MPI Support for Multi-Core Architectures: Optimized Shared Memory Collectives

Richard L. Graham and Galen Shipman

Oak Ridge National Laboratory *
Oak Ridge, TN USA
rlgraham@ornl.gov and gshipman@ornl.gov

Abstract. With local core counts on the rise, taking advantage of shared-memory to optimize collective operations can improve performance. We study several on-host shared memory optimized algorithms for MPI_Bcast, MPI_Reduce, and MPI_Allreduce, using tree-based, and reduce-scatter algorithms. For small data operations with relatively large synchronization costs fan-in/fan-out algorithms generally perform best. For large messages data manipulation constitute the largest cost and reduce-scatter algorithms are best for reductions. These optimization improve performance by up to a factor of three. Memory and cache sharing effect require deliberate process layout and careful radix selection for tree-based methods.
Key Words: Collectives, Shared-Memory, MPI_Bcast, MPI_Reduce, MPI_Allreduce

1 Introduction

As HPC systems continue to grow rapidly the scalability of many scientific applications are limited by the scalability of collective communication. These systems are growing in both node counts and core counts per node. These multi-core nodes provide a way to increase the scalability of collective communication for applications which use more than a single MPI task per node. Implementing these algorithms in terms of on-host and off-host phases reduces network traffic improves their overall scalability. This paper will study in detail the options for implementing the on-host, shared-memory collective algorithms and compare these with Open MPI's point-to-point implementations using shared-memory merely as a transport layer. Memory traffic, cache conflicts and synchronization are barriers to the scalability and performance of shared memory collectives. These algorithms are aimed at reducing memory traffic by limiting the number of writers to a given memory segment and balancing synchronization and memory access costs. In addition to reducing memory traffic on socket and balancing synchronization we also aim take advantage of shared caches and reduce inter-socket memory traffic.

* Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

This paper describes the shared memory implementation of MPI_Reduce, MPI_Allreduce, and MPI_Bcast, and provides the results of benchmark studies of these collective algorithms. These are used by a large number of scientific simulation codes with reductions frequently being the collective communications pattern that hinders scalability the most. The range of reduction sizes used by scientific codes varies from single word sizes used to determine the convergence of iterative algorithms to many megabytes used to aggregate simulation results. Broadcast collective communication is often used to initialize application data structures.

The remainder of this paper is organized as follows; Section 2 provides an overview of previous work on shared memory optimizations of collective operations. Section 3 describes the shared-memory MPI_Reduce, MPI_Allreduce, and MPI_Bcast algorithms implemented within the framework of Open MPI. Results of numerical experiments are discussed in Section 5. Conclusions and future work are then discussed in Section 6.

2 Background

The recent interest in shared memory communications optimizations has come primarily from the desire to take advantage of the performance gain opportunities this gives in the context of hierarchical collectives on systems with a non-uniform memory hierarchy, such as clusters of Shared Memory nodes. As such, the main goal of these studies has been to show collective communications performance improvements of these hierarchical collectives, over standard implementations. Typical collective cost models [1,2] include a network communications term and, for operations such as reductions, a local processing term. On shared memory systems where one MPI process can directly access another's memory or a copy of this memory reduction operations can avoid the data transfer step by directly operating on another processes data. Operations such as broadcast can reduce the number of memory transfers with multiple consumers accessing a shared buffer. This paper studies the benefits of such optimizations over traditional point-to-point based collective communications with shared-memory as a mere transport mechanism.

Several MPI implementations have provided support for shared memory optimized collectives. These include but are not limited to LA-MPI [3], Sun MPI [4], and NEC's MPI [5].

Mamidala et al. [6] have studied the performance of shared-memory MPI_Bcast, MPI_Allgather, MPI_Allreduce, and MPI_Alltoall on a four core Intel Clovertown system. Their focus was on the interaction with the underlying hardware rather than the characteristics of the algorithms. Mamidala et al. [7], also developed a shared memory MPI_Allgather algorithm for use in a hierarchical implementation of this algorithm.

Tipparaju et al. [8] studied the effect of taking the shared memory hierarchies for several collective algorithms. They focused on the hierarchical collectives, and the performance gains from exploiting the memory hierarchies.

Wu et al. [9] proposed a general approach for optimizing shared memory collective operations using several communications primitives. They implemented MPI_Bcast and MPI_Scatter using these primitives showing performance improvements for these collective algorithms over point-to-point based mechanisms.

Unpublished work on shared-memory collective operations in the context of LA-MPI [10] on the 128 processor SGI Origin2000 machines showed that as the process count increases, read and write contention for the shared memory segments can greatly affect the performance of these operations. As a result, in this paper we describe collective algorithms designed to take advantage of the ability to directly access another processes shared memory segment with carefully controlled memory access. We study these algorithms to examine the characteristics of MPI collective operations on the emerging multi-core systems.

3 Algorithms

The ability of multiple processes to directly access common memory brings with it several opportunities for collective communications optimization with memory operations being the means of inter-process communications. However, factors such as process affinity and cache and memory access have a substantial impact on the performance of these algorithms. Cache thrashing causes severe performance penalties, shared caches reduce the access times to volatile memory, and memory bus contention reduces the memory bandwidth available to each MPI process. We proceed to develop algorithms that selectively eliminate extra memory traffic by taking into account cache and memory characteristics.

The approach we use in all these algorithms is to assign a fixed size segment of shared memory to each MPI communicator for use by all the MPI collective algorithms. This segment is contiguous in virtual memory, is memory-mapped at communicator creation, and is freed at communicator destruction. This provides a means of allocating the resources only when they are needed and for adjusting to the changing needs of an application. This shared memory has a control region for managing the working memory and a scratch (or working) memory region used by the collective communication routines. The scratch region is divided into two banks of memory with each bank having several segments of memory. The number of banks and segments is determined at run-time with the default values being two and eight. A bank is either available for use, or not, and once available, the buffers are used in-order without any additional availability checks. The control region is used to manage the availability of these memory banks and a non-blocking barrier structure is associated with each memory bank. When the last segment in the bank is used, a non-blocking barrier is initiated, and if not complete when an attempt to re-use the first segment is made then the process blocks completing this barrier. Multiple banks are used to allow the non-blocking barrier to complete while another block is in use reducing the synchronization costs.

The memory segments used by the collective algorithms are also divided into a control region and data regions. The control region consists of a flag the algorithms use for signaling other processes and the data region is where each

process puts its data. There is one control region and one data region per process in the communicator. Each of these regions is page aligned with the size of the control region being fixed and the size of the data region being set at run-time. By default a single page is allocated for each process's data region. First touch is used to ensure memory locality (if process affinity has been enabled) and a given process only ever writes to it's own control and data region within the memory segment but may read other processes' control and data regions.

The memory cost is constant on a per-process basis with the overall segment sizes scaling linearly with the number of processes in the shared memory communicator. Memory costs also scale linearly with the number of pages used in the data segment. The overall cost of the memory bank control region per process and per bank is the cost for the data structure (a 64 bit field) for the non-blocking barrier. For two memory banks this amounts to one page per process as these data structures share pages on a per process basis.

The shared memory scratch space provides all processes in the communicator access to common data. Since the size of these data segments is fixed and the number of data segments is limited all algorithms process the user data a segment at a time. The reduction routines described in this paper can only be applied to data types that fit within the per-process data segment.

Communication patterns are pre-computed and cached at communicator creation time. The nodes in partial leaf levels of a tree are assigned a parent by distributing these uniformly across the parent layer. Process affinity is used to control process locality taking into account memory and cache hierarchies.

3.1 MPI_Allreduce

Three different algorithms are implemented for MPI_Allreduce, recursive doubling, reduce-scatter followed by an all-gather and fan-in/fan-out.

The recursive-doubling algorithm is useful for large data reductions in which data manipulation tends to dominate reduction time. Each process uses it's data-control flag to signal when it's data is ready for use. To allow both processes involved in a pair to process their data simultaneously the data segment is divided into to two sections. One section is the read section for both processes and other is the write section for the process owning that memory. The roles of these regions are reversed at each step in the reduction so that data can be used in place. For non-power of two communicators with M processes, if N is the largest power of two less than M, rank $N+K$ is paired with rank K, where $K = 0, 1, .., M-N$. Before the recursive doubling algorithm is used each pair reduces it's data, with each process reducing half the data, and the lower rank copying the higher rank's reduced data. After the recursive doubling phase ranks N and higher copy the results from their partner directly into the user's buffer. A single segment is used for all sections of a large single reduction. The only parameter that can be varied is the size of the data segment.

The reduce-scatter algorithm followed by an all-gather is efficient for large data reductions. It typically performs better than the recursive-doubling algorithm. At each step of the reduce scatter each process in the pair reduces it's portion of the data into it's temporary buffer and then reads the data directly

from its partner's shared-memory buffer. The all-gather step is a simple data read from the scratch space of the other processes. Data readiness is signaled using the data-control flags and for non-powers of two are handled in a manner similar to that used in the recursive doubling algorithm. A single segment is used for all sections of a single reduction. The only parameter that can be varied is the size of the data segment.

The fan-in/fan-out algorithm is aimed at minimizing synchronization between processes participating in the reduction operation and is suited well for small data reductions where synchronization costs are prominent. We implement the fan-in and fan-out with different radixes. The fan-in phase involves synchronization of n+1 process in a tree of radix-n, as a single process, the parent process, serially reduces the data from the other n processes onto it's own data. In the fan-out phase the parent process signals M processes in a tree of radix-m that the data is ready to be read and these m processes can attempt to read this data simultaneously. To keep the synchronization cost down we use a new shared memory segment for each section of the user data to amortize the cost of "freeing" these buffers. The size of the data segments can also be varied.

3.2 MPI_Reduce

Two different algorithms are implemented for MPI_Reduce a fan-in algorithm and a reduce-scatter followed by a gather to the root. These are implemented in a manner similar to that of the MPI_Allreduce algorithms. The fan-in algorithm is just the first half of the MPI_Allreduce. For a given communicator we cache the fan-out tree for rank zero as the root and translate the nodes of the tree by n (with wrap around) for root n. The reduce-scatter algorithm differs from the MPI_Allreduce in that the results of the reduce-scatter are gathered back only to the root of the operation.

3.3 MPI_Bcast

The MPI_Bcast is implemented as a fan-out tree of radix-m which can be specified at run-time. For a given communicator we cache the fan-out tree for rank zero as the root and translate the nodes of the tree by n (with wrap around) for root n.

4 Experimental Setup

The shared-memory collective routines are implemented within the Open MPI code base [11] as a separate collective component. The working code is revision 18489 of the trunk. We take advantage of Open MPI's process affinity to control process locality and control memory locality using a first-touch approach.

The performance measurements were all taken on a 16 processor quad-socket, quad-core, 2 Gigahertz (GHz) Barcelona Opteron system with 512 kilobytes (KB) secondary cache and 2 megabytes (MB) shared tertiary cache per socket. The system is running Linux version 2.6.18-53.1.13ccs.el5.

The performance measurements were taken using simple benchmark codes with an outer loop wrapped around an inner loop of calls to the collective routine.

A barrier is called right before and right after the inner loop with the time being measured between the ends of both barrier calls. For rooted collectives an inner-most loop is added which rotates the root of the operation with all processes being the root an equal number of times in a particular measurement. We use integer data in all the measurements and we use the MPI_SUM reduction operations in the reduction operations. We report the latency of the collective operation as the average time per call.

5 Results and Discussion

Experiments were performed to study the shared-memory optimized MPI_Bcast, MPI_Reduce, and MPI_Allreduce. The following sections summarize the results of these experiments. To help keep the discussion brief we will discuss the results in two classes; short data in the range of eight to 1024 bytes; and large data in the range of 1 to 16 MB. These ranges of data sizes were selected as they are the most relevant to applications with which we are familiar. We also restrict the discussion to sixteen process MPI jobs as our experimental hardware was limited to 16 cores per node.

5.1 Memory Hierarchies

The memory layout of the quad-core Barcelona shared-memory nodes offers several opportunities. The tertiary cache shared between cores on a single socket offer an improved multi-process memory access for volatile data. When going off socket, multi-process volatile data access must go to main memory, with the cores on a given socket all sharing that socket's bandwidth. To get an idea on the order of magnitude of these effects we measured the latency of an eight byte and sixteen MB MPI_Allreduce operation laying out the MPI processes in several different configurations. For two process reductions we found that sharing the socket improved the small data operation by about 15% and the large data operation by 10%. For an eight process reduction, using all the cores on two sockets improved the small data operation by about 10% over spreading the eight processes across all four sockets. However the increased memory contention with only two sockets in the large data case reduces it's performance by about 28% relative to the four socket case. While these affects are not as large as those going between hosts, they are significant, and need to be taken into account. In the current set of experiments we used this information as a guide to setting up the communication patterns by using the Linux process affinity capability to lay out the MPI processes in a manner that results in the desired memory traffic patterns. Different layouts are used for different collective routines and different data sizes. Planned future work will explicitly include these hierarchies in algorithm implementation.

5.2 MPI_Bcast

Experiments were carried out on small and large data while varying the radix of the fan-out tree. We choose radix values of 2–8,12,16 and a fragment size of $32KB$. Figure 1 shows the best set of results from these experiments carried

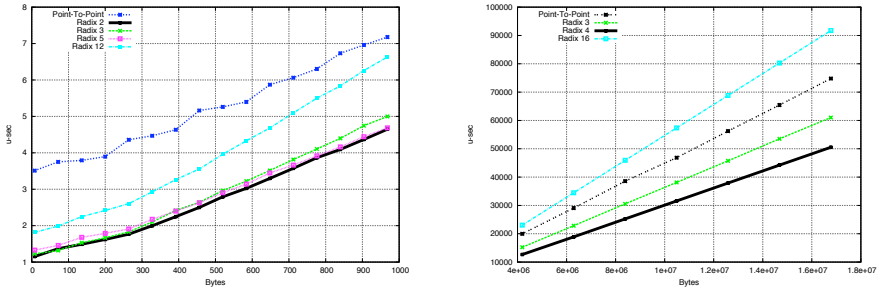

Fig. 1. MPI_Bcast as a function of fan-in radix for sixteen process communicators.

out with 16 processes and compares this data with the point-to-point based MPI_bcast algorithm on the same machine configuration.

For small data the shared memory broadcast using a radix-2 broadcast tree gives the best performance over the range of 8 bytes to 1KB, but the data values using radix-3 and 5 have similar performance. The radix-4 tree also gives similar performance but the data is omitted so as not to clutter the figure. The worst performance is obtained using a radix-12 tree. In all cases the shared memory broadcast performs better than the point-to-point algorithm using shared-memory communications with the radix-2 shared memory optimized broadcast algorithm being about two microseconds more efficient over the range of small data; a factor of three faster at eight bytes. Synchronization for the shared memory optimized routine is far simpler and amounts to reading a flag at a pre-computed address whereas the synchronization in the point-to-point based method occurs via the MPI general-purpose send/receive matching logic.

The large data shared memory broadcast using a radix-4 broadcast tree performs noticeably better than with other reduction trees with a tree of radix-3 being about 25% slower at a message size of four MB. It should come as no surprise that the worst performance is obtained when using a tree of radix-16, with all processes trying to read one buffer thereby creating a large amount of memory contention. The radix-16 algorithm performs even worse than the point-to-point based method. The latter performs about 60% worse than radix-3 broadcast algorithm which is not surprising, given the reduced number of memory copies in the shared-memory optimized algorithm.

5.3 MPI_Reduce

Experiments were carried out using both small and large data varying the radix of the fan-in tree. We chose radix values of 2–8,12,16. As expected, for small data fan-in reduction tree algorithms are the algorithm of choice. For large data a reduce-scatter followed by a gather to the root is the most suitable algorithm of those used. We limit our discussion to these combinations of data-size and algorithm and compare them with point-to-point based implementations of the

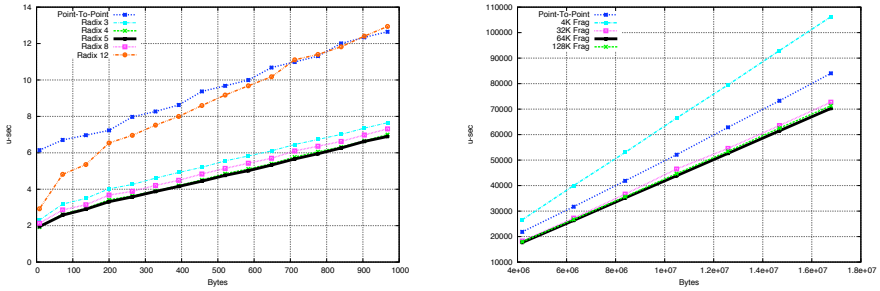

Fig. 2. Left: MPI_Reduce as a function of Fragment size for a sixteen process communicator.

MPI_Reduce algorithm. Figure 2 shows a select set of results from these experiments carried out at a sixteen process count.

For small data the best results are obtained with a radix 5 fan-in tree which gives virtually identical performance to that of a radix-4 tree. The results using a tree of radix-8 and radix-3 are slightly worse. Using a tree of radix-12 results in much worse performance and at the upper end of the size range this performance is even worse than that of the point-to-point based method. The point-to-point reduction routine performs quite a bit worse than the radix-5 fan-in tree based shared memory reduction. At eight bytes the shared memory version is about a factor of three faster than the point-to-point based method and at 968 bytes it is about a factor of 1.8 times faster with the slope of the point-to-point based method being steeper than that of the shared-memory based method.

For large data, communications time is reported as a function of shared memory segment data size. The best performance in this case is obtained with segments of size 64KB, with comparable performance using fragments of size 128KB and 32KB. The best shared memory results are about 23% better than the point-to-point based method for a 4 MB transfer size. However, if a segment size of 4KB is used, the shared memory implementation is slower than the point-to-point based method. At this message size the dominant factor in the overall time are the memory operations and the shared memory algorithm with 4KB segments can not use the memory subsystem as well as the 32KB sized point-to-point shared memory segments used by the point-to-point communications layer.

5.4 MPI_Allreduce

The experiments performed for the MPI_Allreduce function are very similar to those of the MPI_Reduce. These results are reported in Figure 3. In addition, a recursive doubling algorithm was implemented, but this is not competitive with the reduce-scatter/allgather algorithm used for large data and therefore we do not report these results.

For small data, the results from a small set of algorithms using fan-in/fan-out tree algorithms are reported. Overall, a fan-in and fan-out radix of five and five,

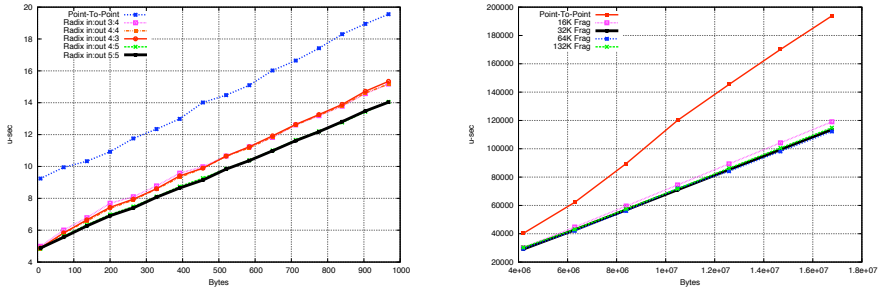

Fig. 3. MPI_Allreduce for a sixteen process communicator. Left: short message fanin/fanout. Right: Reduce-scatter, followed by Allgather.

respectively, give the best performance, but values of four and five, respectively, give virtually identical results. The all-reduce algorithms with fan-in/fan-out radix pairs of 4:3, 4:4, and 4:3 have slightly worse performance at this range of sizes. At eight bytes the performance of all these methods is virtually identical as they are dominated by synchronization but at 968 bytes where memory operations take a larger portion of time the first set of methods are about 8% more efficient than the second group of shared memory optimized methods. Similar to the MPI_Reduce case, the shared memory algorithms perform better than the point-to-point based algorithms. The method using a fan-in radix of five and a fan-out radix of five performs about 89% better than the point-to-point based algorithm at eight bytes and about 29% better at 968 bytes. In addition to improved latency, the slope of the point-to-point based method is higher than that of the shared-memory optimized method.

For large data, the best performance is obtained when using memory segments of 32KB with the reduce-scatter/allgather algorithm which is only marginally better performance than that obtained using both 64KB and 128KB segments and slightly worse at 16KB segment size. With the reduced memory traffic of the shared-memory optimized algorithms relative to the point-to-point based algorithm it is not surprising that these perform better than the point-to-point based method. At 4MB, the 32KB reduce-scatter/allgather method is about 38% more efficient than the point-to-point based method and at 16MB it is about 71% more efficient.

6 Conclusions

Taking advantage of memory hierarchies in the implementation of hierarchical collective operations is a good way to improve overall collectives performance. In this paper we have examined the benefits of creating shared memory optimized collectives for on-node operations. We have presented optimizations of the MP_Bcast, MPI_Reduce, and MPI_Allreduce routines aimed at taking advantage of shared memory architectures. We use instance specific control regions to set flags indicating when data is ready to be used, bypass the point-to-point match-

ing logic, read other processes data directly out of their shared memory buffers reducing memory accesses and write only to one's "own" shared memory region. These algorithms also take into account cache and memory layout in assigning process affinity.

Shared memory optimization improve the performance of the local collective operations over point-to-point based methods. Performance improvement of 3 fold have been demonstrated for small data operations such as eight byte broadcasts and reductions across 16 processes. Improvements on the order of tens of percent were measured across a wider size of messages. In terms of absolute time per collective call the difference in performance between the shared-memory optimized and the point-to-point based algorithms increases with data size. These performance improvements require careful attention to memory layout as care must be taken not to overwhelm the memory subsystem. By varying the arity of tree based algorithms and varying the size of shared memory data regions performance can be improved dramatically.

This paper has described approaches to improve the performance of collective operations on-node. Future work will include exploring the use of these shared memory based collectives in conjunction with the hierarchical collectives framework within Open MPI or weather a tightly coupled approach in which the shared memory based collectives are integrated with point-to-point approaches for off-node communication are in order.

References

[1] Thakur, R., Gropp, W.: Improving the performance of collective operations in mpich. In: Lecture Notes In Computer Science. (2006) 257–267

[2] Rabenseifner, R.: Optimization of collective reduction operations. In: Lecture Notes In Computer Science. (2004) 1–9

[3] : LA-MPI. (`http://public.lanl.gov/lampi`)

[4] Sistare, S., vande Vaart, R., Loh, E.: Optimization of mpi collectives on clusters of large-scale smp's. In Proceedings of SC99: High Performance Networking and Computing (1999)

[5] : NEC web page. (`http://www.nec.de`)

[6] Mamidala, A.R., et al.: Mpi collectives on modern multicore clusters: Performance optimizations and communication characteristics. Accepted for publication at CCGRID 2008 (2008)

[7] Mamidala, A.R., Vishnu, A., Panda, D.K.: Efficient shared memory and rdma based design for mpi_allgather over infiniband. (In: Lecture Notes In Computer Science)

[8] Tipparaju, V., Nieplocha, J., Panda, D.: Fast collective operations using shared and remote memory access protocols on clusters. Proceedings of the International Parallel and Distributed Processing Symposium (2003)

[9] Wu, M.S., Kendall, R.A., Aluru, S.: Exploring collective communications on a cluster of smps. In: Proceedings, HPCAsia2004. (2004) 114–117

[10] Graham, R.L., Choi, S.E., Daniel, D.J., Desai, N.N., Minnich, R.G., Rasmussen, C.E., Risinger, L.D., Sukalksi, M.W.: A network-failure-tolerant message-passing system for terascale clusters. International Journal of Parallel Programming **31(4)** (2003)

[11] : Open MPI. (`http://www.open-mpi.org`)