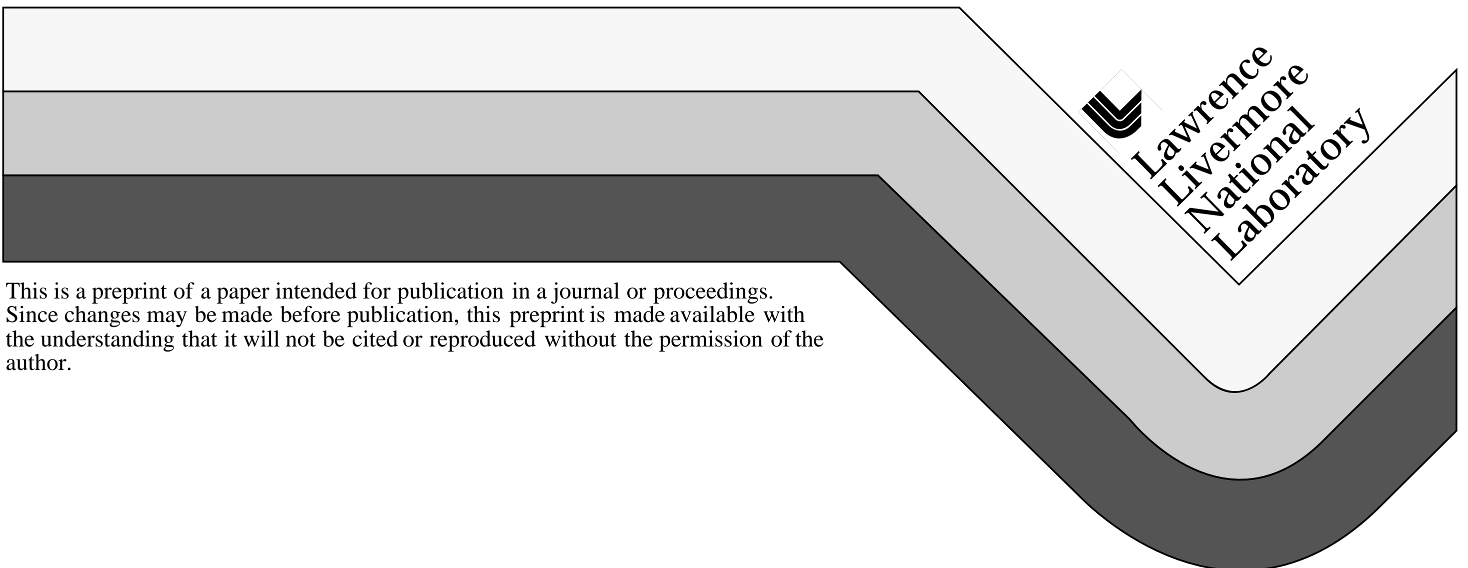
UCRL-JC-133177 Rev. 1
PREPRINT

Accurately Measuring MPI Broadcasts in a Computational Grid

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

May 6, 1999



This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Accurately Measuring MPI Broadcasts in a Computational Grid

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

An MPI 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, even in a challenging grid environment.

Measuring broadcast performance is not easy. Simply sending one broadcast after another allows them to proceed through the network concurrently, thus resulting in inaccurate 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. This problem becomes even more challenging in grid environments. Latencies along different links can vary significantly. Thus, an algorithm's performance is difficult to predict from it's communication pattern. Even when accurate prediction is possible, the pattern is often unknown. Our method introduces a measurable overhead to eliminate the pipelining effect, regardless of variations in link latencies.

1. Introduction

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. Also, accurate and complete measurements could guide use of a given implementation to improve application performance. These choices will become even more important as grid-enabled MPI libraries [6, 7] become more common since bad choices are likely to cost significantly more in grid environments. In short, the distributed processing community needs accurate, succinct and complete measurements of collective communications performance.

Since successive collective communications can often proceed concurrently, accurately measuring them is difficult. Some benchmarks use knowledge of the communication algorithm to predict the timing of events and, thus, eliminate 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 [4]. 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 method that can accurately measure broadcast performance, even in a grid environment.

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; Release No. UCRL-JC-133177.

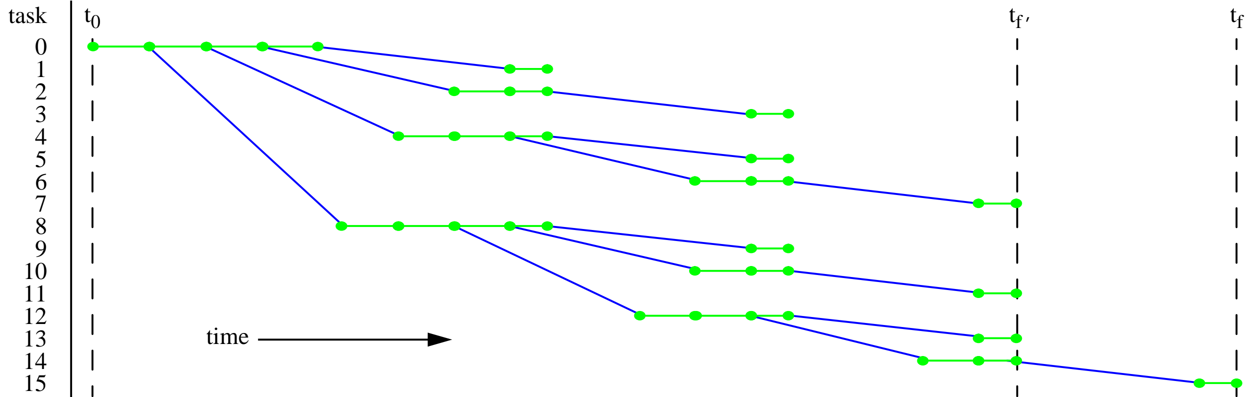


Figure 1: Time Line of a 16 Task MPICH Broadcast

2. Collective Communications Model

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]. 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 overlapped with the message transmission. The (wire) latency, **L**, is 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 $\mathbf{o_s} + \mathbf{L} + \mathbf{o_r}$. 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 with *per processor* parameters to capture collective communications more accurately. Our extensions apply to both asymmetrical (collective communications with a root) and symmetrical collective communications [11]. 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 overheads can be measured with a method similar to that used to measure the overhead of point-to-point communications [5]. The minimum interval of time, $\mathbf{g_i}$, between consecutive occurrences of the same collective communication at processor **i** is the per processor gap, which can be measured simply by timing repeated occurences of the operation at each processor.

Figure 1 shows the time line of a 16 task broadcast for the binomial tree algorithm used in MPICH [8], a popular implementation of the MPI standard [12]. In our figures, we assume **L** and the time spent sending or receiving a message are constant. Our broadcast benchmark method does not rely on these assumptions, which do not hold in general, particularly in grid environments, where latency along different links can vary highly.

Most collective communication benchmarks try to measure *operation latency*, **OL**, the total time that it takes to complete the communication. In Figure 1, $\mathbf{OL} = \mathbf{t_f} - \mathbf{t_0}$, 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 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. 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**.

```
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 2: Broadcast Benchmark Methods

3. Existing Benchmark Methods

In this section, we experimentally evaluate four previously proposed broadcast benchmark methods, which are shown in Figure 2. 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 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 combined technology refresh (CTR) SP2 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 305 nodes and the operating system was AIX 4.3.2. 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 in blocks to nodes and used IBM's OS bypass mechanism (user space) 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. 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** = 100. 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.

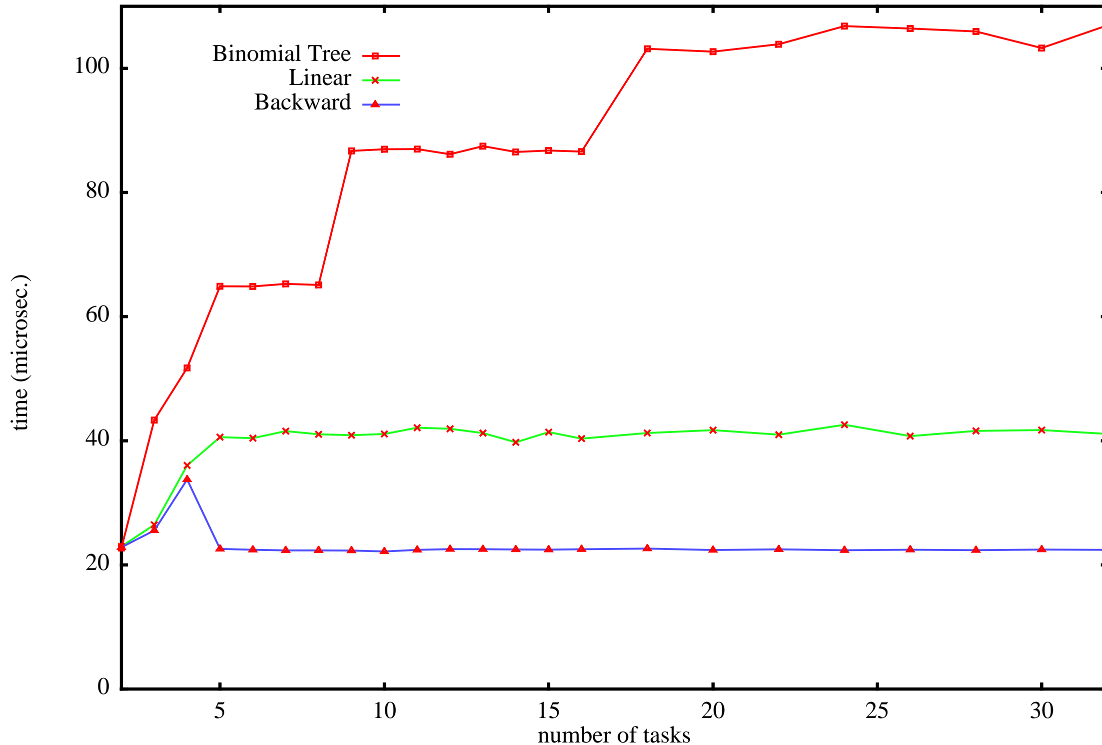


Figure 3: Send Latency Benchmark

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 MPI_Alltoallv and, thus causes considerable network congestion). 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, which measures the time that the root takes to send several broadcasts [1, 2, also MPICH performance test suite], actually measures $\mathbf{g_0}$, the minimum interval between broadcasts at the root. Figure 3 shows the results obtained for our three implementations with a 256 byte message. Larger (64KB) messages yield similar results. As our experimental results demonstrate, $\mathbf{g_0}$ is essentially constant for a linear broadcast. Thus, this benchmark can erroneously lead one to conclude that the linear broadcasts scale well. We note that the backward implementation outperforms the linear implemnetation slightly since off node bandwidth is slightly higher than on node bandwidth on the CTR machine.

Several researchers use the *broadcast rounds* benchmark method, which measures the time to complete some large number of broadcast rounds [1, 3, 9]. 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 4 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 6 shows our experimental results for a 256 byte broadcast, using the root order {0, 1,..., size-1}. Our results, show that the linear implementations can appear significantly different, under this method; worse, this method can make the linear and binomial tree implementations look comparable. 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**.

Another drawback of this method is that it does not scale well. Even if the number of rounds (**M**) is reduced, the broadcast rounds method has a tendency to flood the network. As a result, it is often difficult to obtain data points with the required standard deviation $< 3\%$ of mean.

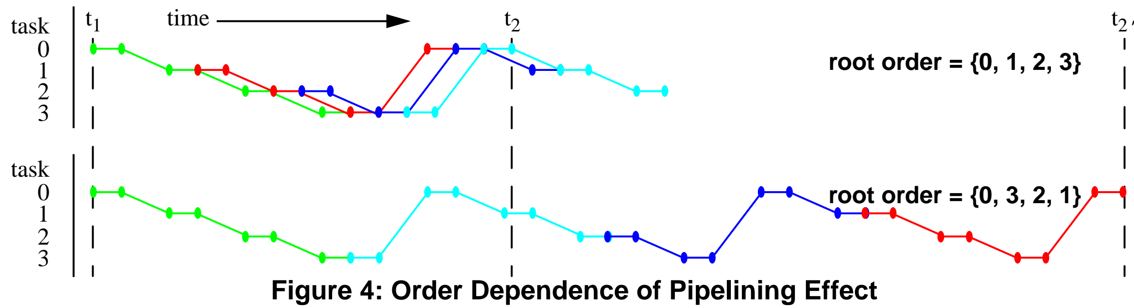


Figure 4: Order Dependence of Pipelining Effect

The *broadcast barrier* method measures the time required for a large number of broadcast-barrier pairs [14]. 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 tested the broadcast barrier method using the linear barrier shown in Figure 5. In this 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. Pipelining can vary significantly with this barrier. The linear broadcast overlaps with the second half of the preceding barrier and the first half of the following one. Alternatively, the backward implementation has almost no overlap since its messages travel in the opposite direction.

Figure 7 shows results for 256 byte and 64K broadcasts for each of the implementations using this barrier; we also include results for MPICH's standard binomial tree algorithm and hypercube barrier implementation. When broadcast messages are small, the pipelining effects and

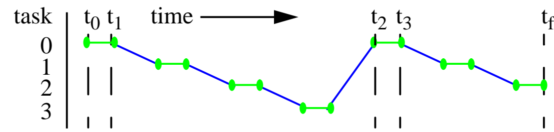


Figure 5: Inefficient Barrier

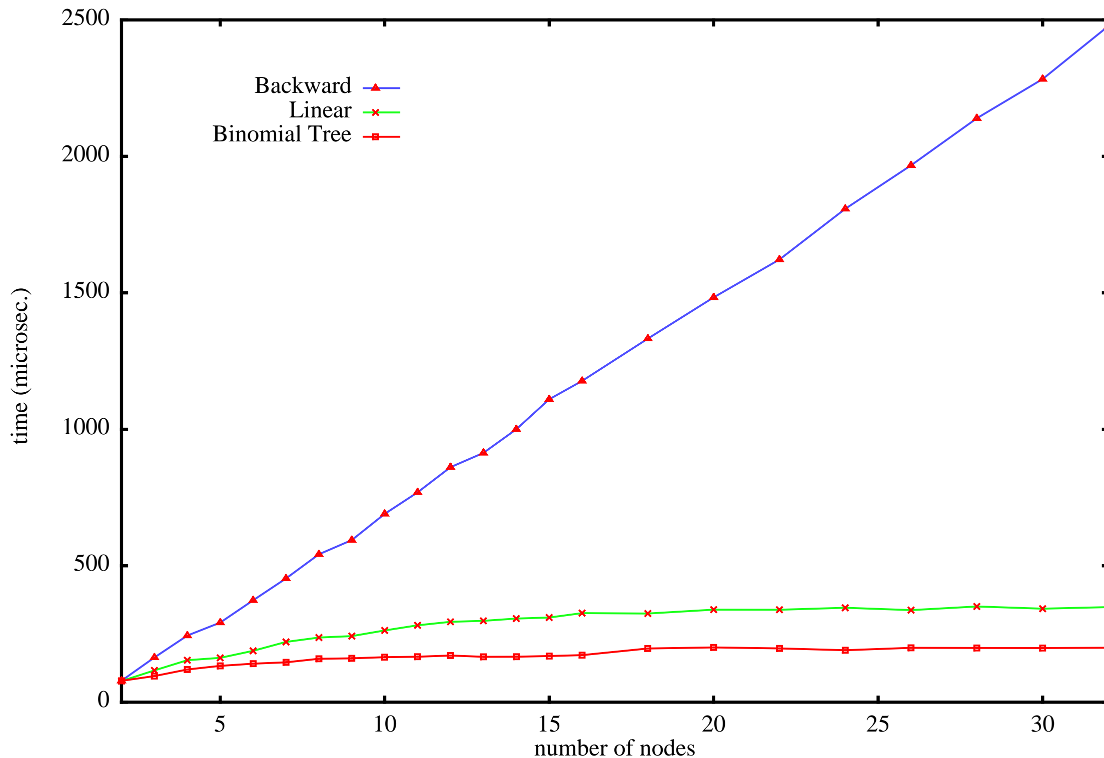


Figure 6: Broadcast Rounds Benchmark

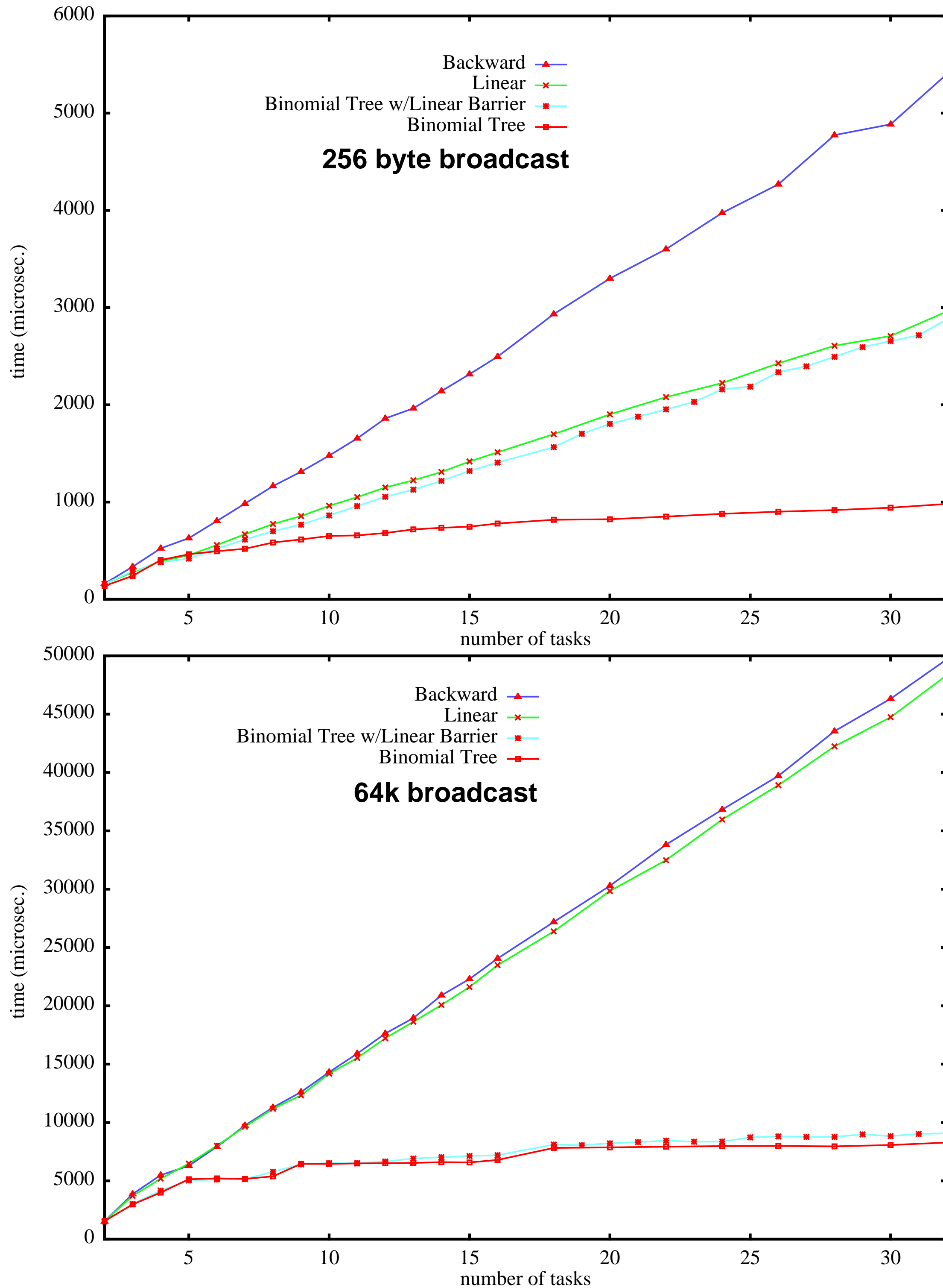


Figure 7: Broadcast Barrier Benchmark

the cost of the barrier dominate this method's measurements, which are similar to those of the broadcast rounds method. Larger broadcast messages reduce the importance of the obscuring effects; however, they again dominate if we use 64K messages in the barrier. Our results demonstrate that this method is unsatisfactory: it is highly dependent on the library's barrier implementation; measurements can be dominated by the barriers; and we

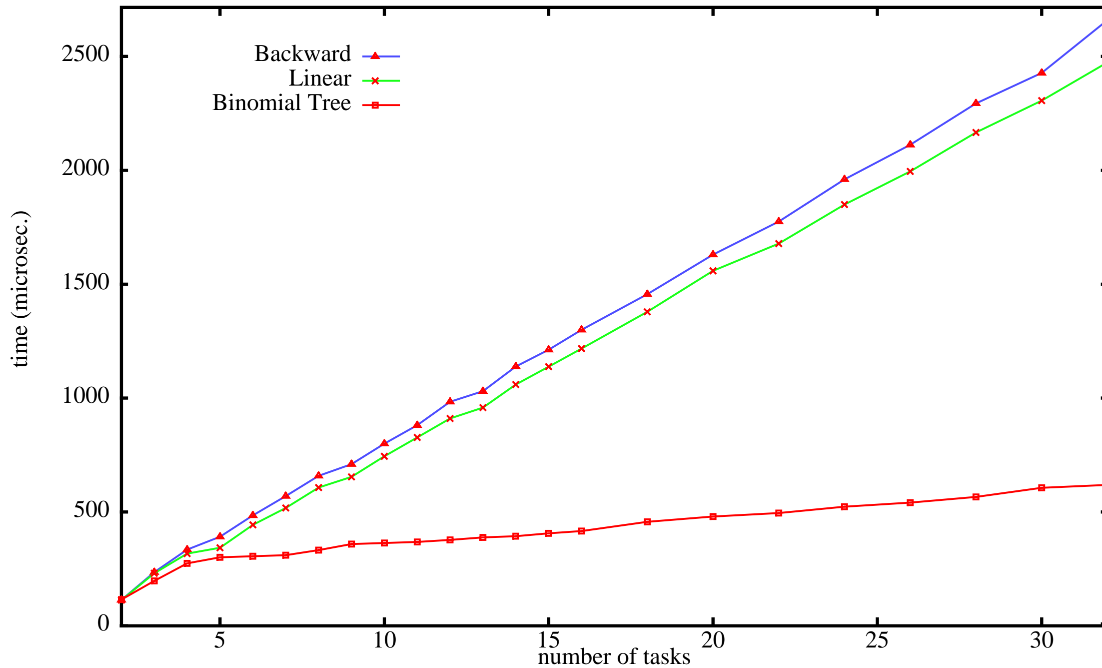


Figure 8: Broadcast Acknowledge Benchmark

cannot simply subtract some measure of the barrier performance since the barriers can overlap with the broadcasts.

In the *broadcast acknowledge* method, each task sends an explicit acknowledgments to the root [13]. This method also 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. This method can work well for small numbers of tasks, as the results for a 256 byte broadcast shown in Figure 8 demonstrate. However, many of the acknowledgments overlap with the broadcast. Since the amount of overlap varies with the broadcast implementation, it is not possible to correct for the overhead introduced by the acknowledgments accurately. Our results in the next section show that the overhead dominates the measurement for large number of tasks since it increases linearly with communicator size.

4. An Accurate 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_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 9: $\mathbf{OL_i}$ Benchmark Method

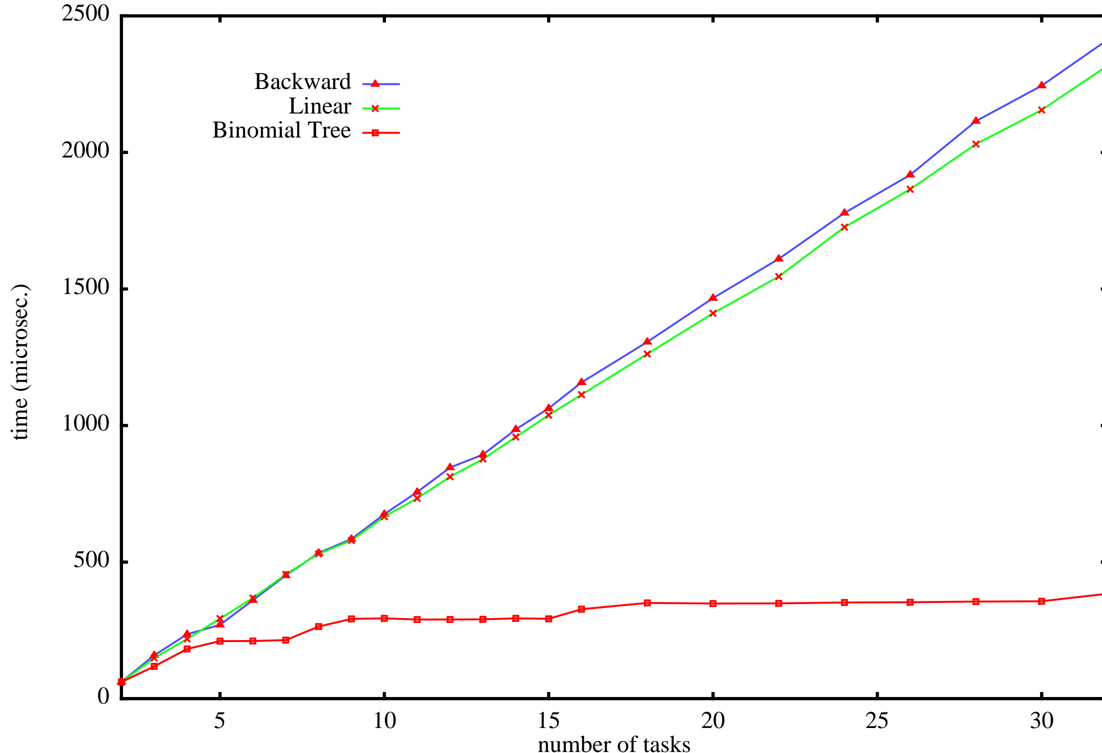


Figure 10: Maximum $\mathbf{OL_i}$ Benchmark Results

Figure 9 shows our method for measuring $\mathbf{OL_i}$, which we repeat for each possible **i**. In our method, task **i** sends an acknowledgment to the root. The root does not begin the next broadcast until it receives this acknowledgment. We can accurately correct for the overhead of the acknowledgment since it is exactly the latency of a single point-to-point message from **i** to the root.

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 does not arrive at the root before the root has finished the broadcast and that no task **j** on **i**'s broadcast path delays 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. Since these requirements are violated only if **j** is sending more messages for the broadcast, they must hold for largest $\mathbf{E_i}$, the basis of our estimate of $\mathbf{OL_i}$. The requirements hold for all **i** for most broadcast implementations although some implementations, such as a flat broadcast in which the root sends directly to every other task, can violate this assumption. 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. We are designing a method to determine this information when source code is unavailable.

Figure 10 shows our estimates of **OL** for a 256 byte broadcast agree with our algorithmic analysis. Figure 11 compares the results of the different methods with the binomial tree implementation. Our estimates 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. At about 20 tasks, acknowledgment overhead dominates the measurements of the broadcast acknowledge method. The broadcast rounds method consistently underestimates **OL** by a factor of two, while the broadcast barrier method overestimates it by an even larger factor. Our max($\mathbf{OL_i}$) method scales well and conforms to our expectation of flat performance between powers of two.

5. 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. Our results demonstrate that previously proposed broadcast benchmark methods cannot measure even a simple linear broadcast reliably. The pipelining effect of

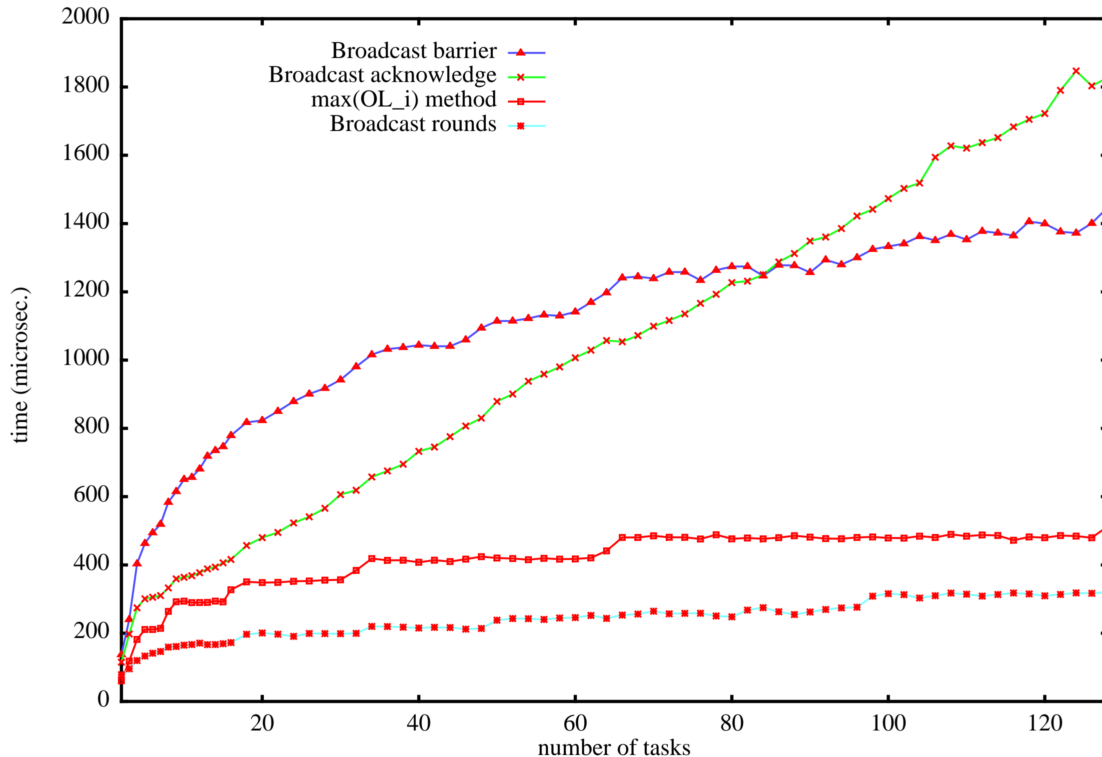


Figure 11: Comparison of Benchmark Methods

message overlap between successive broadcasts makes benchmarking broadcasts hard.

We presented a new, accurate broadcast benchmark method that scales well. It works because it measures the latency to individual tasks instead of trying to measure the latency of the entire broadcast directly. Our results demonstrate that our method is accurate. We will extend our method to other collective communications. Reduction benchmarks suffer from similar problems but require a different solution since the communication pattern is a fan-in logically. Symmetrical collective communications, such as global reductions, are also difficult since they can be implemented asymmetrically.

6. References

[1] G.A. Abandah, "Modeling the Communication and Computation Performance of the IBM SP2," *Master's Thesis*, Univ. 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] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal and S. Tuecke, "Wide-Area Implementation of the Message Passing Interface," *Parallel Computing*, 1998, Vol. 24, No. 11, pp. 1735-1749.

[7] I. Foster and N. Karonis, "A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems," *Proc. of the 1998 ACM/IEEE SC98 Conference*, 1998.

[8] W. Gropp, E. Lusk, N. Doss and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, 1996, Vol. 22, No. 6, pp. 789-828.

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

[10] 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.

[11] T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat and R.A.F. Bhoedjang, "MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems," *Proc. of the 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1999, pp. 131-140.

[12] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard," *International Journal of Supercomputing Applications*, 1994, Vol. 8, No. 3/4, pp. 165-414.

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

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