Cluster Comput (2008) 11: 45–55
DOI 10.1007/s10586-007-0050-7

The design and implementation of MPI collective operations for clusters in long-and-fast networks

Motohiko Matsuda · Tomohiro Kudoh · Yuetsu Kodama · Ryousei Takano · Yutaka Ishikawa

Received: 14 March 2007 / Accepted: 29 October 2007 / Published online: 8 December 2007

Abstract Several MPI systems for Grid environment, in which clusters are connected by wide-area networks, have been proposed. However, the algorithms of collective communication in such MPI systems assume relatively low bandwidth wide-area networks, and they are not designed for the fast wide-area networks that are becoming available. On the other hand, for cluster MPI systems, a bcast algorithm by van de Geijn, et al. and an allreduce algorithm by Rabenseifner have been proposed, which are efficient in a high bi-section bandwidth environment. We modify those algorithms so as to effectively utilize fast wide-area inter-cluster networks and to control the number of nodes which can transfer data simultaneously through wide-area networks to avoid congestion. We confirmed the effectiveness of the modified algorithms by experiments using a 10 Gbps emulated WAN environment. The environment consists of two clusters, where each cluster consists of nodes with 1 Gbps Ethernet links and a switch with a 10 Gbps upper link. The two clusters are connected through a 10 Gbps WAN emulator which can insert latency. In a 10 millisecond latency environment, when the message size is 32 MB, the proposed bcast and allreduce are 1.6 and 3.2 times faster, respectively, than the algorithms used in existing MPI systems for Grid environment.

Keywords Message passing interface (MPI) · Collective communication · Wide-area network · Grid · Broadcast · Allreduce

M. Matsuda (✉) · T. Kudoh · Y. Kodama · R. Takano
Grid Technology Research Center, National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba, Japan
e-mail: m-matsuda@aist.go.jp

Y. Ishikawa
The University of Tokyo, Tokyo, Japan

1 Introduction

There are several MPI systems for clusters in Grid environment, such as MPICH-G2, MagPIe, and PACX-MPI, and they have implemented a set of efficient algorithms of collective communication for high latency networks [6, 11, 12]. The set of algorithms assumes that the bandwidth of inter-cluster links is much lower than that of the network links of the cluster nodes.

However, recently, the bandwidth of wide-area networks has become much wider, and far exceeds the bandwidth of the network interfaces of typical cluster nodes [7, 19]. This situation will continue at least for a while, because the optical network technology for wide-area networks will continue to advance, while providing the fastest network interfaces to all the cluster nodes costs too much and is impractical. Thus, new algorithms of collective communication are needed, which match the fast inter-cluster networks.

Among the set of collective operations, *bcast* and *allreduce* are two important ones [16]. The *bcast* operation (`MPI_Bcast`) is a broadcast, in which data on one node (called a root node) is copied to all other nodes. The *allreduce* operation (`MPI_Allreduce`) is a reduction operation to all nodes, in which a reduction result is copied to all nodes. It can be thought as a reduction followed by a broadcast. It is reported that the *allreduce* operation took 37% of MPI execution time in 5-year profiling on a Cray T3E [16, 20].

For these two operations, efficient algorithms have been invented targeting a high bi-section bandwidth environment: the *bcast* algorithm by van de Geijn et al. [2, 3], and the *allreduce* algorithm by Rabenseifner [17]. Both the *bcast* and *allreduce* algorithms are based on a similar idea of splitting a message, where a message is split and distributed to the nodes, and then gathering the split messages again in every node. The van de Geijn *bcast* can be implemented by *scatter* followed by *allgather*. The Rabenseifner *allreduce* can be implemented by *reduce-scatter* followed by *allgather*. Here, *scatter*, *reduce-scatter*, and *allgather* are all implemented by log(P)-step algorithms, and can effectively utilize the available bandwidth of each node.

We have been investigating algorithms of collective communication for systems where multiple clusters are connected by wide-area networks. In such systems, the network interface of each node has one bi-directional link, and intra-cluster communication is through a switched network, while inter-cluster communication is through a long-and-fast network. Here, long-and-fast roughly means:

long: $L(\text{inter-cluster}) \gg L(\text{inter-node}) * \log(P)$

fast: $B(\text{inter-cluster}) > B(\text{node-link})$

where, L is the latency, B is the bandwidth, and P is the number of nodes in a cluster. The reason for adding $\log(P)$ factor to the latency is to assume that arbitrary $\log(P)$-step algorithms can be performed inside a cluster without considering inter-cluster communication issues. Also, we focus only on large messages, since the efficiency of communication for short messages depends almost only on the inter-cluster latency. Under these assumptions, the inter-node latency inside a cluster is marginal and can be omitted from consideration.

A low bandwidth environment, in which the inter-cluster bandwidth is less than the bandwidth of a network link of a node, is out of the scope of this paper. In such an environment, only one node will perform inter-cluster communication at a time, and regulating the amount of transmission from that node is the issue. Bandwidth limiting mechanisms such as PSPacer [18] and Linux's Token Bucket Filter will work for that purpose, but they are not discussed in this paper.

We have shown in a previous paper that the practical upper-bound of the latency of inter-cluster communication is about 10 milliseconds, when benchmark programs for clusters, such as the NAS Parallel Benchmarks, are not modified to tolerate latency [14]. Although the effect of latency naturally depends on the application, most benchmarks have shown good performance up to a 10 millisecond latency. Out of this range, however, most benchmarks run poorly and connecting two clusters is meaningless in terms of the computing performance, whereas there is still a benefit of using large amounts of resources such as memory and disks. A 10 millisecond latency roughly corresponds to 1000 miles in actual networks, and there may exist some large-scale clusters in this range. Thus, all experiments in this paper are performed with a 10 millisecond latency.

In the following, the designs and the implementations of the algorithms for long-and-fast networks are described in Sect. 2 and Sect. 3. Section 2 describes the *bcast* and *allreduce* operations, and Section 3 briefly describes the extensions to the other collective operations. The experimental results are shown in Sect. 4. We mention very briefly related work in Sect. 5, and conclude the paper in Sect. 6.

2 Design and implementation

2.1 Design overview

Our objective is to design algorithms of collective communication to utilize the available bandwidth of wide-area networks, which is a number of times larger than the bandwidth of the network link of each node, e.g., the inter-cluster bandwidth is 10 Gbps while the inter-node bandwidth is 1 Gbps. In such an environment, a number of nodes should send messages simultaneously to the inter-cluster network to fully utilize the bandwidth. However, contention among the messages should be avoided when nodes send messages simultaneously, especially when the TCP/IP protocol is used on long-and-fast networks [15]. The total transmission rate of the sending nodes should be limited to the bandwidth of the inter-cluster network.

For a cluster environment, good algorithms for *bcast* and *allreduce* operations have been proposed. van de Geijn et al. [2, 3] proposed a *bcast* algorithm. Rabenseifner [17] proposed an *allreduce* algorithm. Our algorithms are based on those algorithms, which are modified to efficiently utilize the bandwidth of wide-area networks which connect clusters.

The van de Geijn *bcast* is algorithmically equivalent to *scatter* followed by *allgather*. The *scatter* operation splits a message and distributes the parts of that message to all nodes. The *allgather* operation collects parts of the message from all nodes, and rebuilds the message from the collected parts on each node. We modified this algorithm to extend it for inter-cluster communication. The modified van de Geijn *bcast* inserts a copy operation between the *scatter* and *allgather* stages, which copies the message between the clusters through a wide-area network. Details are described following this section.

The Rabenseifner *allreduce* is algorithmically equivalent to *reduce-scatter* followed by *allgather*. The *reduce-scatter* operation splits a message and distributes the parts of that message to all nodes. In addition, it performs a reduction on the part of the message. The *allgather* operation collects parts of the reduced message from all nodes, and rebuilds the result in each node. We modified this algorithm to extend it for inter-cluster communication. The modified Rabenseifner *allreduce* inserts copy and reduction operations between the *reduce-scatter* and *allgather* stages in a way similar to the case of *bcast*. Details are described following this section, too.

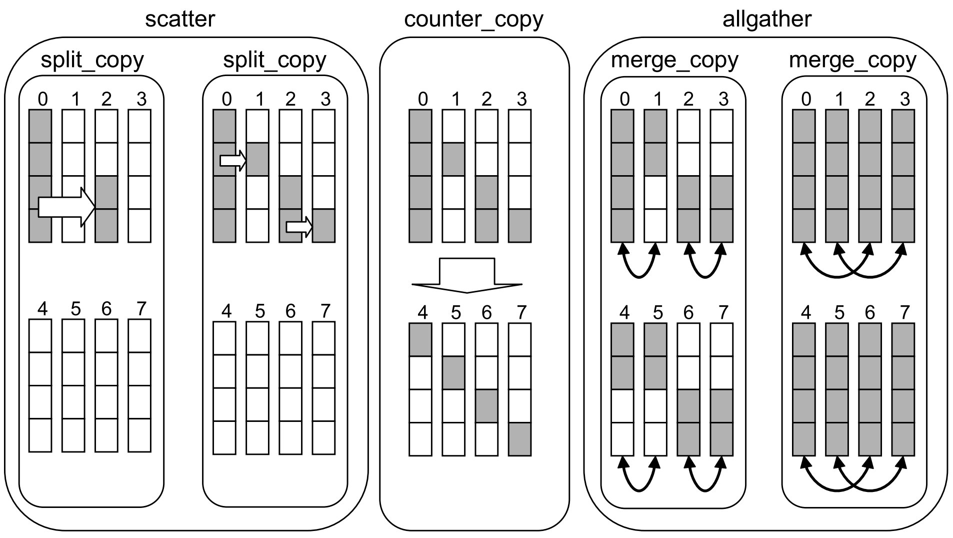


Fig. 1 Data movement of the modified van de Geijn bcast

Note that in the beginning, data represented by the gray area is in node 0.

```
void
vandegeijn(void *buf, int siz)
{
  if (rank < (nprocs/2)) {
    for (i = 0; i < (log2(nprocs)-1); i++) {
      split_copy(buf, siz, i, hemisphere);
    }
  }
  counter_copy(buf, siz);
  for (i = 0; i < (log2(nprocs)-1); i++) {
    merge_copy(buf, siz, i, hemisphere);
  }
}
```

Fig. 2 Skeleton of the modified van de Geijn bcast

Based on the modifications of the algorithms as stated above, all nodes in a cluster participate in inter-cluster communication. Now, we need to regulate the total transmission rate to avoid contention among the messages. We took a simple forwarding approach in the implementation, in which only selected nodes are allowed to send messages to the opposite cluster, while the other nodes send messages to the selected nodes inside the cluster.

2.2 Bcast

Figures 1 and 2 show an outline of the modified van de Geijn *bcast* and its data movement example. The algorithm is identical to the van de Geijn *bcast* except for the addition of `counter_copy` and working in the `hemisphere` communicator. The `hemisphere` communicator represents a half of `MPI_COMM_WORLD` and corresponds to each cluster. The `nprocs` variable holds the number of processes in `MPI_COMM_WORLD`.

The steps of the modified van de Geijn *bcast* are as follows:

1. The `split_copy` function performs the steps of *scatter* in the original *bcast* algorithm. In the i-th step, it splits a message in half and copies the half to a $2^{(nprocs/2-i)}$-apart process. It follows a pattern called *recursive-halving*. Note that the `split_copy` function is only performed in a cluster which contains the root rank (the source of the *bcast* data).
2. The `counter_copy` function copies the scattered messages from the cluster containing the root rank to the other cluster. Basically, each node sends its part of the message to the opposite node in the other cluster as shown in Fig. 1. That is, there is one-to-one correspondence between nodes in each cluster.
3. The `merge_copy` function performs the steps of *allgather* in the original *bcast* algorithm. In the i-th step, it merges a message from a 2^i-apart process to its holding message. It follows a pattern called *recursive-doubling*.

The `counter_copy` function can overflow the bottleneck link when all the nodes simultaneously send their messages. Thus, `counter_copy` takes a parameter for the number of nodes which are allowed to send messages simultaneously. Only selected nodes may send messages to the other cluster. The other nodes forward their messages to selected nodes inside the cluster. The number of selected nodes can be any value between 1 and $nprocs/2$ according to the available inter-cluster bandwidth.

2.3 Allreduce

Figures 3 and 4 show an outline of the modified Rabenseifner *allreduce* and its data movement example. The al-

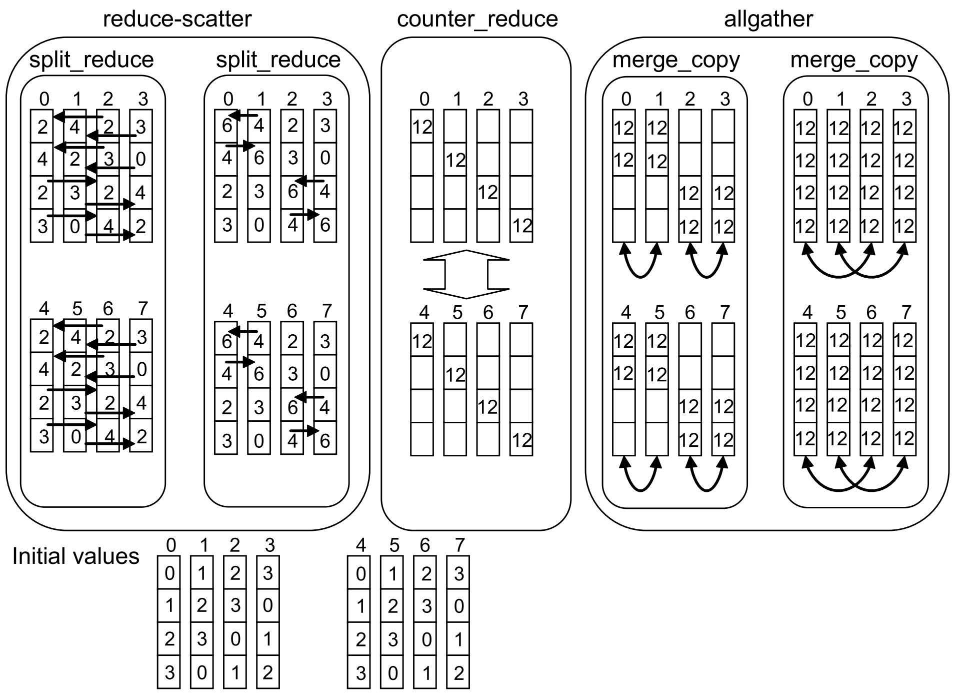


Fig. 3 Data movement in the modified Rabenseifner allreduce with the addition operation

```
void
rabenseifner(void *buf, void *tmp, int siz)
{
  for (i = 0; i < (log2(nprocs)-1); i++) {
    split_reduce(buf, tmp, siz, i, hemisphere);
  }
  counter_reduce(buf, tmp, siz);
  for (i = 0; i < (log2(nprocs)-1); i++) {
    merge_copy(buf, siz, i, hemisphere);
  }
}
```

Fig. 4 Skeleton of the modified Rabenseifner allreduce

gorithm is identical to the Rabenseifner *allreduce* except for the addition of `counter_reduce` and working with `hemisphere`. The *allreduce* algorithm is very similar to *bcast*, where each step performs a reduction in place of a simple copy.

The steps of the modified Rabenseifner *allreduce* are as follows:

1. The `split_reduce` function performs the steps of *reduce-scatter* in the original *allreduce* algorithm. In the i-th step, it splits a message in half and copies the half to a $2^{(nprocs/2-i)}$-apart process. It then performs reduction. It follows a pattern called *recursive-halving*.
2. The `counter_reduce` function performs a bi-directional copy and a reduction on the exchanged message. The `counter_reduce` function works in the same way as `counter_copy`, but it is followed by a reduction.
3. The `merge_copy` function performs the steps of *allgather* in the original *allreduce* algorithm. It is the same as the one in the modified van de Geijn *bcast* algorithm.

`counter_reduce` takes a parameter for the number of nodes which are allowed to send messages simultaneously, too. It can take any value between 1 and $nprocs/2$.

Note that `counter_reduce` is bi-directional and receives messages from the opposite cluster. Therefore, forwarding inside a cluster may cause conflicting use of the receiving link of the node. In the implementation, the order of sends was ad hocly skewed in order to avoid concentration, but we observed no significant effect on the performance of the operation in the experiments.

2.4 Design choice: avoiding contention

The forwarding scheme described in the previous subsections is one choice to reduce the inter-cluster traffic, and there are other ways to reduce the number of nodes simultaneously communicating. We will discuss some design choices below.

One way is to algorithmically reduce the number of nodes. In *bcast*, the `split_copy` operation splits and distributes a message as a part of the *scatter* stage. If the `counter_copy` operation is performed just after the i-th iteration of `split_copy`, 2^i nodes hold the split message. Therefore, only 2^i nodes participate in `counter_copy`. After `counter_copy`, the remaining iterations of `split_copy` are performed. Although this scheme seems attractive, the number of communicating nodes is limited to powers of two, and it is not flexible.

Algorithmically reducing number of sending nodes cannot be done without some penalty in *allreduce*. The `split_reduce` steps in the *reduce-scatter* stage cannot be suspended before completion, because fully reduced values are needed to minimize the message exchanged between clusters. Therefore, to reduce the number of nodes allowed to communicate, a few extra `merge_copy` operations must be performed before the `counter_reduce`. Repeating `merge_copy` i times makes 2^i nodes hold identical copies of the reduced message. Thus, after the i-th `merge_copy`, the $1/2^i$ nodes may participate in the `counter_reduce` function. However, the receiving side also needs to perform some extra `split_copy` operations to get back to the state at which *allgather* can be performed. Note that `split_copy` undoes the effect of `merge_copy` performed in the sending cluster. An arbitrary number of pairs of `split_copy` and `merge_copy` can be used without affecting the correctness of the algorithm.

Another way to reduce the traffic is to select a set of nodes by chaining one set after another, and phasing the use of the bottle-neck link. However, there is no portable way to know the end of transmission in the standard socket API. Messages are buffered in the socket, and there is no way to know when the socket buffer becomes empty. Therefore, chaining the nodes is not considered in our implementation, because it needs support from the system software.

The forwarding mechanism increases the traffic inside a cluster, and also occupies the receiving link on the forwarding node. However it is the simplest way, and allows regulating the sending nodes to an arbitrary number. Thus, simple forwarding is used in our implementation.

2.5 Adaptation for more clusters

Naturally, the two-cluster algorithm described above can be extended to more clusters. Inter-cluster communication can be simple one-to-all for *bcast* and all-to-all for *allreduce*. The algorithms can also be adapted to the available bandwidth between each pair of clusters. When the wide-area network is shared, in a case such as one where one city exists in the middle of two other cities, the number of sending nodes should be reduced.

In addition, there is a situation where imbalance exists in the number of nodes of clusters. The above mentioned method to algorithmically reduce the number of communicating nodes can be used at the larger cluster, in case the numbers of nodes in clusters are different by more than a factor of two. By using the method, the number of communicating nodes in the larger cluster can be matched to that of the opposing cluster.

3 Other MPI collective operations

3.1 MPI collective operations

The operations *bcast* and *allreduce* are very important, but MPI defines 16 collective operations (14 in MPI-1.2). They are shown in Table 1. This section briefly describes applying the modifications to other operations. As we have described in the design section (Sect. 2), collective operations can be implemented by combinations of more primitive operations. Some of the collective operations are sub-algorithms of *bcast* and *allreduce*, and our modifications work for them straightforwardly. Avoiding message contention by limiting the number of nodes which send messages simultaneously should work for some other collective operations.

We omitted *barrier*, *scan*, *exscan*, and operations with the postfixes *v* and *w* from consideration. The operation *barrier* is omitted, because its message size is zero. The operations with the postfixes are omitted, because they are variants of the operations without the postfixes and work similarly. The operations *scan* and *exscan* are also omitted, because data do not take appropriate positions when the modifications are based on splitting and scattering found in the van de Geijn or Rabenseifner algorithms, and applying our modifications is not straightforward.

3.2 Allgather/reduce-scatter

The *allgather* operation gathers parts of data from all nodes and copies the gathered data to all nodes. It is the operation found in the second-half of the van de Geijn *bcast* algorithm. The operation can be carried out by first exchanging data between clusters, and then, by performing *allgather* inside each cluster. Exchanging data can be performed by the `counter_copy` function modified to work in both directions.

The *reduce-scatter* operation scatters parts of the result of the reduction. It is the operation found in the first-half of the Rabenseifner *allreduce* algorithm. The operation can be carried out by first performing *allreduce* inside each cluster, and then, exchanging and reducing data between clusters. Exchanging and reducing data can be performed by the `counter_reduce` function.

Table 1 Collective operations of MPI-2.0

MPI_Barrier	MPI_Bcast	MPI_Gather	MPI_Gatherv	MPI_Scatter	MPI_Scatterv
MPI_Allgather	MPI_Allgatherv	MPI_Alltoall	MPI_Alltoallv	MPI_Alltoallw	MPI_Reduce
MPI_Allreduce	MPI_Reduce_scatter	MPI_Scan	MPI_Exscan		

These operations are sub-algorithms of the van de Geijn *bcast* and Rabenseifner *allreduce* algorithms.

3.3 Reduce

The *reduce* operation can be implemented by *reduce-scatter* followed by *gather*. That is, *allgather* found in the second-half of the Rabenseifner *allreduce* algorithm is replaced by *gather*. This operation is a sub-algorithm of the Rabenseifner *allreduce* algorithm.

3.4 Scatter

The *scatter* operation distributes parts of data on one node (called a root node) to all nodes. In *scatter*, the root node should generate all data to others, and the network of the root node is a bottle-neck. Thus, there is no merit of using multiple inter-cluster connections.

3.5 Gather/alltoall

The *gather* operation gathers parts of data on all nodes to one root node. The *alltoall* operation exchanges data from each node to each node. These operations have no similarity to the van de Geijn *bcast* and Rabenseifner *allreduce* algorithms. But, avoiding contention can be useful, because all nodes send data to one node simultaneously. Limiting the number of nodes which send messages over the inter-cluster connections can be applied to these operations. Since they are only operations essentially differ from the van de Geijn *bcast* and Rabenseifner *allreduce* algorithms, their performance results are included in the evaluation section.

4 Evaluation

4.1 Simple cost model of bcast

The communication cost is modeled by the following parameters in this section: M is the message size, B is the bandwidth of the link of the node, L is the latency of inter-cluster communication. n is the number of connections used in inter-cluster communication.

To compare the performance of the proposed *bcast* algorithm, a simple *bcast* algorithm is implemented. The algorithm is called *far-first bcast* in this paper. In the algorithm, the whole message is sent first using inter-cluster communication. It minimizes time by using long links first. This algorithm is a simplified one used in existing MPI systems for wide-area networks. It uses the van de Geijn *bcast* algorithm inside a cluster.

The cost of *far-first* is $(L + M/B + (M/B + M/B))$, when simply ignoring the latency in the intra-cluster communication. The term $(M/B + M/B)$ is for the *bcast* inside a cluster, where the first M/B in it corresponds to the *scatter* stage, and the second M/B to the *allgather* stage. Note that the cost of the *scatter* stage is M/B, because the `split_copy` operation halves the message in each step, and the sum of the cost of repeating it accumulates to M/B asymptotically. It is similar for the *allgather* stage.

Similarly, the cost of the modified van de Geijn *bcast* is $(L+M/nB+M/B+M/B)$. The second term is changed to M/nB by the effect of using multiple connections. Thus, the modified van de Geijn wins by multiple uses of connections.

Table 2 summarizes the costs of the algorithms.

Table 2 Cost of algorithms

	Bcast
van de Geijn	$(L + M/nB + M/B + M/B)$
far-first	$(L + M/B + (M/B + M/B))$
	Allreduce
Rabenseifner	$(L + M/nB + M/B + M/B)$
two-tier	$(L + M/B + (M/B + M/B) + (M/B + M/B))$
	Others
Allgather	$(L + M/nB + M/B)$
Reduce-scatter	$(L + M/nB + M/B)$
Reduce	$(L + M/nB + M/B + M/B)$
Reduce (*two-tier*)	$(L + M/B + M/B + M/B)$
Scatter	$(L + M/B)$
Gather	$(L + M/B)$
Alltoall	$(L + M/B)$

4.2 Simple cost model of allreduce

To compare the performance of the proposed *allreduce* algorithm, a simple *allreduce* algorithm is implemented. The algorithm is called *two-tier allreduce* in this paper. It first performs the reduction inside each cluster, exchanges the reduced messages between clusters, and then performs *bcast* with the messages inside each cluster. This algorithm is a simplified one used in existing MPI systems for wide-area networks. It performs the reduction inside a cluster by a variant of the Rabenseifner *allreduce*, in which *allgather* of the second stage is replaced with *gather*.

The cost of *two-tier allreduce* is $(L + M/B + (M/B + M/B) + (M/B + M/B))$. The third and fourth terms $(M/B + M/B)$ correspond to the reduction and *bcast* carried out inside a cluster.

Similarly, the cost of the modified Rabenseifner *allreduce* is $(L+M/nB+M/B+M/B)$. The third M/B term corresponds to the *reduce-scatter* stage and the fourth M/B term corresponds to the *allgather* stage.

Note that the modified Rabenseifner performs better algorithmically, without regard to the bandwidth of inter-cluster communication.

Table 2 summarizes the costs of the algorithms.

4.3 Simple cost model of others

The *allgather* operation is the second-half of the van de Geijn *bcast*. The cost can be calculated by adding the cost of the inter-cluster communication to the cost of the second-half of the *bcast*. Similarly, the *reduce-scatter* operation is the first-half of the Rabenseifner *allreduce*. The cost can be calculated by adding the cost of the inter-cluster communication to the cost of the first-half of the *allreduce*, too.

The *reduce* operation is the same as the Rabenseifner *allreduce* whose second-half is replaced by *gather*. The cost is the same, because replacing the operation does not reduce the time taken. The cost can be calculated similarly, when the *two-tier allreduce* is used as the base algorithm. The cost of *reduce* can be reduced by using multiple connections.

The *scatter*, *gather*, and *alltoall* operations are limited by the network of a single node. Thus, the costs are very simply $(L + M/B)$.

Table 2 includes the costs of the above operations. In the table, the message size M for *gather* is the total size of the messages (i.e., the data size multiplied by the number of nodes).

4.4 Experimental setting

Figure 5 shows the experimental setting, and Table 3 shows the specification of the node PC and the Ethernet switches. Two clusters were connected via a WAN emulator. In the experiment, we used GtrcNET-10 [9] to emulate a WAN

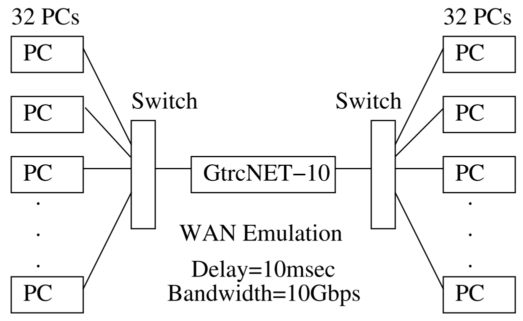


Fig. 5 Experimental setting

Table 3 PC cluster specifications

Node PC	
CPU	Opteron (2.0 GHz)
Memory	6 GB DDR333
NIC	Broadcom BCM5704 (on-board)
OS	SuSE Enterprise Server 9 (Linux-2.6.17)
Switch	Huawei-3Com Quidway S5648 + optional 10 Gbps port

environment, which is a 10 Gbps successor of a well-established network testbed, GtrcNET-1 [13] for 1 Gbps Ethernet. GtrcNET-10 consists of a large-scale Field Programmable Gate Array (FPGA), three 10 Gbps Ethernet XENPAK ports, and three blocks of 1 GB DDR-SDRAM. The FPGA is a Xilinx XC2VP100, which includes three 10 Gbps Ethernet MAC and XAUI interfaces. GtrcNET-10 provides many functions such as traffic monitoring in millisecond resolution, traffic shaping, and WAN emulation at 10 Gbps wire speed. GtrcNET-10 was used to add latency between clusters and to observe precise network traffic. GtrcNET-10 added a 10 millisecond delay (one-way) in the experiment.

We used the MPI system, YAMPII [10], in the experiments, which almost fully implements the MPI-2.0 specification. YAMPII is the base of the MPI system, GridMPI [8], for Grid environment. Both YAMPII and GridMPI are fully functional, but YAMPII was used in the experiment, because GridMPI supports heterogeneity and has overheads in handling messages (e.g., byte-order conversion).

In the experiment, TCP buffer sizes of sockets were set to 128 KB for intra-cluster connections and 2 MB for inter-cluster connections. The 2 MB buffer size was chosen to tolerate a 10 millisecond latency. Also, some kernel TCP parameters were set as in the table below, because standard settings are not adequate for the experiment. tcp_no_metrics_save disables recording of the parameters of the previous connection to reuse them. These TCP parameters can be found in Linux in the directories /proc/sys/net/core and /proc/sys/net/ipv4.

tcp_no_metrics_save	1
wmem_max	3000000
rmem_max	3000000
tcp_rmem	3000000 3000000 3000000
tcp_wmem	3000000 3000000 3000000
tcp_mem	3000000 3000000 3000000

We ran each operation 10 times and took the maximum for stable results. TCP behaves disastrously at congestion, and the variance of performance sometimes reached near 50 percent in the experiment.

4.5 Bcast performance

The left graph of Fig. 6 shows the bandwidth achieved in *bcast* by varying the message size. The unit of the Y-axis is MB/s, but it just represents the value of the total user-level messages over the time $(M * nprocs/T)$. It does not count the actual messages sent by nodes because different algorithms send different amounts of messages.

The results labeled with $n = 1$ to $n = 32$ are for the modified van de Geijn algorithm, and n indicates the number of nodes simultaneously communicating. The label *far-*

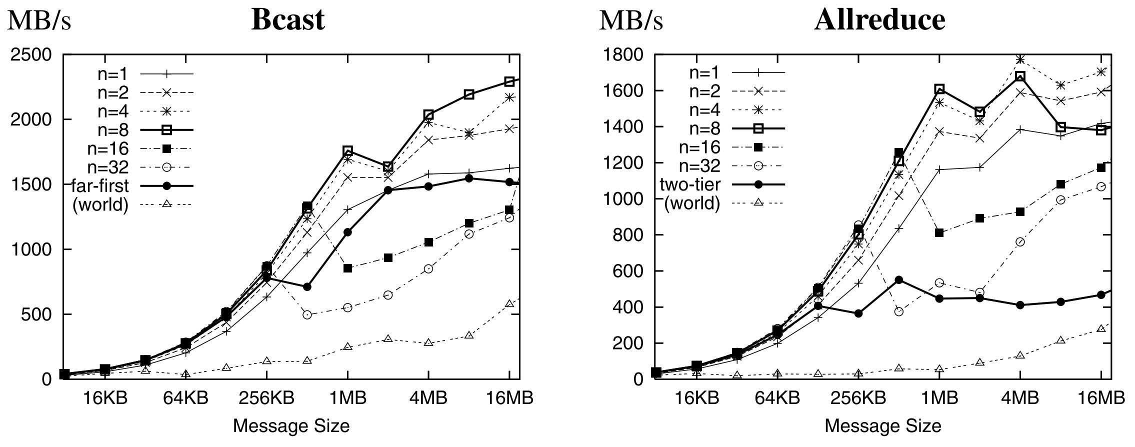


Fig. 6 Throughput of bcast and allreduce (delay = 10 msec)

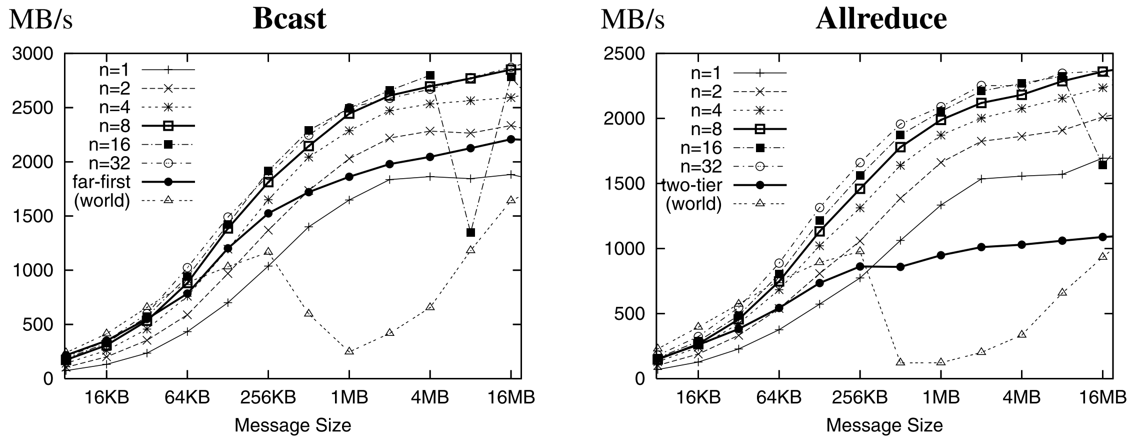


Fig. 7 Throughput of bcast and allreduce (without delay)

first indicates the *far-first* algorithm. The label *(world)* indicates the original van de Geijn algorithm, which works in `MPI_COMM_WORLD` and does not distinguish a bottle-neck link. It is included for comparison.

As the cost model suggests, *far-first bcast* behaves similarly to the modified van de Geijn algorithm for $n = 1$. However, the modified van de Geijn algorithm improves as n increases, up to $n = 8$. When the traffic overwhelms the bandwidth between clusters over $n = 8$, the performance drops but gradually improves as the message size increases.

No resend of TCP was observed in the modified van de Geijn algorithm for $n \leq 8$ in inter-cluster communication.

4.6 Allreduce performance

The right graph of Fig. 6 shows the bandwidth achieved in *allreduce* by varying the message size. The unit of the Y-axis is MB/s, but it just represents the value of the total message size over the time ($M * nprocs^2/T$).

The results labeled with $n = 1$ to $n = 32$ are for the modified Rabenseifner algorithm, and n indicates the number of nodes simultaneously communicating. The label *two-tier* indicates the *two-tier* algorithm. The label *(world)* indicates the original algorithm, which works in `MPI_COMM_WORLD` and does not distinguish a bottle-neck link.

As the cost model suggests, the modified Rabenseifner *allreduce* outperforms the *two-tier* algorithm, even at $n = 1$.

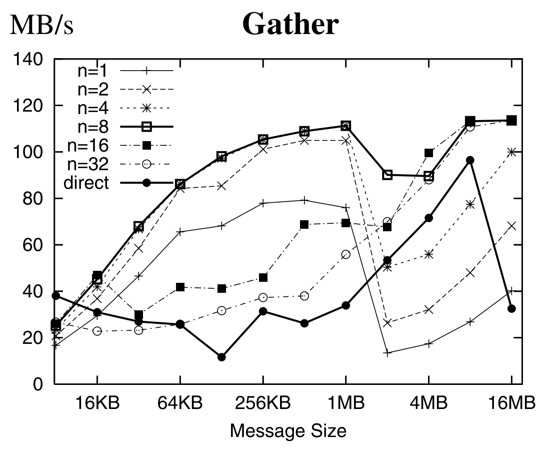
No resend of TCP was observed in the modified Rabenseifner algorithm for $n \leq 8$ in inter-cluster communication.

4.7 Clusters environment

The proposed algorithms are expected to show good performance for clusters with limited bi-section bandwidth, such as clusters with multiple Ethernet switches or fat-trees with a reduced number of links at upper levels.

Figure 7 shows the results without delay. The two Ethernet switches were still connected via GtrcNET-10 and there was a bottle-neck at 10 Gbps bandwidth.

The results labeled with *(world)* are for the original algorithms, which work in `MPI_COMM_WORLD` and do not



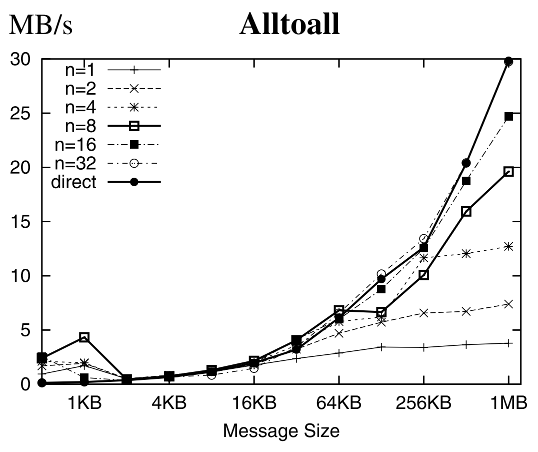


Fig. 8 Throughput of gather and alltoall (delay = 10 msec)

distinguish a bottle-neck link. The performance of the original algorithms is not good. It is because they heavily use the bottle-neck link as if it had a full bi-section bandwidth. Although the modified algorithms generally perform well, the effect of reducing the number of nodes of simultaneous communication was almost not observed in a low latency environment.

4.8 Gather and alltoall performance

Figure 8 shows the results of *gather* and *alltoall* in a 10 millisecond delay.

The graph of *gather* shows the effectiveness of using multiple connections and limiting the number of connections. The performance improves up to eight connections. The label *direct* shows the case when each node sends data directly to the root node. The drop found at 2 MB message size is caused by congestion. The reason of congestion was not precisely analyzed, but the traffic can conflict between inter-cluster and intra-cluster communication, because our algorithm limits only the inter-cluster communication but does nothing for intra-cluster communication.

The graph of *alltoall* shows no effectiveness of limiting the number of connections. The label *direct* shows the case when each node sends data directly to all nodes. The cases of *direct* and the 32 connections are the same. Congestion is not avoided in *alltoall*, and the well-known multiple stream effect of TCP works for the *direct* case, in which the over-supplied connections compensate the congested connections with each other [1]. The overall performance of *alltoall* was not good enough for the available bandwidth.

The results without delay is omitted in this case, because they behaved similarly with and without delay. For *bcast* and *allreduce*, showing the graphs without delay is meaningful because they reduce the communication cost, but our *gather* and *alltoall* have the same cost. The difference is likely due to the timing of messages and the congestion behavior, which are not investigated in this work.

5 Related work

The van de Geijn *bcast* [2, 3] and the Rabenseifner *allreduce* [17] algorithms described in the design section are used in MPICH-1.2.6 and MPICH-2 [20].

PACX-MPI [6], MPICH-G2 [11], and MagPIe [12] are MPI systems designed for Grid environment. They are all designed to exploit the hierarchy of communication media ranging from memory systems to wide-area networks, and to adapt to the complex structure in latency and topology of wide-area networks. Although some optimality results have been presented for their algorithms, they are not designed to exploit the bandwidth of a single wide-area network. Thus, their *bcast* and *allreduce* algorithms are reduced to the simple *far-first* and *two-tier* algorithms shown in the evaluation section, when the setting is a simple two cluster configuration connected by a fast network. We have already shown that the modified van de Geijn and modified Rabenseifner algorithms outperformed the *far-first* and *two-tier* algorithms in such a setting.

Chan, et al discuss algorithms of collective operations for machines capable of sending to or receiving from multiple links [5]. They have proposed the variations of algorithms depending on the capabilities and limitations of the torus network of BlueGene/L.

There is large amount of research in multicast overlay networks (including *bcast*) under the general setting where a network is represented by a weighted graph. Especially, den Burger el al. discuss the use of multiple multicast trees in clusters connected via wide-area networks [4]. However, it cannot directly be compared to the communication algorithms, because the work is general and proposes a method to find near-optimal multicast trees, and thus, the behavior of the algorithm is only implied by the trees.

6 Conclusion

Since the assumption of low bandwidth wide-area networks is now false, the algorithms of collective communication need to be redesigned. In this paper, we have developed algorithms of inter-cluster collective communication for *bcast* and *allreduce*, which are based on the algorithms by van de Geijn, et al and by Rabenseifner. They are designed for fast wide-area networks, with bandwidth larger than that of a network link of a node. The algorithms utilize multiple node-to-node connections while regulating the number of nodes simultaneously communicating, and improve the performance of collective operations on large messages. Experiments using an emulated WAN environment with 10 Gbps bandwidth and a 10 millisecond latency have shown that our *bcast* and *allreduce* algorithms performed 1.6 and 3.2 times faster, respectively, than the algorithms used in existing MPI systems for Grid environment.

Acknowledgements Part of this research was supported by a grant from the Ministry of Education, Sports, Culture, Science and Technology (MEXT) of Japan through the NAREGI (National Research Grid Initiative) Project.

References

1. Allman, M., Kruse, H., Ostermann, S.: An application-level solution to TCP's satellite inefficiencies. In: Proc. of the First Intl. Workshop on Satellite-based Information Services (WOSBIS) (1997)
2. Barnett, M., Gupta, S., Payne, D.G., Shuler, L., van de Geijn, R., Watts, J.: Interprocessor collective communication library (InterCom). In: Proc. of the Scalable High Performance Computing Conference, pp. 357–364 (1994)
3. Barnett, M., Gupta, S., Payne, D.G., Shuler, L., van de Geijn, R., Watts, J.: Building a high-performance collective communication library. In: Proc. of the 1994 Conference on Supercomputing (SC94), pp. 107–116 (1994)
4. den Burger, M., Kielmann, T., Bal, H.E.: Balanced multicasting: high-throughput communication for grid applications. In: Proc. of the 2005 ACM/IEEE Conference on Supercomputing (SC'05) (2005)
5. Chan, E., van de Geijn, R., Gropp, W., Thakur, R.: Collective communication on architectures that support simultaneous communication over multiple links. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP 2006), pp. 2–11 (2006)
6. Gabriel, E., Resch, M., Rühle, R.: Implementing MPI with optimized algorithms for metacomputing. In: Proc. of the Third MPI Developer's and User's Conference (MPIDC'99), pp. 31–41 (1999)
7. GLIF: Global Lambda Integrated Facility, http://www.glif.is
8. GridMPI Project, http://www.gridmpi.org
9. GtrcNET-10, http://www.gtrc.aist.go.jp/gnet
10. Ishikawa, Y.: YAMPII official home page, http://www.il.is.s.u-tokyo.ac.jp/yampii
11. Karonis, N.T., de Supinski, B.R., Foster, I.T., Gropp, W., Lusk, E.L., Lacour, S.: A multilevel approach to topology-aware collective operations in computational grids. Tech. Rep. ANL/MCS-P948–0402 (2002)
12. Kielmann, T., Bal, H.E., Gorlatch, S.: Bandwidth-efficient collective communication for clustered wide area systems. In: Proc. of the 14th Intl. Parallel and Distributed Processing Symp., pp. 492–499 (1999)
13. Kodama, Y., Kudoh, T., Takano, R., Sato, H., Tatebe, O., Sekiguchi, S.: GNET-1: gigabit Ethernet network testbed. In: IEEE Intl. Conf. on Cluster Computing (Cluster2004), pp. 185–192 (2004)
14. Matsuda, M., Ishikawa, Y., Kudoh, T.: Evaluation of MPI implementations on grid-connected clusters using an emulated WAN environment. In: 3rd Intl. Symp. on Cluster Computing and the Grid (CCGrid2003), pp. 10–17 (2003)
15. Matsuda, M., Ishikawa, Y., Kudoh, T., Kodama, Y., Takano, R.: TCP adaptation for MPI on long-and-fat networks. In: IEEE Intl. Conf. on Cluster Computing (Cluster2005), pp. 1–10 (2005)
16. Rabenseifner, R.: Automatic MPI counter profiling of all users: first results on a CRAY T3E 900-512. In: Proc. of the Message Passing Interface Developers and Users Conference 1999 (MPIDC99), pp. 77–85 (1999)
17. Rabenseifner, R.: Optimization of collective reduction operations. In: Intl. Conf. on Computational Science, LNCS 3036, pp. 1–9. Springer (2004)
18. Takano, R., Kudoh, T., Kodama, Y., Matsuda, M., Tezuka, H., Ishikawa, Y.: Design and evaluation of precise software pacing mechanisms for fast long-distance networks. In: 3rd Intl. Workshop on Protocols for Fast Long-Distance Networks (PFLDnet05) (2005)
19. TeraGrid, http://www.teragrid.orgVol
20. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. Int. J. High Perform. Comput. Appl. **19**(1), 49–66 (2005)

Motohiko Matsuda received his BS in science in 1988, and his doctor degree in computer science in 1999, both from Kyoto University. He joined Sumitomo Metal Industries Ltd. in 1988. He was a senior researcher at Grid Technology Research Center, National Institute of Advanced Industrial Science and Technology (AIST) from 2003 to 2007. He is a Project Associate Professor of the University of Tokyo from 2007. His research interests include communication systems and data-parallel computing libraries for clusters and Grid environment.

Tomohiro Kudoh received his Ph.D. degree from Keio University in 1992. He joined National Institute of Advanced Industrial Science and Technology (AIST), Japan in 2002. Currently he is the leader of the Cluster Technology Team of Grid Technology Research Center, AIST. In the past few years his research has focused on network as a Grid infrastructure. His recent work also includes the GridMPI, a high performance MPI for Grid environment, GbE/10GbE hardware network testbed GtrcNET, and the G-lambda project in which we are defining an interface to manage the network as a resource.

Yuetsu Kodama received his B.E., M.E. and Ph.D. degree in engineering from the University of Tokyo in 1986, 1988, and 2003, respectively. He has been engaged in the research of parallel computer architecture since he joined the Electrotechnical Laboratory in 1988. He is currently a senior researcher at the Grid Technology Research Center, National Institute of Advanced Industrial Science and Technology. He is a member of IEEE CS, IEICE, and IPSJ.

Ryousei Takano received his master degree in computer science from Tokyo University of Agriculture and Technology in 1999. He does research in the area of operating systems and high performance computing. In 2003, He joined AXE, Inc. as a Linux kernel engineer, and has been working for the GridMPI project, which focuses on development of high performance MPI executed over a Grid environment. He is a member of IEEE and IPSJ.

Yutaka Ishikawa is a professor of the University of Tokyo, Japan. He was in charge of the SCore cluster system software development at Real World Computing Partnership in Japan. He is currently a chairman of PC Cluster Consortium which continues to develop and support the SCore cluster system software. Ishikawa received the BS, MS, and PhD degrees in electrical engineering from Keio University. From 1987 to 2001, he was a member of AIST (former Electrotechnical Laboratory), METI. He was a visiting researcher at Carnegie Mellon University from 1988 for one year. From 1993 to 2001, he was the chief of Parallel and Distributed System Software Laboratory at Real World Computing Partnership.
He is a member of ACM, the IEEE Computer Society, Information Processing Society of Japan, and Japan Society for Software Science and Technology.