Improving the Performance of MPI Collective Communication on Switched Networks[*]

Rajeev Thakur *William Gropp*
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439, USA
email: {thakur, gropp}@mcs.anl.gov
Tel: (630) 252-7847, Fax: (630) 252-5986

Preprint ANL/MCS-P1007-1102
November 2002

Abstract

In this paper, we present new algorithms for improving the performance of collective communication operations in MPI. Our target architecture is a cluster of machines connected by a switched network such as Myrinet or switched ethernet. We have developed new algorithms for all the MPI collective communication operations, namely, scatter/gather/reduce, allgather/allreduce, broadcast, reduce-scatter, all-to-all, and scan. We compare the performance of our new algorithms with the algorithms currently used in the latest version of MPICH on up to 256 nodes of a Myrinet-connected cluster. For operations such as scatter/gather/reduce, allgather/allreduce, and reduce-scatter, we observe an improvement of up to a factor of 10 for short message sizes. For operations such as broadcast and reduce-scatter and for long messages sizes, the new algorithms are truly scalable: the time taken remains fairly constant as we increase the number of processes participating in the operation.

[*]This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

1 Introduction

MPI, the Message Passing Interface Standard, defines an API (application programming interface) for message passing in parallel programs [10, 18]. MPI defines both point-to-point communication routines, such as sends and receives between pairs of processes, and collective communication routines that involve a group of processes that need to perform some operation together. Examples include broadcasting data from a root process to other processes and finding the global minimum or maximum of data values on all processes (called a reduction operation). Collective communication routines provide the user with a simple interface for commonly required operations. They also enable an implementation to optimize these operations for the particular machine architecture. As a result, collective communication is widely and frequently used in many applications, and the performance of collective communication routines is often critical to the performance of the overall application.

This paper focuses on improving the performance of the collective communication operations in MPI. We have developed new algorithms for all the MPI collective operations, namely, scatter/gather/reduce, allgather/allreduce, broadcast, reduce-scatter, all-to-all, and scan. For lack of space, we do not cover the scan algorithm in this paper. We have implemented the new algorithms on top of MPI point-to-point operations so that they can be used with any MPI implementation. For the purpose of this study, we compare the performance of the new algorithms with the algorithms currently used in the latest release (1.2.4) of the MPICH implementation of MPI [12]. The new algorithms will be included in the next release (1.2.5) of MPICH.

We focus on developing optimized algorithms for the most popular architecture of today, namely, clusters of workstations connected by a switched network, such as Beowulf clusters connected by Myrinet or switched fast ethernet or gigabit ethernet. On such networks, all nodes are more or less "equidistant," and the communication performance depends more on the latency for short messages and bandwidth usage for long messages. We have used this property in our design of the collective communication algorithms. We have strived to design algorithms that minimize latency in the case of short messages and minimize bandwidth usage for long messages. Achieving this goal often means using one algorithm for short messages and a different algorithm for long messages and determining a good cutoff point. We present performance results that compare the performance of the new algorithms with those used in MPICH-1.2.4 on an IA-32 cluster located at the National Center for Supercomputing Applications (NCSA), for up to 256 processes. For operations such as scatter/gather/reduce, allgather/allreduce, and reduce-scatter, we observe an improvement of up to a factor of 10 for short message sizes. For operations such as broadcast and reduce-scatter and for long messages sizes, the new algorithms are truly scalable: the time taken remains fairly constant as we increase the number of processes participating in the operation.

MPI collective communication operations are very general. They are intended to work for non-power-of-two number of processes and on heterogenous systems; the data to be communicated can be noncontiguous (defined by using MPI derived datatypes). Our algorithms handle all these cases.

The rest of this paper is defined as follows. Section 2 describes related work in this area. Section 3 describes the cost model we use to guide the development of the algorithms and describes the environment where we ran our tests. The algorithms and their performance are described in Section 4. We conclude in Section 5 with a brief discussion of future work.

2 Related Work

Collective communication has been an active area of research over the past twelve years or so. Early work on collective communication focused on developing optimized algorithms for particular architectures, such as hypercube, mesh, or fat tree, with an emphasis on minimizing link contention, node contention, or the distance between communicating nodes [2, 3, 4, 6, 13, 15].

Dongarra et al. have developed automatically tuned collective communication algorithms [5, 20]. Their approach consists of running tests to measure system parameters and then tuning their algorithms for those parameters. Researchers in Holland and at Argonne have optimized MPI collective communication for wide-area distributed environments [7, 8, 9]. In such environments, the goal is to minimize communication over slow wide-area links at the expense of more communication over faster local-area connections. Research has also been done on developing collective communication algorithms for clusters of SMPs [14, 17, 19], where communication within an SMP is done differently from communication across a cluster.

Most closely related to our work is the InterCom collective communication library developed by van de Geijn et al. [1, 11, 16]. They also used different algorithms for short messages and long messages and minimized latency for short messages and bandwidth for long messages. The specific algorithms they used, however, differ from ours. For example, we use recursive doubling extensively, whereas they do not. Their algorithms were designed for the mesh interconnect on the Intel Paragon, while ours are designed for a switch-based interconnect. Furthermore, we are not aware of any MPI implementation that has implemented their algorithms. The algorithms proposed in this paper, on the other hand, will be available as part of MPICH-1.2.5.

3 Parallel Computer Model and Test Environment

For the algorithms described in this paper, we make the following assumptions about the parallel computer on which collective communication is being performed.

The parallel computer comprises a number of compute nodes interconnected by a switched communication network. The time taken to send a message between any two nodes is independent of the distance between the nodes and can be modeled as $\alpha + n\beta$, where α is the latency (or startup time) per message, independent of message size, and β is the transfer time per byte. The links between two nodes are bidirectional; that is, a message can be transferred in both directions on the link in the same time as in one direction. The node's network interface is single ported; that is, at most one message can be sent and one message can be received simultaneously. An example of such a parallel computer is a cluster of machines interconnected by Myrinet, switched ethernet, or some other switch fabric.

We used the IA-32 cluster at NCSA for our experiments. The cluster has a total of 484 compute nodes interconnected by a Myrinet-2000 interconnection network. Each compute node is an IBM eServer x330 thin server with two 1 GHz Intel Pentium III processors. In our experiments, we ran only a single process on each compute node. We used MPICH-GM, the implementation of MPICH on top of GM, which is the low-level message-passing library for Myrinet. Our collective communication algorithms are implemented on top of MPI point-to-point communication functions. Therefore, we were able to test our new algorithms by using the existing MPICH-GM implementation on the system.

4 Algorithms

In this section we present the new algorithms and compare their performance with the old algorithms.

4.1 Scatter, Gather, Reduce

We consider the three operations scatter, gather, and reduce together because the algorithmic issues are similar. `MPI_Scatter` scatters data from a root process to all other processes in the communicator, `MPI_Gather` gathers data from all processes in the communicator to the root, and `MPI_Reduce` performs a reduction operation (such as addition, minimum, or maximum) on the data contributed by all processes and stores the result at the root. Without loss of generality, we consider only the scatter operation in the rest of this section. `MPI_Scatter` is defined as follows:

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

The root process sends the first `sendcount` items of type `sendtype` from its `sendbuf` to process 0; the next `sendcount` items to process 1; and so forth. Each process stores the received data in `recvbuf`.

A simple algorithm for scatter is a linear algorithm in which the root first sends data to process 0, then to process 1, and so forth. MPICH-1.2.4 uses this algorithm. A linear algorithm seems bad at first glance, but it is not that bad for scatter. If p is the number of processes and n is the total size of the data to be scattered from the root to all other processes, the time taken by this algorithm is given by

$$T_{linear} = (p-1)\alpha + \frac{p-1}{p} n\beta$$

For long messages, this algorithm is as good as any because the latency term can be ignored and no extra data is communicated (the $\frac{p-1}{p} n\beta$ term cannot be reduced further). For short messages, however, an algorithm that takes fewer than $p-1$ steps, for example $\lg p$ steps, is better. We have implemented such an algorithm.

The new algorithm uses a minimum spanning tree (MST) approach. In the first step, the root sends $\frac{n}{2}$ amount of data to process (root + $\frac{p}{2}$) with wrap-around when the root is nonzero. Each of these processes act as a root within their subtree and recursively subdivides and continues this algorithm. This communication takes a total of $\lceil \lg p \rceil$ steps. The bandwidth requirement is $\frac{n}{2}\beta$ in the first step, $\frac{n}{2^2}\beta$ in the second step, and so forth, up to $\frac{n}{2^{\lceil \lg p \rceil}}\beta$ in the last step. Therefore, the total time taken by this algorithm is given by

$$T_{MST} = \lceil \lg p \rceil \alpha + \frac{p-1}{p} n\beta$$

Comparing with the linear algorithm, we see that the bandwidth requirement is identical, whereas the latency is lower. Therefore, the MST algorithm is the best algorithm for all message sizes. One drawback of the MST algorithm is that it needs temporary buffer space on the intermediate nodes. The maximum buffer space needed is $\frac{n}{2}$. The linear algorithm, on the other hand, does not require any extra buffer space.

For gather and reduce, we use a similar MST algorithm that has the same time complexity.

4.1.1 Performance

Figure 1 shows the performance of scatter with the old and new algorithms.[1] For short messages, the performance improvement provided by the MST algorithm is evident, the reason being the $\lg p$ versus $p-1$ steps. For the same reason, the performance improvement is higher for larger numbers of nodes. For example, on 256 nodes, the performance improves by a factor of 8.7. On 8 nodes, on the other hand, the performance improves only by a factor of 1.8. As the message size gets larger, bandwidth plays a more prominent role in the overall time, and the performance difference between the two algorithms narrows because they have the same bandwidth requirement. For some message sizes, the linear algorithm actually performs better than MST, although we don't know why; perhaps some constants are not reflected in the mathematical equation, or the difference in communication scheduling between the two algorithms makes a difference on the Myrinet interconnect. We also see a few abrupt jumps in timings at certain message sizes; we don't understand why these jumps happen, but they are reproducible. We plan to investigate the causes with the help of our contact at Myricom (Patrick Geoffray) and report the results in the final version of this paper.

4.1.2 Scatterv, Gatherv

MPI also has "v" versions of scatter and gather in which the user can specify a vector of sendcounts in the case of scatter or recvcounts in the case of gather, representing different amounts of data to be sent to or received from each process, for example,

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm);
```

where `sendcounts` is an array specifying the number of elements to send to each process, and `displs` is an array in which entry `i` specifies the displacement (relative to `sendbuf`) from which to take the outgoing data to process `i`.

The problem with using the MST algorithm for scatterv is that the `sendcounts` and `displs` arrays are valid only on the root. As a result, if an MST algorithm is used, at least the `sendcounts` information must be sent from the root process to other processes participating in the store-and-forward MST algorithm. Doing so takes up additional latency, thereby eliminating some of the benefit of using MST for short messages. Therefore, we currently use the linear algorithm for scatterv and gatherv, but we are investigating ways to use MST without incurring additional latency, such as by concatenating the count information with the data into a single message.

4.2 Allgather, Allreduce

Allgather and allreduce are similar to gather and reduce except that all processes get the result instead of just the root. We consider only allgather in the rest of this section. `MPI_Allgather` is defined as follows.

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

[1]Some of the graphs are not clearly legible when they are printed on paper. We will provide more easily readable graphs in the final version of the paper.

The block of data sent from the *j*th process is received by every process and placed in the *j*th block of the buffer `recvbuf`.

One way of implementing `MPI_Allgather` is to first do an `MPI_Gather` to process 0 and then do an `MPI_Bcast`. Better algorithms exist, however, such as the *ring* algorithm used in MPICH-1.2.4. In the first step of the ring algorithm, each process i sends its contribution to process $i+1$ and receives the contribution from process $i-1$ (with wrap-around). From the second step onwards, each process i forwards to process $i+1$ the data it received from process $i-1$ in the previous step. The entire algorithm, therefore, takes $p-1$ steps, and the bandwidth requirement of each step is $\frac{n}{p}\beta$, where n is the total size of the data to be received by any process from all other processes. Therefore, the time taken by this algorithm is given by

$$T_{ring} = (p-1)\alpha + \frac{p-1}{p}n\beta$$

Note that the bandwidth term cannot be reduced further because each process must receive $\frac{n}{p}$ data from $p-1$ other processes. The latency term, however, can be reduced if we use a logarithmic algorithm that takes $\lg p$ steps. We have devised such an algorithm, which we call *recursive doubling*.

In the recursive doubling algorithm, processes exchange data with each other in $\lg p$ steps as follows.

```
mask = 1;
while (mask < p) {
   dst = rank ^ mask;
   exchange all data with dst;
   mask <<= 1;
}
```

In the first step of recursive doubling, each process exchanges data with a destination process given by the exclusive-or of the rank of the process and a mask that is initialized to 1. The mask is left-shifted by one bit before each of the following steps. In the second step, each process exchanges all the data it has (including the data received in the previous step) with a destination process given by the exclusive-or of its rank and mask=2. For a power-of-two number of processes, in $\lg p$ steps, each process gets the data contributed by all other processes. The bandwidth required is $\frac{n}{p}\beta$ in the first step, $\frac{2n}{p}\beta$ in the second step, and so forth, up to $\frac{2^{\lg p-1}n}{p}\beta$ in the last step. Therefore, the total time taken by this algorithm is given by

$$T_{rec\_doub} = \lg p\, \alpha + \frac{p-1}{p}n\beta$$

The algorithm is straightforward for a power-of-two number of processes but is a little tricky to get right for a non-power-of-two number of processes. We have implemented the non-power-of-two case as follows. At each step of recursive doubling, we ensure that if the current subtree is not a power of two, all processes nonetheless get the data they would have gotten if the subtree had been a power-of-two. This approach is necessary for the subsequent steps of recursive doubling to work correctly. We do the correction step for the non-power-of-two subtree also in a logarithmic fashion in order to minimize the number of steps and hence the latency. The total number of steps for the non-power-of-two case is bounded by $2\lfloor \lg p \rfloor$.

Since recursive doubling does not communicate any extra data and it uses a logarithmic number of steps, we use the recursive doubling algorithm for both short and long message sizes. We use the same algorithm to implement `MPI_Allgatherv` because the `recvcounts` and `displacements` parameters to the function are defined on all processes (unlike in `MPI_Gatherv`).

Performance

Figures 2 and 3 show the performance of allgather with the new and old algorithms. We ran tests for three categories of message size: short messages (up to 16 KB), medium-sized messages (16–256 KB), and large messages (256 KB–8 MB), on 8, 16, 32, 64, 128, and 256 nodes of the cluster. In all cases, for short messages, where latency dominates the overall time, the performance improvement provided by the new algorithm ($\lg p$ steps) compared with the old algorithm ($p - 1$ steps) is evident. For the 256-node case and a total message size of 512 bytes, the new algorithm performed a factor of 10.7 times better than the old algorithm. As message size gets larger, latency plays a smaller role in the overall time. The time is now dominated by the bandwidth cost, which is the same for both algorithms. Therefore, we would expect the algorithms to perform the same for large message sizes. However, the new algorithm consistently outperformed the old algorithm by about 10%, even for very large messages. We surmise that the scheduling of communication in recursive doubling better suits the Myrinet interconnect than in the ring algorithm.

4.3 Broadcast

Broadcast is an important kind of collective communication because it is very commonly used. `MPI_Bcast` is defined as follows.

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int
root, MPI_Comm comm)
```

`MPI_Bcast` broadcasts a message from the process with rank `root` to all processes in the group.

A common algorithm for broadcast is to use a minimum spanning tree (MST). In the first step, the root sends data to process (root + p/2). Each of these processes then acts as a root within its own subtree and recursively continues this algorithm. This communication takes a total of $\lceil \lg p \rceil$ steps. The bandwidth requirement of each step is $n\beta$. Therefore, the time taken by this algorithm is given by

$$T_{MST} = \lceil \lg p \rceil (\alpha + n\beta)$$

This algorithm is used in MPICH-1.2.4. It is good for short messages because it has a logarithmic latency term. For long messages, where latency can be ignored, an algorithm that reduces the bandwidth requirement is better. Van de Geijn et al. proposed implementing a broadcast for long messages as a scatter followed by an allgather [1, 16]. That is, the message to be broadcast is first divided up and scattered among the processes, similar to an `MPI_Scatter`, and the scattered data is then collected back to all processes, similar to an `MPI_Allgather`. They proposed using an MST algorithm for the scatter and a ring algorithm for the allgather. We use the same scatter-allgather technique for long-message broadcast but instead of the ring allgather, we use the recursive doubling allgather described above, because it performs better on a switched network. The time taken by this algorithm is the sum of the times taken by the scatter and the allgather and is given by

$$T_{long\_msg} = 2 \lg p \, \alpha + 2 \frac{p-1}{p} n\beta$$

If we compare this time with that for the MST algorithm, we see that for large messages where latency is negligible and when $\lg p > 2$ (or $p > 4$), the scatter-allgather method is better than MST. The maximum improvement in performance over MST that can be expected is $(\lg p)/2$. In other words, the larger the number of processes, the greater the expected improvement in performance.

Our new algorithm for broadcast uses MST for short messages and also for long messages if the number of processes is less than 8. For long messages on 8 or more processes, we use the scatter-allgather method. The cutoff point between short and long message sizes for selecting between the two algorithms is system dependent. For the NCSA Myrinet cluster we found it to be 12 KB.

Performance

Figures 4 and 5 show the performance of broadcast with the old and new algorithms. We show the performance only for medium and long messages, because for short messages we use the old algorithm. The difference between the two algorithms is more evident for large numbers of processes, because the expected improvement is proportional to $(\lg p)/2$. On 256 nodes, for 8 MB messages, we see an improvement of a factor of 3, whereas the theoretical maximum is a factor of $(\lg 256)/2 = 4$. Another striking result is the scalability of the new algorithm for long messages: the time taken hardly changes as we increase the number of processes. This is most evident in the lower graph in Figure 5 where the lines for the new algorithm are almost overlapping. (Since the lines for the 64-node and 128-node cases overlap very closely, the 64-node case is not visible.) The reason for this result is that the bandwidth requirement of the new algorithm is effectively $2n\beta$, which is independent of the number of processes, whereas in the old algorithm it is $n \lg p \, \beta$.

4.4 Reduce-Scatter

Reduce-scatter is a variant of reduce, where the result, instead of being stored at the root, is scattered among all processes. It is equivalent to calling an `MPI_Reduce` followed by an `MPI_Scatter`.

`MPI_Reduce_scatter` is defined as follows.

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

`MPI_Reduce_scatter` does an elementwise reduction on the vector of elements in the send buffer defined by `sendbuf`, `count`, and `datatype`. The resulting vector of results is split into *p* disjoint segments. Segment *i* contains `recvcounts[i]` elements. The *i*th segment is sent to process *i* and stored in the receive buffer defined by `recvbuf`, `recvcounts[i]`, and `datatype`.

MPICH-1.2.4 implements reduce-scatter by doing an MST reduce to rank 0 followed by a linear scatterv. This takes $\lg p + p - 1$ steps and requires $2\frac{p-1}{p}n\beta$ bandwidth:

$$T_{old} = (\lg p + p - 1)\alpha + 2\frac{p-1}{p}n\beta$$

Our new algorithms do better than that for both short and long messages. For short messages, we use a recursive doubling algorithm, which takes $\lg p$ steps. At step 1, processes exchange $(n - \frac{n}{p})$ amount of data; at step 2, $(n - 2\frac{n}{p})$ amount of data; at step 3, $(n - 2^2\frac{n}{p})$ amount of data; and so forth. Therefore, the time taken by this algorithm is given by

$$T_{short} = \lg p\alpha + n(\lg p - \frac{p-1}{p})\beta$$

For long messages, we use a pairwise exchange algorithm. At step i, each process sends $\frac{n}{p}$ amount of data to $(rank + i)$ and receives $\frac{n}{p}$ amount of data from $(rank - i)$ (with wraparound). The time taken by this algorithm is given by

$$T_{long} = (p-1)\alpha + \frac{p-1}{p} n\beta$$

The cutoff point for switching between the two algorithms is a bit tricky in the case of reduce-scatter. We found it to depend on the message size as well as the number of processes. For 8, 16, and 32 processes, the cutoff point was 512 bytes. For 64 processes it was 1500 bytes. For 128 processes it was 15 KB. For 256 processes it was 50 KB. The reason is the $n \lg p$ term in the bandwidth requirement of the short-message algorithm. On small numbers of processes, the effect of $\lg p$ steps wears off quickly as the message size increases, because of the $n \lg p$ term. On larger numbers of processes, the effect of taking $\lg p$ steps is greater, and the short-message algorithm performs better up to a larger message size. For the final version of this paper, we will study this issue further and provide a formula for determining the cutoff point based on the message size and the number of processes.

Performance

Figure 6 shows the performance of reduce-scatter with the old and new algorithms. For short messages, we see the usual benefits of $\lg p$ versus ($\lg p + p - 1$) steps. (Some inexplicable, but reproducible, spikes occur in the graphs for the old algorithm.) For long messages, the new algorithm performs better because it requires half the bandwidth compared with the old algorithm. As in the case of broadcast, another striking result for long messages is the scalability of the new algorithm: the time taken hardly increases as we increase the number of processes. In the two graphs for large messages, the lines depicting 8, 16, and 32 processes and the lines depicting 64, 128, and 256 processes almost overlap. This result is because the bandwidth cost in the long-message algorithm is effectively $n\beta$, which is independent of the number of processes. This is true even for the old algorithm, where the bandwidth requirement is $2n\beta$, but in practice we see that the time taken increases for larger numbers of processes with the old algorithm.

4.5 Alltoall

All-to-all communication is a common form of collective communication used in parallel quicksort, some implementations of the 2D FFT, matrix transpose, array redistribution, and so forth. It is a dense form of communication involving exchange of data from all processes to all other processes. `MPI_Alltoall` is defined as follows.

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

Each process sends `sendcount` elements of type `sendtype` to all other processes. The data sent to each process is distinct and is taken from `sendbuf` in order of process rank. The *j*th block sent from process *i* is received by process *j* and is placed in the *i*th block of `recvbuf`.

MPICH-1.2.4 implements all-to-all by simply posting all the nonblocking receives and sends and then calling `MPI_Waitall` on all of them.

```
for (i=0; i<p; i++) {
    MPI_Irecv(src=i);
    MPI_Isend(dest=i);
}
MPI_Waitall();
```

The drawback of this algorithm is that messages are not ordered. All messages get sent first to process 0, then to process 1, and so forth, resulting in a single-node bottleneck.

A better approach is to implement all-to-all as a series of exchanges among pairs of processes. Such an algorithm will take $p-1$ steps. The question then is: How do we form the pairs at each step? One method that has been proposed for hypercube machines is the following [3, 4, 15].

```
for (i=1; i<p; i++) {
    dest = rank ^ i;
    exchange data with dest;
}
```

In each step, each process exchanges data with the destination given by the rank of the process exclusive-or'ed with the step number. This algorithm works only for a power-of-two number of processes. It has been shown that for hypercube systems, this algorithm minimizes link contention at each step. The algorithm can be modified to work for non-power-of-two systems, but the number of steps increases to the next higher power of two.

Since we assume a switched communication network, where pairs of processes can communicate without contention, we use a different method for pairing that always takes $p-1$ steps even for a non-power-of-two number of processes. The algorithm is simple: At step i, each process receives from process $i-1$ and sends to process $i+1$ (with wrap-around).

```
for (i=1; i<p; i++) {
    src = (rank - i + p) % p;
    dest = (rank + i) % p;
    MPI_Sendrecv(from src, to dest);
}
```

If n is the total amount of data to be sent from one process to all other processes, the time taken by this algorithm is given by

$$T_{PEX} = (p-1)\alpha + \frac{p-1}{p} n\beta$$

This is a good algorithm for long messages because it consumes no more bandwidth than is necessary (the bandwidth term cannot be reduced further). For short messages, however, we can do better if we use an algorithm that has a logarithmic number of steps, even at the expense of some extra bandwidth. We have devised such an algorithm, derived from the recursive doubling algorithm described above for allgather. The algorithm is as follows.

```
mask = 1;
while (mask < p) {
   dst = rank ^ mask;
   exchange with dst all data received so far;
   mast <<= 1;
}
```

Process pairings in each step are done in the same way as for allgather. In the first step, each process sends all the data it has to the destination process (including data meant for other processes). In the second step, each process sends its own data as well as the data received in the first step to the destination process.

This procedure continues for $\lg p$ steps. At the end, all processes have all the data that all processes had in the beginning. Each process then selects the data intended for itself and discards the rest. The bandwidth required in the first step is $n\beta$, the bandwidth required in the second step is $2n\beta$, and so on up to $2^{\lg p - 1} n\beta$ in the last step. Therefore, the time taken by this algorithm is given by

$$T_{PEX} = \lceil \lg p \rceil \alpha + np\beta$$

Although it consumes a lot more bandwidth than pairwise exchange, this algorithm is better for latency-sensitive short messages. We therefore use recursive doubling for very short messages and pairwise exchange for long messages.

4.5.1 Performance

Because of lack of time, we did not complete our experimental performance study for all-to-all by the deadline for this conference. We will report performance results for all-to-all in the final version of this paper. Preliminary results demonstrate the benefit of using the $\lg p$ algorithm for very short messages. For long messages, however, the pairwise exchange algorithm sometimes performs better than the old algorithm and sometimes does not. We don't understand why simply posting all the receives and sends would work better than an ordered pairwise exchange. We plan to investigate this issue with the help of our contact at Myricom and report the results in the final version of the paper.

4.5.2 Alltoallv

MPI also supports a "v" version of all-to-all, where a process can send a different amount of data to each of the other processes. We have implemented alltoallv by using the pairwise exchange algorithm for both short and long messages. We do not use recursive doubling for short messages because a process does not know any other process's `sendcounts` or `recvcounts` arrays. If we were to communicate that information, it would defeat the low-latency benefit of recursive doubling.

5 Conclusions and Future Work

We have presented new algorithms for improving the performance of MPI collective communication on switched networks. The algorithms will be released as part of the next release of MPICH (1.2.5). Our approach to designing algorithms that minimize latency for short messages and minimize bandwidth usage for large messages has paid off well. We have observed performance improvement of up to a factor of 10 for short messages in the case of scatter/gather/reduce, allgather/allreduce, and reduce-scatter. We have also achieved scalable performance for long messages in the case of broadcast and reduce-scatter, where the time taken remains constant even as we increase the number of processes.

In the final version of the paper we will provide complete performance results and explain the unusual spikes in some of the graphs. We will also provide a mathematical formula for determining the cutoff point between short-message and long-message algorithms in the case of reduce-scatter and broadcast.

In the future, we plan to extend these algorithms to run efficiently on clusters of SMPs, where processes running on the same node can communicate via shared memory.

References

[1] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor collective communication library (InterCom). In *Proceedings of Supercomputing '94*, November 1994.

[2] M. Barnett, R. Littlefield, D. Payne, and R. van de Geijn. Global combine on mesh architectures with wormhole routing. In *Proceedings of the* 7^{th} *International Parallel Processing Symposium*, April 1993.

[3] S. Bokhari. Complete exchange on the iPSC/860. Technical Report 91–4, ICASE, NASA Langley Research Center, 1991.

[4] S. Bokhari and H. Berryman. Complete exchange on a circuit switched mesh. In *Proceedings of the Scalable High Performance Computing Conference*, pages 300–306, 1992.

[5] Graham E. Fagg, Sathish S. Vadhiyar, and Jack J. Dongarra. ACCT: Automatic collective communications tuning. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, pages 354–361. Lecture Notes in Computer Science 1908, Springer, September 2000.

[6] S. L. Johnsson and C. T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, pages 1249–1268, September 1989.

[7] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the Fourteenth International Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 377–384, 2000.

[8] T. Kielman, H. E. Bal, and S. Gorlatch. Bandwidth-efficient collective communication for clustered wide-area systems. In *Proceedings of the Fourteenth International Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 492–499, 2000.

[9] 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. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140. ACM, May 1999.

[10] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 1.1, June 1995. `http://www.mpi-forum.org/docs/docs.html`.

[11] P. Mitra, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Fast collective communication libraries, please. In *Proceedings of the Intel Supercomputing Users' Group Meeting*, June 1995.

[12] MPICH – A portable implementation of MPI. `http://www.mcs.anl.gov/mpi/mpich`.

[13] R. Ponnusamy, Rajeev Thakur, Alok Choudhary, and G. Fox. Scheduling regular and irregular communication patterns on the CM-5. In *Proceedings of Supercomputing '92*, pages 394–402, November 1992.

[14] Peter Sanders and Jesper Larsson Träff. The hierarchical factor algorithm for all-to-all communication. In B. Monien and R. Feldman, editors, *Euro-Par 2002 Parallel Processing*, pages 799–803. Lecture Notes in Computer Science 2400, Springer, August 2002.

[15] D. Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *Proceedings of the* 6^{th} *Distributed Memory Computing Conference*, pages 398–403, 1991.

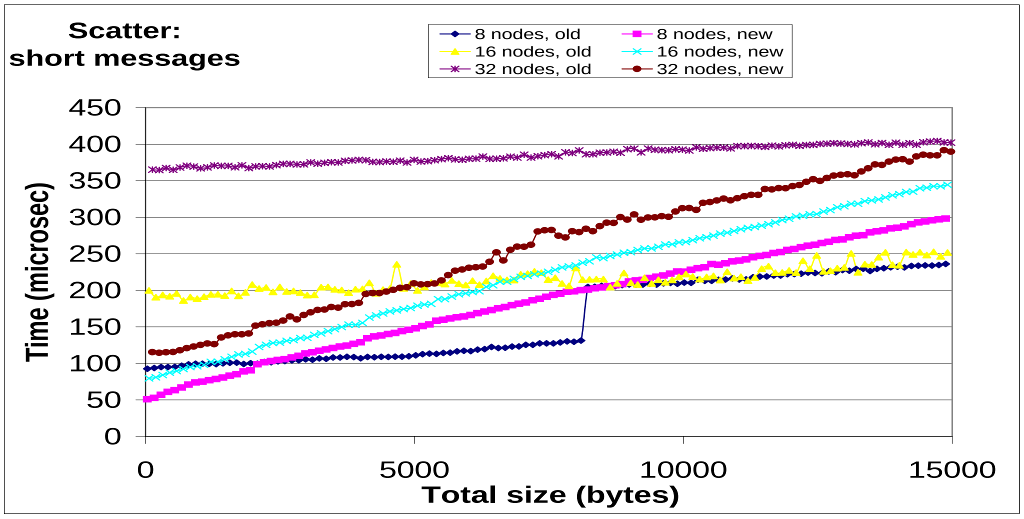
[16] Mohak Shroff and Robert A. van de Geijn. CollMark: MPI collective communication benchmark. Technical report, Dept. of Computer Sciences, University of Texas at Austin, December 1999.

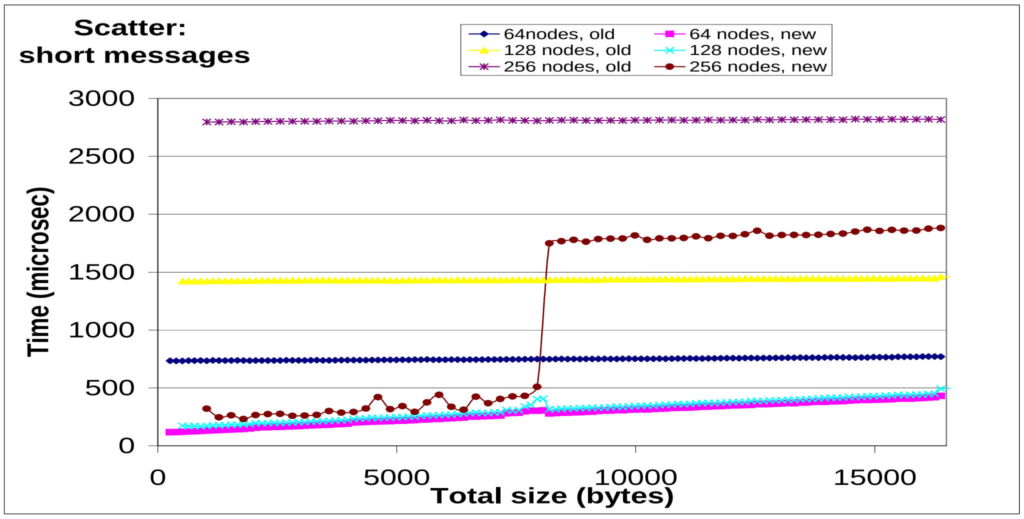
[17] Steve Sistare, Rolf vandeVaart, and Eugene Loh. Optimization of MPI collectives on clusters of large-scale SMPs. In *Proceedings of SC99: High Performance Networking and Computing*, November 1999.

[18] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core,* 2nd edition. MIT Press, Cambridge, MA, 1998.

[19] Jesper Larsson Träff. Improved MPI all-to-all communication on a Giganet SMP cluster. In Dieter Kranzlmuller, Peter Kacsuk, Jack Dongarra, and Jens Volkert, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface,* pages 392–400. Lecture Notes in Computer Science 2474, Springer, September 2002.

[20] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings of SC99: High Performance Networking and Computing,* November 1999.





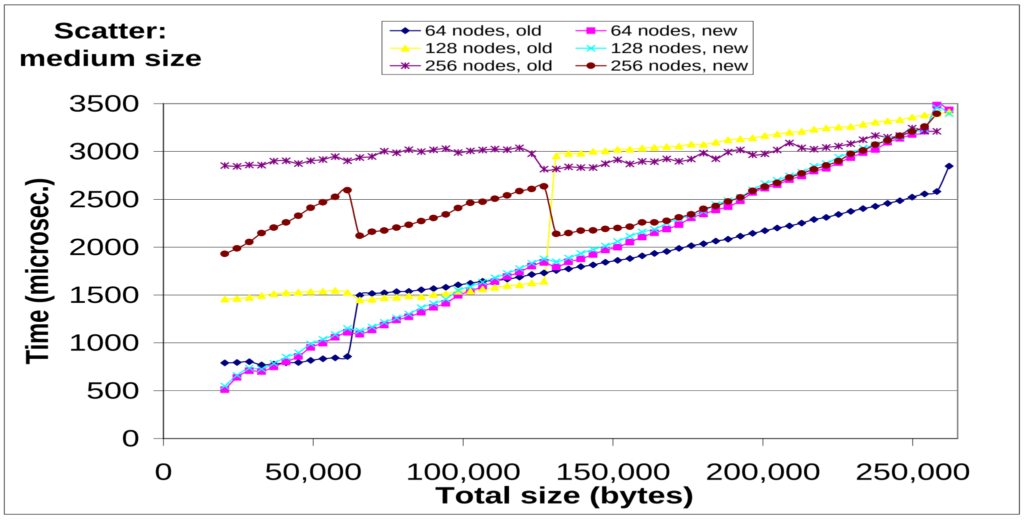


Figure 1: Performance of scatter. The old algorithm is a linear algorithm; the new algorithm uses a minimum spanning tree.

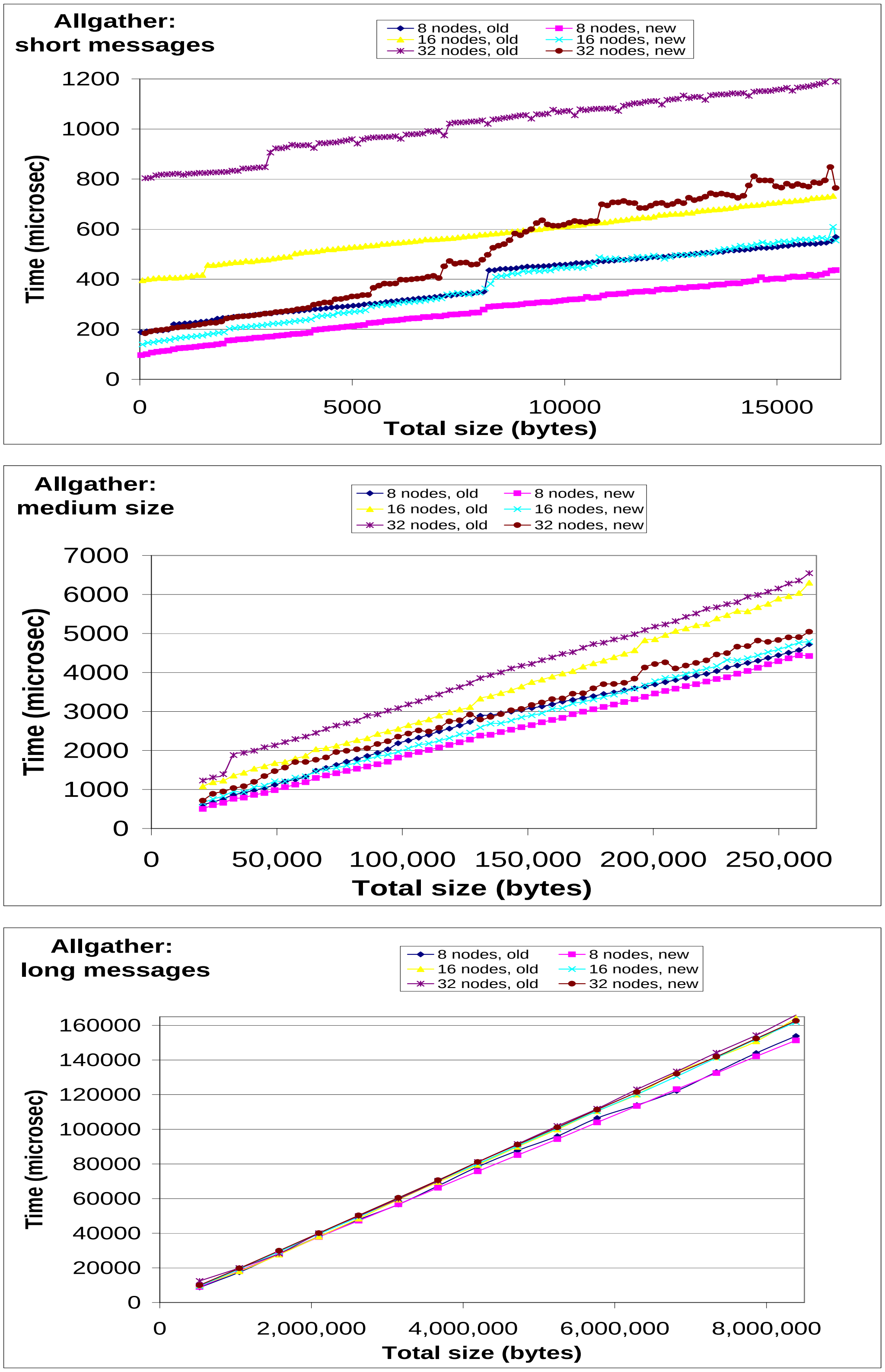
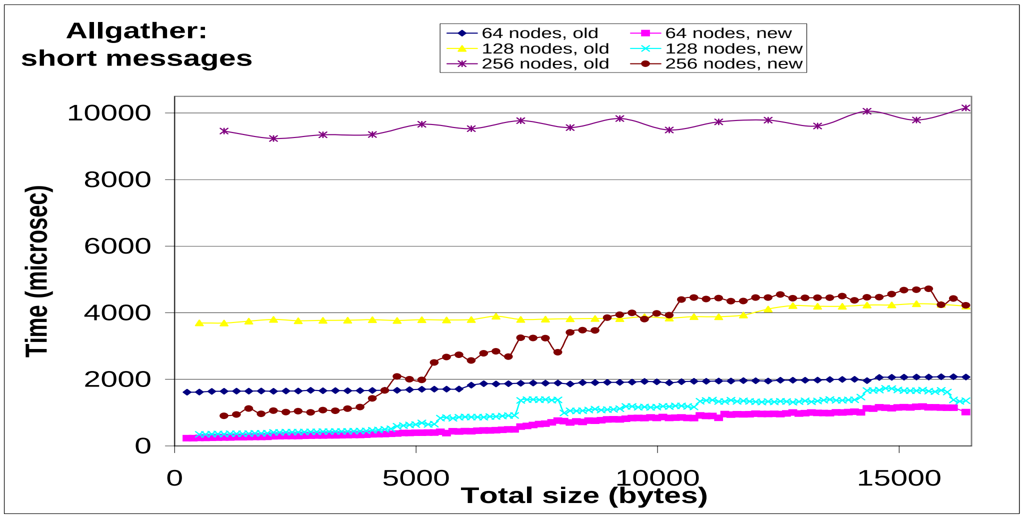
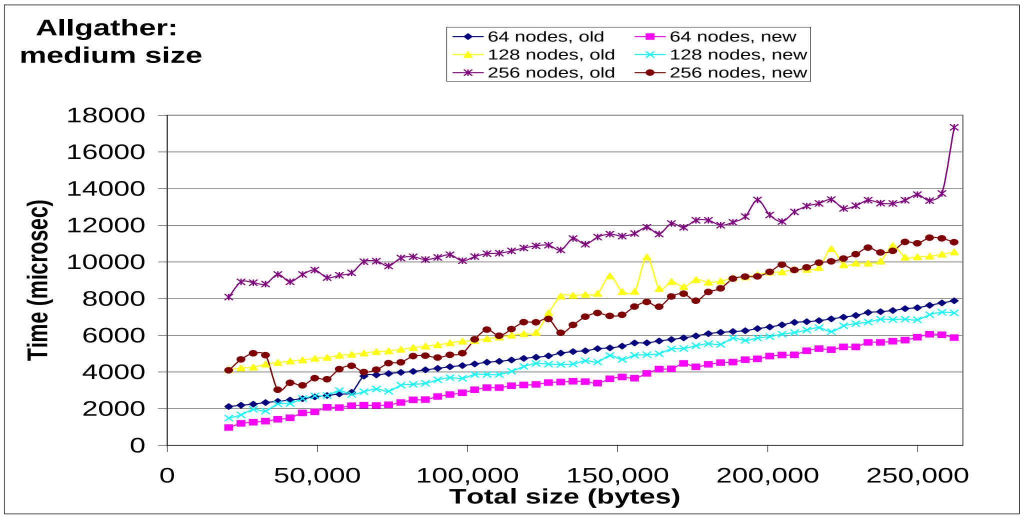


Figure 2: Performance of allgather on 8, 16, and 32 processes. The old algorithm is a ring algorithm; the new algorithm uses recursive doubling.





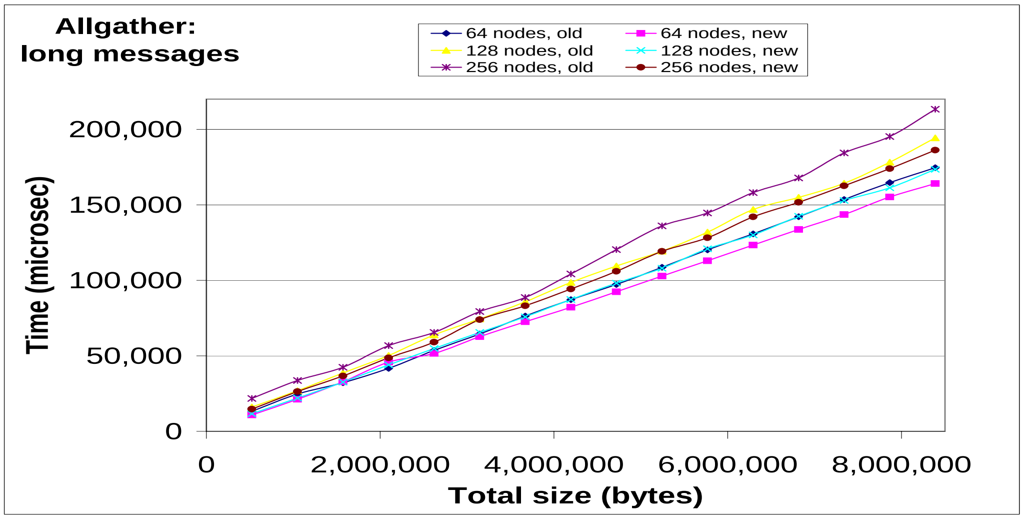
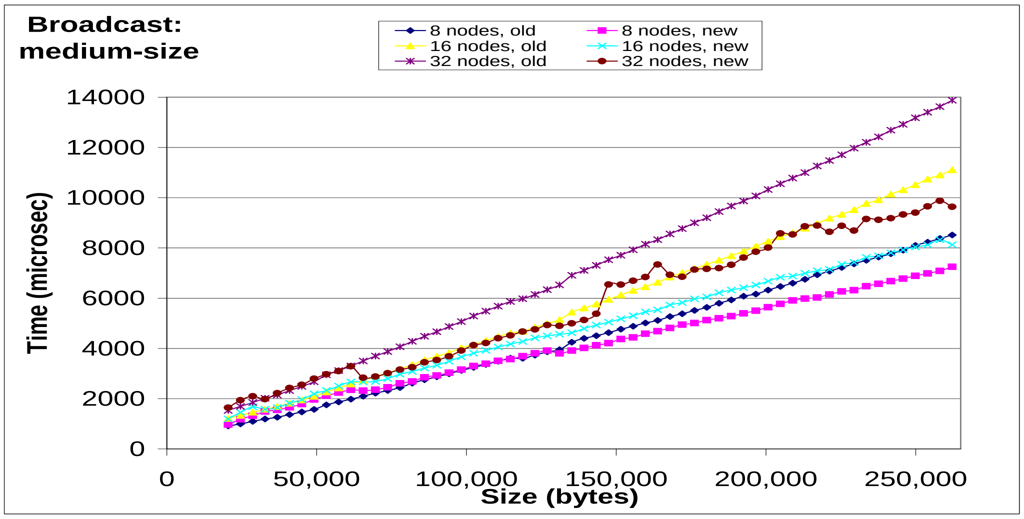


Figure 3: Performance of allgather on 64, 128, and 256 processes. The old algorithm is a ring algorithm; the new algorithm uses recursive doubling.



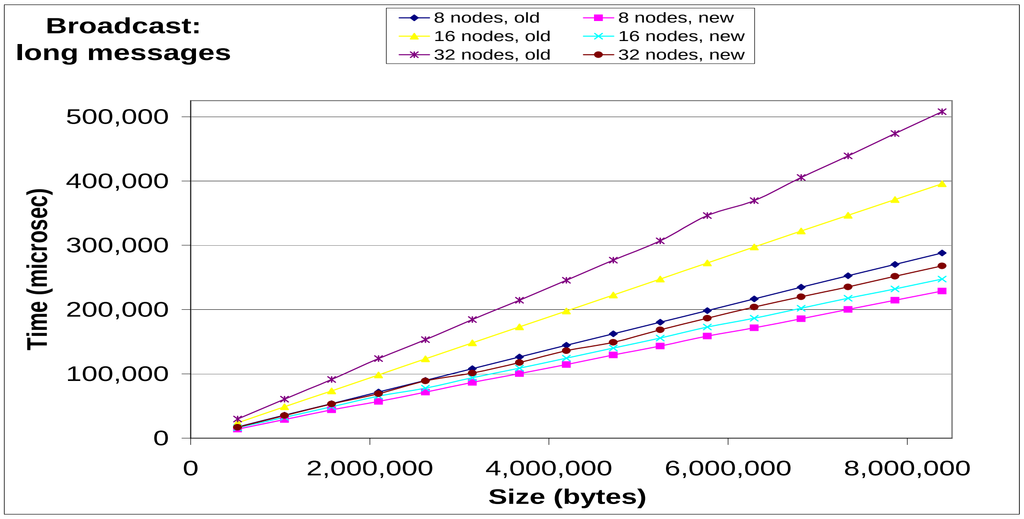
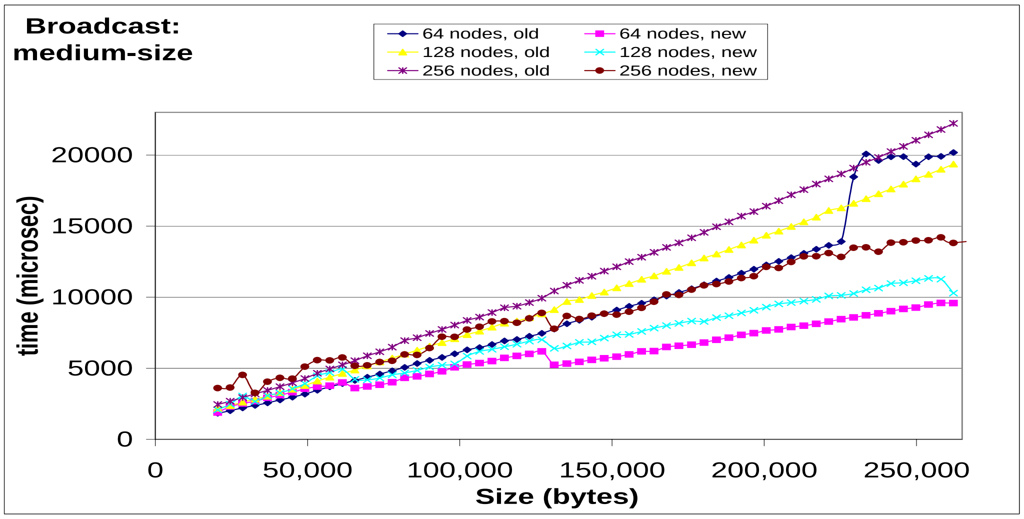


Figure 4: Performance of broadcast on 8, 16, and 32 processes. The old algorithm is a minimum spanning tree (MST) algorithm; the new algorithm uses MST for short messages and scatter-allgather otherwise.



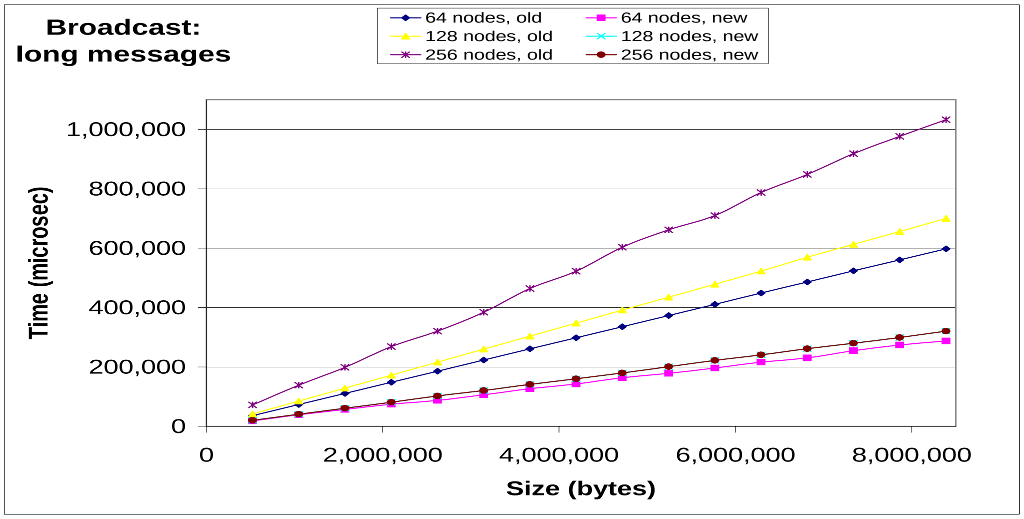
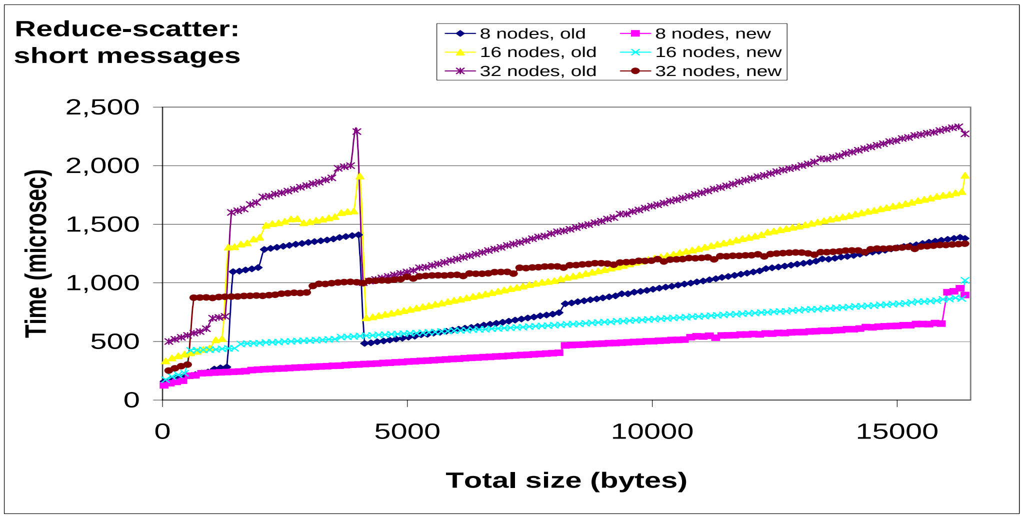
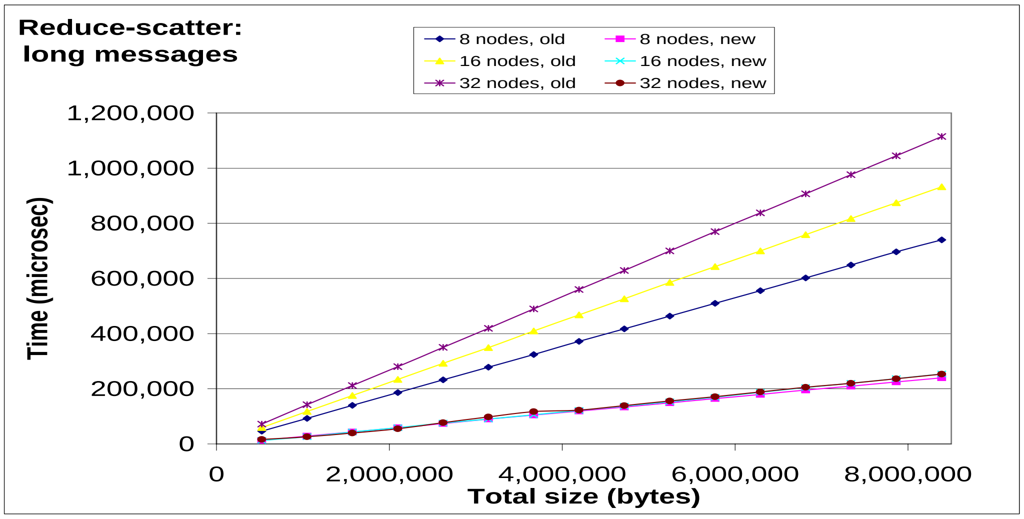


Figure 5: Performance of broadcast on 64, 128, and 256 processes. The old algorithm is a minimum spanning tree (MST) algorithm; the new algorithm uses MST for short messages and scatter-allgather otherwise.





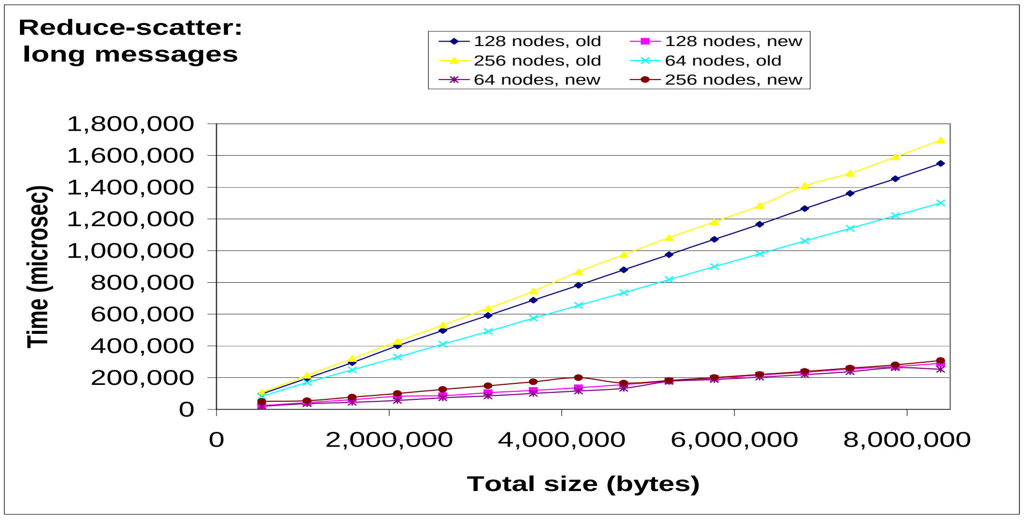


Figure 6: Performance of reduce-scatter. The old algorithm does an MST reduce to rank 0 followed by a linear scatterv. The new algorithm uses recursive doubling for short messages and pairwise exchange otherwise.