Efficient Allgather for Regular SMP-Clusters

Jesper Larsson Träff

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
`traff@ccrl-nece.de`

Abstract. We show how to adapt and extend a well-known *allgather* (all-to-all broadcast) algorithm to parallel systems with a hierarchical communication system such as clusters of SMP nodes. For small problem sizes, the new algorithm requires a logarithmic number of communication rounds in the number of SMP nodes, and *gracefully degrades* towards a linear algorithm as problem size increases. The algorithm has been used to implement the `MPI_Allgather` collective operation of MPI in the MPI/SX library. Performance measurements on a 72 node SX-8 system shows that graceful degradation provides a smooth transition from logarithmic to linear behavior, and significantly outperforms a standard, linear algorithm. The performance of the latter is furthermore highly sensitive to the distribution of MPI processes over the physical processors.

1 Introduction

An important and well-studied collective communication primitive for message-passing systems is the *allgather* or *all-to-all broadcast* operation [6], in which each processor has data which have to be distributed (i.e. broadcast) to all other processors. This primitive has been extensively studied in a variety of settings and, correspondingly, is known also as (for instance) *total exchange* [5,4], *catenation* [3], and *gossip* [9]. We will use the term *allgather* here.

The allgather primitive is incorporated as a *collective communication operation* in the *Message-Passing Interface* (MPI) standard [11] in two flavors. The `MPI_Allgather` collective is *regular* in the sense that the size of the data to be broadcast by each MPI process must be the same for all processes. The more general, *irregular* `MPI_Allgatherv` collective does not have this restriction, and each process may contribute data of different size. A peculiarity of both MPI primitives, however, is that the size of the data contributed by each process is known by all processes in advance.

There has recently been much interest in improving the collective operations in various MPI libraries, see for instance [1,10,12] (and the references therein). Various allgather algorithms for MPI were discussed and evaluated in [2]. However, the collective operations in many MPI libraries are not adapted to systems with hybrid, hierarchical communication systems such as clusters of SMP nodes (see [7,8,10] for exceptions).

In this paper we give an improvement to a well-known allgather algorithm which makes it suitable to the SMP case. In this context an SMP cluster is simply a collection of multi-processor nodes interconnected by a communication network. Communication between processors on the same node (typically via shared memory) is assumed to be faster (lower latency, higher bandwidth) than between processors on different nodes. Most importantly, the number of processors per node that can simultaneously communicate with processors on different nodes is restricted, typically to only one processor, although some modern high-performance interconnects offer multiple communication ports. We assume that communication within nodes is homogeneous, and likewise that the interconnect over the nodes is homogeneous, that is the cost of communication between any two processors on two different nodes is independent of the location of the two processors.

MPI is a process based model. Sets of processes are represented by so-called *communicators*. The semantics of the MPI collectives is defined in terms of the numbering of the processes in the given communicator. Since new communicators can be defined arbitrarily from existing ones, no assumptions about the numbering of MPI processes residing on an SMP node can be made. In particular, it cannot be assumed that the processes on a node form a consecutively numbered subset. Since allgather is a symmetric operation, it is desirable that the performance of `MPI_Allgather` be independent of the numbering of the processes.

In this paper we are concerned with *regular* SMP clusters, where the SMP nodes have the same number of processors. Additionally, the performance bounds for the allgather algorithm requires each node to run the same number of MPI processes. Again, since MPI allows arbitrary creation of new communicators, also for regular clusters it is possible to create communicators that do not fulfill this assumption. The algorithm can be used for the general case also, but can incur load imbalance. Better performance could possibly be achieved by a dedicated, non-regular algorithm. The algorithm can also be used for the irregular `MPI_Allgatherv` collective, but for very irregular problems better performance could possibly be achieved by a dedicated, irregular allgather algorithm.

2 An Allgather Algorithm with Graceful Degradation

We first present the regular allgather algorithm independent of MPI for systems with a homogeneous communication system (non-SMP case). The new feature which makes the algorithm better suited to clusters of SMP nodes is a smooth transition from logarithmic to linear behavior as the problem size grows. We term this feature *graceful degradation*.

We let p denote the number of processors, which are numbered from 0 to $p-1$. Each processor r has a block of data $\texttt{block}[r]$ of size b. For the regular allgather problem, b is the same for all processors. The *total size* of the allgather problem at hand is $m = pb$. The task of the allgather operation is to collect all blocks $\texttt{block}[0], \texttt{block}[1], \ldots, \texttt{block}[p-1]$ on all processors (in that order). By convention, for $i \leq j$, we let $\texttt{block}[i, j]$ denote the consecutive sequence of blocks

$\texttt{block}[i], \texttt{block}[i+1], \ldots, \texttt{block}[j]$, and for $j < i$, we let $\texttt{block}[i,j]$ denote the "wrapped" consecutive sequence of blocks $\texttt{block}[i], \texttt{block}[i+1], \ldots, \texttt{block}[p-1], \texttt{block}[0], \ldots, \texttt{block}[j]$.

The algorithm consists of a *logarithmic phase*, a *linear phase*, and a *last round*, either of which can be empty. In the logarithmic phase, each processor in each round doubles the number of blocks that it has collected. The algorithm used in this phase is the catenation algorithm of [3] (whose communication pattern is a regular, so-called *circulant graph*). The number of rounds of the logarithmic phase is determined by K, which can be any integer less than or equal to $\lfloor \log p \rfloor$. In the linear phase, larger, consecutive chunks consisting of 2^K input blocks are pipelined through rings of processors, until in the last round a last chunk of size strictly less than 2^K blocks is sent and received by each processor. In each round each processor sends and receives the same number of blocks. Below follows a more precise description of the *combined algorithm*.

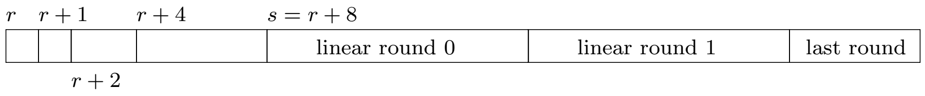

Fig. 1. The three phases of the combined allgather algorithm illustrated from a single processors point of view. In the *logarithmic phase* processor r receives blocks of size $1, 2, 4$ from processors $r+1, r+2, r+4$ respectively. In the *linear phase* processor r receives two blocks of size 8 from processor $s = r+8$. In the *last round* the remaining smaller block of size 5 is finally received from processor s.

1. In round k of the *logarithmic phase*, for $k = 0, \ldots K-1$, processor r receives blocks $\texttt{block}[s, (s+2^k-1) \bmod p]$ from processor $s = (r+2^k) \bmod p$ and sends blocks $\texttt{block}[r, (r+2^k-1) \bmod p]$ to processor $(r-2^k) \bmod p$.
2. Let $s = (r+2^K) \bmod p$, and $t = (r-2^K) \bmod p$, where K is the number of rounds of the logarithmic phase. In round k of the *linear phase*, $k = 0, \ldots, \lfloor p/2^K \rfloor - 2$, processor r receives blocks $\texttt{block}[(s+k2^K) \bmod p, (s+(k+1)2^K-1) \bmod p]$ from processor s and sends blocks $\texttt{block}[(r+k2^K) \bmod p, (r+(k+1)2^K-1) \bmod p]$ to processor t.
3. In the case that $\lceil p/2^K \rceil 2^K > p$, in the *last round* processor r receives blocks $\texttt{block}[(s+(\lfloor p/2^K \rfloor -1)2^K) \bmod p, (s+p-2^K-1) \bmod p]$ from processor s and sends blocks $\texttt{block}[(r+(\lfloor p/2^K \rfloor -1)2^K) \bmod p, (r+p-2^K-1) \bmod p]$ to processor t.

The three phases of the combined algorithm are illustrated in Figure 1. Correctness follows, since in each round, each processor receives a consecutive segment of new blocks, and sends a consecutive segment of blocks received in the previous round.

The number of rounds required is $K + \lceil p/2^K \rceil - 1$, and the total number of blocks sent and received per processor is $p-1$ for a total communication volume per processor of $(p-1)b = m-b$. Each round entails either two communication steps (a send and a receive) for uni-directional interconnects, or one communication step (a combined send-receive) for interconnects supporting full bi-directional communication.

For $K = \lfloor \log p \rfloor$ the algorithm coincides with the algorithm in [3], and for $K = 0$ with a trivial, linear time ring algorithm. By choosing $K = \lfloor \log(B/b) \rfloor$ for some fixed *intermediate buffer size* the algorithm switches from logarithmic to linear behavior before the size of the consecutive segments received and sent in a round exceeds B. Thus, with increasing block size b the algorithm gracefully changes from purely logarithmic to linear behavior.

3 Implementation on SMP Clusters

The allgather operation allows a simple *hierarchical decomposition* to exploit the faster communication between processors on the same SMP node. The hierarchical allgather algorithm looks as follows.

1. Choose a local root on each SMP node
2. On all nodes gather input blocks to local root
3. Perform allgather over local roots
4. On all nodes broadcast result from local root

A straightforward implementation of this scheme would be inefficient for medium and large problems, since non-root processes would sit idle throughout the allgather step. For the implementation of `MPI_Allgather` an additional complication is caused by the fact that MPI processes are not necessarily consecutively numbered within the SMP nodes. Thus the blocks gathered at the local roots in the second step will either be nonconsecutive at the local root, or will have to be stored consecutively in an intermediate buffer. Both solutions have undesirable drawbacks.

For the broadcast (and the gather) operation, an SMP implementation would presumably use shared memory. In many cases, shared memory used for communication between MPI processes has to be specially allocated outside of process memory, and cannot be arbitrarily large.

By using the allgather algorithm of Section 2 each of these problems can be effectively addressed for allgather problems up to a certain size. For now we consider *regular* SMP systems with the same number of MPI processes per node. We let N denote number of nodes, and n the number of processes per node such that $p = nN$.

A shared memory buffer is used for the gather and broadcast operations, and is chosen to be of a fixed, maximum size B. The number of logarithmic rounds of the allgather algorithm is chosen as $K = \min(\lfloor \log N \rfloor, \lfloor B/nb \rfloor)$, where nb is the total size of the input blocks on each SMP node.

The hierarchical allgather algorithm is implemented as follows. A local root process r is chosen for each SMP node, and allocates a shared memory communication buffer of size B. For each node the blocks `block`$[i]$ for the processes on the node are packed consecutively into the shared memory buffer using a node local consecutive numbering of the processes. The local roots execute the allgather algorithm of Section 2 with the modification that after each round of the linear phase, the blocks sent to processor t are unpacked into the result buffer of *all* processes on the SMP node. After the last round, the broadcast is completed by unpacking the last segment of blocks. This implementation effectively pipelines the allgather and the broadcast step of the hierarchical algorithm. We note that for communication systems that support concurrent communication and computation, unpacking of the blocks being sent to t into the result buffer of the local root can be performed concurrently with sending these blocks and receiving the next blocks from process s.

This algorithm can be used for allgather problems for which $nb \leq B$, i.e. for problems where the blocks of the processes on each SMP node can fit into the shared memory buffer. For larger problems a *linear ring algorithm* can be used. This should be implemented as follows. The MPI processes are sorted according to their SMP node id. The index of each process in this sorted sequence is used as virtual rank. In $p-1$ rounds, each process with virtual rank r receives a block from virtual process $(r+1) \bmod p$ and sends a block to virtual process $(r-1) \bmod p$.

For systems with large SMP nodes (say, more than 8 processors per node) the transition from SMP algorithm to linear ring (which occurs when $nb > B$) may be too coarse. This can be avoided by introducing a linear algorithm similar to the linear phase for "medium sized" problems. The number of input blocks that can be kept in the shared memory buffer is $\lfloor B/b \rfloor$, so the p processors are divided into $p/\lfloor B/b \rfloor$ *virtual nodes* each of size $\lfloor B/b \rfloor$. A local root is chosen for each virtual node, and the linear phase of the allgather algorithm is executed over the virtual roots.

4 Performance Evaluation

The SMP allgather algorithm has been incorporated into the MPI/SX implementation for NEC's SX series of parallel vector computers [10]. In this section we evaluate the implementation using the 72 node SX-8 system at HLRS (Hochleistungsrechenzentrum, Stuttgart).

The basic performance of the combined algorithm for $N = 36$ nodes and $n = 1, 4, 8$ processes per node is illustrated in Figure 2 by comparing it to a linear ring algorithm. Running time is given as a function of the block size per process b. For small blocks up to a few KBytes the improvement over the linear algorithm is more than a factor of 3 for $n = 1$ process per node, and more than 13 for $n = 8$ processes per node. As block size increases the performance of the combined algorithm converges towards that of the linear algorithm. For $n = 8$ processes per node the switch to linear ring occurs after 64 KBytes (per process; the maximum shared memory buffer size is set to $B = 1$ MByte), and incurs a performance decrease by a factor of two (thus, the additional linear algorithm over virtual nodes described above would be worth considering).

The effect of graceful degradation towards the linear performance is further illustrated in Figure 3. This compares the combined algorithm to an SMP implementation of the logarithmic algorithm of [3], which switches to a linear ring

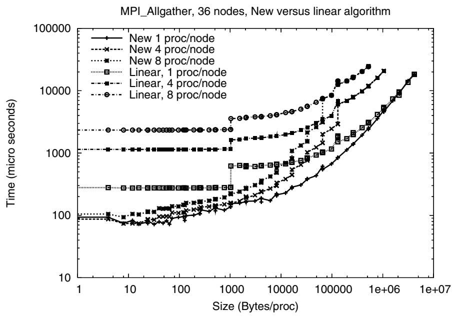

Fig. 2. The combined allgather algorithm with graceful degradation over ordered (`MPI_COMM_WORLD`) communicator compared to the linear ring algorithm for $N = 36$ nodes and $1, 4, 8$ processes per node

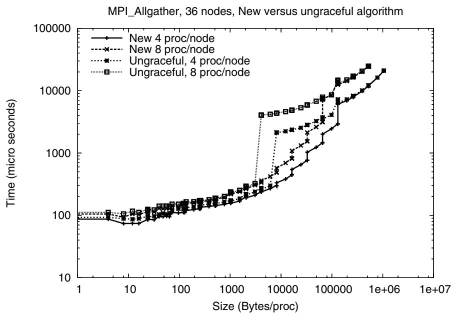

Fig. 3. The combined allgather algorithm over `MPI_COMM_WORLD` compared to an algorithm with without graceful degradation for $N = 36$ nodes and $n = 4, 8$ processes per node

as soon as the gathered result cannot fit into the shared memory buffer. This hybrid algorithm exhibits a very sharp jump in running time, which for $n = 8$ processes per node occurs at 8 KBytes, and is about a factor 7.

The potential sensitivity of a non-SMP algorithm to the numbering of of the MPI processes over the SMP nodes is illustrated in Figure 4. The combined algorithm (shown left) is compared to a linear ring algorithm over the MPI ranks (shown right) for the ordered `MPI_COMM_WORLD` communicator and a communicator in which the processes have been randomly permuted. In the latter case, the successor and predecessor of process r (namely $(r-1) \bmod p$, and $(r+1) \bmod p$) are almost always on a different SMP node, so in each communication round, almost all n processes on each node attempt to communicate with a process on another node, leading to serialization at the nodes. As expected, for $n = 4$ the random communicator performance is from a factor 2 for small block sizes up to almost a factor 4 for large block sizes worse than the ordered communicator. For $n = 8$ the performance degradation ranges from a factor of 3 up to a factor of 6. For somewhat larger block sizes, the combined algorithm is insensitive to the

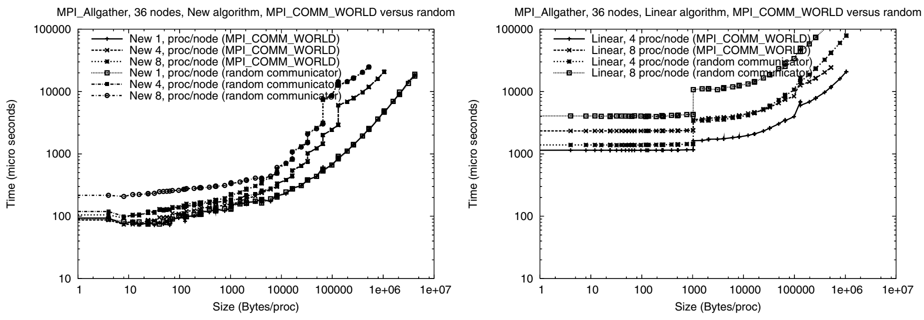

Fig. 4. The combined allgather algorithm (left) compared to non-SMP aware linear algorithm (right) for ordered `MPI_COMM_WORLD` and random communicator for $N = 36$ nodes and $n = 1, 4, 8$ processes per node

MPI process distribution. The difference for $n = 8$ for small block sizes is due to the fact that for the random communicator the gathered result does not form a consecutive segment of blocks, and must be unpacked as p individual blocks. On a vector machine like the NEC SX-8, copying of small blocks is penalized.

5 Concluding Remarks

We presented an allgather algorithm which combines well-known logarithmic and linear round allgather algorithms, and efficiently makes use of potentially limited intermediate communication buffer space. This makes the new algorithm suitable for use in *regular* SMP clusters, in which the number of processors (and MPI processes) per SMP node is the same for all nodes. The algorithm can be implemented also for non-regular SMP clusters, and has been used for the implementation of both `MPI_Allgather` and `MPI_Allgatherv` collectives. However, for very irregular problem instances, dedicated irregular algorithms might give better performance.

The combined algorithm was developed assuming single-port communication of the SMP nodes. It is worth pointing out that both the logarithmic and the linear phase can easily be generalized to the case where the SMP nodes have $k > 1$ communication ports.

References

1. G. Almási, P. Heidelberger, C. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. D. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *19th ACM International Conference on Supercomputing (ICS 2005)*, pages 253–262, 2005.
2. G. D. Benson, C.-W. Chu, Q. Huang, and S. G. Caglar. A comparison of MPICH allgather algorithms on switched networks. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 335–343, 2003.

3. J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.
4. P. Fraigniaud and E. Lazard. Methods and problems of communication in usual networks. *Discrete Applied Mathematics*, 53(1–3):79–133, 1994.
5. S. M. Hedetniemi, T. Hedetniemi, and A. L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.
6. S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.
7. N. T. Karonis, B. R. Toonen, and I. T. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.
8. T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, volume 34 of *ACM Sigplan Notices*, pages 131–140, 1999.
9. D. W. Krumme, G. Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111–139, 1992.
10. H. Ritzdorf and J. L. Träff. Collective operations in NEC's high-performance MPI libraries. In *International Parallel and Distributed Processing Symposium (IPDPS 2006)*, page 100, 2006.
11. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.
12. R. Thakur, W. D. Gropp, and R. Rabenseifner. Improving the performance of collective operations in MPICH. *International Journal on High Performance Computing Applications*, 19:49–66, 2004.