Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters

Vinod Tipparaju Jarek Nieplocha
Computational Sciences & Mathematics
Pacific Northwest National Laboratory
{vinod.tipparaju, jarek.nieplocha}@pnl.gov

Dhabaleswar Panda
Department of Computer &Information Sciences
Ohio State University
panda@cis.ohio-state.edu

Abstract

This paper describes a novel methodology for implementing a common set of collective communication operations on clusters based on symmetric multiprocessor (SMP) nodes. Called Shared-Remote-Memory collectives, or SRM, our approach replaces the point-to-point message passing, traditionally used in implementation of collective message-passing operations, with a combination of shared and remote memory access (RMA) protocols that are used to implement semantics of the collective operations directly. Appropriate embedding of the communication graphs in a cluster maximizes the use of shared memory and reduces network communication. Substantial performance improvements are achieved over the highly optimized commercial IBM implementation and the open-source MPICH implementation of MPI across a wide range of message sizes on the IBM SP. For example, depending on the message size and number of processors, SRM implementation of broadcast, reduce, and barrier outperforms IBM MPI_Bcast by 27-84%, MPI_Reduce by 24- 79%, and MPI_Barrier by 73% on 256 processors, respectively.

1. Introduction

Collective communication operations such as barrier, broadcast, reduce, allreduce [18] are important for many scientific applications based on the MPI model, e.g., [19]. For example, they are used for synchronizing processes, broadcasting data, updating distributed vectors, calculating stopping criteria in iterative algorithms, and for many other purposes. This paper describes a novel methodology for implementing this common set of collective communication operations on clusters of symmetric multiprocessor (SMP) nodes equipped with a network that supports remote memory access (RMA) operations. Called Shared-Remote-Memory collectives, or SRM, this approach replaces the point-to-point message passing (MPI send/receive), traditionally used in implementation of message-passing collectives, with a direct implementation of these operations based on a combination of shared and remote memory. These two protocols are carefully coupled to minimize data movements, streamline flow control, and eliminate internal overheads associated with implementations based on higher-level protocols, i.e., of MPI send/receive. The current paper extends on our previous work [17] on the implementation of barrier based on shared and remote memory access protocols for the SMP clusters with VIA networks. We take the next step extending this methodology to address challenges presented by the other common collective operations.

The paper shows that by eliminating point-to-point message passing as the communication protocol underlying the usual implementations of collective operations and building the semantics of the collective operations at a much lower level, significant performance gains can be achieved. This is accomplished by implementing collectives directly on top of the fastest communication method for two of the hardware domains in a cluster -- shared memory on the SMP node and RMA across the network. The performance is improved due to a combination of several other factors as well. First, the number of data movement operations in SRM is less than that in the point-to-point message-passing implementations of collectives. Second, additional savings are achieved by avoiding overheads associated with tag matching and dealing with early message arrivals and buffer management complexities associated with implementations of general-purpose point-to-point message-passing protocols. Third, appropriate embedding of the multi-method communication graphs is critical to exploit locality information and maximize the benefit of the fastest communication protocol on an SMP cluster — shared memory. Finally, pipelined data transfers across the two communication domains can be employed and tuned to maximize performance without running into scalability issues due to extensive buffer space consumption.

Indeed, the proposed approach achieves substantial performance improvements over the collective operations in the highly optimized commercial IBM implementation of MPI over the entire tested range of message sizes (up to 8 MB) on the IBM SP in all tested processor configurations. For example, depending on the message size and number of processors, our implementation of broadcast, reduce, and barrier outperforms the IBM MPI_Bcast by 27% to 84%, MPI_Reduce by 24% to 79%, MPI_Allreduce by 30%-73%, and MPI_Barrier by 73% on 256 processors, respectively. In addition, SRM outperforms MPICH, an open source implementation of MPI, by similar or better margins.

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

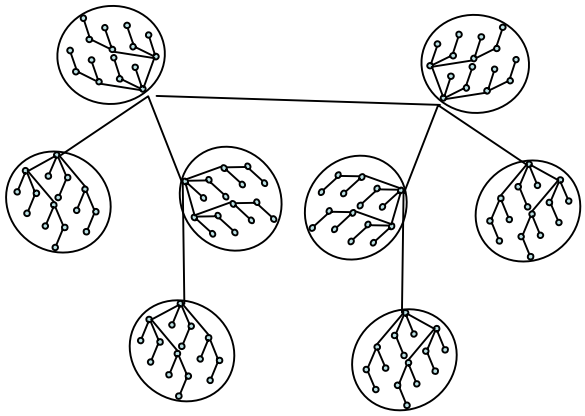

Figure 1: Embedding of the 128-processor binomial tree in an 8-node 16-way SMP cluster

The SRM approach is most appropriate for clusters with "fat" SMP nodes such as those used in the current generation IBM SP, clusters built around the HP Superdome or Sun Fire servers. The current trend of increased size of SMP nodes in the high-end commercial systems is exemplified by the evolution of processing nodes in the IBM SP, from uniprocessor in 1993 to two- four- eight-, 16-, and 32-processor node ("Regatta") systems today. With improved scalability in the Linux kernel and the Intel-based processor chipsets (especially in the IA-64 line), we also expect larger SMP nodes to appear in commodity Linux clusters. The SRM collective operations, in addition to shared memory, rely on the remote memory access and, in particular, the put operation. This capability is offered by supercomputer vendors in their low-level communication interfaces (e.g., LAPI on the IBM SP, RDMA on the Hitachi SR-8000, MPlib on the Fujitsu VPP) and is supported by all the popular high-performance networks like Myrinet, Giganet/VIA, Quadrics, SCI, and InfiniBand networks. For this study, we used LAPI, the lowest-level interface available on the IBM SP. LAPI offers RMA interfaces in addition to the first commercial implementation of Active Messages [20].

This paper is organized as follows. First, we provide technical details of the SRM implementation for SMP and network domains in a cluster, then present results of experiments conducted on the IBM SP using SRM and two implementations of MPI, IBM's and Argonne's MPICH. Next, we compare our approach to previous work in this area and conclude with suggestions for future work.

2. Technical Approach

The SRM implementation of collective operations relies on the performance and flexibility advantages of shared and remote memory protocols and the appropriate embedding of the broadcast/reduce trees to match the SMP cluster topology so that the full benefit of these protocols can be realized. Specifically, the primary goal is to maximize the usage of the fastest protocol, shared memory.

2.1 Embedding Collective Communication Trees in SMP Clusters

The previous implementations of collective operations on clusters relied on binary, Fibonacci (also known as λ-trees) [5], or binomial trees. We implemented and experimented with the three tree types and found binomial trees (distance power-of-two [6]) perform the best, for inter-node communication, in our target environment (IBM SP). These trees also are used in the MPICH implementation of collective operations, such as reduce and broadcast.

The fastest communication protocol available on SMP clusters is shared memory. However, shared memory is applicable only to the intra-node communication. Our implementation attempts to maximize the amount of processing that can be done using shared memory. Maximization is accomplished by the appropriate embedding of the broadcast and reduce trees into a cluster (see Figure 1). For example, the binomial tree for reduce operation is built by assembling binomial subtrees, each embedded in a different SMP node. The height (h) of a binomial tree for P processor is defined by

$$h(P) = \lfloor \log(P) \rfloor \qquad (1)$$

We observe that if the number of tasks p on each of the n SMP nodes is the same, then the embedding of a binomial tree into an SMP cluster does not add more steps to the execution of the reduce operation (does not increase the tree height). Evidently, this is true because:

$$\lfloor \log(P) \rfloor \geq \lfloor \log(n) \rfloor + \lfloor \log(p) \rfloor .$$

To minimize the impact of the system daemons running on each node, some applications on the IBM SP leave out one processor and use only 15 of the 16 processors per node. For that case, too, our embedding is optimal. By using the SMP-oriented embedding of binomial trees, we can effectively decouple and optimize the broadcast/reduce operations for the intra-node and inter-node sub-domains. In the following subsections, we describe the algorithms used for these sub-domains and then how they are combined.

2.2 Collective Operations on the SMP Node

Shared Memory Reduce

The reduce operation is used often in message-passing applications. It combines data stored on each processor to make the final result available on one specified process. Examples of operators available through MPI_Reduce include sum, min, or max. Figure 2 explains the implementation and demonstrates some of the benefits of shared memory in the reduce operation. SRM reduce within an SMP node involves a memory copy for processes that are at lowest level in a binomial tree. This operation is required to make contributions of these

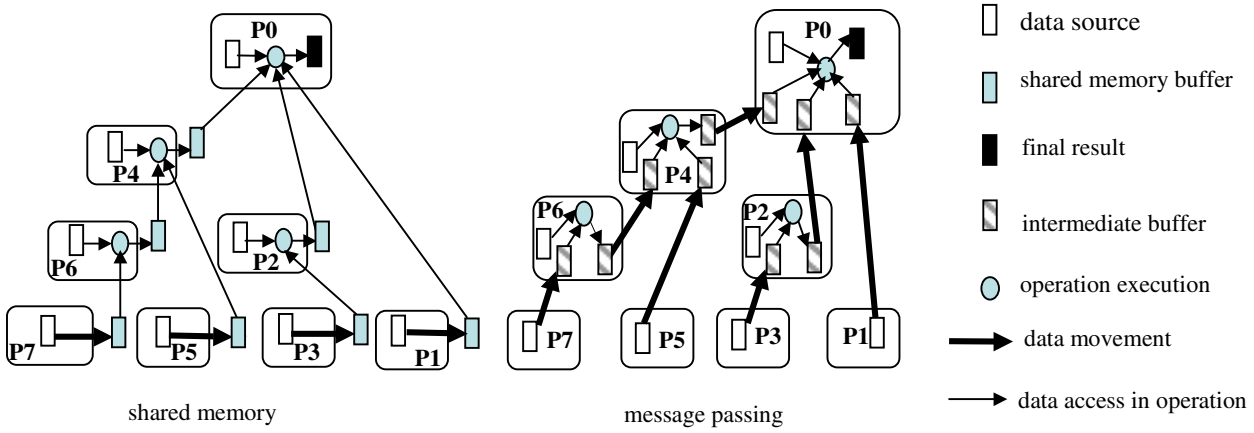

Figure 2: Reduce operation using shared memory and point-to-point message passing on eight processors

processes available to processes that execute the operation. For eight processes, there are four memory copies. The remainder of the tree simply involves execution of the operator by the CPU and is free of any additional data movements. For the same eight-process tree, the message-passing implementation requires seven data movement operations (message passing between sender and receiver buffers). Depending on the MPI implementation, these seven operations might internally involve 7 or even 14 memory copies. Note that even the collective communication operations based on the very efficient message passing implementations on shared memory (like the one described in [1,12,14]) will have at least 7 memory copies in the example described in Figure 2. Because the implementation based on the message passing requires data movement at every level of the tree, the shared memory approach is even more competitive as the tree (process count) grows.

Shared Memory Broadcast

We implemented binomial, binary and Fibonacci broadcast trees using shared memory buffers and flags. The flags synchronize access to the shared memory buffers between parent and child tasks in the tree. Our tree-based algorithms divide data into chunks and use two sets of buffers on each non-parent task in a tree. The two sets are needed to implement a two-stage pipeline that overlaps memory copies between the root and its leaves at each node of the tree. Pipelining also was used to overlap memory copy operations in and out of shared memory area. Surprisingly, experiments showed that the most successful approach is even more straightforward. It relies on the SMP hardware to manage simultaneous read and write operations by the multiple processes.

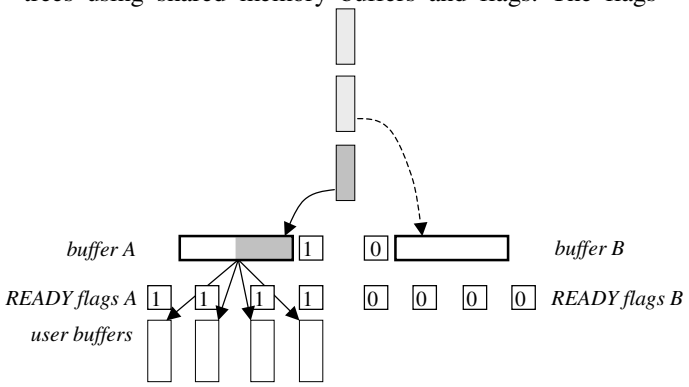

Figure 3: Broadcast using two shared memory buffers on a 4-way SMP node

The algorithm uses one set of two shared memory buffers (*A* and *B*) per node and two sets of shared memory flags (*READY A*, *B*), with each flag associated with a single process (see Figure 3). In addition, each buffer is protected by a corresponding flag located in shared memory. Root process in a broadcast acquires a shared memory buffer (*A* or *B*) copies data to the buffer and then sets *READY* flags for the other processes to indicate that the buffer is full. The other processes copy data from that buffer to their destination user buffer. When the copy operation is complete, each process clears its shared memory flag. Two buffers are used to allow the root process to copy the next chunk of a message to the second buffer while the other processes access the data in the first buffer, i.e., facilitate pipelining. For small message sizes, pipelining is not used. However, consecutive broadcast operations alternate between the buffers to improve concurrency. The algorithm supports the arbitrary root without any extra copies. Despite the contention in simultaneous read access to the shared memory buffer, this algorithm has achieved a much better performance than the tree-based algorithms.

Shared Memory Barrier

An SMP barrier algorithm is implemented using a flat tree. For moderately sized SMP nodes such as in the IBM SP, we found this approach faster than tree-based algorithms

and sufficiently scalable. The algorithm is very simple and requires only one flag variable per process. That flag is located in shared memory, and we ensure that each flag is located on a different cache line. The flag is set by the corresponding process to indicate its arrival at a barrier. Each process waits until its flag is reset, which indicates that all the processes on the SMP node were synchronized. One process on an SMP node is selected as a master. The master process waits for all processes to check in by setting their flags, and then it resets the value of flags for all the other processes.

Shared Memory Allreduce

The allreduce operation is normally implemented using reduce followed by broadcast. Although a combination of the two could be implemented as a single operation in shared memory, we did not do it due to its limited usefulness. In nontrivial (more than one node) clustered environments, the SMP reduce is followed by the internode reduce operation. Then the internode broadcast is followed by the SMP broadcast.

2.3 Inter-Node Protocols Using LAPI

LAPI offers RMA capabilities such as put, get, atomic read-modify-write, and active messages operations [20]. It is the lowest-level protocol offered by IBM on the SP. LAPI is a complementary and alternative protocol to MPI (IBM MPI is not relying on LAPI despite successful research experiences [4]). Performance of LAPI RMA operations is similar to that of MPI send-receive. LAPI has several advantages over MPI point-to-point operations for implementing collective communications. First, LAPI eliminates dependence on internal MPI protocols (e.g., Eager, Rendezvous) designed and tuned for a general-purpose point-to-point communication rather than collective operations. Second, it provides full control and ability to monitor progress in the actual data movement thanks to its origin, target and completion counter interfaces. A value of the counter is incremented by LAPI dispatcher when a corresponding phase of the communication completes, and a process can probe or block waiting for a counter to reach a certain value [20]. Finally, LAPI decouples synchronization from data transfer present in the message-passing operations, thus increasing the opportunities for overlapping shared memory operations on the SMP node with the inter-node communication. SRM relies primarily on the LAPI put operation to implement communication trees between the SMP nodes. On each node, only one selected process ("master") communicates across the network.

Management of LAPI Interrupts

Special care is required to manage interrupts generated by LAPI when data arrives before the destination task makes a LAPI library call (e.g., *LAPI_Waitcntr*). We attempt to minimize the number of interrupts, especially for small messages, but cannot completely avoid them. The interrupt mode of data reception is needed to overlap intra-node processing with network communication. Usually, interrupts are disabled in SRM when entering a collective operation for small message size and enabled when the operation is completed. For larger messages the relative cost of interrupt compared to the time it takes to transfer a message is less significant. It is important for LAPI to make progress as the put operation would not be able to complete without implicit cooperation of the destination task (e.g., polling in another LAPI call) *if* interrupts are disabled while the calling process is engaged in the SMP communication through shared memory.

Buffer Space Management in Network Communication

One of the issues in implementation of collective operations on top of point-to-point message passing is the implicit reliance on internal data transfer protocols in MPI (e.g., Eager and Rendezvous) and the corresponding tradeoffs between performance and memory consumption. On each task, the Eager mode usually requires P-1 buffers of size sufficient to accommodate the largest message sent in that mode. Larger messages are sent in the Rendezvous mode, which involves an extra short control message that notifies the sender about the posted receive buffer. The IBM MPI switches between Eager and Rendezvous protocols at different message sizes, depending on the number of tasks, to reduce the overall consumption of memory for larger task counts. It means that for a larger number of tasks, messages that normally should be sent using the faster Eager mode protocol; end up being sent using the slower Rendezvous protocol. By replacing point-to-point message passing with RMA, we are able to explicitly control buffer space consumption and tailor buffer sizes to performance characteristics of the collective operations. Unlike the Eager mode of MPI that requires P-1 buffers on each task, SRM on each SMP node needs to maintain only as many buffers as the degree of the master process in the internode binomial tree. Thus, the optimal buffer size can be used without running into resource scalability problems that motivated IBM MPI to vary the switch point between Eager and Rendezvous protocols as a function of the number of tasks.

2.4 Integration of Shared Memory and RMA Protocols in SRM

Shared memory buffers are used as targets of the LAPI put operations on each node to implement reduce, allreduce, barrier, and broadcast for short to medium-sized messages. Network and shared memory protocols are tightly integrated to ensure (whenever it is appropriate) that the

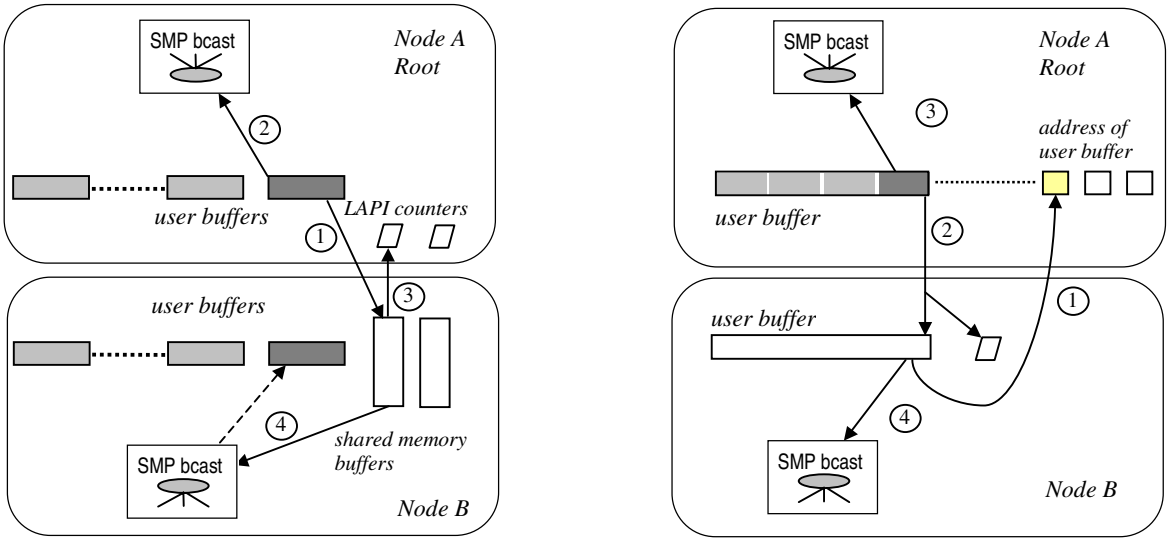

Figure 4: Integrated broadcast for small (left) and large (right) messages

data moved by LAPI is directly available to all the tasks running on that node without the need for copying the data.

Another issue to consider is the thread and CPU management. The implementation of LAPI uses two additional threads created implicitly at the startup time for each single-threaded user task. For example, on a 16-way node, 48 threads are used (96 if the application uses MPI and LAPI). To ensure efficient operation of LAPI, the SMP protocols that control access to buffers by spinning on the flag variables located in shared memory (see Section 2.2) had to be modified to yield the CPU (the current time slice) after a certain number of unsuccessful spins. This provides CPU cycles to the LAPI threads and improves the overall efficiency of the integrated protocols.

Broadcast

Two protocols are used, one for small messages and one for large messages (see Figure 4). The switching point is 64 KB. For small messages, two shared memory buffers are used. In addition, two LAPI counters are used to communicate the state of the two buffers on a leaf node to the parent. The parent alternates between the two buffers and sends the data after verifying that the appropriate buffer is free. If it is not, the process blocks in LAPI_Waitcntr call for the corresponding counter. The reason for using LAPI counter rather than spin on an integer variable is to avoid an interrupt when a message arrives and pass control to the LAPI dispatcher that polls the network. In Step 1, the process issues a nonblocking put call to initiate data transfer to the shared memory buffer on a leaf node, and returns immediately. In Step 2, SMP broadcast is performed on node A. On leaf node B, the received data is sent down the tree, and then SMP broadcast is performed. The SMP broadcast recognizing that the data is in shared memory avoids unnecessary data copies. Upon completion of this operation, a zero-byte LAPI nonblocking put is sent to the parent node to increment the counter corresponding to the current buffer (Step 3). As a further refinement, messages larger than 8 KB and smaller than 32 KB are split into 4KB chunks and sent in a pipelined fashion using the two buffers.

The SRM broadcast operation for larger messages does not rely on intermediate buffers whatsoever (Figure 4, right side). The operation involves four stages: 1) initialization, in which each leaf in an inter-node tree sends an address of the user buffer to its parent; 2) data movement across the network to the user buffer followed by SMP broadcasts on root 3) and 4) leaf nodes. To facilitate efficient pipelining in the SMP broadcast, two buffers are used as shown in Figure 3. We alternate between them to overlap memory copies within the SMP node with the internode communication. This approach enables the inter-node broadcast to proceed at its own natural rate and not be slowed by the availability of the intermediate buffer space. Because the message size is not small, the cost of that memory copy is at least partially hidden by pipelining in the SMP broadcast (Figure 3).

Reduce

The SRM reduce operation relies on pipelining to overlap memory copy, network communication, and computations within each SMP node. The intra- and inter-node protocols are tightly integrated to maximize the degree of overlapping. The reduce operation uses a binomial tree within each node and between the master task on the SMP nodes. The combined algorithm uses sets of two buffers and pipelining to overlap data movement in intra and internode communication.

Figure 5: Four-stage pipeline in Allreduce operation for large messages

Allreduce

In principle, the allreduce algorithm can be represented as the reduce operation followed by broadcast. However, for messages up to 16 KB, we use an integrated pairwise exchange based on recursive doubling [15] between the nodes, and reduce followed by broadcast within each node. For larger messages, we are combining reduce and broadcast in a manner that allows pipelining over the entire message range (see Figure 5).

Barrier

In the SRM barrier algorithm, the master task first waits until all other tasks on the node check in at the barrier. It then participates in the inter-node barrier algorithm (similar to pairwise exchange with recursive doubling [15]) involving one master on each node. Finally, it resets the value of all flags to notify the other tasks on the node that the global barrier operation is complete [17].

3. Experimental Results

The numerical experiments were carried out on up to 256 processors of the IBM SP equipped with 16-way SMP nodes and the high-performance "Colony" switch. For performance comparison, we timed the equivalent collective operations available in two different MPI implementations: 1) IBM's implementation of MPI and 2) ANL's MPICH implementation on top of MPL (native message passing interface for the original SP-2). MPL and MPI are implemented on top of a lower-level messaging layer called MPCI (Message Passing Client Interface). In that comparison, MPI (MPCI) was configured to use shared memory. Within the SMP node, the primary difference between SRM and MPI was that in MPI, shared memory was used to implement point-to-point message passing topped by collective operations, whereas SRM used shared memory to implement collective operations directly. The experimental results presented in Figures 6 through 12 correspond to the average execution time for 1000 calls of a given operation in SRM or MPI using the 16 tasks per node configuration. For reduce and allreduce operations, the sum operator, and double data type were tested. The number of elements was varied from one element to two million. Similarly, in the broadcast operation, message size varied from 8 bytes to 8MB.

On the left-hand side, in Figures 6 through 8, absolute performance numbers for the SRM broadcast, reduce, and allreduce are given as functions of the data size for 16, 32, 64, 128, and 256 processors (one curve for each fixed processor count) on a log-log scale. On the right-hand side, the SRM execution times are compared IBM-MPI and MPICH counterpart operations for messages up to 64KB on a log-linear scale.

The performance comparison between SRM and MPI implementations for the entire broad range of tested message sizes is shown in Figures 9 through 11 on the log-log scale for broadcast, reduce, and allreduce operations. The graphs represent the ratio of the SRM execution time, T_{SRM}, relative to the execution time of the same operation in MPI, T_{MPI}, i.e., the quantity: $T_{SRM} / T_{MPI} \cdot 100\%$ for the IBM MPI (left) and MPICH (right). The numbers less than 100% indicate that SRM is faster than MPI, which is the case for all our test runs in Figures 9 through 11. For example, the value of 20% indicates SRM is five times faster than MPI. The figures provide additional insight regarding the performance advantages of SRM.

The rate of improvement of SRM performance over MPI varies primarily as a function of the message size and processor count. This is due to a combination of several factors, including how protocols are switched internally in the MPI implementation and in SRM, and different buffering, pipelining and protocol/method coupling schemes. For example, the MPI implementation switches between the Eager and Rendezvous modes for different message size, depending on the number of tasks, to conserve the overall buffer consumption on each node. The buffer sizes and pipelining scheme in SRM do not depend on the number of processors. However, the number of leaves in the binomial tree for the internode communication does. In addition, some amount of variability visible in the graphs is attributed to the system daemons running on each node of the IBM SP.

For very small and large messages, the performance differences are easier to analyze than for medium-sized messages since the protocols are not switched. We can assume that both implementations of MPI use the Eager mode for shortest messages and Rendezvous for largest messages for all the processor configurations in Figures 9 through 11. Due to the embedding scheme for the binomial trees in SRM (Figure 1), the performance advantage of shared memory is maximized. It has a more profound effect when a larger fraction of the processors can communicate through shared memory. Therefore, the performance advantage of SRM over MPI is somewhat lower for the largest processor counts due to the increased height of the tree and corresponding increased amount of the inter-node communication. However, the higher efficiency from the one-sided nature of the RMA protocols helps SRM to remain competitive to the message-passing implementation for inter-node communication as well.

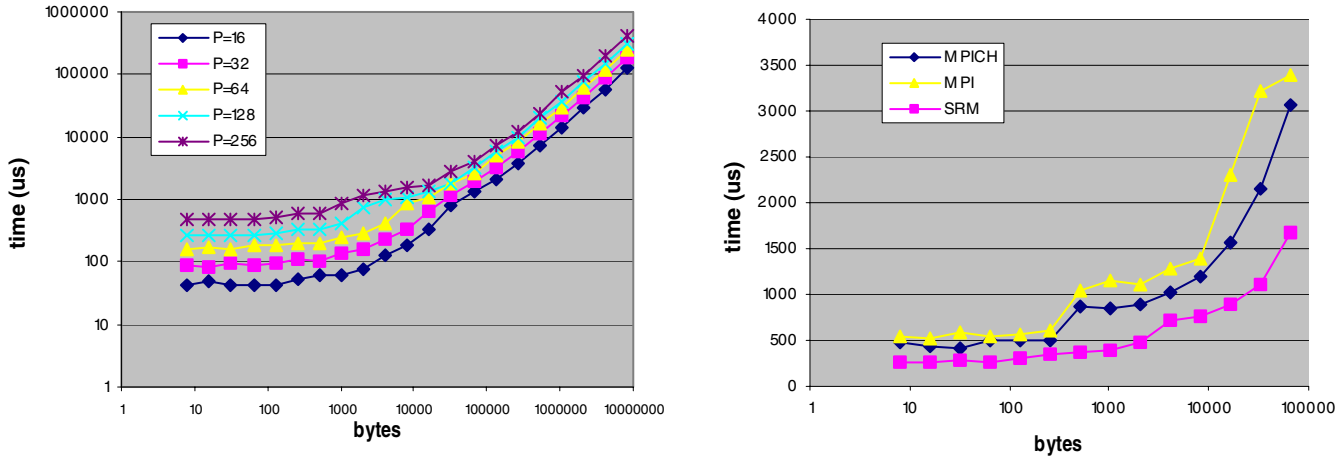

Figure 6: Performance of SRM broadcast: on left - performance on log-log scale for 8byte-8MB range on 16-256 CPUs, on right- in comparison to IBM and MPICH MPI_Bcast on log-linear scale for 8byte-64KB sub-ranges on 256 CPUs

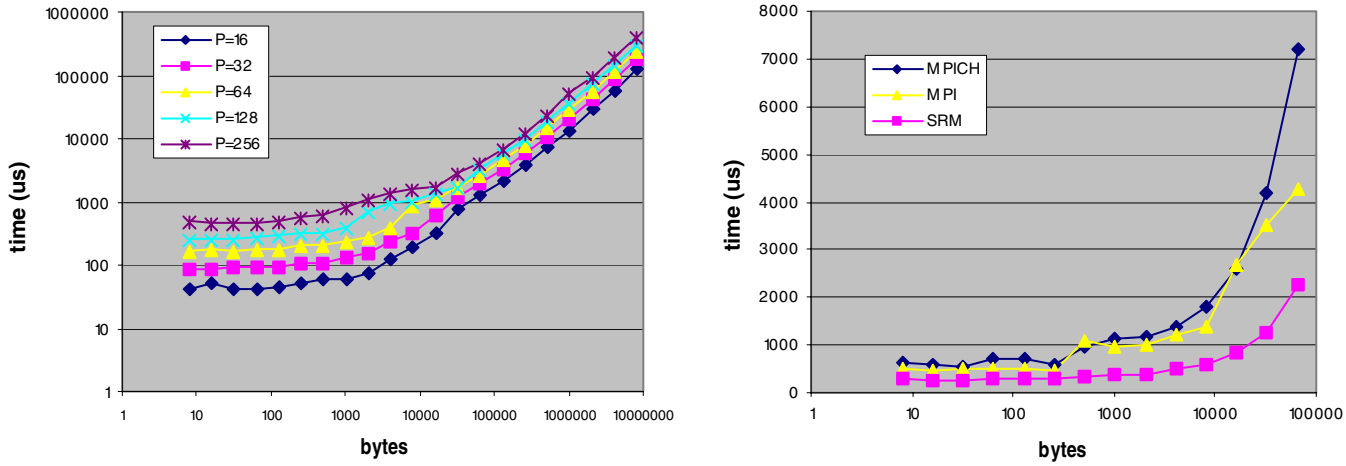

Figure 7: Performance of SRM reduce: left - absolute performance on log-log scale for 8byte-8MB range on 16-256 CPUs, on right- in comparison to MPI and MPICH MPI_Reduce on log-linear scale for 8byte-64KB sub-range on 256 CPUs.

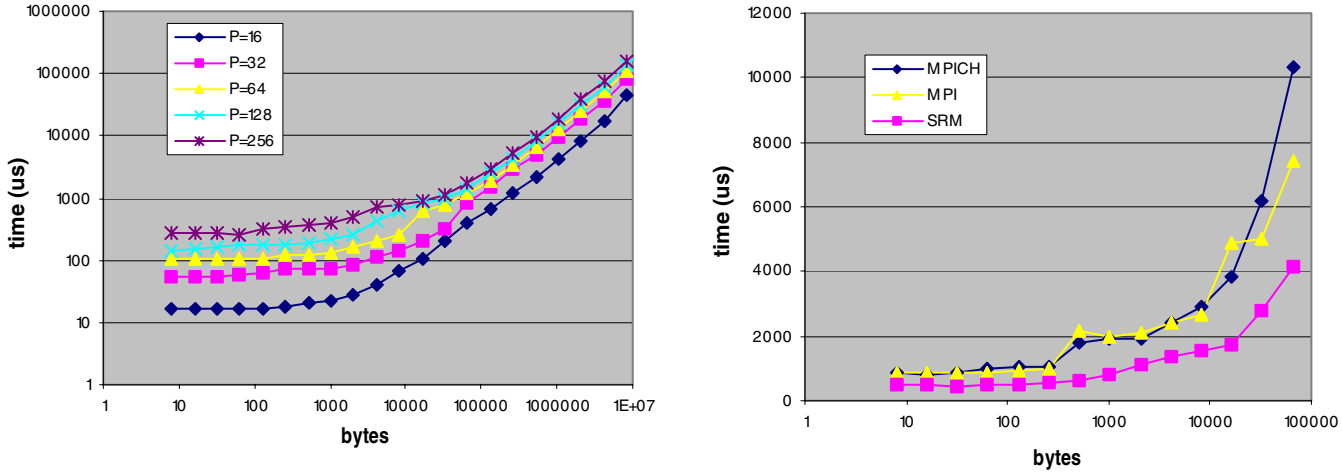

Figure 8: Performance of SRM allreduce. Left: absolute performance on log-log scale for 8byte-8MB range on 16-256 CPUs. Right: in comparison to MPI and MPICH MPI_Allreduce on log-linear scale for 8byte-64KB sub-range on 256 CPUs

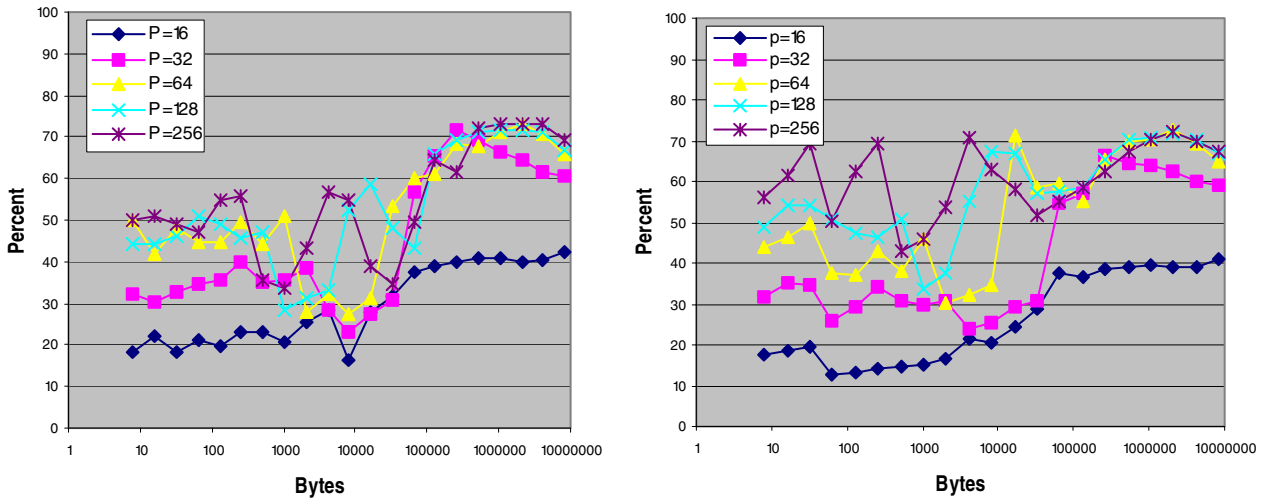

Figure 9: SRM broadcast time as a fraction of the execution time in IBM MPI (left) and MPICH (right) MPI_Bcast (the lower the better)

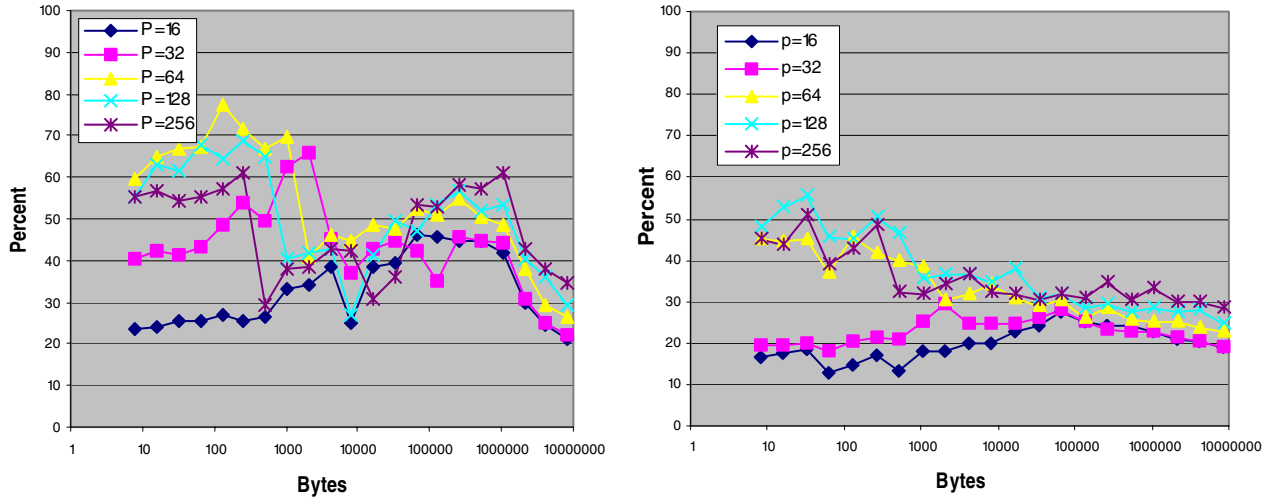

Figure 10: SRM reduce time as a fraction of the execution time in IBM MPI (left) and MPICH (right) MPI_Reduce (the lower the better)

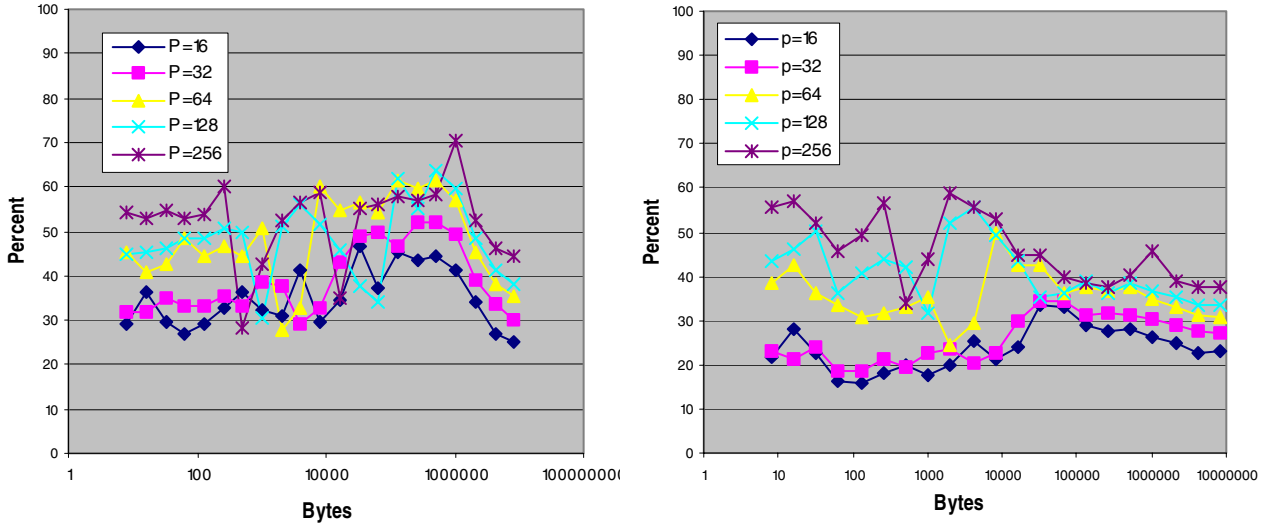

Figure 11: SRM allreduce time as a fraction of the execution time in IBM MPI (left) and MPICH (right) MPI_Allreduce (the lower the better)

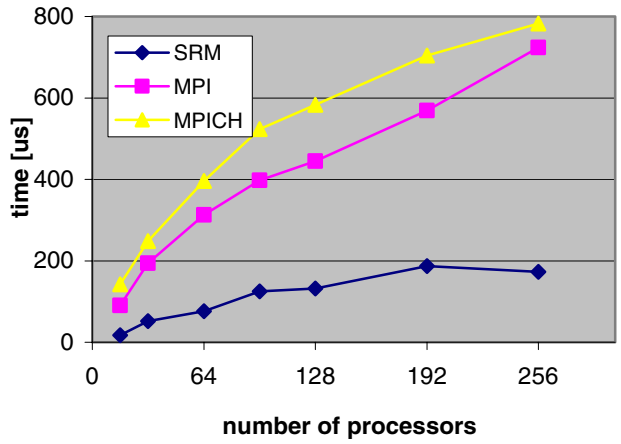

Figure 12: Performance of barrier operation

Depending on the message size and number of processors, SRM broadcast outperforms IBM MPI_Bcast by 27% to 84%. Similarly, for the reduce operation, savings ranged from 24% to 79% over MPI_Reduce. For allreduce, SRM is faster than MPI by 30% to 73%, again depending on the number of processors and message sizes.

Figure 12 compares performance of the SRM barrier with both MPI implementations on up to 256 processors. The performance and scaling advantage of our approach is clear — on 256 processors, an improvement of over 73% was achieved.

4. Related Work

With exception of our earlier paper devoted to barrier [17], to our knowledge, there is no published work that employs combined shared memory and RMA-based protocols for direct implementation of collective communication operations on SMP clusters. Multiple papers described issues and methods involved in implementation of point-to-point message passing on top of one-sided communication protocols e.g., [4] or shared memory e.g., [1,14]. A sizable number of papers on collective communications focused on aspects other than selection of communication protocols.

Several previous efforts focused on designing algorithms and communication structures for collective communication operations. Banikazemi et al [2] discuss multicast operations on heterogeneous networks of workstations. Paper [5] describes degree-D trees and generalized Fibonacci trees in the context of message passing and discusses methods for pipelining and repeating a collective operation. Huse [6] compares different communication structures for collective operations. Sophisticated algorithms are available to determine non-binomial optimal spanning trees. The interplay between cluster organizations and broadcast algorithms was investigated in [21]. Some studies [7,10] focused on collective algorithms designed within the framework of LogP model [7]. Other algorithms [5, 8] were discussed in context of the postal model [5]. Both of the models were created for operations based on point-to-point message-passing communication.

With a few exceptions, previous implementations of collective operations have been based on point-to-point message passing. One exception is the paper by Sistare et al. that describes SMP optimization of collective operations performed on a 64-way Sun server and a small 4-node cluster of 8-way Sun machines [11]. However, between the nodes, we use RMA instead of message passing in [11]. Despite both using shared memory; there are several key differences. First, in [11] a barrier was used to synchronize access to shared memory buffers, whereas SRM uses shared memory flags to coordinate access to buffers between the interacting task pairs. This weaker form of synchronization makes the overall algorithm faster and less susceptible to the processor late arrivals and delays. Second, the SMP reduce operation in [11] involves an extra memory copy by the SMP node root task, which SRM avoids by placing the result of the last reduce operation directly in the destination rather an intermediate buffer. Third, the reduce and allreduce operations based on shared memory in [11] are not competitive with the message-passing implementation in the SUN MPI for small messages, most likely due to the internal barrier used to arbitrate access to shared memory buffers, and the extra memory copy. The SRM counterparts of these operations do not use barrier and are faster than MPI for all message sizes. Finally, the SRM barrier employs RMA protocols and a less costly synchronization scheme within the SMP node while the barrier in [11] uses a spanning tree within each node and message passing between. Our barrier scales very well whereas it is hard to evaluate scalability of the barrier in [11] based on the 4-node results.

Some previous papers dealt with the shared memory barrier, as a topic independent of other collective operations (even outside the message-passing model context). For example, [3] describes the collection of efficient barrier algorithms for the scalable shared-memory COMA architecture of the KSR-2. These ideas could be considered in the SMP part of the SRM barrier for clusters with larger than the current shared memory nodes. A shared memory broadcast on the Sun Enterprise-10000 was discussed in [9]. It is similar to the SMP broadcast in [11] but it targets a single large machine. The dissemination algorithm described in [22] has similar properties to the pairwise exchange -like algorithm in SRM. Because SRM relies on RMA operations that allow overlapping communication between the process pairs to proceed simultaneously on modern networks, this effectively reduces for each process the number of operations on the critical path to ~log(P), the same as in the dissemination algorithm.

5. Summary and Future Work

This paper outlined a novel approach for optimizing collective operations using a combination of shared and remote memory access protocols. The experimental results obtained on the IBM SP show that SRM outperforms the highly optimized IBM implementation of MPI and the open source MPICH implementation across a wide range of message sizes and processor counts.

Despite the current performance improvements, more work is needed to exploit the full potential of integrated shared and remote memory protocols. Our plans for future work involve development of an analytical performance model of the SRM collectives to better understand, model, and evaluate effectiveness of this technique under different assumptions and parameter values such as the SMP node size, intra-SMP memory bandwidth, and performance of inter-node communication. That model also should be helpful in tuning the pipeline parameters in SRM. Some research issues related to the optimal embedding spanning trees for arbitrary MPI task groups in the SMP clusters remain open and we plan to pursue them in another paper.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL) and Ohio State University. PNNL is operated for DOE by Battelle. This work was supported by the Center for Programming Models for Scalable Parallel Computing sponsored by the MICS/ASCR program in the DOE Office of Science. Computational resources at the National Energy Research Scientific Computing Center (NERSC) and Environmental Molecular Sciences Laboratory (EMSL) at PNNL were used in this research.

References

1. W. Gropp and E. Lusk. A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer. Parallel Computing, 22, 1997.
2. M. Banikazemi, V. Moorthy, D. Panda. Efficient Collective Communication on Heterogeneous Networks of Workstations. ICPP, 1998.
3. D. Grunwald, S. Vajracharya, Efficient barriers for distributed shared memory computers, 8th IPPS, 1994.
4. M. Banikazemi, R.K. Govindaraju R. Blackmore, D.K. Panda. Implementing Efficient MPI on LAPI for IBM RS/6000 SP Systems: Experiences and Performance Evaluation, IPPS'99, 1999.
5. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. ACM Symposium on Parallel Algorithms and Architectures, 1992.
6. L.P Huse. Collective Communication on Dedicated Clusters of Workstations. 6th EuroPVM/MPI'99.
7. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken. LogP: Towards a realistic model of parallel computation. 4th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, 1993.
8. L. Gargano, A.A. Rescigno, Fast Collective Communication by Packets in the Postal Model. Networks, 31, pp. 67-79, 1998.
9. M. Bernaschi and G. Richelli. MPI Collective Communication Operations on Large Shared Memory Systems. 9th Euromicro Workshop PDP'01.
10. R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal Broadcast and Summation in the LogP model. Symposium on Parallel Algorithms and Architectures, 1993.
11. S. Sistare, R. van de Vaart, E Loh. Optimization of MPI collectives on clusters of large-scale SMPs, SC'99.
12. M. Bernaschi. Efficient Message Passing on UNIX Shared Memory Multiprocessors. Future Generation Computer System Journal, vol. 13, 443, 1998.
13. M. Bernaschi and G. Iannello. Collective Communication Operations: Experimental Results vs. Theory. Concurrency: Practice and Experience, vol. 10, no. 5, pp. 359-386, 1998.
14. MPI Collective Communication on the Convex Exemplar SPP-1000 Series Scalable Parallel Computer www.mpi.nd.edu/downloads/mpidc95/abstracts/html/fleischman
15. Eric F. Van de Velde, Concurrent Scientific Computing, Springer-Verlag 1994..
16. Luo. Y. Shared memory vs. message passing: the COMOPS benchmark experiment., 31st Hawaii International Conference on System Sciences, Vol: 7, 1998
17. R. Gupta, V. Tipparaju, J. Nieplocha, D.K. Panda, Efficient Barrier using Remote Memory Operations on VIA based Clusters, Proc. IEEE Cluster Computing, 2002.
18. W. Gropp, E. Lusk, A. Skjellum, Using MPI, 2nd ed. MIT Press, 1999.
19. J.S. Vetter, F. Mueller, Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures, Proc. IPDPS'2002.
20. G.H. Shah, J. Nieplocha, J. Mirza, C. Kim, R. K. Govindaraju, K. J. Gildea, R. Harrison, C. A. Bender, "LAPI: A Low Level Communication Interface on the IBM RS/6000 SP: Experience and Performance Evaluation", Proc. IPPS'98. 1998.
21. D. Basak, D.K. Panda, Designing processor-cluster based systems: Interplay between cluster organizations and broadcasting algorithms, Proc. ICPP'96, 1996.
22. D. Hengsen, R. Finkel, and U. Manber, Two Algorithms for Barrier Synchronization, International Journal of Parallel Programming, vol. 17, no. 1, 1988.