Hierarchical Collectives in MPICH2

Hao Zhu[1], David Goodell[2], William Gropp[1], and Rajeev Thakur[2]

[1] Department of Computer Science,
University of Illinois, Urbana, IL, 61801, USA
[2] Mathematics and Computer Science Division,
Argonne National Laboratory, Argonne, IL 60439, USA

Abstract. Most parallel systems on which MPI is used are now hierarchical: some processors are much closer to others in terms of interconnect performance. One of the most common such examples are systems whose nodes are symmetric multiprocessors (including "multicore" processors). A number of papers have developed algorithms and implementations that exploit shared memory on such nodes to provide optimized collective operations, and these show significant performance benefits compared to implementations that do not exploit the hierarchical structure of the nodes. However, shared memory between processes is often a scarce resource. How necessary is it to use shared memory for collectives in MPI? How much of the performance benefit comes from tailoring the algorithm to the hierarchical topology of the system? In this paper, we describe an implementation based entirely on message-passing primitives but that exploits knowledge of the two-level hierarchy. We discuss both rootless collectives (such as Allreduce) and rooted collectives (such as Reduce), and develop a performance model. Our results show that for most collectives, exploiting shared memory directly will bring small additional benefit, and the places where shared memory is beneficial suggest design approaches that make best use of a pool of shared memory.

Key words: MPI, Collective Communication

1 Introduction

Most modern parallel computing systems have a hierarchical structure, often with several levels of hierarchy. For years, these systems have been composed of a (possibly hierarchical) high-speed network connecting many symmetric multiprocessor (SMP) nodes with a handful of processor cores on each node. While node counts will likely increase in many HPC systems, core counts per node are expected to increase dramatically [18] in the coming years. As this trend continues, the current approximation of "one process per node" that is currently used by many MPI collective algorithm implementations [15] will become increasingly inaccurate.

Optimized collectives that make use of shared memory have been developed for many of the collectives defined by the MPI Standard [7]. This approach has two disadvantages. The first is that sharing memory is difficult and error prone.

Shared memory does not behave portably between different architectures or even between different operating systems on the same architecture. Shared memory is often a scarce resource; it may also be a persistent resource as in the case of System V shared memory.

The second disadvantage to this approach is that shared-memory algorithms often require careful tuning and can be extremely sensitive to architectural specifics. Furthermore, algorithms that work for shared memory generally will not translate into solutions for other instances of non-uniformity such as a hierarchical network.

On platforms where shared memory is the fastest communications substrate for message passing, most MPI implementations already use shared memory for point-to-point communication [1]. In this paper we posit that building MPI collective algorithms based on point-to-point messages and taking into account the hierarchical structure of a system provides a general and portable way to achieve performance that is nearly optimal.

2 Related Work

Much work has been done in the areas of both hierarchical and shared memory collective algorithms. Hierarchical algorithms are discussed in the context of wide-area network (WAN) MPI implementations in [4–6]. The WAN/LAN collectives scenario is analogous to the network/shared memory scenario discussed in this paper. Of particular note, [5] provides a flexible parameterized LogP model for analyzing collective communication in systems with more than one level of network hierarchy.

Sanders and Träff present hierarchical algorithms for `MPI_Alltoall` [10, 17] and `MPI_Scan` [11] on clusters of SMP nodes. Other than their effort, most hierarchical work has centered around algorithms for `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Barrier`, and `MPI_Allgather`.

Graham and Shipman recently investigated techniques for performing intranode, shared-memory collectives in [2]. This includes improved shared-memory algorithms for `MPI_Bcast`, `MPI_Reduce`, and `MPI_Allreduce`. However, an important conclusion from that work is that shared memory collectives remain fickle and difficult to implement efficiently because performance is highly dependent on characteristics of a particular architecture's memory subsystem. Non-uniform memory architectures, cache sizes and hierarchy, together with process placement greatly influence the performance of shared-memory collective algorithms.

Implementations that combine both hierarchical and shared memory collectives are discussed in many works [12, 14, 16]. None of these papers clearly separate the mostly orthogonal issues of knowledge of hierarchy from the availability of shared memory as a communication substrate. This makes it difficult to understand the impact of hierarchical knowledge alone on now-common clusters of SMP nodes.

Wu et al. provide an excellent discussion [19] of SMP-aware collective algorithm construction in terms of shared-memory collectives, shared-memory point-to-point communication, and regular network communication. Their algorithmic framework also overlaps inter-node communication with intra-node communication. This approach generally pays the largest dividends in the case of medium-sized messages, where the message is large enough to amortize the additional overhead introduced by the non-blocking communication, yet is small enough that it is not dominated by the inter-node message transfer time. We do not examine this optimization in the work presented here.

3 Algorithms and Implementation

In the rest of the paper we identify the MPI collectives as belonging to one of two groups: one for routines which have a distinguished root (such as Bcast or Reduce), which we call "rooted;" and one for those routines without a distinguished root (such as Allreduce), which we call "rootless."

Our hierarchical algorithms for broadcast, reduce, allreduce and barrier have similar structures:

1. If necessary, perform local node operation and collect data in the master process of each node, such as in broadcast and barrier. Here master process means the representative of the node, which is the only process participating inter-node communication. We select the master processes as the lowest ranked process on each node in the communicator passed to the collective. This information is determined at communicator creation time and cached in the communicator object for later use by the collective routines.
2. Perform inter-node operation among all the master processes. Send/collect data from/to the root in rooted collectives, or send/collect data in all the master processes.
3. If necessary, perform local node operation such as broadcast data received in step 2 from master process to other processes in the node.

We can easily implement hierarchical algorithms for many collectives, because non-hierarchical collectives can be used in for both intra-node and inter-node phases of communication. A few operations, such as scan, are more difficult; we discuss scan Section 3.5.

3.1 Broadcast

We have two implementations for hierarchical broadcast: one is a non-pipelined version which has the same algorithm as described above in the overview. The other is a pipelined version that divides a long message into segments with the same size and then broadcast those segments in a pipelined manner.

In the pipelined implementation, we use a binomial-tree algorithm in the local node broadcast and a binary-tree algorithm in the inter-node broadcast. Assume the segment size in the pipelined implementation is s, which is chosen as 5KB by default. When the message size is less than s, the only difference between pipelined and non-pipelined is the tree structure of inter-node broadcast. When the message size is larger than s, we pipeline the broadcast of different segments and therefore can overlap their communication.

Theoretical Analysis of the Pipelined Implementation When the message size is less than s, there is only one segment. The cost is composed of that of the inter-node broadcast and local node broadcast. Suppose the node number is p and every node contains c cores. The inter-node broadcast cost is between $log(p) * t_{inter}$ and $2log(p) * t_{inter}$, and the actual value depends on the percentage of the ping-ping latency in the one-way ping-pong latency for the network. If the fraction is α, the cost is $(1 + \alpha)log(p) * t_{inter}$. The local node broadcast cost is $log(c) * t_{local}$. Here, t_{inter} and t_{local} are the inter-node and local node one time transmission cost of the message.

When the message size is larger than s, there is more than one segment. Using more than one segment allows different segments to overlap each other in inter-node communication. If n_{seg} is the number of segments and t_0 is the time to broadcast one segment, the total cost is not $n_{seg} * t_0$ but $n_{seg} * t_1$. Here t_1 is the part of t_0 that cannot be overlapped by the pipeline, such as the end of the whole broadcast. This total cost can be further simplified to $\beta * s * n_{seg}$, which equals $\beta * l_{msg}$. Here, l_{msg} is the length of message and β is inter-node transmission rate.

When the message size is larger than s, the local node broadcast is similar to the inter-node broadcast. The cost is approximately $\beta_{local} * l_{msg}$, where β_{local} is the local node transmission rate. By comparing the local node cost and inter-node cost, we see that inter-node transmission cost becomes dominant since β is typically ten times or more of β_{local}. Therefore, using shared memory provides only a small benefit because it only reduces local node collective cost. Similar results hold for reduce, scan, and allreduce because our hierarchical algorithms for those collectives have similar properties. Table 1 sumarizes our measured results for a wide range of collective operations.

3.2 Reduce

Our hierarchical reduce first performs a local node reduce among the master processes of each node, then reduces the temporary result among all the master processes. We have both pipelined and non-pipelined implementations. In the pipelined implementation, we choose different algorithms for the local node reduce based on the message size, which is the same as non-hierarchical reduce. In the inter-node reduce step, we use a binary tree similar to the pipelined smp broadcast.

3.3 Allreduce

Allreduce and Barrier are rootless collectives. Their hierarchical algorithms are the same as described in the overview. Allreduce has three steps:

1. Perform local node reduce to collect the partial result in the master processes of each node.
2. Allreduce on the partial result among all master processes to get final result.
3. Broadcast the final result within the local node.

3.4 Barrier

The hierarchical barrier has three steps:

1. Local node barrier.
2. Inter-node barrier across master processes of all nodes.
3. Release the local node processes with a 1-byte broadcast. The reason we are broadcasting 1-byte message instead of 0-byte is that, as an optimization, our broadcast doesn't perform actual send/recv when message size is 0, and the overhead of sending this extra byte is low.

3.5 Scan

Scan is a rootless collective, whose hierarchical algorithm is quite different from broadcast. Our algorithm requires that all the processes in the same node have consecutive ranks. The algorithm has these four steps:

1. All processes participate in an inclusive-scan on their local node, which yields a partial result on every process.
2. In step 1, the process with minimum rank (which we call the master process) on each node already has the partial result from the process with the maximum rank on the local node because we are exchanging data using a binomial tree algorithm.
3. All master processes participate in an exclusive-scan. At this point all master processes have all the data needed to compute their result.
4. All master processes broadcast the data collected in step 3 within the local node. All processes compute their final result.

4 Performance Experiments

All performance results presented in this section were gathered on a 16-node, quad-core AMD Opteron cluster at Argonne National Laboratory. A modified version of MPICH2 [8] was used. Each collective was measured at a variety of data sizes and process counts using the SKaMPI [9] MPI benchmarking software. For rootless collectives, the time on the plot represents the value of the "result time" field reported by SKaMPI; that is, the total time for the collective across all processes.

For rooted collectives, a separate measurement was taken for each root, and the minimum, median, and maximum "result time" are plotted with the minimum and maximum as plot whiskers. The difference between the minimum and

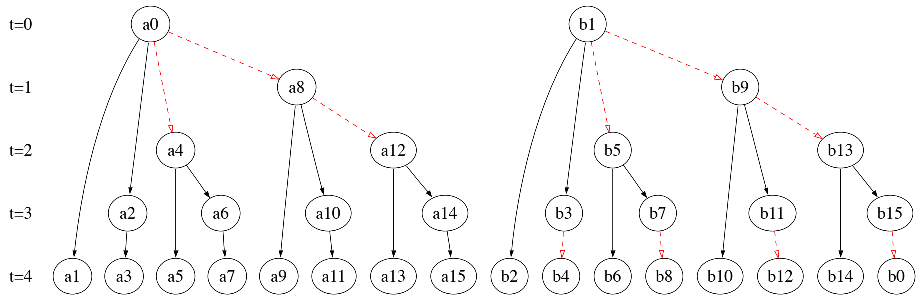

Fig. 1. Two schedules for a 16-process binomial broadcast algorithm using root=0 (a) and root=1 (b). Numbers in the node label indicate the process rank in the communicator. The solid edges indicate faster intra-node communication and the dashed edges indicate slower inter-node communication.

the maximum serves as an indicator of the performance variation with different root selections for a given algorithm. This is a much more rigorous method of performance testing for rooted collectives because, for certain systems and certain roots (especially the commonly chosen root 0), a topology-ignorant algorithm may result in an optimal or nearly-optimal communication schedule entirely by accident. Figure 1 is an example of one such scenario. Two communication schedules are shown for the same algorithm but using different roots. Schedule (b) will require more time to complete than schedule (a) because there are three inter-node messages in the longest path (b1, b9, b13, b15, b0) instead of the maximum of two inter-node messages in schedule (a).

One could claim that the nearly optimal schedule chosen when the root of the broadcast is 0 covers the most common case (broadcast from rank 0). However this “happy accident” will only occur for the binomial algorithm when the number of processes on a node is a power of two. Unfortunately, non-powers of two are not uncommon and are becoming more common as we look to the future. For example, Intel has introduced a six-core Nehalem processor [3]. Similarly, the SiCortex machine has six-cores per node [13]. It is also possible to arrive at this situation by creating a communicator that contains a non-power-of-two number of processes from some or all nodes.

4.1 Broadcast

Figures 2, 3, and 4 show the performance results for broadcast, including the comparison between non-smpcoll broadcast and smpcoll broadcast. The theoretical performance is calculated using $\alpha = 1/3$. From these experiments, we see that an advantage of smp bcast is that the cost is independent of the choice of root.

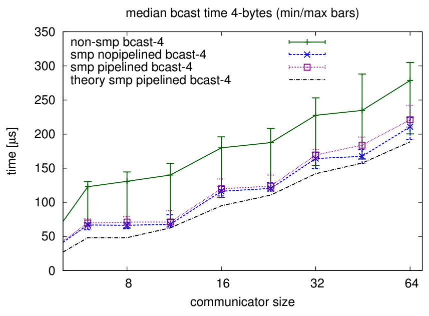

Fig. 2. 4B `MPI_Bcast` times

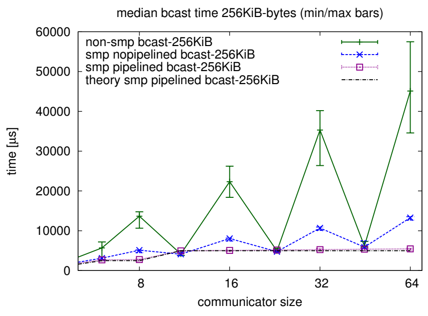

Fig. 3. 256KiB `MPI_Bcast` times

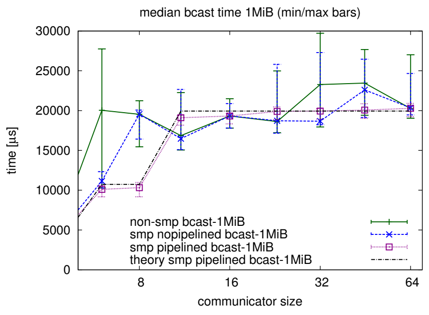

Fig. 4. 1MiB `MPI_Bcast` times

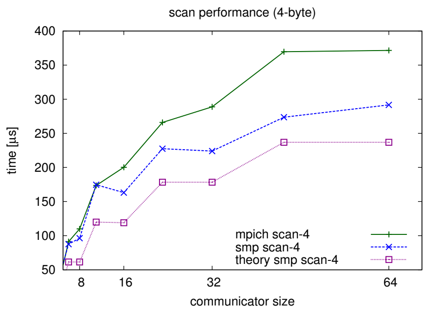

Fig. 5. 4B `MPI_Scan` times

4.2 Scan

Figures 5, 6, and 7 show the performance results for our new scan algorithm compared to the existing non-hierarchical scan algorithm in MPICH2. Hierarchical scan has a better performance in all the cases. When message size is 256KiB and 1MiB, we achieve up to 20 times improvement as shown in figure 6 and 7.

4.3 Reduce, Allreduce and Barrier

Figures 8, 9, and 10 show the performance results for reduce. The theoretical model is calculated in the same way as with broadcast. Our pipelined hierarchical reduce and nopipelined hierarchical reduce have similar good performance when message size is 4 bytes, which is the same as broadcast. In figure 9 and 10, pipelined hierarchical reduce shows an overall better performance against the other algorithms.

Figures 11 to 14 show the performance results for allreduce and barrier. As shown in Figures 11 and 12, hierarchical allreduce has better performance in most cases. It has an overall better performance when message size is 1MiB. Hierarchical barrier provides an approximately 100% improvement in all cases.

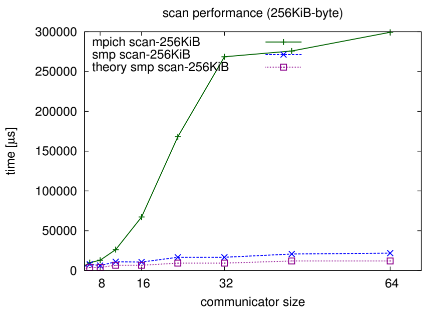

Fig. 6. 256KiB `MPI_Scan` times

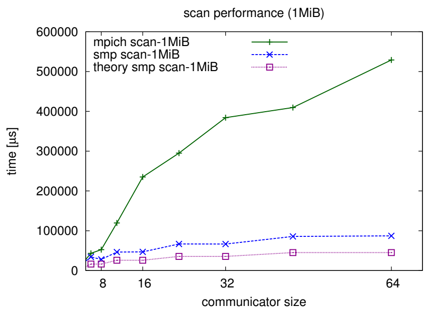

Fig. 7. 1MiB `MPI_Scan` times

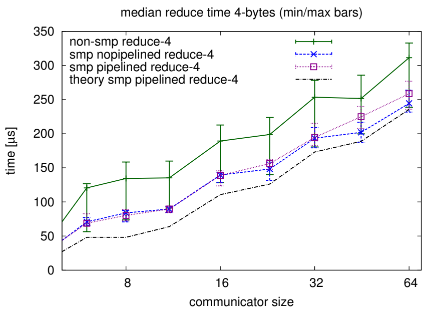

Fig. 8. 4B `MPI_Reduce` times

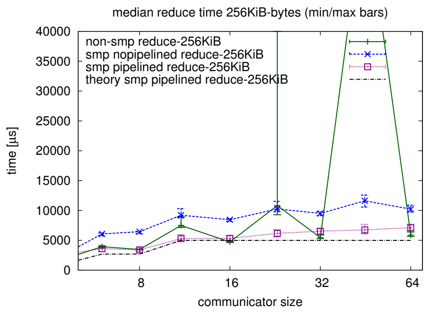

Fig. 9. 256KiB `MPI_Reduce` times

4.4 Are Shared-Memory Optimizations Worthwhile?

For all of the collective algorithms discussed here, the inter-node communication time dominates the overall communication time. By measuring the time spent in the inter-node phase of the collective and assuming an ideal (zero time) intra-node phase we can use Amdahl's Law to obtain a lower-bound for total collective time when using a shared-memory collective algorithm. Table 1 lists values for selected collective algorithms with 64 processes on 16 nodes with various data sizes. It is clear from this analysis that while shared-memory collective algorithms may improve performance somewhat, the vast majority of the performance is won via algorithmic awareness of the system's hierarchy.

Even for nodes with more processes, we believe a more detailed performance model will show that, at least for long messages, the performance benefit of using shared memory will be small.

5 Conclusions and Future Work

Our results show that significant performance benefits can be realized by exploiting the hierarchical nature of the interconnect topology. For many of the collective routines, particularly those that do not make many copies of the same data

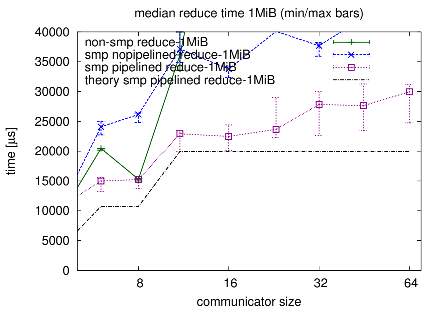

Fig. 10. 1MiB `MPI_Reduce` times

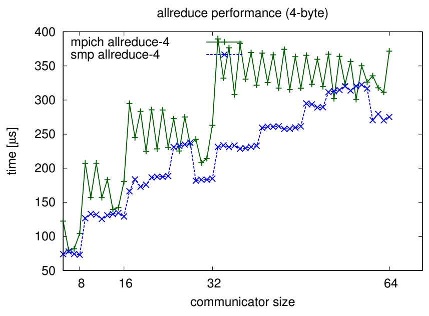

Fig. 11. 4B `MPI_Allreduce` times

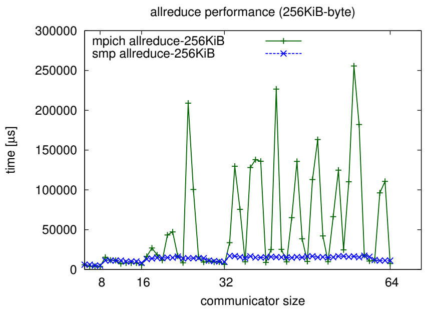

Fig. 12. 256KiB `MPI_Allreduce` times

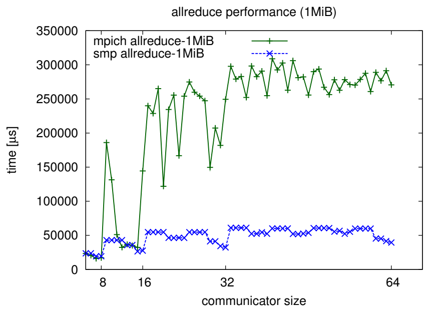

Fig. 13. 1MiB `MPI_Allreduce` times

(which Bcast does), our performance models suggest that for the long-message case, directly using shared memory will not significantly improve performance beyond what exploiting the topology achieves.

For short messages, where overheads dominate, using shared memory may offer a relatively greater advantage. This suggests an implementation strategy that uses small amounts of shared memory for short data transfers within an SMP node for collectives, switching to message-passing for longer data transfers.

Acknowledgments

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357 and Award DE-FG02-08ER25835.

References

1. Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. *Parallel Computing*, 33(9):634–644, September 2007.

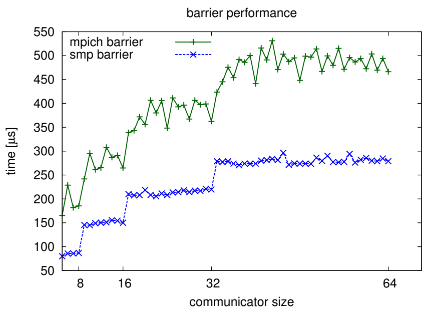

Fig. 14. `MPI_Barrier` times

Table 1. Intra-node time versus total time for collective operations, and potential speedup if the intra-node time dropped to 0 (ideal shared-memory collective algorithm). All values are for 64 processes on 16 nodes.

Algorithm	Data Size	Total (μs)	Intra-node (μs)	Ideal Speedup
Bcast	4B	37	8	1.276
Bcast	256KiB	9313	406	1.046
Reduce	4B	412	8	1.020
Reduce	256KiB	12052	648	1.057
Reduce	1MiB	71907	3044	1.044
Scan	4B	257	14	1.058
Scan	256KiB	17120	2082	1.138
Scan	1MiB	68994	10104	1.172
Allreduce	4B	271	14	1.054
Allreduce	256KiB	34695	1350	1.040
Allreduce	1MiB	71821	6981	1.108
Barrier	N/A	252	14	1.059

2. Richard L. Graham and Galen Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140, 2008.
3. Intel Corporation. *Intel Xeon Processor 7400 Series Datasheet*, October 2008.
4. Nicholas T. Karonis, Bronis R. de Supinski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Fourteenth International Parallel and Distributed Processing Symposium*, pages 377–384, May 2000.
5. Thilo Kielmann, Henri E. Bal, Sergei Gorlatch, Kees Verstoep, and Rutger F. H. Hofman. Network performance-aware collective communication for clustered wide-area systems. *Parallel Computing*, 27(11):1431 – 1456, 2001.
6. Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *SIGPLAN Not.*, 34(8):131–140, 1999.
7. Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–

299, 1998.

8. MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`.

9. Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Müller. SKaMPI: A detailed, accurate MPI benchmark. In Vassil N. Alexandrov and Jack Dongarra, editors, *PVM/MPI*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59. Springer, 1998.

10. Peter Sanders and Jesper Larsson Träff. The hierarchical factor algorithm for all-to-all communication. In *Euro-Par 2002 Parallel Processing*, pages 799–803, 2002.

11. Peter Sanders and Jesper Larsson Träff. Parallel prefix (scan) algorithms for MPI. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings*, volume 4192 of *Lecture Notes in Computer Science*, pages 49–57. Springer, 2006.

12. Steve Sistare, Rolf vandeVaart, and Eugene Loh. Optimization of MPI collectives on clusters of large-scale SMP's. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 23, New York, NY, USA, 1999. ACM.

13. Lawrence C. Stewart, David Gingold, Jud Leonard, and Peter Watkins. RDMA in the SiCortex cluster systems. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2007.

14. Hong Tang and Tao Yang. Optimizing threaded MPI execution on SMP clusters. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 381–392, New York, NY, USA, 2001. ACM.

15. Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *International Journal of High-Performance Computing Applications*, 19(1):49–66, Spring 2005.

16. V. Tipparaju, J. Nieplocha, and D. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, April 2003.

17. Jesper Larsson Träff. Improved MPI all-to-all communication on a giganet SMP cluster. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 392–400, 2002.

18. S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w TeraFLOPS processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, 2008.

19. Meng-Shiou Wu, Ricky A. Kendall, and Kyle Wright. Optimizing collective communications on SMP clusters. *Parallel Processing, International Conference on*, pages 399–407, 2005.