OPTIMIZATION OF COLLECTIVE COMMUNICATION OPERATIONS IN MPICH

Rajeev Thakur[1]
Rolf Rabenseifner[2]
William Gropp[1]

Abstract

We describe our work on improving the performance of collective communication operations in MPICH for clusters connected by switched networks. For each collective operation, we use multiple algorithms depending on the message size, with the goal of minimizing latency for short messages and minimizing bandwidth use for long messages. Although we have implemented new algorithms for all MPI (Message Passing Interface) collective operations, because of limited space we describe only the algorithms for allgather, broadcast, all-to-all, reduce-scatter, reduce, and allreduce. Performance results on a Myrinet-connected Linux cluster and an IBM SP indicate that, in all cases, the new algorithms significantly outperform the old algorithms used in MPICH on the Myrinet cluster, and, in many cases, they outperform the algorithms used in IBM's MPI on the SP. We also explore in further detail the optimization of two of the most commonly used collective operations, allreduce and reduce, particularly for long messages and non-power-of-two numbers of processes. The optimized algorithms for these operations perform several times better than the native algorithms on a Myrinet cluster, IBM SP, and Cray T3E. Our results indicate that to achieve the best performance for a collective communication operation, one needs to use a number of different algorithms and select the right algorithm for a particular message size and number of processes.

Key words: Collective communication, message passing, MPI, reduction

The International Journal of High Performance Computing Applications,
Volume 19, No. 1, Spring 2005, pp. 49–66
DOI: 10.1177/1094342005051521
© 2005 Sage Publications

1 Introduction

Collective communication is an important and frequently used component of MPI (Message Passing Interface) and offers implementations considerable room for optimization. Although widely used as an MPI implementation, MPICH (a portable implementation of MPI; see http://www.mcs.anl.gov/mpi/mpich) has until recently had fairly rudimentary implementations of the collective operations. This paper describes our efforts at improving the performance of collective operations in MPICH. Our initial target architecture is one that is very popular among our users, namely, clusters of machines connected by a switch, such as Myrinet or the IBM SP switch. Our approach has been to identify the best algorithms known in the literature, improve on them or develop new algorithms where necessary, and implement them efficiently. For each collective operation, we use multiple algorithms based on message size: the short-message algorithms aim to minimize latency, and the long-message algorithms aim to minimize bandwidth use. We use experimentally determined cutoff points to switch between different algorithms depending on the message size and number of processes. We have implemented new algorithms in MPICH (MPICH 1.2.6 and MPICH2 1.0) for all the MPI collective operations: scatter, gather, allgather, broadcast, all-to-all, reduce, allreduce, reduce-scatter, scan, barrier, and their variants. Because of limited space, however, we describe only the new algorithms for allgather, broadcast, all-to-all, reduce-scatter, reduce, and allreduce.

A five-year profiling study of applications running in production mode on the Cray T3E 900 at the University of Stuttgart revealed that more than 40% of the time spent in MPI functions was spent in the two functions MPI_Allreduce and MPI_Reduce and that 25% of all execution time was spent on program runs that involved a non-power-of-two number of processes (Rabenseifner 1999). We therefore investigated in further detail how to optimize allreduce and reduce. We present a detailed study of different ways of optimizing allreduce and reduce, particularly for long messages and non-power-of-two numbers of processes, both of which occur frequently according to the profiling study.

The rest of this paper is organized as follows. In Section 2, we describe related work in the area of collective communication. In Section 3, we describe the cost model used to guide the selection of algorithms. In Section 4, we describe the new algorithms in MPICH and their perform-

[1]MATHEMATICS AND COMPUTER SCIENCE DIVISION ARGONNE NATIONAL LABORATORY ARGONNE, IL 60439, USA (THAKUR@MCS.ANL.GOV)

[2]RECHENZENTRUM UNIVERSITAT STUTTGART (RUS) HIGH PERFORMANCE COMPUTING CENTER (HLRS) UNIVERSITY OF STUTTGART D-70550 STUTTGART, GERMANY

ance. In Section 5, we investigate in further detail the optimization of reduce and allreduce. In Section 6, we conclude with a brief discussion of future work.

2 Related Work

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 (Bokhari 1991; Scott 1991; Bokhari and Berryman 1992; Barnett et al. 1993). More recently, Vadhiyar et al. (1999) have developed automatically tuned collective communication algorithms. They run experiments to measure the performance of different algorithms for a collective communication operation under different conditions (message size, number of processes) and then use the best algorithm for a given set of conditions. Researchers in Holland and at Argonne have optimized MPI collective communication for wide-area distributed environments (Kielmann et al. 1999; Karonis et al. 2000). 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. Researchers have also developed collective communication algorithms for clusters of SMPs (Sistare et al. 1999; Träff 2002; Sanders and Träff 2002; Tipparaju et al. 2003), where communication within an SMP is done differently from communication across a cluster. Some efforts have focused on using different algorithms for different message sizes, such as the work by van de Geijn and co-workers (Barnett et al. 1994; Mitra et al. 1995; Shroff and van de Geijn 1999; Chan et al. 2004), by Rabenseifner on reduce and allreduce (http://www.hlrs.de/organization/par/services/models/mpi/myreduce.html), and by Kale et al. (2003) on all-to-all communication. Benson et al. (2003) studied the performance of the allgather operation in MPICH on Myrinet and TCP networks and developed a dissemination allgather based on the dissemination barrier algorithm. Bruck et al. (1997) proposed algorithms for allgather and all-to-all that are particularly efficient for short messages. Iannello (1997) developed efficient algorithms for the reduce-scatter operation in the LogGP model.

3 Cost Model

We use a simple model to estimate the cost of the collective communication algorithms in terms of latency and bandwidth use, and to guide the selection of algorithms for a particular collective communication operation. This model is similar to the one used by van de Geijn (Barnett et al. 1994; Mitra et al. 1995; Shroff and van de Geijn 1999), Hockney (1994), and others. Although more sophisticated models such as LogP (Culler et al. 1993) and LogGP (Alexandrov et al. 1997) exist, this model is sufficient for our needs.

We assume that the time taken to send a message between any two nodes can be modeled as $\alpha + n\beta$, where α is the latency (or startup time) per message, independent of message size, β is the transfer time per byte, and n is the number of bytes transferred. We assume further that the time taken is independent of how many pairs of processes are communicating with each other, independent of the distance between the communicating nodes, and that the communication links 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 assumed to be single ported; that is, at most one message can be sent and one message can be received simultaneously. In the case of reduction operations, we assume that γ is the computation cost per byte for performing the reduction operation locally on any process.

This cost model assumes that all processes can send and receive one message at the same time, regardless of the source and destination. Although this is a good approximation, many networks are faster if pairs of processes exchange data with each other, rather than if a process sends to and receives from different processes (Benson et al. 2003). Therefore, for the further optimization of reduction operations (Section 5), we refine the cost model by defining two costs: $\alpha + n\beta$ is the time taken for bidirectional communication between a pair of processes, and $\alpha_{uni} + n\beta_{uni}$ is the time taken for unidirectional communication from one process to another. We also define the ratios $f_\alpha = \alpha_{uni} / \alpha$ and $f_\beta = \beta_{uni} / \beta$. These ratios are normally in the range 0.5 (simplex network) to 1.0 (full-duplex network).

4 Algorithms

In this section we describe the new algorithms and their performance. We measured performance by using the SKaMPI benchmark (Worsch et al. 2002) on two platforms: a Linux cluster at Argonne connected with Myrinet 2000 and the IBM SP at the San Diego Supercomputer Center. On the Myrinet cluster, we used MPICH-GM and compared the performance of the new algorithms with the old algorithms in MPICH-GM. On the IBM SP, we used IBM's MPI and compared the performance of the new algorithms with the algorithms used in IBM's MPI. On both systems, we ran one MPI process per node. We implemented the new algorithms as functions on top of MPI point-to-point operations, so that we could compare performance simply by linking or not linking the new functions.

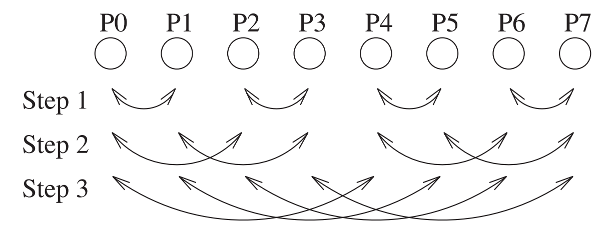


Fig. 1 Recursive doubling for allgather.

4.1 ALLGATHER

`MPI_Allgather` is a gather operation in which the data contributed by each process are gathered on all processes, instead of just the root process as in `MPI_Gather`. The old algorithm for allgather in MPICH uses a ring method in which the data from each process are sent around a virtual ring of processes. In the first step, 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 onward, each process i forwards to process $i + 1$ the data it received from process $i - 1$ in the previous step. If p is the number of processes, the entire algorithm takes $p - 1$ steps. If n is the total amount of data to be gathered on each process, then at every step each process sends and receives n / p amount of data. Therefore, the time taken by this algorithm is given by $T_{ring} = (p - 1)\alpha + ((p - 1) / p)n\beta$. Note that the bandwidth term cannot be reduced further because each process must receive n / p data from $p - 1$ other processes. The latency term, however, can be reduced by using an algorithm that takes $\lg p$ steps. We consider two such algorithms: recursive doubling and the Bruck algorithm (Bruck et al. 1997).

4.1.1 Recursive Doubling. Figure 1 illustrates how recursive doubling works. In the first step, processes that are a distance 1 apart exchange their data. In the second step, processes that are a distance 2 apart exchange their own data as well as the data they received in the previous step. In the third step, processes that are a distance 4 apart exchange their own data as well the data they received in the previous two steps. In this way, for a power-of-two number of processes, all processes obtain all the data in $\lg p$ steps. The amount of data exchanged by each process is n / p in the first step, $(2n) / p$ in the second step, and so forth, up to $(2^{\lg p - 1}n) / p$ in the last step. Therefore, the total time taken by this algorithm is $T_{rec\_dbl} = \lg p\alpha + ((p - 1) / p)n\beta$.

Recursive doubling works very well for a power-of-two number of processes but is 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, if any set of exchanging processes is not a power of two, we perform additional communication in the peer (power-of-two) set in a logarithmic fashion to ensure that all processes obtain the data they would have obtained had the number of processes been a power of two. This extra communication is necessary for the subsequent steps of recursive doubling to work correctly. The total number of steps for the non-power-of-two case is bounded by $2\lfloor \lg p \rfloor$.

4.1.2 Bruck Algorithm. The Bruck algorithm for allgather (Bruck et al. 1997), referred to as concatenation, is a variant of the dissemination algorithm for barrier, described in Hensgen et al. (1988). Both algorithms take $\lceil \lg p \rceil$ steps in all cases, even for non-power-of-two numbers of processes. In the dissemination algorithm for barrier, in each step k ($0 \le k < \lceil \lg p \rceil$), process i sends a (zero-byte) message to process $(i + 2^k)$ and receives a (zero-byte) message from process $(i - 2^k)$ (with wrap-around). If the same order were used to perform an allgather, it would require communicating non-contiguous data in each step in order to obtain the right data for the right process (see Benson et al. 2003 for details). The Bruck algorithm avoids this problem nicely by a simple modification to the dissemination algorithm in which, in each step k, process i sends data to process $(i - 2^k)$ and receives data from process $(i + 2^k)$, instead of the other way around. The result is that all communication is contiguous, except that, at the end, the blocks in the output buffer must be shifted locally to place them in the right order, which is a local memory-copy operation.

Figure 2 illustrates the Bruck algorithm for an example with six processes. The algorithm begins by copying the input data on each process to the top of the output buffer. In each step k, process i sends to the destination $(i - 2^k)$ all the data it has so far and stores the data it receives (from rank $(i + 2^k)$) at the end of the data it currently has. This procedure continues for $\lfloor \lg p \rfloor$ steps. If the number of processes is not a power of two, an additional step is needed in which each process sends the first $(p - 2^{\lfloor \lg p \rfloor})$ blocks from the top of its output buffer to the destination and appends the data it receives to the data it already has. Each process now has all the data it needs, but the data are not in the right order in the output buffer: The data on process i are shifted up by i blocks. Therefore, a simple local shift of the blocks downwards by i blocks brings the data into the desired order. The total time taken by this algorithm is $T_{bruck} = \lceil \lg p \rceil\alpha + ((p - 1) / p)n\beta$.

4.1.3 Performance. The Bruck algorithm has lower latency than recursive doubling for non-power-of-two numbers of processes. For power-of-two numbers of processes, however, the Bruck algorithm requires local memory permutation at the end, whereas recursive doubling does not. In practice, we find that the Bruck algorithm is

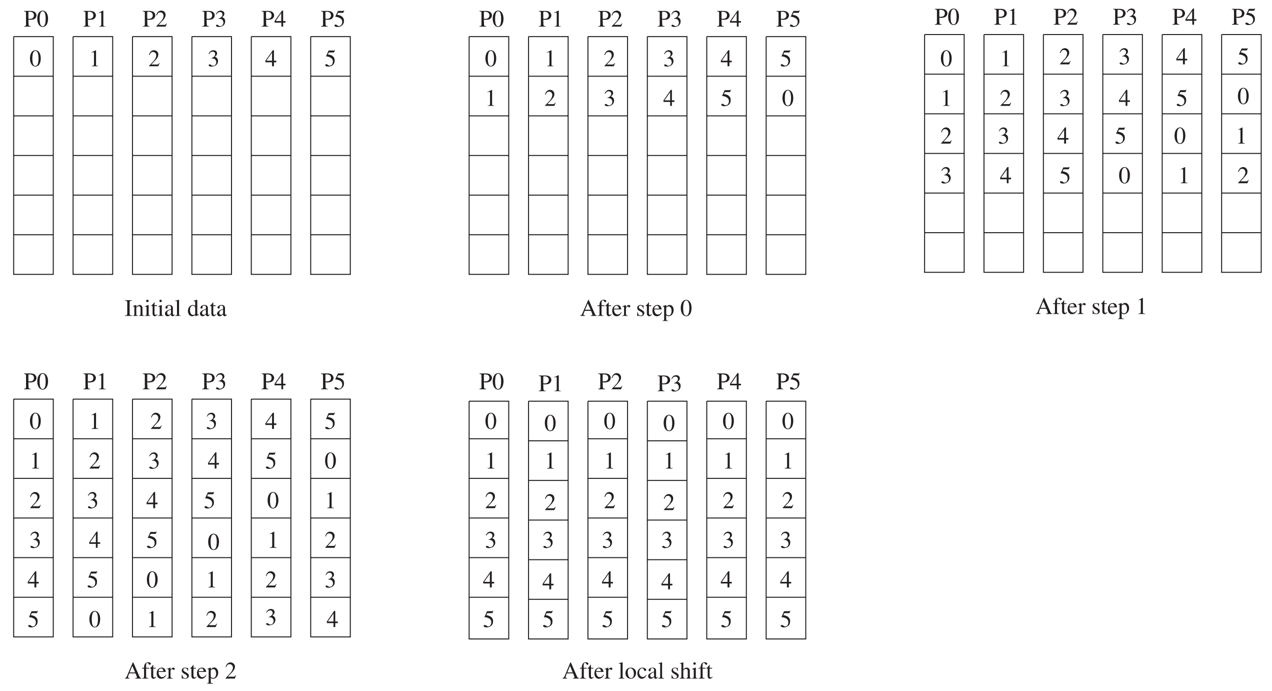


Fig. 2 Bruck allgather.

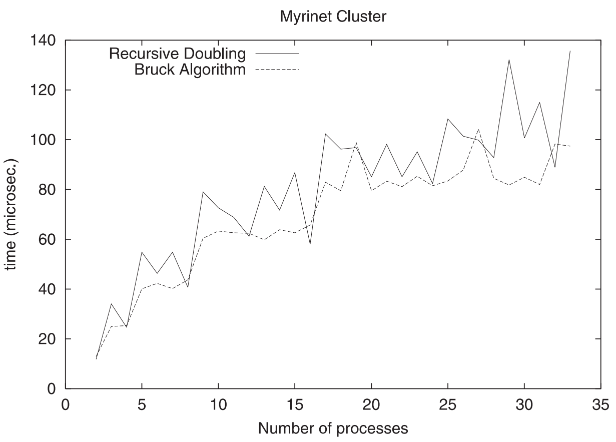


Fig. 3 Performance of recursive doubling versus Bruck allgather for power-of-two and non-power-of-two numbers of processes (message size 16 bytes per process).

best for short messages and non-power-of-two numbers of processes; recursive doubling is best for power-of-two numbers of processes and short or medium-sized messages; and the ring algorithm is best for long messages and any number of processes and also for medium-sized messages and non-power-of-two numbers of processes.

Figure 3 shows the advantage of the Bruck algorithm over recursive doubling for short messages and non-power-of-two numbers of processes because it takes fewer steps. For power-of-two numbers of processes, however, recursive doubling performs better because of the pairwise nature of its communication pattern and because it does not need any memory permutation. As the message size increases, the Bruck algorithm suffers because of the memory copies. In MPICH, therefore, we use the Bruck algorithm for short messages (< 80 KB total data gathered) and non-power-of-two numbers of processes, and recursive doubling for power-of-two numbers of processes and short or medium-sized messages (< 512 KB total data gathered). For short messages, the new allgather performs significantly better than the old allgather in MPICH, as shown in Figure 4.

For long messages, the ring algorithm performs better than recursive doubling (see Figure 5). We believe this is because it uses a nearest-neighbor communication pat-

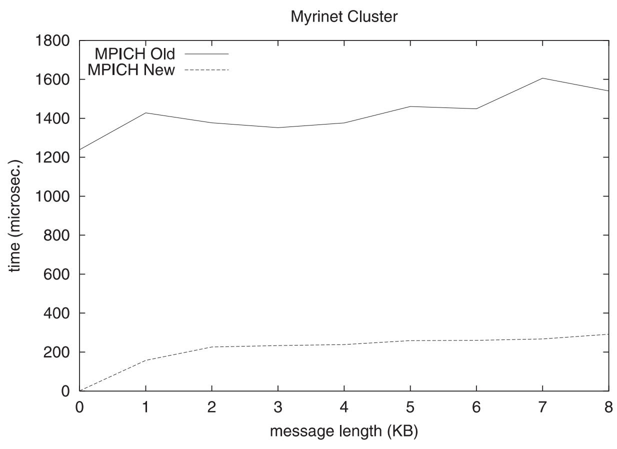
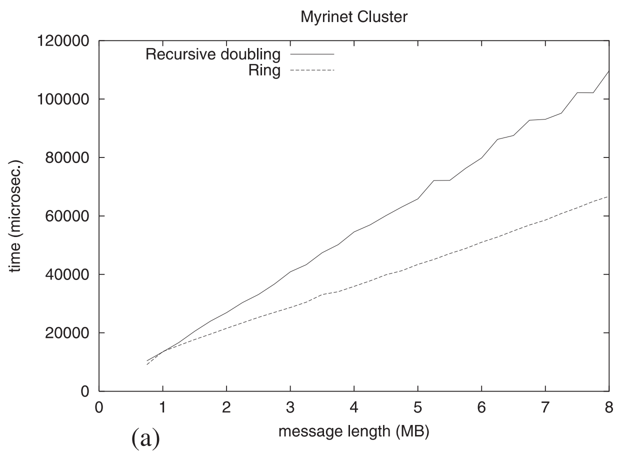


Fig. 4 Performance of allgather for short messages (64 nodes). The size on the x-axis is the total amount of data gathered on each process.



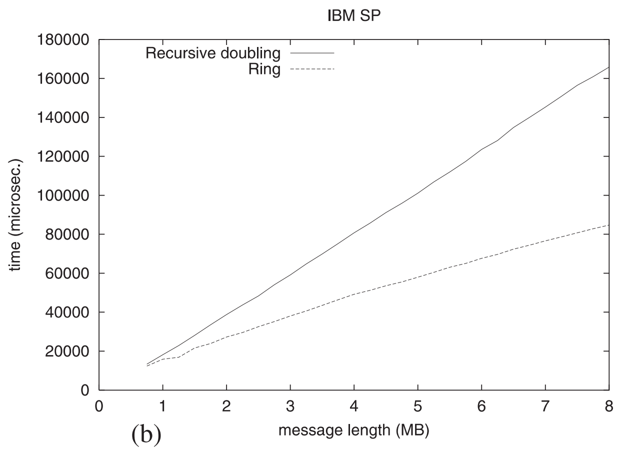


Fig. 5 Ring algorithm versus recursive doubling for long-message allgather (64 nodes). The size on the x-axis is the total amount of data gathered on each process.

tern, whereas in recursive doubling, processes that are much farther apart communicate. To confirm this hypothesis, we used the *b_eff* MPI benchmark (http://www.hlrs.de/mpi/b_eff/), which measures the performance of about 48 different communication patterns, and found that, for long messages on both the Myrinet cluster and the IBM SP, some communication patterns (particularly nearest-neighbor) achieve more than twice the bandwidth of other communication patterns. In MPICH, therefore, for long messages ($\geq$ 512 KB total data gathered) and any number of processes and also for medium-sized messages ($\geq$ 80 KB and < 512 KB total data gathered) and non-power-of-two numbers of processes, we use the ring algorithm.

4.2 BROADCAST

The old algorithm for broadcast in MPICH is the commonly used binomial tree algorithm. In the first step, the root sends data to process ($root + (p / 2)$). This process and the root then act as new roots within their own subtrees and recursively continue this algorithm. This communication takes a total of $\lceil \lg p \rceil$ steps. The amount of data communicated by a process at any step is n. Therefore, the time taken by this algorithm is $T_{tree} = \lceil \lg p \rceil(\alpha + n\beta)$.

This algorithm is good for short messages because it has a logarithmic latency term. For long messages, however, a better algorithm has been proposed by Barnett et al. (1994) and Shroff and van de Geijn (1999) that has a lower bandwidth term. In this algorithm, the message to be broadcast is first divided up and scattered among the processes, similar to an `MPI_Scatter`. The scattered data are then collected back to all processes, similar to an `MPI_Allgather`. The time taken by this algorithm is the sum of the times taken by the scatter, which is ($\lg p\,\alpha + ((p - 1) / p)n\beta$) for a binomial tree algorithm, and the allgather for which we use either recursive doubling or the ring algorithm depending on the message size. Therefore, for very long messages where we use the ring allgather, the time taken by the broadcast is $T_{vandegeijn} = (\lg p + p - 1)\alpha + 2((p - 1) / p)n\beta$.

Comparing this time with that for the binomial tree algorithm, we see that for long messages (where the latency

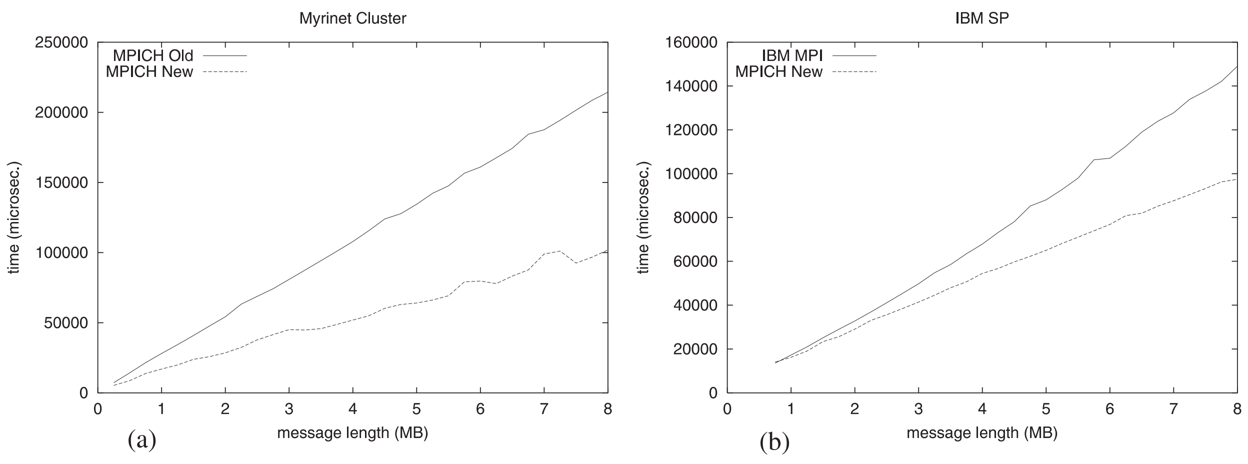


Fig. 6 Performance of long-message broadcast (64 nodes).

term can be ignored) and when lg $p > 2$ (or $p > 4$), the van de Geijn algorithm is better than binomial tree. The maximum improvement in performance that can be expected is (lg p) / 2. In other words, the larger the number of processes, the greater the expected improvement in performance. Figure 6 shows the performance for long messages of the new algorithm versus the old binomial tree algorithm in MPICH as well as the algorithm used by IBM's MPI on the SP. In both cases, the new algorithm performs significantly better. In MPICH, therefore, we use the binomial tree algorithm for short messages (< 12 KB) or when the number of processes is less than 8, and the van de Geijn algorithm otherwise (long messages and number of processes ≥ 8).

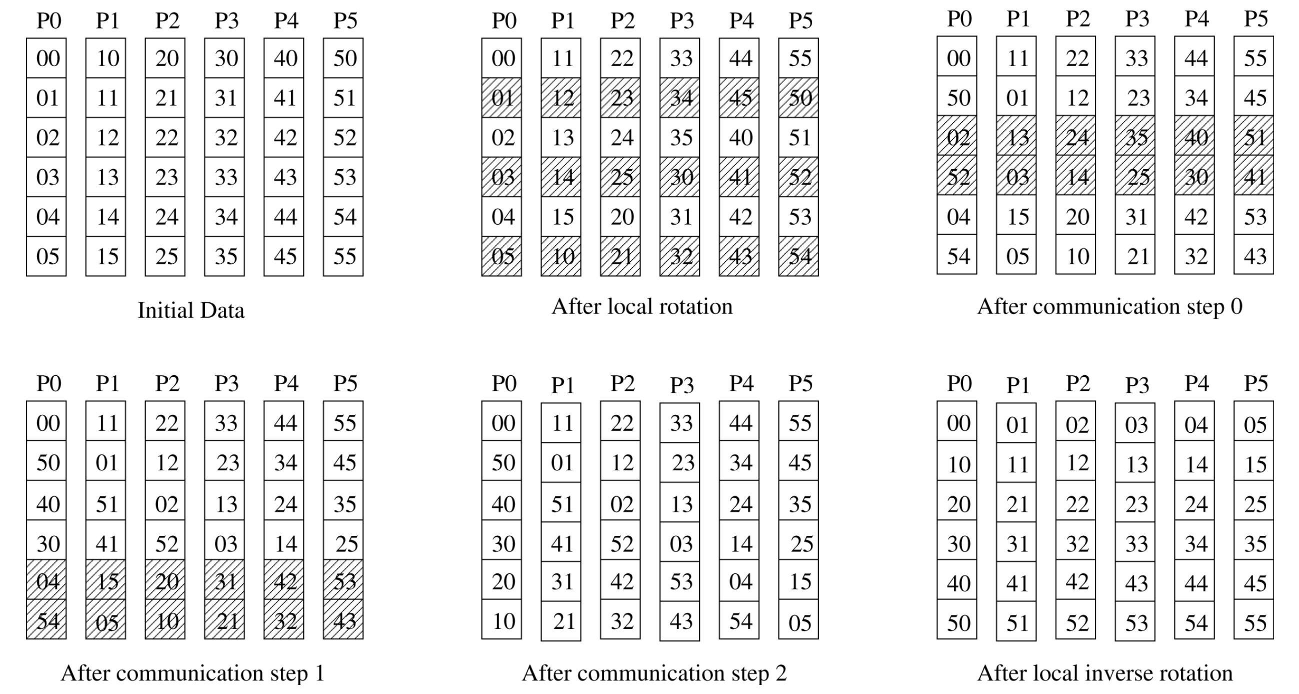
4.3 ALL-TO-ALL

All-to-all communication is a collective operation in which each process has unique data to be sent to every other process. The old algorithm for all-to-all in MPICH does not attempt to schedule communication. Instead, each process posts all the `MPI_Irecvs` in a loop, then all the `MPI_Isend` in a loop, followed by an `MPI_Waitall`. Instead of using the loop index i as the source or destination process for the irecv or isend, each process calculates the source or destination as (*rank* + *i*) modulo p, which results in a scattering of the sources and destinations among the processes. If the loop index were directly used as the source or target rank, all processes would try to communicate with rank 0 first, then with rank 1, and so on, resulting in a bottleneck.

The new all-to-all in MPICH uses four different algorithms depending on the message size. For short messages (≤ 256 bytes per message), we use the index algorithm by Bruck et al. (1997). It is a store-and-forward algorithm that takes $\lceil \lg p \rceil$ steps at the expense of some extra data communication ($(n/2)\lg p\beta$ instead of $n\beta$, where n is the total amount of data to be sent or received by any process). Therefore, it is a good algorithm for very short messages where latency is an issue.

Figure 7 illustrates the Bruck algorithm for an example with six processes. The algorithm begins by doing a local copy and upward shift of the data blocks from the input buffer to the output buffer such that the data block to be sent by each process to itself is at the top of the output buffer. To achieve this, process i must rotate its data up by i blocks. In each communication step k ($0 \leq k < \lceil \lg p \rceil$), process i sends to rank ($i + 2^k$) (with wrap-around) all those data blocks whose kth bit is 1, receives data from rank ($i - 2^k$), and stores the incoming data into blocks whose kth bit is 1 (that is, overwriting the data that was just sent). In other words, in step 0, all the data blocks whose least significant bit is 1 are sent and received (blocks 1, 3, and 5 in our example). In step 1, all the data blocks whose second bit is 1 are sent and received, namely, blocks 2 and 3. After a total of $\lceil \lg p \rceil$ steps, all the data become routed to the right destination process, but the data blocks are not in the right order in the output buffer. A final step in which each process does a local inverse shift of the blocks (memory copies) places the data in the right order.

The beauty of the Bruck algorithm is that it is a logarithmic algorithm for short-message all-to-all that does



Initial Data
After local rotation
After communication step 0
After communication step 1
After communication step 2
After local inverse rotation

Fig. 7 Bruck algorithm for all-to-all. The number *ij* in each box represents the data to be sent from process *i* to process *j*. The shaded boxes indicate the data to be communicated in the next step.

not need any extra bookkeeping or control information for routing the right data to the right process–-that is taken care of by the mathematics of the algorithm. It does need a memory permutation in the beginning and another at the end, but for short messages, where communication latency dominates, the performance penalty of memory copying is small.

If n is the total amount of data a process needs to send to or receive from all other processes, the time taken by the Bruck algorithm can be calculated as follows. If the number of processes is a power of two, each process sends and receives $n / 2$ amount of data in each step, for a total of $\lg p$ steps. Therefore, the time taken by the algorithm is $T_{bruck} = \lg p\alpha + (n / 2) \lg p\beta$. If the number of processes is not a power of two, in the final step, each process must communicate $(n / p)(p - 2^{\lfloor \lg p \rfloor})$ data. Therefore, the time taken in the non-power-of-two case is $T_{bruck} = \lceil \lg p \rceil\alpha + ((n / 2)\lg p + (n / p)(p - 2^{\lfloor \lg p \rfloor}))\beta$.

Figure 8 shows the performance of the Bruck algorithm versus the old algorithm in MPICH (isend-irecv) for short messages. The Bruck algorithm performs signif-

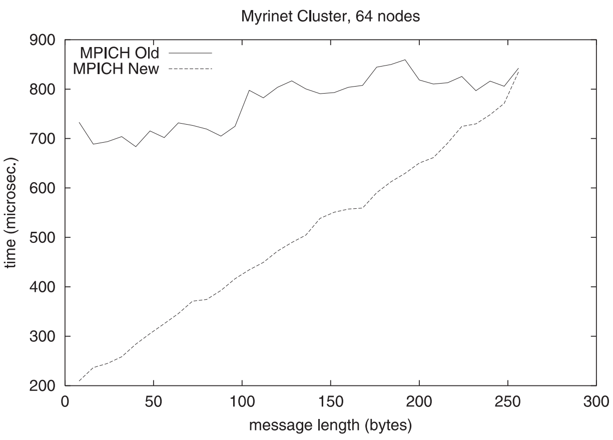


Fig. 8 Performance of Bruck all-to-all versus the old algorithm in MPICH (isend-irecv) for short messages. The size on the x-axis is the amount of data sent by each process to every other process.

icantly better because of its logarithmic latency term. As the message size is increased, however, latency becomes less of an issue, and the extra bandwidth cost of the Bruck algorithm begins to show. Beyond a per process message size of about 256 bytes, the isend-irecv algorithm performs better. Therefore, for medium-sized messages (256 bytes to 32 KB per message), we use the irecv-isend algorithm, which works well in this range.

For long messages and power-of-two number of processes, we use a pairwise-exchange algorithm, which takes $p - 1$ steps. In each step k, $1 \le k < p$, each process calculates its target process as ($rank$ ^ k) (exclusive-or operation) and exchanges data directly with that process. However, this algorithm does not work if the number of processes is not a power of two. For the non-power-of-two case, we use an algorithm in which, in step k, each process receives data from $rank - k$ and sends data to $rank + k$. In both these algorithms, data are directly communicated from source to destination, with no intermediate steps. The time taken by these algorithms is given by $T_{long} = (p - 1)\alpha + n\beta$.

4.4 REDUCE-SCATTER

Reduce-scatter is a variant of reduce in which the result, instead of being stored at the root, is scattered among all processes. It is an irregular primitive: the scatter in it is a scatterv. The old algorithm in MPICH implements reduce-scatter by doing a binomial tree reduce to rank 0 followed by a linear scatterv. This algorithm takes $\lg p + p - 1$ steps, and the bandwidth term is $(\lg p + ((p - 1) / p))n\beta$. Therefore, the time taken by this algorithm is $T_{old} = (\lg p + p - 1)\ \alpha + (\lg p + ((p - 1) / p))n\beta + n\lg p\gamma$.

In our new implementation of reduce-scatter, for short messages, we use different algorithms depending on whether the reduction operation is commutative or non-commutative. The commutative case occurs most commonly because all the predefined reduction operations in MPI (such as `MPI_SUM`, `MPI_MAX`) are commutative.

For commutative operations, we use a recursive-halving algorithm, which is analogous to the recursive-doubling algorithm used for allgather (see Figure 9). In the first step, each process exchanges data with a process that is a distance $p / 2$ away: each process sends the data needed by all processes in the other half, receives the data needed by all processes in its own half, and performs the reduction operation on the received data. The reduction can be done because the operation is commutative. In the second step, each process exchanges data with a process that is a distance $p / 4$ away. This procedure continues recursively, halving the data communicated at each step, for a total of $\lg p$ steps. Therefore, if p is a power of two, the time taken by this algorithm is $T_{rec\_half} = \lg p\alpha + ((p - 1) / p)\ n\beta + ((p - 1) / p)n\gamma$. We use this algorithm for messages up to 512 KB.

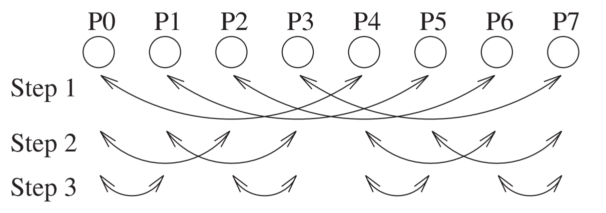
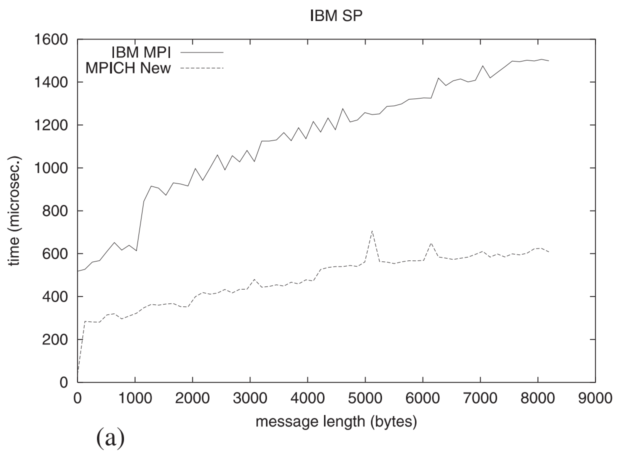


Fig. 9 Recursive halving for commutative reduce-scatter.

If p is not a power of two, we first reduce the number of processes to the nearest lower power of two by having the first few even-numbered processes send their data to the neighboring odd-numbered process ($rank + 1$). These odd-numbered processes do a reduce on the received data, compute the result for themselves and their left neighbor during the recursive halving algorithm, and, at the end, send the result back to the left neighbor. Therefore, if p is not a power of two, the time taken by the algorithm is $T_{rec\_half} = (\lfloor \lg p \rfloor + 2)\alpha + 2n\beta + n(1 + ((p - 1) / p))\gamma$. This cost is approximate because some imbalance exists in the amount of work each process does, since some processes do the work of their neighbors as well.

If the reduction operation is not commutative, recursive halving will not work (unless the data are permuted suitably; J. L. Träff, private communication). Instead, we use a recursive-doubling algorithm similar to that in allgather. In the first step, pairs of neighboring processes exchange data; in the second step, pairs of processes at distance 2 apart exchange data; in the third step, processes at distance 4 apart exchange data; and so forth. However, more data are communicated than in allgather. In step 1, processes exchange all the data except the data needed for their own result ($n - (n / p)$); in step 2, processes exchange all data except the data needed by themselves and by the processes they communicated with in the previous step ($n - (2n / p)$); in step 3, it is ($n - (4n / p)$); and so forth. Therefore, the time taken by this algorithm is $T_{short} = \lg p / \alpha + n(\lg p - ((p - 1) / p))\beta + n(\lg p - ((p - 1) / p))\gamma$. We use this algorithm for very short messages (< 512 bytes).

For long messages ($\ge$ 512 KB in the case of commutative operations and $\ge$ 512 bytes in the case of non-commutative operations), we use a pairwise exchange algorithm that takes $p - 1$ steps. In step i, each process sends data to ($rank + i$), receives data from ($rank - i$), and performs the local reduction. The data exchanged are only the data needed for the scattered result on the process (n / p). The time taken by this algorithm is $T_{long} = (p - 1)\alpha + ((p - 1) / p)n\beta + ((p - 1) / p)n\gamma$. Note that this algorithm has the same bandwidth requirement as the recursive halving algorithm. Nonetheless, we use this algorithm for long



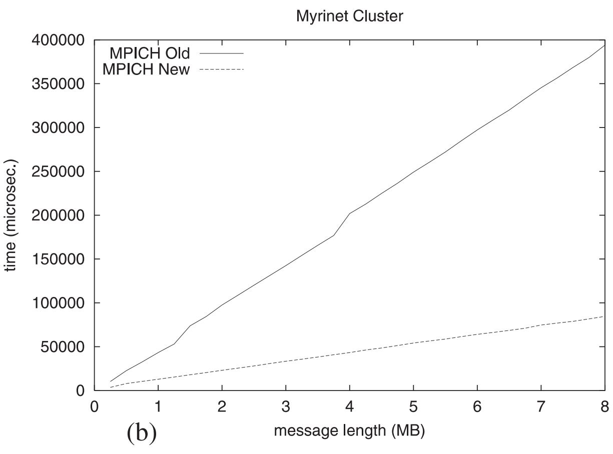


Fig. 10 Performance of reduce-scatter for short messages on the IBM SP (64 nodes) and for long messages on the Myrinet cluster (32 nodes).

messages because it performs much better than recursive halving (similar to the results for recursive doubling versus ring algorithm for long-message allgather).

The SKaMPI benchmark, by default, uses a non-commutative user-defined reduction operation. Since commutative operations are more commonly used, we modified the benchmark to use a commutative operation, namely, `MPI_SUM`. Figure 10 shows the performance of the new algorithm for short messages on the IBM SP and on the Myrinet cluster. The performance is significantly better than that of the algorithm used in IBM's MPI on the SP and several times better than the old algorithm (reduce + scatterv) used in MPICH on the Myrinet cluster.

The above algorithms will also work for irregular reduce-scatter operations, but they are not specifically optimized for that case.

4.5 REDUCE AND ALLREDUCE

`MPI_Reduce` performs a global reduction operation and returns the result to the specified root, whereas `MPI_Allreduce` returns the result on all processes. The old algorithm for reduce in MPICH uses a binomial tree, which takes lg p steps, and the data communicated at each step are n. Therefore, the time taken by this algorithm is $T_{tree} = \lceil \lg p \rceil(\alpha + n\beta + n\gamma)$. The old algorithm for allreduce simply performs a reduce to rank 0 followed by a broadcast.

The binomial tree algorithm for reduce is a good algorithm for short messages because of the lg p number of steps. For long messages, however, a better algorithm exists, proposed by Rabenseifner (http://www.hlrs.de/mpi/myreduce.html). The principle behind Rabenseifner's algorithm is similar to that behind van de Geijn's algorithm for long-message broadcast. Van de Geijn implements the broadcast as a scatter followed by an allgather, which reduces the $n \lg p\beta$ bandwidth term in the binomial tree algorithm to a $2n\beta$ term. Rabenseifner's algorithm implements a long-message reduce effectively as a reduce-scatter followed by a gather to the root, which has the same effect of reducing the bandwidth term from $n \lg p\beta$ to $2n\beta$. The time taken by Rabenseifner's algorithm is the sum of the times taken by reduce-scatter (recursive halving) and gather (binomial tree), which is $T_{rabenseifner} = 2\lg p\alpha + 2((p-1)/p)n\beta + ((p-1)/p)n\gamma$.

For reduce, in the case of predefined reduction operations, we use Rabenseifner's algorithm for long messages (> 2 KB) and the binomial tree algorithm for short messages (≤ 2 KB). In the case of user-defined reduction operations, we use the binomial tree algorithm for all message sizes because, unlike with predefined reduction operations, the user may pass derived data types, and breaking up derived data types to perform the reduce-scatter is tricky. Figure 11 shows the performance of reduce for long messages on the Myrinet cluster. The new algorithm is more than twice as fast as the old algorithm in some cases.

For allreduce, we use a recursive doubling algorithm for short messages and for long messages with user-defined reduction operations. This algorithm is similar to the recur-

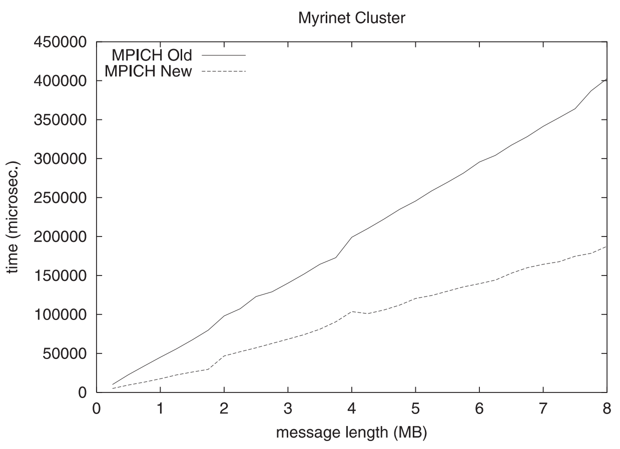


Fig. 11 Performance of reduce (64 nodes).

sive doubling algorithm used in allgather, except that each communication step also involves a local reduction. The time taken by this algorithm is $T_{rec-dbl} = \lg p\ \alpha + n \lg p\ \beta + n \lg p\ \gamma$.

For long messages and predefined reduction operations, we use Rabenseifner's algorithm for allreduce (http:// www.hlrs.de/organization/par/services/models/mpi/myreduce.html), which performs a reduce-scatter followed by an allgather. If the number of processes is a power of two, the cost for the reduce-scatter is $\lg p\alpha + ((p-1)/p)n\beta + ((p-1)/p)n\gamma$. The cost for the allgather is $\lg p\alpha + ((p-1)/p)n\beta$. Therefore, the total cost is $T_{rabenseifner} = 2\lg p\ \alpha + 2((p-1)/p)n\beta + ((p-1)/p)n\gamma$.

5 Further Optimization of Allreduce and Reduce

As the profiling study in Rabenseifner (1999) indicated that allreduce and reduce are the most commonly used collective operations, we investigated in further detail how to optimize these operations. We consider five different algorithms for implementing allreduce and reduce. The first two algorithms are binomial tree and recursive doubling, which were explained above. Binomial tree for reduce is well known. For allreduce, it involves performing a binomial-tree reduce to rank 0 followed by a binomial-tree broadcast. Recursive doubling is used for allreduce only. The other three algorithms are recursive halving and doubling, binary blocks, and ring. For explaining these algorithms, we define the following terms.

- *Recursive vector halving*: the vector to be reduced is recursively halved in each step.
- *Recursive vector doubling*: small pieces of the vector scattered across processes are recursively gathered or combined to form the large vector
- *Recursive distance halving*: the distance over which processes communicate is recursively halved at each step ($p/2, p/4, \ldots, 1$).
- *Recursive distance doubling*: the distance over which processes communicate is recursively doubled at each step ($1, 2, 4, \ldots, p/2$).

All algorithms in this section can be implemented without local copying of data, except if user-defined non-commutative operations are used.

5.1 VECTOR HALVING AND DISTANCE DOUBLING ALGORITHM

This algorithm is a combination of a reduce-scatter implemented with recursive vector halving and distance doubling, followed either by a binomial-tree gather (for reduce) or by an allgather implemented with recursive vector doubling and distance halving (for allreduce).

Since these recursive algorithms require a power-of-two number of processes, if the number of processes is not a power of two, we first reduce it to the nearest lower power of two ($p' = 2^{\lfloor \lg p \rfloor}$) by removing $r = p - p'$ extra processes as follows. In the first $2r$ processes (ranks 0 to $2r-1$), all the even ranks send the second half of the input vector to their right neighbor ($rank + 1$), and all the odd ranks send the first half of the input vector to their left neighbor ($rank - 1$), as illustrated in Figure 12. The even ranks compute the reduction on the first half of the vector and the odd ranks compute the reduction on the second half. The odd ranks then send the result to their left neighbors (the even ranks). As a result, the even ranks among the first $2r$ processes now contain the reduction with the input vector on their right neighbors (the odd ranks). These odd ranks do not participate in the rest of the algorithm, which leaves behind a power-of-two number of processes. The first r even-ranked processes and the last $p - 2r$ processes are now renumbered from 0 to $p' - 1$, p' being a power of two.

Figure 12 illustrates the algorithm for an example on 13 processes. The input vectors and all reduction results are divided into eight parts (A, B, ... ,H), where 8 is the largest power of two less than 13, and denoted as A–H_{ranks}. After the first reduction, process P0 has computed A–D_{0-1}, which is the reduction result of the first half (A–D) of the input vector from processes 0 and 1. Similarly, P1 has computed E–H_{0-1}, P2 has computed A–D_{2-3}, and so forth. The odd ranks then send their half to the even ranks on their left: P1 sends E–H_{0-1} to P0, P3 sends E–H_{2-3} to P0, and so forth. This completes the first step, which takes $(1 + f_\alpha)\alpha + (n/2)(1 + f_\beta)\beta + (n/2)\gamma$ time. P1, P3, P5, P7,

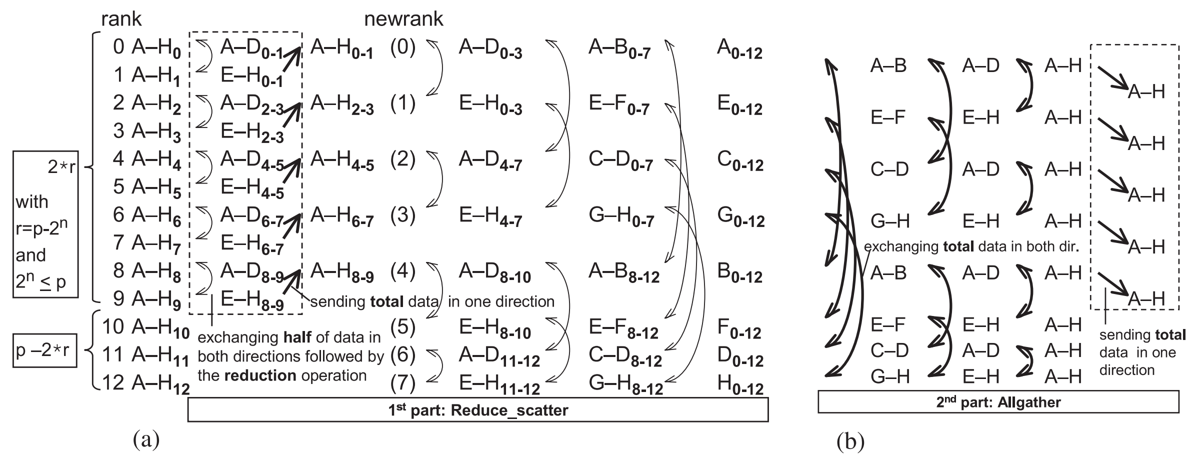

rank
newrank
2*r
with r=p-2^n and 2^n ≤ p
p –2*r
exchanging half of data in both directions followed by the reduction operation
sending total data in one direction
1st part: Reduce_scatter
(a)
exchanging total data in both dir.
sending total data in one direction
2nd part: Allgather
(b)

Fig. 12 Allreduce using the recursive halving and doubling algorithm. The intermediate results after each communication step are shown, including the reduction operation in the reduce-scatter phase. The dotted frames show the additional overhead caused by a non-power-of-two number of processes.

and P9 do not participate in the remainder of the algorithm, and the remaining processes are renumbered from 0–7.

The remaining processes now perform a reduce-scatter by using recursive vector halving and distance doubling. The even-ranked processes send the second half of their buffer to *rank′* + 1 and the odd-ranked processes send the first half of their buffer to *rank′* – 1. All processes then compute the reduction between the local buffer and the received buffer. In the next $\lg p' - 1$ steps, the buffers are recursively halved, and the distance is doubled. At the end, each of the p' processes has $1/p'$ of the total reduction result. All these recursive steps take $\lg p'\alpha + ((p'-1)/p')(n\beta + n\gamma)$ time. The next part of the algorithm is either an allgather or gather depending on whether the operation to be implemented is an allreduce or reduce.

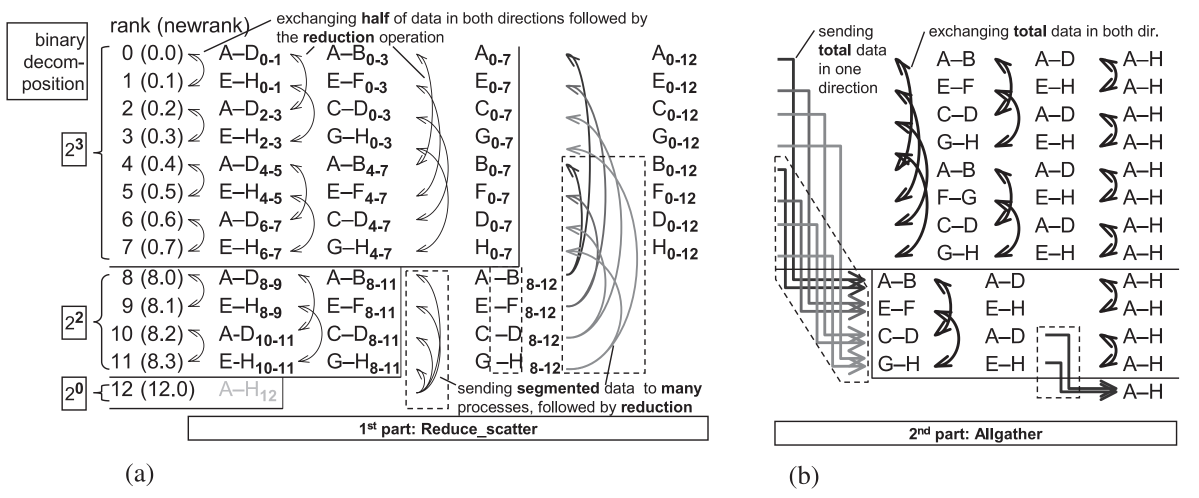
Allreduce: To implement allreduce, we perform an allgather using recursive vector doubling and distance halving. In the first step, process pairs exchange $1/p'$ of the buffer to achieve $2/p'$ of the result vector; in the next step $2/p'$ of the buffer is exchanged to obtain $4/p'$ of the result, and so forth. After $\lg p'$ steps, the p' processes receive the total reduction result. This allgather part costs $\lg p'\alpha + ((p'-1)/p')n\beta$. If the number of processes is not a power of two, the total result vector must be sent to the r processes that were removed in the first step, which results in additional overhead of $\alpha_{uni} + n\beta_{uni}$. The total allreduce operation therefore takes the following time.

- If p is a power of two: $T_{all,\,h\&d,\,p=2^{exp}} = 2\lg p\alpha + 2n\beta + n\gamma - (1/p)(2n\beta + n\gamma) \simeq 2\lg p\alpha + 2n\beta + n\gamma$.
- If p is not a power of two: $T_{all,\,h\&d,\,p \neq 2^{exp}} = (2\lg p' + 1 + 2f_\alpha)\alpha + (2 + ((1+3f_\beta)/2))n\beta + (3/2)n\gamma - (1/p')(2n\beta + n\gamma) \simeq (3 + 2\lfloor \lg p \rfloor)\alpha + 4n\beta + (3/2)n\gamma$.

This algorithm is good for long vectors and power-of-two numbers of processes. For non-power-of-two numbers of processes, the data transfer overhead is doubled, and the computation overhead is increased by 3 / 2. The binary blocks algorithm described in Section 5.2 can reduce this overhead in many cases.

Reduce: For reduce, a binomial tree gather is performed by using recursive vector doubling and distance halving, which takes $\lg p'\alpha_{uni} + ((p'-1)/p')n\beta_{uni}$ time. In the non-power-of-two case, if the root happens to be one of those odd-ranked processes that would normally be removed in the first step, then the role of this process and its partner in the first step are interchanged after the first reduction in the reduce-scatter phase, which causes no additional overhead. The total reduce operation therefore takes the following time.

- If p is a power of two: $T_{red,\,h\&d,\,p=2^{exp}} = \lg p(1+f_\alpha)\alpha + (1+f_\beta)n\beta + n\gamma - (1/p)((1+f_\beta)n\beta + n\gamma) \simeq 2\lfloor \lg p \rfloor\alpha + 2n\beta + n\gamma$.
- If p is a not a power of two: $T_{red,\,h\&d,\,p \neq 2^{exp}} = \lg p'(1 + f_\alpha)\alpha + (1+f_\alpha)\alpha + (1 + ((1+f_{beta})/2) + f_\beta)n\beta + (3/2)n\gamma$



binary decom-position
rank (newrank)
exchanging half of data in both directions followed by the reduction operation
0 (0.0)
1 (0.1)
2 (0.2)
3 (0.3)
4 (0.4)
5 (0.5)
6 (0.6)
7 (0.7)
8 (8.0)
9 (8.1)
10 (8.2)
11 (8.3)
12 (12.0)
A–H12
sending segmented data to many processes, followed by reduction
1st part: Reduce_scatter
sending total data in one direction
exchanging total data in both dir.
2nd part: Allgather
(a)
(b)

Fig. 13 Allreduce using the binary blocks algorithm.

$– (1 / p')((1 + f_{\beta})n\beta + n\gamma) \simeq (2 + 2\lfloor \lg p \rfloor)\alpha + 3n\beta + (3 / 2)n\gamma$.

5.2 BINARY BLOCKS ALGORITHM

This algorithm reduces some of the load imbalance in the recursive halving and doubling algorithm when the number of processes is not a power of two. The algorithm starts with a binary-block decomposition of all processes in blocks with power-of-two numbers of processes (see the example in Figure 13). Each block executes its own reduce-scatter with the recursive vector halving and distance doubling algorithm described above. Then, starting with the smallest block, the intermediate result (or the input vector in the case of a 2^0 block) is split into the segments of the intermediate result in the next higher block and sent to the processes in that block, and those processes compute the reduction on the segment. This does cause a load imbalance in computation and communication compared with the execution in the larger blocks. For example, in the third exchange step in the $\boxed{2^3}$ block, each process sends one segment, receives one segment, and computes the reduction of one segment (P0 sends B, receives A, and computes the reduction on A). The load imbalance is introduced by the smaller blocks $\boxed{2^2}$ and $\boxed{2^0}$: in the $\boxed{2^2}$ block, each process receives and reduces two segments (for example, A–B on P8), whereas in the $\boxed{2^0}$ block (P12), each process has to send as many messages as the ratio of the two block sizes (here $2^2 / 2^0$). At the end of the first part, the highest block must be recombined with the next smaller block, and the ratio of the block sizes again determines the overhead.

We see that the maximum difference between the ratio of two successive blocks, especially in the low range of exponents, determines the load imbalance. Let us define $\delta_{\text{expo, max}}$ as the maximal difference of two consecutive exponents in the binary representation of the number of processes. For example, $100 = 2^6 + 2^5 + 2^2$, $\delta_{\text{expo, max}} = \max(6 – 5, 5 – 2) = 3$. If dexpo, max is small, the binary blocks algorithm can perform well.

Allreduce: For allreduce, the second part is an allgather implemented with recursive vector doubling and distance halving in each block. For this purpose, data must be provided to the processes in the smaller blocks with a pair of messages from processes of the next larger block, as shown in Figure 13.

Reduce: For reduce, if the root is outside the largest block, then the intermediate result segment of rank 0 is sent to the root, and the root plays the role of rank 0. A binomial tree is used to gather the result segments to the root process.

We note that if the number of processes is a power of two, the binary blocks algorithm is identical to the recursive halving and doubling algorithm.

5.3 RING ALGORITHM

This algorithm uses a pairwise-exchange algorithm for the reduce-scatter phase (see Section 4.4). For allreduce,

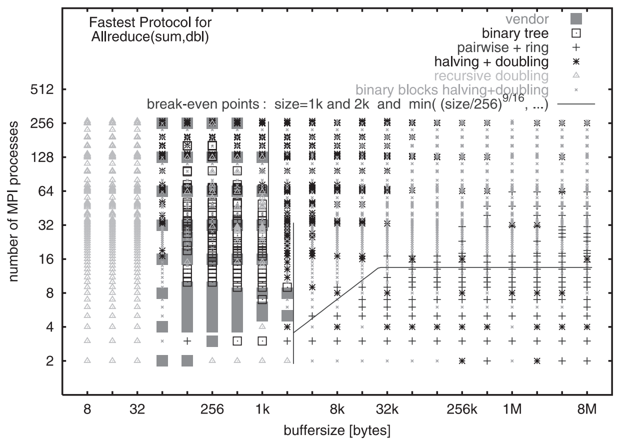


Fig. 14 The fastest algorithm for allreduce (`MPI_DOUBLE`, `MPI_SUM`) on a Cray T3E 900.

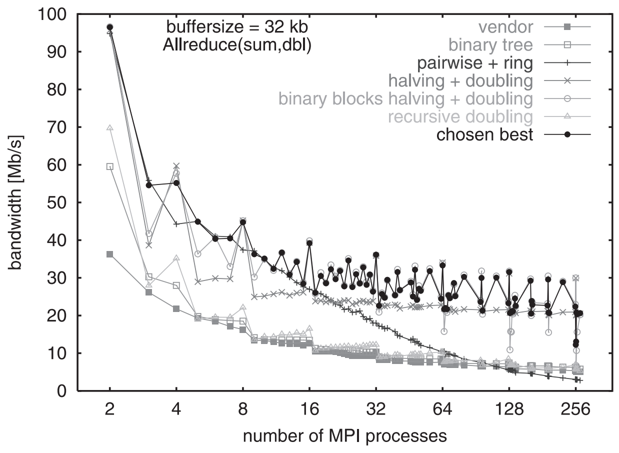


Fig. 15 Bandwidth comparison for allreduce (`MPI_DOUBLE`, `MPI_SUM`) with 32 KB vectors on a Cray T3E 900.

it uses a ring algorithm to perform the allgather, and, for reduce, all processes directly send their result segment to the root. This algorithm is good in bandwidth use when the number of processes is not a power of two, but the latency scales with the number of processes. Therefore, this algorithm should be used only for small or medium number of processes or for large vectors. The time taken is $T_{all,ring} = 2(p - 1)\alpha + 2n\beta + n\gamma - (1 / p)(2n\beta + n\gamma)$ for allreduce and $T_{red,ring} = (p - 1)(\alpha + \alpha_{uni}) + n(\beta + \beta_{uni}) + n\gamma - (1 / p)(n(\beta + \beta_{uni}) + n\gamma)$ for reduce.

5.4 CHOOSING THE FASTEST ALGORITHM

Based on the number of processes and the buffer size, the reduction routine must decide which algorithm to use. This decision is not easy and depends on a number of factors. We experimentally determined which algorithm works best for different buffer sizes and number of processes on the Cray T3E 900. The results for allreduce are shown in Figure 14. The figure indicates which is the fastest allreduce algorithm for each parameter pair (number of processes, buffer size) and for the operation `MPI_SUM` with data type `MPI_DOUBLE`. For buffer sizes less than or equal to 32 bytes, recursive doubling is the best; for buffer sizes less than or equal to 1 KB, the vendor's algorithm (for power-of-two) and binomial tree (for non-power-of-two) are the best, but not much better than recursive doubling; for longer buffer sizes, the ring algorithm is good for some buffer sizes and some number of processes less than 32. In general, on a Cray T3E 900, the binary blocks algorithm is faster if $\delta_{expo,\ max}$ < lg(vector length in bytes) / 2.0 – 2.5 and vector size $\geq$ 16 KB and more than 32 processes are used. In a few cases, for example, 33 processes and less than 32 KB, recursive halving and doubling is the best.

Figure 15 shows the bandwidths obtained by the various algorithms for a 32 KB buffer on the T3E. For this buffer size, the new algorithms are clearly better than the vendor's algorithm (Cray MPT.1.4.0.4) and the binomial tree algorithm for all numbers of processes. We observe that the bandwidth of the binary blocks algorithm depends strongly on $\delta_{expo,\ max}$ and that recursive halving and doubling is faster on 33, 65, 66, 97, 128–131 processes. The ring algorithm is faster on 3, 5, 7, 9–11, and 17 processes.

5.5 COMPARISON WITH VENDOR'S MPI

We also ran some experiments to compare the performance of the best of the new algorithms with the algorithm in the native MPI implementations on the IBM SP at San Diego Supercomputer Center, a Myrinet cluster at the University of Heidelberg, and the Cray T3E. Figures 16–18 show the improvement achieved compared with the allreduce/reduce algorithm in the native (vendor's) MPI library. Each symbol in these figures indicates how many times faster the best algorithm is compared with the native vendor's algorithm.

Figure 16 compares the algorithm based on two different application programming models on a cluster of SMP nodes.

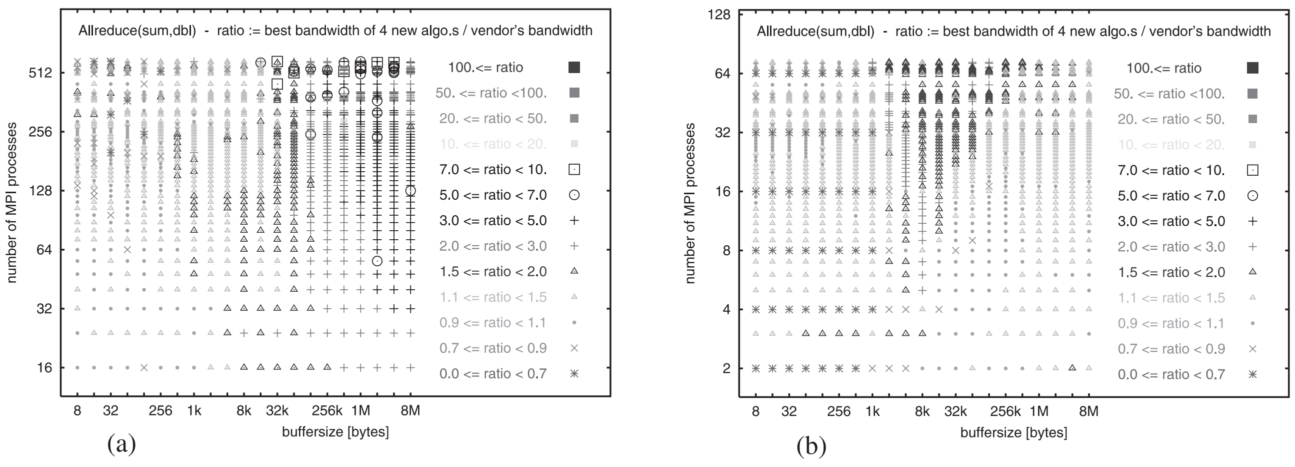


Fig. 16 Ratio of the bandwidth of the fastest of the new algorithms (not including recursive doubling) and the vendor's allreduce on the IBM SP at SDSC with one MPI process per CPU (left) and per SMP node (right).

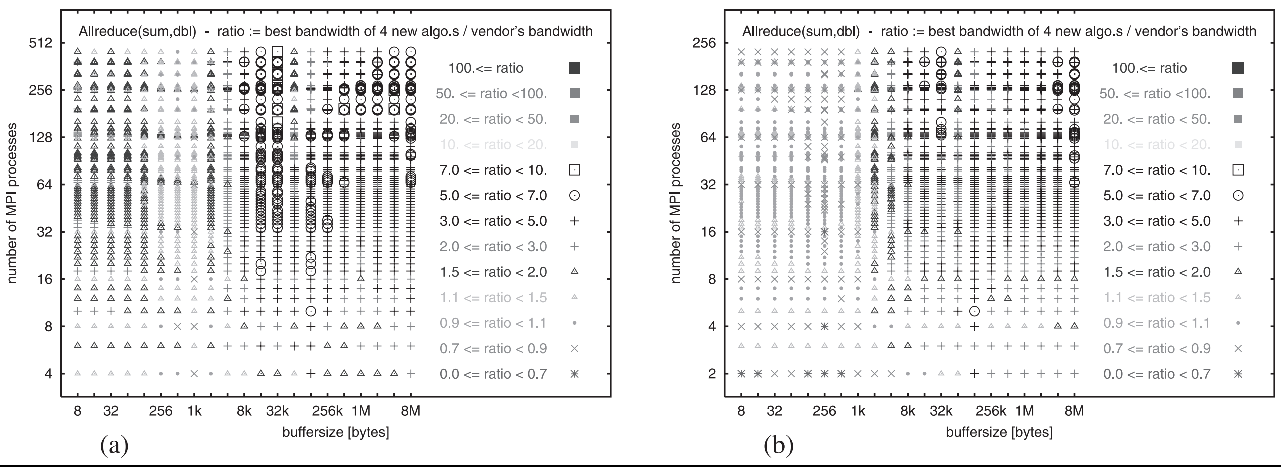


Fig. 17 Ratio of the bandwidth of the fastest of the new algorithms (not including recursive doubling) and the old MPICH-1 algorithm on a Myrinet cluster with dual-CPU PCs (HELICS cluster, University of Heidelberg) and one MPI process per CPU (left) and per SMP node (right).

The left graph shows that with a pure MPI programming model (one MPI process per CPU) on the IBM SP, the fastest algorithm performs about 1.5 times better than the vendor's algorithm for buffer sizes of 8–64 KB and two to five times better for larger buffers. In the right graph, a hybrid programming model comprising one MPI process per SMP node is used, where each MPI process is itself SMP-parallelized (with OpenMP, for example) and only the master thread calls MPI functions (the master-only style in Rabenseifner and Wellein 2003). The performance is about 1.5 to three times better than the vendor's MPI for buffer sizes 4–128 KB and more than four processes.

Figure 17 compares the best of the new algorithms with the old MPICH-1 algorithm on the Heidelberg

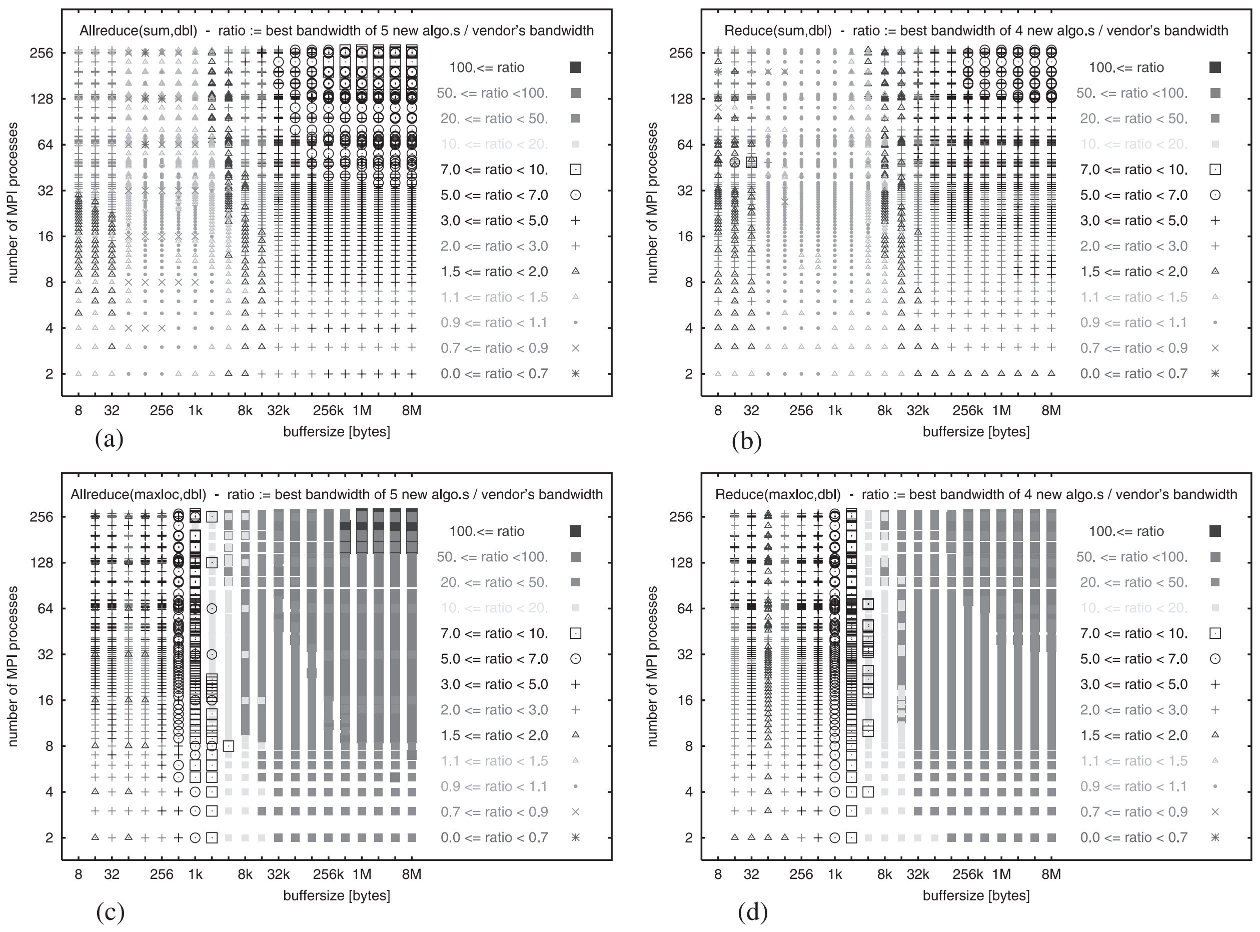


Fig. 18 Ratio of the bandwidth of the fastest of the new algorithms and the vendor's algorithm for allreduce (left) and reduce (right) with operation `MPI_SUM` (first row) and `MPI_MAXLOC` (second row) on a Cray T3E 900.

Myrinet cluster. The new algorithms show a performance benefit of three to seven times with pure MPI and two to five times with the hybrid model. Figure 18 shows that on the T3E, the new algorithms are three to five times faster than the vendor's algorithm for the operation MPI_SUM and, because of the very slow implementation of structured derived data types in Cray's MPI, up to 100 times faster for MPI_MAXLOC.

We ran the best-performing algorithms for the usage scenarios indicated by the profiling study in Rabenseifner (1999) and found that the new algorithms improve the performance of allreduce by up to 20% and that of reduce by up to 54%, compared to the vendor's implementation on the T3E, as shown in Figure 19.

6 Conclusions and Future Work

Our results demonstrate that optimized algorithms for collective communication can provide substantial performance benefits and, to achieve the best performance, one needs to use a number of different algorithms and select the right algorithm for a particular message size and number of processes. Determining the right cutoff points for switching between the different algorithms is tricky, however, and they may be different for different machines and networks. At present, we use experimentally determined cutoff points. In the future, we intend to determine the cutoff points automatically based on system parameters.

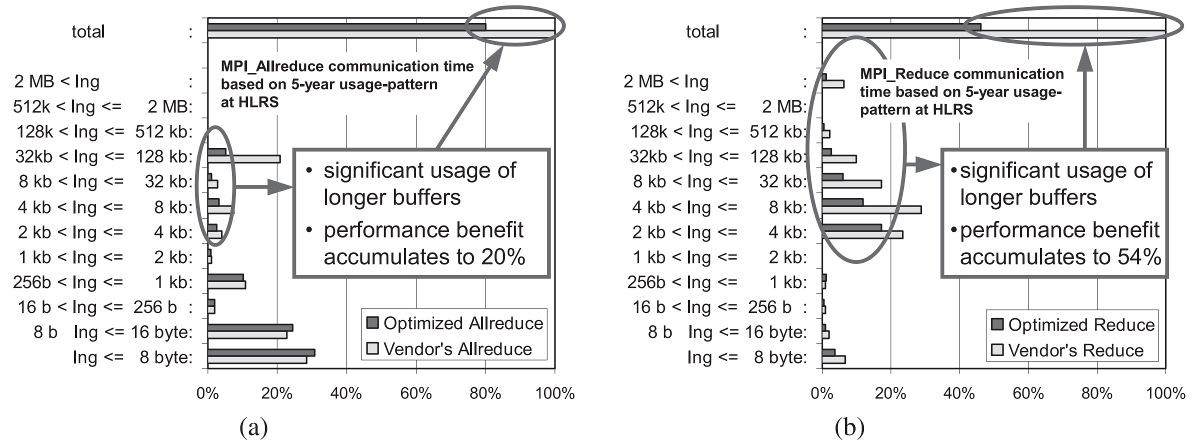


Fig. 19 Benefit of new allreduce and reduce algorithms optimized for long vectors on the Cray T3E.

MPI also defines irregular ("v") versions of many of the collectives, where the operation counts may be differ ent on different processes. For these operations, we currently use the same techniques as for the regular versions described in this paper. Further optimization of the irregular collectives is possible, and we plan to optimize them in the future.

In this work, we assume a flat communication model in which any pair of processes can communicate at the same cost. Although these algorithms will work even on hierarchical networks, they may not be optimized for such networks. We plan to extend this work to hierarchical networks and develop algorithms that are optimized for architectures comprising clusters of SMPs and clusters distributed over a wide area, such as the TeraGrid (http://www.teragrid.org). We also plan to explore the use of one-sided communication to improve the performance of collective operations.

The source code for the algorithms in Section 4 is available in MPICH-1.2.6 and MPICH2 1.0. Both MPICH-1 and MPICH2 can be downloaded from http://www.mcs.anl.gov/mpi/mpich.

ACKNOWLEDGMENTS

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38. The authors would like to acknowledge their colleagues and others who provided suggestions and helpful comments. They would especially like to thank Jesper Larsson Träff for helpful discussion on optimized reduction algorithms and Gerhard Wellein, Thomas Ludwig, and Ana Kovatcheva for their benchmarking support. We also thank the reviewers for their detailed comments.

AUTHOR BIOGRAPHIES

Rajeev Thakur is a Computer Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. He received a B.E. in Computer Engineering from the University of Bombay, India, in 1990, an M.S. in Computer Engineering from Syracuse University in 1992, and a Ph.D. in Computer Engineering from Syracuse University in 1995. His research interests are in the area of high-performance computing in general and high-performance communication and I/O in particular. He was a member of the MPI Forum and participated actively in the definition of the I/O part of the MPI-2 standard. He is also the the author of a widely used, portable implementation of MPI-IO, called ROMIO. He is currently involved in the development of MPICH-2, a new portable implementation of MPI-2. Rajeev Thakur is a co-author of the book *Using MPI-2: Advanced Features of the Message Passing Interface* published by MIT Press. He is an associate editor of IEEE Transactions on Parallel and Distributed Systems, has served on the program committees of several conferences, and has also served as a co-guest editor for a special issue of the International Journal of High

Performance Computing Applications on "I/O in Parallel Applications".

Rolf Rabenseifner (http://www.hlrs.de/people/rabenseifner/) studied mathematics and physics at the University of Stuttgart. He is head of the Department of Parallel Computing at the High-Performance Computing Center Stuttgart (HLRS). He led the projects DFN-RPC, a remote procedure call tool, and MPI-GLUE, the first metacomputing MPI combining different vendor's MPIs without losing the full MPI interface. In his dissertation work at the University of Stuttgart, he developed a controlled logical clock as global time for trace-based profiling of parallel and distributed applications. He is an active member of the MPI-2 Forum. In 1999, he was an invited researcher at the Center for High-Performance Computing at Dresden University of Technology. His current research interests include MPI profiling, benchmarking, and optimization. Each year he teaches parallel programming models in a workshop format at many universities and labs in Germany.

William Gropp received his B.S. in Mathematics from Case Western Reserve University in 1977, a M.S. in Physics from the University of Washington in 1978, and a Ph.D. in Computer Science from Stanford in 1982. He held the positions of assistant (1982–1988) and associate (1988–1990) professor in the Computer Science Department at Yale University. In 1990, he joined the Numerical Analysis group at Argonne, where he is a Senior Computer Scientist and Associate Director of the Mathematics and Computer Science Division, a Senior Scientist in the Department of Computer Science at the University of Chicago, and a Senior Fellow in the Argonne–Chicago Computation Institute. His research interests are in parallel computing, software for scientific computing, and numerical methods for partial differential equations. He has played a major role in the development of the MPI message-passing standard. He is co-author of the most widely used implementation of MPI, MPICH, and was involved in the MPI Forum as a chapter author for both MPI-1 and MPI-2. He has written many books and papers on MPI including *Using MPI* and *Using MPI-2*. He is also one of the designers of the PETSc parallel numerical library, and has developed efficient and scalable parallel algorithms for the solution of linear and nonlinear equations.

References

Alexandrov, A., Ionescu, M.F., Schauser, K.E., and Scheiman, C. 1997. LogGP: incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing* 44(1):71–79.

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

Barnett, M., Littlefield, R., Payne, D., and van de Geijn, R. 1993. Global combine on mesh architectures with wormhole routing. *Proceedings of the 7th International Parallel Processing Symposium*, Newport Beach, CA, April.

Benson, G.D., Chu, C-W., Huang, Q., and Caglar, S.G. 2003. A comparison of MPICH allgather algorithms on switched networks. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science* Vol. 2840, J. Dongarra, D. Laforenza, and S. Orlando, editors, Springer-Verlag, Berlin, pp. 335–343.

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

Bokhari, S. and Berryman, H. 1992. Complete exchange on a circuit switched mesh. *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, VA, April, pp. 300–306.

Bruck, J., Ho, C-T., Kipnis, S., Upfal, E., and Weathersby, D. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems* 8(11):1143–1156.

Chan, E.W., Heimlich, M.F., Purakayastha, A., and van de Geijn, R.A. 2004. On optimizing collective communication. *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, San Diego, CA, September.

Culler, D.E., Karp, R.M., Patterson, D.A., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., and von Eicken, T. 1993. LogP: towards a realistic model of parallel computation. *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May, pp. 1–12.

Hensgen, D., Finkel, R., and Manbet, U. 1988. Two algorithms for barrier synchronization. *International Journal of Parallel Programming* 17(1):1–17.

Hockney, R.W. 1994. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing* 20(3):389–398.

Iannello, G. 1997. Efficient algorithms for the reduce-scatter operation in LogGP. *IEEE Transactions on Parallel and Distributed Systems* 8(9):970–982.

Kale, L.V., Kumar, S., and Vardarajan, K. 2003. A framework for collective personalized communication. *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, Nice, France, April.

Karonis, N., de Supinski, B., Foster, I., Gropp, W., Lusk, E., and Bresnahan, J. 2000. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS '00)*, Cancun, Mexico, May, pp. 377–384.

Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., and Bhoedjang, R.A.F. 1999. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Symposium on Principles and Practice of Par-*

allel Programming (PPoPP'99), Atlanta, GA, May, pp. 131–140.

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

Rabenseifner, R. 1999. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. *Proceedings of the Message Passing Interface Developer's and User's Conference 1999 (MPIDC '99)*, Atlanta, GA, March, pp. 77–85.

Rabenseifner, R. and Wellein, G. 2003. Communication and optimization aspects of parallel programming models on hybrid architectures. *International Journal of High Performance Computing Applications* 17(1):49–62.

Sanders, P. and Träff, J.L. 2002. The hierarchical factor algorithm for all-to-all communication. *Euro-Par 2002 Parallel Processing, Lecture Notes in Computer Science* Vol. 2400, B. Monien and R. Feldman, editors, Springer-Verlag, Berlin, pp. 799–803.

Scott, D. 1991. Efficient all-to-all communication patterns in hypercube and mesh topologies. *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April, pp. 398–403.

Shroff, M. and van de Geijn, R.A. 1999. CollMark: MPI collective communication benchmark. Technical Report, Department of Computer Sciences, University of Texas at Austin, December.

Sistare, S., vandeVaart, R., and Loh, E. 1999. Optimization of MPI collectives on clusters of large-scale SMPs. *Proceedings of Supercomputing: High Performance Networking and Computing*, Portland, OR, November.

Tipparaju, V., Nieplocha, J., and Panda, D.K. 2003. Fast collective operations using shared and remote memory access protocols on clusters. *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, Nice, France, April.

Träff, J.L. 2002. Improved MPI all-to-all communication on a Giganet SMP cluster. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science* Vol. 2474, D. Kranzlmuller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, Springer-Verlag, Berlin, pp. 392–400.

Vadhiyar, S.S., Fagg, G.E., and Dongarra, J. 1999. Automatically tuned collective communications. *Proceedings of Supercomputing 99: High Performance Networking and Computing*, Portland, OR, November.

Worsch, T., Reussner, R., and Augustin, W. 2002. On benchmarking collective MPI operations. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science* Vol. 2474, D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, Springer-Verlag, Berlin, pp. 271–279.