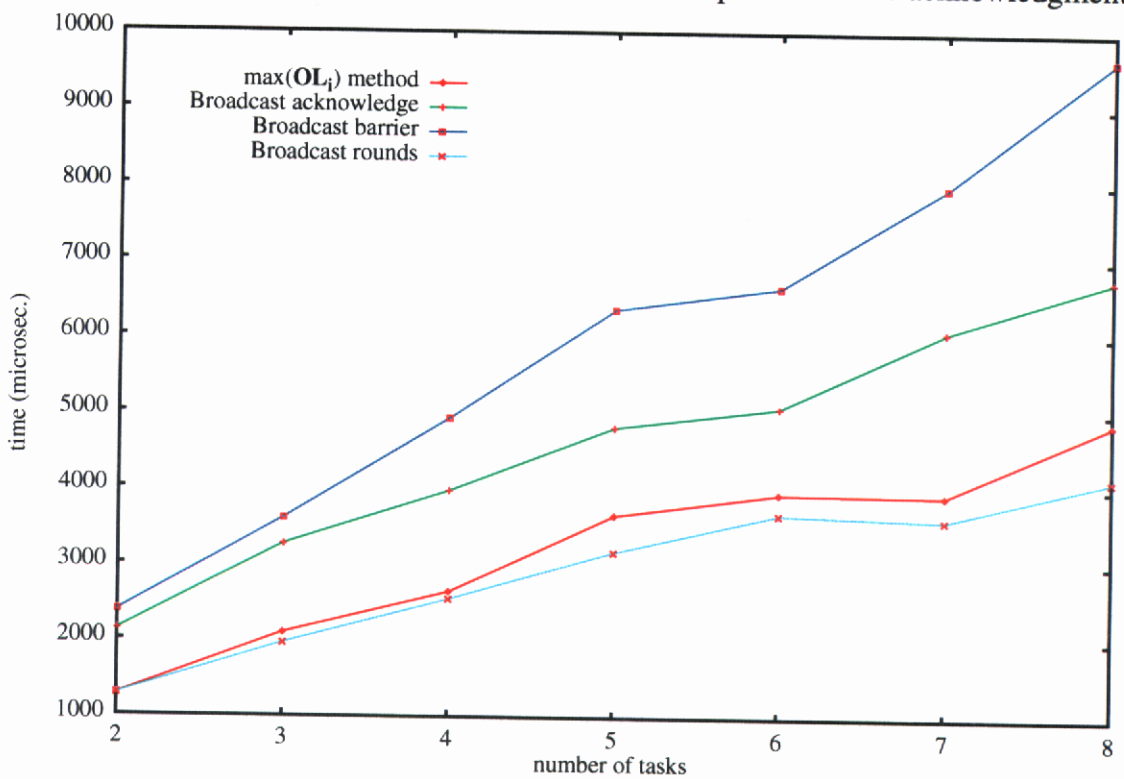


Figure X: Initial Comparison of Benchmark Methods

the root arrives at the root after the broadcast has finished at the root and that no task **j** on **i**'s broadcast path must delay the next broadcast. These assumptions hold if the measured time, $\mathbf{E_i}$ is greater than the broadcast gap, $\mathbf{g_j}$ for any task **j**, including the root, on **i**'s broadcast path, which is true for most broadcast implementations. Nonetheless, some implementations, such as a flat broadcast in which the root sends directly to every other task, can violate this assumption.

We can prove our measurement of $\mathbf{OL_i}$ is accurate if $\mathbf{E_i}$ is greater than any of the broadcast gaps, which holds for all of our $\mathbf{OL_i}$ measurements for the implementations that we discuss. We have designed a method for measuring $\mathbf{OL_i}$ accurately when this assumption is violated. The benchmarker must know **i**'s broadcast path in order to use this method. This information is clearly available to the library implementer; we are currently designing a method to determine the communication pattern of an implementation when source code is unavailable.

Due to a major hardware and software upgrade, our test platform became unavailable as we were preparing this paper. This upgrade will double the number of nodes and change the OS version. We will rerun our results under the new configuration for the full paper. Initial results on a older and smaller SP2 confirm that our method provides an accurate estimate of **OL**. Figure X shows that our estimates of **OL** for the binomial tree implementation always fall between the lower bound provided by the broadcast rounds method and the upper bounds obtained from the broadcast barrier and broadcast acknowledge methods. The full paper will include results using our method for all three broadcast implementations and extend Figure X to at least 64 nodes.

Section 4: Conclusions

Broadcast communication is an important factor in the performance of message passing applications. Therefore, reliable measurement of broadcast performance is an important criteria for evaluating message passing libraries. Previously proposed broadcast benchmark methods were inaccurate. We have presented an new and accurate broadcast benchmark method.

Our methodology is accurate and reliable for implementers of broadcast algorithms. When implementation source code is unavailable (i.e. when a "black box" methodology is required), it is easy to determine when our measurements may be inaccurate. Further, we expect to extend our methodology to an accurate and reliable "black box" methodology.

Section 5: References

[1] G.A. Abandah, "Modeling the Communication and Computation Performance of the IBM SP2," *Master's Thesis*, University of Michigan, 1995.

[2] G.A. Abandah and E.S. Davidson, "Modeling the Communication Performance of the IBM SP2," *Proc. of the 10th International Parallel Processing Symp.*, 1996.

[3] M. Bernaschi and G. Iannello, "Collective communication operations: experimental results vs. theory," *Concurrency: Practice and Experience*, 1998, Vol. 10, No. 5, pp. 359-386.

[4] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1993, pp. 1-12.

[5] D.E. Culler, L.T. Liu, R.P. Martin and C.O. Yoshikawa, "Assessing Fast Network Interfaces," *IEEE Micro*, 1996, Vol. 16, No. 1, pp. 35-43.

[6] P. Husbands and J.C. Hoe, "MPI-StarT: Delivering Network Performance to Numerical Applications," *Proc. of the 1998 ACM/IEEE SC98 Conference*, 1998.

[7] R.M. Karp, A. Sahay, E. Santos and K.E. Schauser, "Optimal Broadcast and Summation in the LogP Model," *Proc. of the 5th Annual Symp. on Parallel Algorithms and Architectures*, 1993, pp. 142-153.

[8] P.J. Mucci and K.S. London, "Low Level Architectural Characterization Benchmarks for Parallel Computers," *Tech. Report ut-cs-98-394*, University of Tennessee, 1998.

[9] R.H. Reussner, "User Manual of SKaMPI, Special Karlsruher MPI-Benchmark," *Tech. Report*, University of Karlsruhe, 1998.