A Pipelined Algorithm for Large, Irregular All-gather Problems*

Jesper Larsson Träff[1] Andreas Ripke[1] Christian Siebert[1]
Pavan Balaji[2] Rajeev Thakur[2]
William Gropp[3]
[1]NEC Laboratories Europe, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany

[2]Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439, USA

[3]Department of Computer Science
University of Illinois, Urbana, IL 61801, USA

Abstract

We describe and evaluate a new, pipelined algorithm for *large, irregular all-gather problems*. In the irregular all-gather problem each process in a set of processes contributes individual data of possibly different size, and all processes have to collect all data from all processes. The pipelined algorithm is useful for the implementation of the `MPI_Allgatherv` collective operation of MPI (the Message-Passing Interface) for large problems. By conception the new algorithm is well suited to implementation on clustered multiprocessors, like for instance SMP clusters. The new algorithm has been implemented within different MPI libraries. Benchmark results on NEC SX-8, Linux clusters with InfiniBand and Gigabit Ethernet, IBM Blue Gene/P, and SiCortex systems show huge performance gains in accordance with the expected behavior.

*This paper is a revised version of the conference presentation "A Simple, Pipelined Algorithm for Large, Irregular All-gather Problems" that appeared in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 15th European PVM/MPI Users' Group Meeting*, volume 5205 of *Lecture Notes in Computer Science*, pages 84–93, Springer, 2008. This work was supported in part 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 DE-AC02-06CH11357.

1 Introduction

The *all-gather problem* is a basic collective (or group) communication problem, in which *each* participant of a predefined group of processes wants to broadcast personal data to all other processes of the group. In the Message-Passing Interface (MPI) standard [8], the *de-facto* standard for parallel programming in the message-passing paradigm, this functionality is embodied in two, so-called collective communication operations [8, Chapter 5]. In the *regular* `MPI_Allgather` operation each process contributes the same amount of data, whereas in the *irregular* (or *vector*) `MPI_Allgatherv` operation the amount of data may vary among the processes. By the end of such an operation, all processes have gathered all contributed data in some prescribed order. For both MPI collectives, all participating processes know the sizes of the data to be broadcast by all other processes. Both operations are useful, symmetric (i.e. non-rooted) data gathering operations with many applications. The irregular variant is used for instance in linear algebra kernels for matrix multiplication and LU factorization [1].

The regular all-gather problem has been intensively studied (theoretically under the term *gossiping*, but is also known as *broadcast-to-all*, *all-to-all-broadcast*, and other names) [5, 6], and many algorithms have been proposed and/or implemented as part of MPI libraries for various systems and communication models [1, 2, 3, 7, 9, 10]. The more challenging, irregular all-gather problem has received much less attention, and MPI libraries typically use the same algorithm for both `MPI_Allgather` and `MPI_Allgatherv`. For irregular problems with considerable differences between the amount of data contributed by the processes, this can have huge performance drawbacks. For extreme cases, the resulting performance loss can amount to orders of magnitude (cf. Section 3).

In this note, we present an algorithm for large, irregular all-gather problems. The underlying idea is quite simple and can be viewed as an adaptation to the irregular problem of a ring-based algorithm for regular all-gather problems for single-ported, clustered multiprocessors. The resulting algorithm is a *pipelined* (or *blocked*), linear ring, similar to a linear pipeline as sometimes used for implementing broadcast and reduction operations for large problem sizes. By conception the new algorithm is well suited for implementation on clustered multiprocessors, like clusters of SMP nodes. The algorithm has been implemented for several MPI libraries, and evaluated on diverse systems, namely an NEC SX-8, two Linux clusters, IBM Blue Gene/P, and a SiCortex 5832. We demonstrate significant performance improvements over a standard `MPI_Allgatherv` algorithm, depending on the amount of irregularity in the benchmark scenarios.

2 Algorithm and Implementation(s)

In the following, p is the number of participating (MPI) processes, numbered consecutively from 0 to $p-1$. We let m_i denote the size of the data contributed by process i, and $m = \sum_{i=0}^{p-1} m_i$ the total amount of data that eventually has to be gathered by all processes. For large data, we assume that the time for transmitting a message of size m' is simply $O(m')$. For most of the following discussion, a detailed communication cost model is unnecessary.

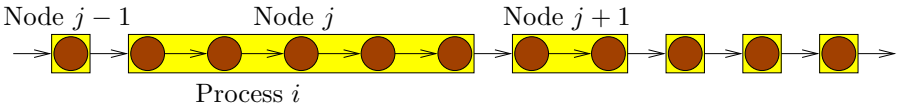

Figure 1: The linear ring algorithm on a cluster of SMP nodes with different number of MPI processes per node. The processes are (virtually) ranked such that one process at each node receives data from another node, and one process sends data to another node in each round.

2.1 Standard, linear ring Algorithm

A basic (folklore) algorithm for large, *regular* all-gather problems is the *linear ring*. All processes contribute data of size $m_i = m'$. The linear ring algorithm steps through $p-1$ communication rounds. In each round process i sends (starting with its own data) an already known block of data of size m' to process $(i+1) \bmod p$ and receives an unknown block of data from process $(i-1) \bmod p$. Since $p-1$ blocks are sent, and $p-1$ blocks are received by the processes in parallel, the completion time of the linear ring algorithm is $O((p-1)m') = O(m-m')$. The number of communication start-ups (latency) scales linearly with p. This is unproblematic for large m', but for small problems, an algorithm with a logarithmic number of start-ups is clearly preferable [1, 3, 10].

The linear ring algorithm is straightforward to implement. For systems with single-ported, bidirectional communication capabilities (where each process can at the same time send data to another process and receive data from a possibly different process) it uses the system communication bandwidth to full capacity, since each processes both sends and receives data in each round. For irregular all-gather problems, where the data sizes m_i can vary arbitrarily over the processes, the linear ring algorithm can perform poorly. The running time is determined by the largest amount of data $m' = \max_{i=0}^{p-1} m_i$, which has to be sent along the ring in each round, and is therefore $O((p-1)m')$. In particular, $(p-1)m'$ can be much larger, up to a factor of $(p-1)$, than the total amount of data m.

2.2 Pipelined (blocked) ring Algorithm

We first observe that the linear ring algorithm can also be used for the regular all-gather problem on clustered multiprocessors (like clusters of SMP nodes) with a single-ported communication network. In such a system a set of compute nodes each consisting of one or more processors is connected by an interconnection network such that on each node at most one processor can at any one instant be sending and at most one processor be receiving data from processors on other nodes.

The ring is organized such that exactly one process i per node has its predecessor $(i-1) \bmod p$ on another node, and exactly one process i' per node has its successor $(i'+1) \bmod p$ on another node. To accomplish this, a (virtual) reranking of the MPI processes might be necessary. The clustered, linear ring algorithm is now communication-bandwidth optimal, because in each round one process on each node receives a block of data and one process

sends a block of data. This holds also for the case where the number of MPI processes per cluster node is not identical. The idea is illustrated in Figure 1 which shows a situation with $1, 5, 2, 1, 1, 1, \ldots$ processes on the nodes.

In [11] it is observed that regular collective communication problems (like the all-gather problem) induce corresponding irregular problems over the set of nodes in a clustered system. In Figure 1 the *regular* all-gather problem, when viewed over the set of processes (each process contributes data of the same size), becomes an *irregular* all-gather problem when viewed over the set of nodes (each node contributes data of size equal to the sum of the data sizes contributed by the processes on the node). If the communication capabilities of processors and nodes in a cluster are similar (for instance, single ported), an algorithm for solving a regular problem on a clustered system (with possibly different number of processes per cluster node) can therefore be converted into an algorithm for solving its irregular counterpart over a set of processors. This is done by letting each processor simulate the actions of a whole node in the clustered algorithm. Since the linear ring algorithm solves the regular all-gather problem on a clustered system, the observation can be exploited to convert the clustered linear ring algorithm into an algorithm for the irregular all-gather problem.

We now consider the irregular all-gather problem over a set of processes. The data of process i of size m_i is associated with a virtual cluster node, and divided into $b_i = \max(1, \lceil m_i/B \rceil)$ blocks of size at most B. This corresponds to the number of virtual processes on the virtual node, and each block is associated with a virtual processor in the node. The total number of blocks is $b = \sum_{i=0}^{p-1} b_i$ (note that $b \geq p$). Each actual process with data size m_i simulates the role of a virtual cluster node with b_i virtual processors. In each round a new block of size at most B is received by each virtual process and a known block of size at most B is sent. The virtual processes simulated by actual process i of course do not have to actually send and receive blocks among themselves, therefore each actual process in each communication round sends and receives a block from two other actual processes. Each actual process terminates as soon as it has received all blocks from the other processes. By these observations the linear ring algorithm with regular blocks of size (at most) B solves the irregular all-gather problem in

$$\begin{aligned} r &= \max_{i=0}^{p-1}(b - b_i) \\ &= b - \min_{i=0}^{p-1} b_i \end{aligned} \tag{1}$$

instead of $p - 1$ communication rounds. Let in the following i' be a process with $b_{i'} = \min_{i=0}^{p-1} b_i$.

The resulting, *pipelined* (or *blocked*) *ring* algorithm is illustrated in Figure 2. Compared to the linear ring, the advantage of the pipelined ring algorithm is that (more) regular blocks are sent and received in each round, for a total time of $O(rB)$. A small value for B increases the number of start-ups, and a large value increases the possible round up error. Therefore a proper balancing must be applied to find an optimal value for the block size parameter. We note that for extremely irregular all-gather problems where only one process has all the data, the pipelined ring algorithm is equivalent to a linear broadcast pipeline. For regular

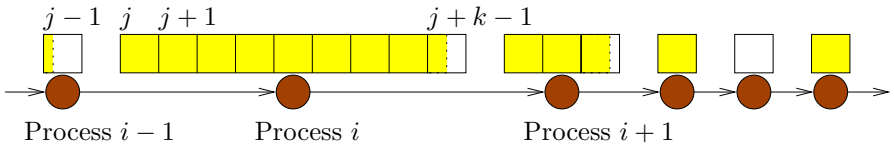

Figure 2: The clustered, linear ring algorithm viewed as a pipelined (blocked) algorithm for solving the irregular all-gather problem. For each process, the data m_i is divided into blocks of some maximum block size B (partially full blocks are partially colored). Process i starts sending block $j+k-1$ and receiving block $j-1$. After r rounds (see equation 1) all processes have gathered all the data.

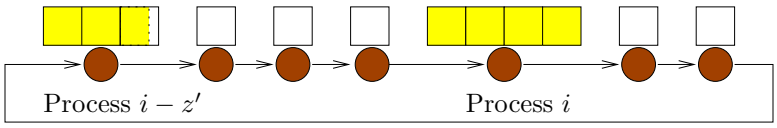

Figure 3: Reduction in number of communication rounds by placing processes with non-zero data equidistantly among the processes contributing no data. The actual number of rounds required in this example is 9, whereas r is 11, an improvement of 2 rounds.

problems where $m_i = m'$ for all i, the block size B can be set to m', in which case the algorithm becomes identical to the standard, linear ring. Thus, by choosing B properly, the pipelined ring algorithm should never perform worse than the linear ring algorithm.

2.3 Implementation improvements

As described the algorithm divides all contributed data into blocks of size B, which determines the number of blocks $b_i = \max(1, \lceil m_i/B \rceil)$ to be sent by each process. Another possibility would be to divide the m_i data into b_i roughly even sized blocks of size at most B. With the first possibility where each process may have a partially full block of size $m_i \bmod B$, we note that for such partially full blocks, only actual data have to be sent and received (see again Figure 2). In particular, the empty blocks that arise for processes with $m_i = 0$ are neither sent nor received. Let z denote the number of such processes. This latter observation can be used to reduce the number of communication rounds for the case where $0 < z < p-1$ (that is, at least two processes with $m_i > 0$ and at least one process with $m_i = 0$). Assume that process i in the ring is preceded by z' processes that contribute no data. In the first z' rounds of the algorithm, process i does not receive any actual blocks, and consequently does not have to send any of these blocks further on after round b_i when all non-zero blocks of process i have been sent. Effectively, if $b_i > z'$ the number of rounds have been reduced by $z'+1$; otherwise by $z'+1-b_i$ since in this case process i stands idle for $z'+1-b_i$ rounds before the next actual block is received and can be sent further on.

The maximal reduction in number of rounds is achieved by organizing the ring such

that $b_i > z'$ for all processes i as far as this is possible. In this case the number of rounds needed for an arbitrary process i with $m_i > 0$ to have delivered all its blocks to all other processes in the ring can be calculated as follows. Process, say 0, has to send b_0 blocks in a pipelined fashion. The first block is delivered to the last process after $p-1$ rounds (assuming that this is the block that is piped through the ring), and the remaining $b_0 - 1$ blocks require another $b_0 - 1$ rounds plus the number of rounds incurred by blocks at intermediate processes in the ring. For all intermediate processes a saving due to empty processes of $b_i - (z_i + 1)$ rounds is achieved, thus a total delay in the pipelining of the blocks of process 0 of $(b_1 - z_1 - 1) + (b_2 + z_2 - 1) + \ldots$ rounds is incurred (where $b_1, b_2, \ldots$ are the processes with $m_i > 0$). This gives

$$\begin{aligned} b_0 - 1 + (p-1) + (b_1 - z_1 - 1) + (b_2 + z_2 - 1) + \ldots &= \\ (\sum_{i=0}^{p-1} \lceil \frac{m_i}{B} \rceil) + (p-1) - (p-z) - (z - z_0) &= \\ (\sum_{i=0}^{p-1} \lceil \frac{m_i}{B} \rceil) - 1 + z_0 &= b - z - 1 + z_0 \end{aligned}$$

since $(\sum_{i=0}^{p-1} \lceil \frac{m_i}{B} \rceil) = b - z$.

An easy, reasonable solution is to place the $p - z$ processes with $m_i > 0$ equidistantly with $z' = \lfloor \frac{z}{p-z} \rfloor$ processes with $m_i = 0$ in between. Assuming that $b_i > z'$ for all processes with $m_i > 0$, the total number of rounds is therefore

$$(\sum_{i=0}^{p-1} \lceil \frac{m_i}{B} \rceil) - 1 + \lfloor \frac{z}{p-z} \rfloor$$

This improvement is illustrated in Figure 3.

As for the linear ring for the regular problem, the pipelined ring for the irregular problem can also be implemented to run with full bandwidth utilization on clusters of SMP nodes. As shown in Figure 1 the processes are reranked such one (actual) process per node send to a process on another node (call this the last process), and one (actual) process per node receives from a process on another node (call this the first process). Node internally processes are ordered such that the last processes have $m_i > 0$. With this the nodes start sending actual blocks immediately in the first round of the algorithm.

2.4 Determining an optimal block size

The number of blocks and the number of processes with $m_i = 0$ together determine the number of rounds and the number of start-up latencies of the algorithm, and the size of the blocks determines the time of each round. Imbalance is caused by partial blocks. Furthermore, the best possible block size depends on the concrete communication capabilities of the underlying system. We can therefore only roughly indicate how a best block size can be determined.

- For regular problems with all $m_i = m'$ we take $B = m'$. The algorithm will coincide with the standard, linear ring algorithm which performs optimally for large problems.
- If $z = p-1$ the irregular all-gather problem degenerates into a broadcast problem, and the algorithm into a linear pipeline. The block size should be chosen accordingly.
- Otherwise we try to minimize the time needed for $b-z-1+z'$ communication rounds with $z' = \lfloor \frac{z}{p-z} \rfloor$. Assuming that m_i/B needs rounding up for half of the $p-z$ processes with $m_i > 0$ we can simplify $\sum_{i=0}^{p-1} \lceil m_i/B \rceil$ to $m/B + \frac{p-z}{2}$ for a total number of rounds of $m/B + \frac{p+z}{2} - 1 + \lfloor \frac{z}{p-z} \rfloor$. With an appropriate cost model we can use this to estimate the best value of B.

  Assuming for instance linear communication costs, where sending and receiving messages of size m' takes time $\alpha + \beta m'$, the estimated total running time is

$$(b-z-1+z')(\alpha+\beta B) \quad = \quad (m/B + \frac{p+z}{2} - 1 + \lfloor \frac{z}{p-z} \rfloor)(\alpha+\beta B)$$

  Minimizing this term gives an (approximated) optimal block size of

$$B \quad = \quad \sqrt{m \frac{\alpha}{\beta} \frac{1}{\frac{p+z}{2} - 1 + \lfloor \frac{z}{p-z} \rfloor}} \tag{2}$$

3 Experimental Evaluation

The pipelined, irregular all-gather algorithm has been used to implement the `MPI_Allgatherv` collective within (or on top of) MPI implementations for different target systems. We have benchmarked these `MPI_Allgatherv` implementations with the following distributions of *contiguous data* over the p MPI processes. A base count c (which is varied over some interval) is used as seed for the following distributions:

1. **Regular:** all $m_i = c$ are identical, therefore $m = pc$.
2. **Broadcast:** $m_0 = c$, all other $m_i = 0$, therefore $m = c$.
3. **Spike:** similar to broadcast but all processes contribute some data, $m_0 = c/2$ and $m_i = c\frac{1}{2(p-1)}$, therefore $m = c$.
4. **Half full:** $m_{2\lfloor i/2 \rfloor} = 2c$, and $m_{2\lfloor i/2 \rfloor + 1} = 0$, therefore $m = pc$.
5. **Linearly decreasing:** $m_i = 2c\frac{(p-1-i)}{p-1}$, therefore $m = pc$.
6. **Geometric curve:** $m_{i-1+j} = c\frac{p}{i \log p}$ for $i = 1, 2, 4, \ldots$ and $j = \{0, \ldots, i-1\}$, therefore $m = pc$.

In distributions (2) and (3) the same total amount of data $m = c$ is gathered by all processes, so similar running times can be expected (comparable to the regular distribution with p times smaller data size). The case for distributions (1), (4), (5) and (6) is analogous, where the total amount of data is $m = pc$.

We compare our implementations of the new `MPI_Allgatherv` algorithm with implementations of the standard linear ring algorithm that is still used in many MPI libraries [9]. The reported running times are minimum times for the last process to finish over a (small) number of iterations [4]. For the pipelined algorithm, the implementations as reported here did not attempt to compute an optimal block size. Rather the block size was fixed in the algorithm or determined from the outside. We include experiments showing the effects of the choice of block size on the performance achieved with the new algorithm.

3.1 Results on an NEC SX-8 vector system

The pipelined ring has been implemented for MPI/SX for the NEC SX-series of parallel vector computers. It has been benchmarked with the distributions described above on 30 SX-8 nodes at HLRS in Stuttgart, with 1 and 8 MPI processes per node, respectively. Selected results are shown in Figure 4.

For the extreme broadcast distribution (2) the pipelined ring outperforms the standard linear ring by more than a factor of 10 on 30 SX-8 nodes. For 32 MBytes with a fixed block size B of 1 MByte an improvement of a factor $\frac{32\times29}{29+31} \approx 15$ would have been best possible. Significant improvements can also be observed for the other distributions. Note that the standard ring algorithm is twice as slow on the broadcast (2) as on the spike distribution (3), which is in accordance with the analysis, since $m_0 = c$ for broadcast and $m_0 = c/2$ for spike. The pipelined algorithm performs similarly on both. The performance of the standard ring and the pipelined ring are similar for the regular (1) and the half full (4) distributions. Running on a randomly permuted communicator instead of `MPI_COMM_WORLD` gives almost identical results. This is a desirable property of an algorithm for a symmetric (i.e. non-rooted) collective operation like `MPI_Allgatherv` [12].

3.2 Results on a Linux Cluster with InfiniBand

To show the effect of the block size B, the algorithm has also been integrated into NEC's MPI/PC version and evaluated on an Intel Xeon based SMP cluster with InfiniBand interconnect. The running time is compared to the standard, non-pipelined algorithm for $B = 32\text{KByte}, 64\text{KByte}, 128\text{KByte}, 512\text{KByte}, 1024\text{KByte}$. Results are shown in Figure 5. For the spike distribution (3) the pipelined algorithm is faster for all block sizes. However, the best block size depends not only on the size of the problem but also on the distribution of data over the processes. This can be seen in the case of the decreasing distribution (5) where a too small block size makes the pipelined algorithm perform worse than the standard ring. We also observed that even for regular distributions (1) blocking into smaller blocks than m_i (e.g. $B = 1024\text{KByte}$) can sometimes improve performance.

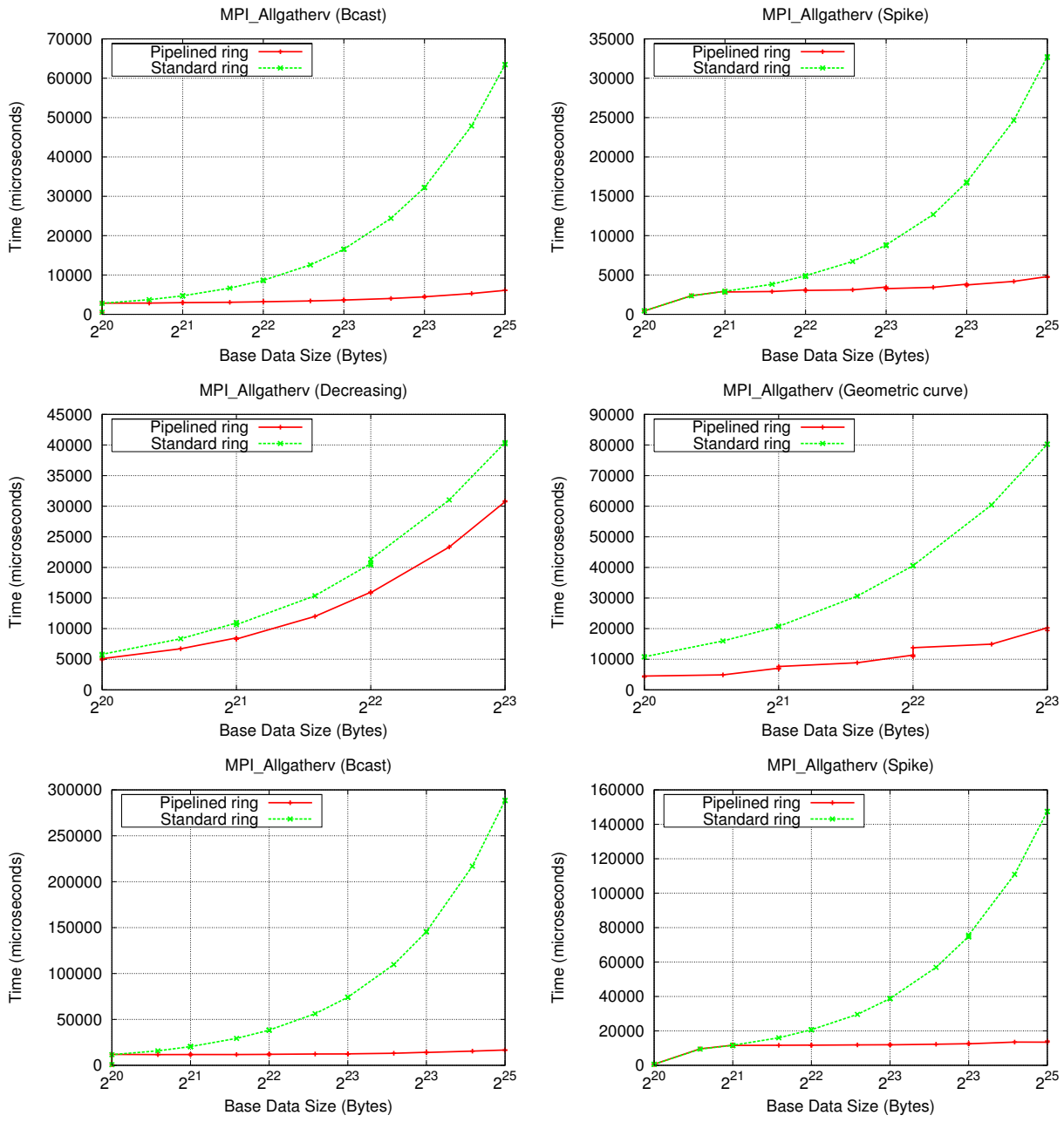

Figure 4: Results (left to right, top to bottom) for distributions (2), (3), (5) and (6) on an NEC SX-8 with 30 nodes and 1 MPI process per node, and distributions (2) and (3) with 8 MPI processes per node. A fixed block size $B = 1$MByte has been used. The base data size is the base count c multiplied by the size of an `MPI_INT`.

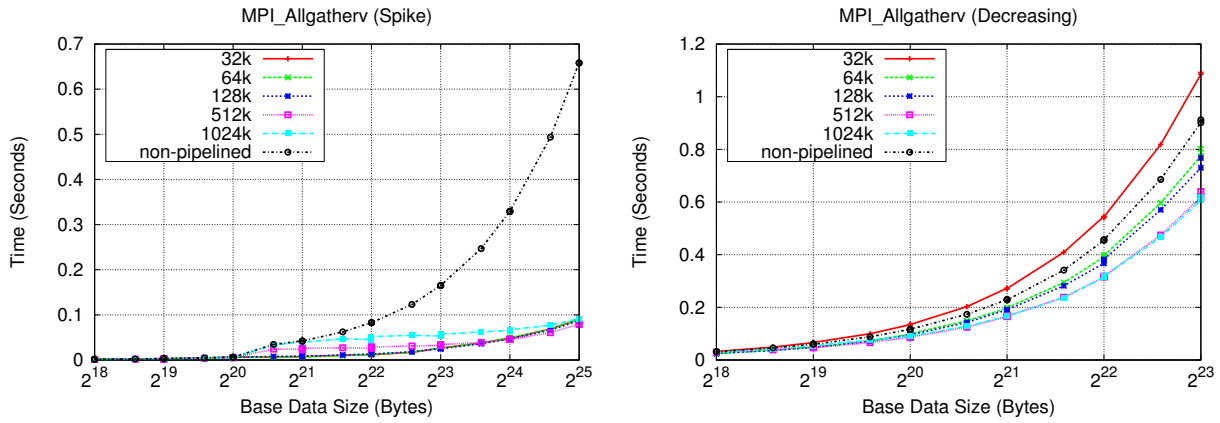

Figure 5: Results from a Linux Xeon/InfiniBand cluster with 16×2 processes with spike (left) and linearly decreasing (right) distributions, and block size $B = 32\text{KByte}, 64\text{KByte}, 128\text{KByte}, 512\text{KByte}, 1024\text{KByte}$ compared to the non-pipelined (standard ring) algorithm.

3.3 Results on a Linux Cluster with Gigabit Ethernet

We ran the benchmarks on a Linux cluster at Argonne National Laboratory with 24 nodes, each with two dual-core 2.8 GHz AMD Opteron CPUs (total of 4 cores per node or 96 cores in the system), and Gigabit Ethernet. We used MPICH2 1.0.7 as the MPI implementation. Selected results are shown in Figures 6 and 7. For small problem sizes, the pipelined algorithm performs only slightly better than the standard algorithm, but as problem size increases, the difference in performance becomes considerable. Figure 7(right) shows the distribution of communication and idle times for the two algorithms.

As expected, the standard linear ring algorithm suffers because many processes remain idle for a long time, whereas in the pipelined algorithm, communication is more balanced. To show this, we collected traces of the program execution and plotted them using the Jumpshot tool, as shown in Figure 8. The penalty due to idle time incurred by the standard algorithm is clearly visible as the lighter bars.

3.4 Results on SiCortex

Benchmarks were also performed on a SiCortex 5832 system at Argonne National Laboratory. This machine has 972 nodes, each with 6 cores, for a total of 5832 processors. The nodes are connected by a Kautz graph network. Ten of the processors (60 cores) of the system at Argonne are pre-assigned for system management tasks; our experiments utilized the remaining 5772 cores available.

While the system is shipped with a binary version of the vendor native MPI implementation (based on MPICH2), we did not have access to a working source code. Hence, we implemented both the original as well as the pipelined all-gather algorithms on top of MPI,

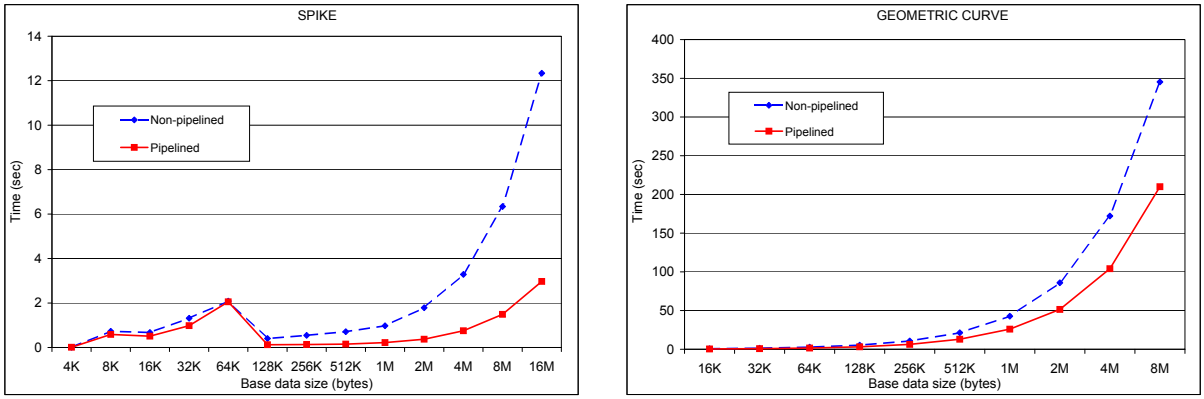

Figure 6: Results with 96 processes on Linux cluster: (left) Spike distribution (right) Geometric curve distribution. A fixed block size B = 32KByte was used. "Base data size" is the value of the parameter c determining the data size per process in the distributions.

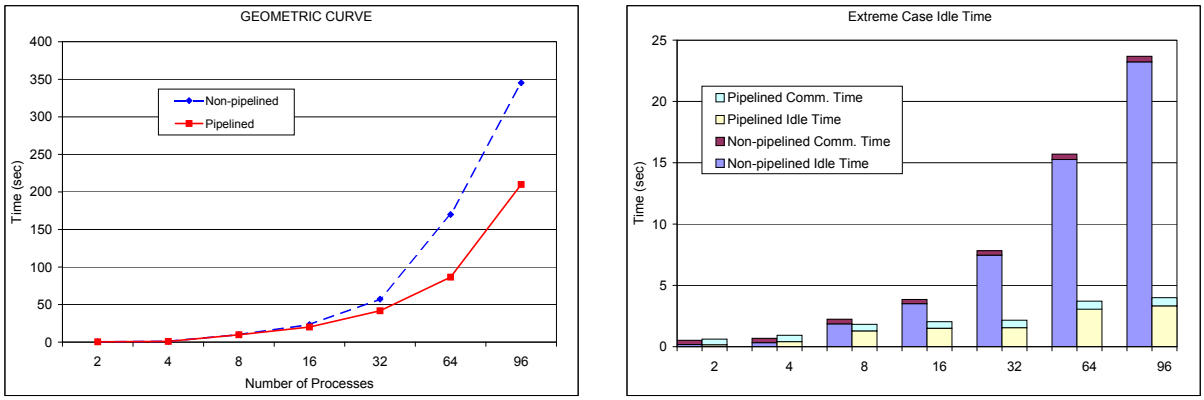

Figure 7: Linux cluster: (left) Geometric curve distribution with varying number of processes, (right) Communication versus idle time in the extreme case of broadcast distribution.

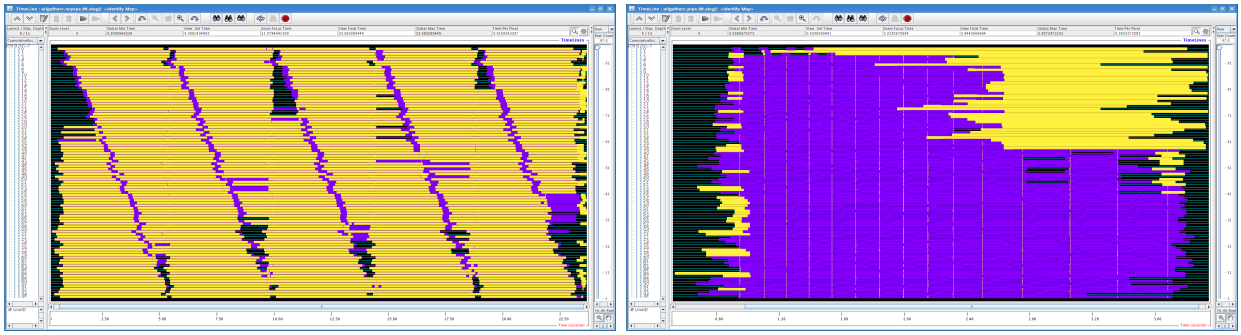

Figure 8: Jumpshot plot of program trace on Linux cluster for several iterations of allgatherv with broadcast distribution: (left) Non-pipelined algorithm, (right) Pipelined algorithm. Light shading is idle time, dark shading is communication time.

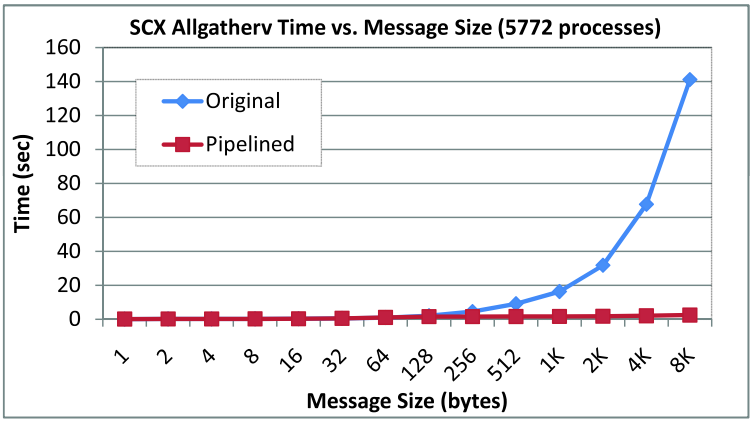

Figure 9: Results for the geometric curve distribution with 5772 processes on the SiCortex machine and a fixed block size of $B = 32$KByte. "Message size" is the value of the parameter c determining the data size per process in the distributions.

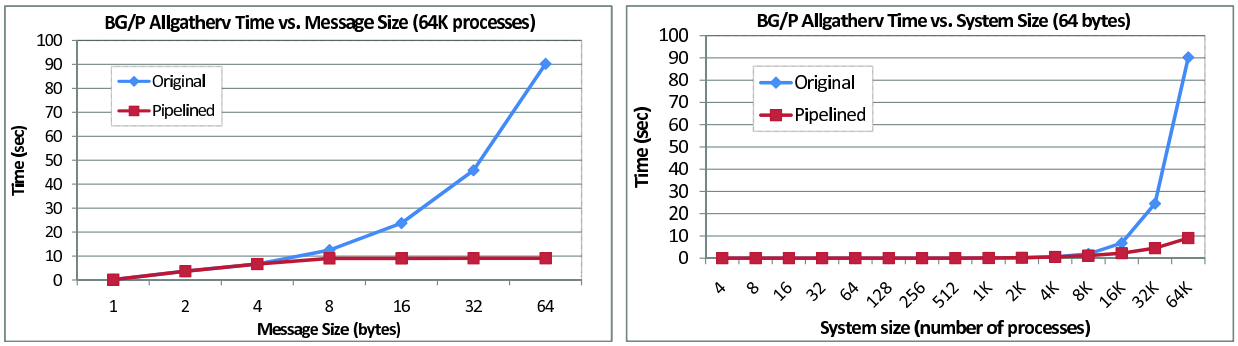

Figure 10: Results for the geometric curve distribution on Blue Gene/P with a fixed block size of $B = 32$KByte: (left) against message size with a fixed system size of 65536 processes, (right) against system size with a fixed message size of 64 bytes. "Message size" is the value of the parameter c determining the data size per process in the distributions.

instead of within the MPI stack. This adds a small amount of overhead due to additional function calls, but that is negligible for large problems. This was confirmed by comparing the performance of the original algorithm on top of MPI against the internal implementation of the vendor native MPI stack; the performance difference was insignificant (results not shown here).

Figure 9 shows the results for a test run with a geometric curve distribution on 5772 processors. The pipelined algorithm significantly outperforms the standard algorithm as the message size increases.

3.5 Results on IBM Blue Gene/P

Finally, we performed the tests on up to 16 racks of the IBM Blue Gene/P at Argonne National Laboratory (65536 cores). The native implementation of `MPI_Allgatherv` in the Blue Gene/P's MPI library uses a very fast hardware-supported algorithm for simple cases including contiguous data communication or pre-defined communicators (such as `MPI_COMM_WORLD`). However, as the communication patterns become more complex (e.g., derived datatypes used over split communicators that involve only a subset of processes), the native MPI implementation falls back to MPICH2's default collective implementation. As discussed in this paper, this default implementation lacks the proposed pipelining capability.

Figure 10 shows the performance comparison for one such case: `MPI_Allgatherv` using a derived datatype that uses a non-contiguous list of bytes to be communicated over a sub-communicator that involves all processes in `MPI_COMM_WORLD` except the last process. Clearly, the pipelined algorithm significantly outperforms the original algorithm even on this machine.

4 Concluding Remarks

We described a simple, pipelined ring algorithm for large, irregular all-gather problems. The algorithm was implemented within different MPI libraries and benchmarked on various systems, and in all cases showed considerable improvements over a commonly used linear ring algorithm for problems with significant irregularity in the individual message sizes. We indicated how to estimate a best possible block size as a function of the number of processes with no data contribution and the distribution of such processes. Analytic block size computation is however also dependent on the communication cost model, and will thus vary from system to system. We leave it as future work to experiment in this direction. On regular problem instances the pipelined algorithm performs similarly to the linear ring, which is bandwidth optimal for that case. Ring algorithms can likewise be implemented to be largely independent on process placement in an SMP system. This is an important property for users expecting (self-)consistent performance of their MPI library [12].

References

[1] P. Balaji, D. Buntinas, S. Balay, B. F. Smith, R. Thakur, and W. Gropp. Nonuniformly communicating noncontiguous data: A case study with PETSc and MPI. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–10, 2007.

[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] W. Gropp and E. Lusk. Reproducible measurements of MPI performance characteristics. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 11–18, 1999.

[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] D. W. Krumme, G. Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111–139, 1992.

[7] A. R. Mamidala, A. Vishnu, and D. K. Panda. Efficient shared memory and RDMA based design for mpi_allgather over InfiniBand. In *Recent Advances in Parallel Vir-*

tual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting, volume 4192 of *Lecture Notes in Computer Science*, pages 66–75, 2006.

[8] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 2.1*, September 4th 2008. `www.mpi-forum.org`.

[9] 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.

[10] J. L. Träff. Efficient allgather for regular SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 58–65. Springer-Verlag, 2006.

[11] J. L. Träff. Relationships between regular and irregular collective communication operations on clustered multiprocessors. *Parallel Processing Letters*, 19(1):85–96, 2009.

[12] J. L. Träff, W. Gropp, and R. Thakur. Self-consistent MPI performance requirements. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI Users' Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 36–45. Springer-Verlag, 2007.

Biographies

Jesper Larsson Träff received an M.Sc. in computer science in 1989, and, after two years at the industrial research center ECRC in Munich, a Ph.D. in 1995, both from the University of Copenhagen. After postdoc positions at the Max-Planck Institute for Computer Science and Technical University of Munich, he is since 1998 working at the NEC Laboratories Europe in Sankt Augustin, Germany. His interests are broadly in the design and efficient implementation of parallel programming interfaces.

Andreas Ripke is working as a Research Scientist at the NEC Laboratories Europe in Sankt Augustin, Germany. His focus is on the implementation of the Message Passing Interface (MPI) for PC clusters.

Christian Siebert holds a Diploma in Computer Science from the Chemnitz University of Technology. Since 2007 he is working at the NEC Laboratories Europe in Sankt Augustin, Germany. His main research interests are MPI and other parallel programming interfaces, especially in collective operations in relation to parallel applications.

Dr. Pavan Balaji holds a joint appointment as an Assistant Computer Scientist at the Argonne National Laboratory and as a research fellow of the Computation Institute at the University of Chicago. His research interests include high-speed interconnects, efficient protocol stacks, parallel programming models and middleware for communication and I/O, and job scheduling and resource management. He has nearly 60 publications in these areas and has delivered more than 60 talks and tutorials at various conferences and research institutes.

Rajeev Thakur is a Computer Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. He is also a Fellow in the Computation Institute at the University of Chicago and an Adjunct Associate Professor in the Department of Electrical Engineering and Computer Science at Northwestern University. He received a Ph.D. in Computer Engineering from Syracuse University. His research interests are in the area of high-performance computing in general and particularly in parallel programming models and message-passing and I/O libraries.

William Gropp is the Paul and Cynthia Saylor Professor in the Department of Computer Science and Deputy Directory for Research for the Institute of Advanced Computing Applications and Technologies at the University of Illinois in Urbana-Champaign. He received his Ph.D. in Computer Science from Stanford University in 1982 and worked at Yale University and Argonne National Laboratory. His research interests are in parallel computing, software for scientific computing, and numerical methods for partial differential equations.