Optimizing MPI Collectives for X1

Howard Pritchard, Jeff Nicholson, *and* ***Jim Schwarzmeier,*** *Cray Inc.*

ABSTRACT: *Traditionally MPI collective operations have been based on point-to-point messages, with possible optimizations for system topologies and communication protocols. The Cray X1 scatter/gather hardware and shared memory mapping features allow for significantly different approaches to MPI collectives leading to substantial performance gains over standard methods, especially for short message lengths and higher process counts. This paper describes some of the algorithms used, implementation features, and relevant performance data.*
KEYWORDS: *MPI, MPI collectives, CAF, POP, CAM*

1. Introduction

Almost all message passing applications make use of collective communications at one or more steps in their operation. In some cases, the performance of these collective communication steps can be an important factor in the overall scalability of a parallel application [1]. The importance of collective communication operations for message passing style applications is reflected in the voluminous amount of literature on the subject.

The MPI standard [2] defines a number of frequently used collective operations including broadcasts, global reductions, and all-to-all communication patterns. Since its adoption as a *de facto* standard for message passing parallel applications, a number of computer vendors and research groups have optimized these operations for specific computer architectures and interconnects.

Efforts at optimization of collective communication operations for some of the first MPI implementations focused largely on mapping point to point send/recv approaches efficiently to various mesh, hypercube, and torus topologies [3]. As these types of systems were supplanted by clusters of workstations and SMPs, more effort was directed toward optimization of MPI for these topologies[4-6]. On these types of systems, efficient MPI collective implementations typically divide operations into an on-host, shared memory based component, and a point-to-point based approach using a proprietary RDMA or TCP protocol for the inter-host component of the collective.

Some of the most recently reported investigations into collective communications optimizations have focused on various OS-bypass networks for clusters of commodity one to four way SMP servers. These include Quadric's MPI implementation [7] and MPI over Infiniband [8]. Some of the largest computer systems have specialized hardware which MPI can use for certain types of collective operations. The Earth Simulator(ES) MPI uses hardware support for its barrier implementation [9]. For other collective operations, the ES MPI is optimized along similar principles as those used for clusters of SMPs. The MPI for IBM BlueGene/L will make use of dedicated networks for global sums and barriers [10].

In this paper we describe ongoing work to improve the performance of MPI collective operations on the CRAY X1 and follow-on systems. Earlier work on shared memory optimizations for MPI collectives for X1 has been previously described [11].

The rest of this paper is organized as follows: In Section 2, we give some background on aspects of the X1 hardware and parallel job structure of importance to an MPI implementation. In Section 3 we describe briefly some of the initial optimization work done on X1 MPI, then proceed to describe how these optimizations have been enhanced by adopting algorithms that more efficiently utilize the vector processor units and network bandwidth. In Section 4 we describe algorithms for specific collective operations and present micro-benchmark results for these operations. In Section 5, results using a real applications are presented. Conclusions and future work are described in Section 6.

2. Background

X1 Hardware and OS Essentials

The distributed, shared memory architecture of the CRAY X1 has a number of features important to the implementation of efficient MPI collective operations.

The X1 is organized as a group of nodes, with each node containing 16 memory controllers (M-chip) coupled via a set of vector caches to four multistreaming processors

(MSPs). Each MSP consists of four tightly coupled subprocessors (SSPs). Each M-chip on a node is connected by two 1.6 GB/sec ports to an external network. This network connects the memory controller with the corresponding memory controller on each of the other nodes of the system. Pages of memory are striped across the sixteen controllers in 32 byte (cacheline size) chunks. Each M-chip contains remote translation table (RTT) hardware to efficiently translate incoming virtual memory addresses to physical addresses. The RTT eliminates the need for extensive page tables for processes which make many remote memory accesses [12].

MPI applications can be built to run one MPI process on each MSP (MSP mode), or to run one MPI process on each SSP (SSP mode).

As with most distributed shared memory architectures, memory on remote nodes is accessed via loads and stores generated by the processor. There is no need for OS-bypass or system calls to move data between nodes. Unlike most other contemporary cache-coherent, distributed shared memory architectures, only local memory references are cacheable. As will be shown in the next section, this type of cache coherency matches closely the requirements for efficient shared memory collective communication operations. In addition, vector load/store operations can be given a non-cacheline allocating attribute even for local, cacheable memory. This makes it easier for the X1 MPI implementation to avoid cache pollution problems that occur with shared memory MPI implementations on some cache coherent shared memory systems.

The X1 provides additional hardware support for shared memory communication protocols including atomic memory operations (AMO's) and memory fencing instructions (*gsync)*.

In order to effectively use the X1 RTT hardware and vector processors, parallel applications are organized as application teams. The virtual memory layout of processes within an application team differs substantially from that of a standard UNIX process. Although each process has private data, heap, and stack segments, these segments are cross mapped into the address space of the other processes within the team at regularly spaced intervals. In addition to these segments, the processes also share a symmetric heap segment analogous to that supported on older CRAY MPP platforms. This virtual memory structure allows for very efficient strided vector load/store operations across processes in the team.

The application team virtual memory layout also greatly simplifies the design of MPI for X1. There is no need for shared memory buffers and the resulting buffer management code, except for performance reasons. There is also no need to worry about whether or not a target virtual memory region in another process has been prepared for access by a DMA device. Processes merely need to exchange memory pointers to access data from other processes. These features are used extensively in the optimization of MPI collectives.

3. Improving the Performance of MPI Collectives on X1

The X1 MPI is derived from the SGI/CRAY MPI version for SGI Origin clusters and CRAY PVP systems. This implementation used cluster topology aware point-to-point algorithms for several MPI collective operations, retaining older MPICH algorithms for the remainder. As a first step to improving the performance on X1, the cluster awareness was removed, and explicit point-to-point MPI send/recvs were replaced with analogous shared memory constructs involving pointer exchanges and AMO's for synchronization. This eliminated the substantial amount of scalar overhead associated with MPI send/receive operations. Significant improvements in performance were observed on smaller system configurations. Unfortunately some of these changes led to severe memory bank/network contention problems on larger system configurations. The use of AMOs for large scale synchronization also led to scalability issues on larger systems. For certain operations, the aggregate bandwidth for long messages was also relatively low.

Analysis of these performance problems indicated that several strategies needed to be pursued, depending primarily on the amount of data each process exchanged with every other process in a collective operation. It is not atypical for MPI collectives to use different point-to-point algorithms for different size message lengths. For example, MPICH2 uses three different strategies for MPI_Alltoall depending on the message size [13].

To maximize the reusability of this optimization work in the event that it is found to be useful in other software, the algorithmic improvements described in the next several sections are being introduced into the X1 MPI software in a way which does not require significant dependence on X1 MPI specific internal data structures. One internal data structure, the communicator data structure, did have to be extended to include a pointer to a new data structure used by the optimized collective routines. This had to be done to avoid the use of communicator attribute functions, which are slow on the X1. The new *collopt* data structure contains state information, a synchronization flag array, and a scratch buffer for short message optimizations. The state information includes a phase variable, counters, and any incomplete polling requirements arising from previous non-blocking collectives like MPI_Bcast and MPI_Reduce. The state information and the double buffering method reduce synchronization requirements for the short message algorithms from two or more barriers to at most one.

For user derived communicators, the *collopt* structure is only allocated and initialized when a collective operation is invoked using the derived communicator . At job startup the *collopt* structure for the MPI_COMM_WORLD communicator is optimized. For multiple program, multiple data (MPMD) MPI jobs, communicators are created which span each individual application. In creating these application wide communicators, the ranks within applications are ordered based on the value returned by *shmem_my_pe*. A *collopt* data structure is allocated and initialized for each application communicator. When possible, the data buffer and flag array are allocated out of the symmetric heap, thereby allowing for constant stride vector operations in some cases. The *collopt* structure is freed when the application invokes MPI_Comm_free for a communicator.

Short Message Optimizations

Optimization of collectives involving small message per process (≤64 bytes) proved to be the area requiring the most significant algorithmic changes. Work by applications programmers using Co-Array Fortran (CAF) indicated the type of algorithmic changes required [14]. Generally these CAF algorithms recognize that the point-to-point type of message passing models one usually uses for MPI implementations are not appropriate for the X1 architecture for these message size. The RTT hardware allows the X1 vector units to store elements across many nodes of the system in one operation. The vector units also allow for very efficient polling over long vectors of flag variables in a way that does not generate excessive local memory traffic, even when using non-allocating vector loads of the flag variables. These capabilities allow the MPI implementor to recast many of the collectives as various combinations of vector polling and strided vector load/store (or gather/scatter) operations. For short message transfers, the data motion associated with a collective operation can be recast as loops, with an outer loop over message length, and an inner loop over the number of ranks(or a subset thereof) involved in the operation.

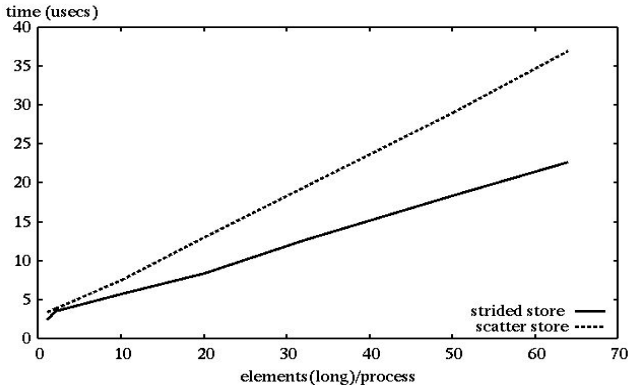

Figure 1. Overhead for multiple vector stores followed by a gsync and flag vector store across 128 MPI processes.

The performance of this type of strategy is illustrated in Figure 1. The plot shows the overhead for one process to execute differing numbers of strided vector stores (*puts*) of a 8 byte scalar quantity into the memories of all of the other processes within a 128 process application team followed by a *gsync* and a succeeding strided vector store of a flag quantity. This sequence of operations is the type required to send data and a notification that data is ready to be consumed. The remarkable thing to note here is that by using vector operations, one is able to send 80 bytes of data to 128 other processes in under 6 usecs, a time more typical of the overhead for a single point-to-point message exchange on most architectures. The figure also shows the results when using non-strided scatter vector stores. Although not as fast, the latency for the operation is still impressive.

Figure 2 shows the overhead for executing a single strided vector store followed by a *gsync* and second strided store as a function of application team size. The important thing to note here is the very weak dependence of the latency on application team size. It is only takes about 33% more time to execute the operation across 240 processes as 4.

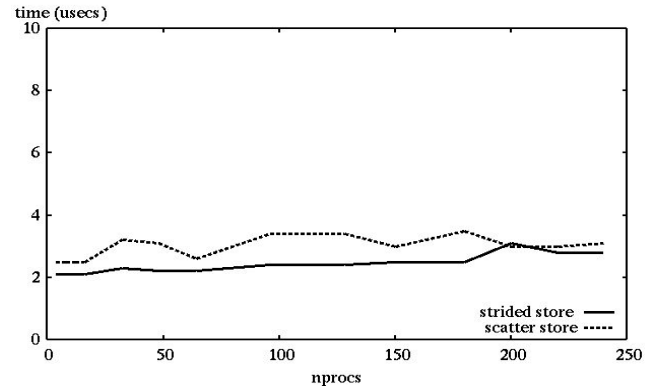

Figure 2. Overhead for a single vector store, followed by a gsync and flag vector store as a function of number of MPI processes across which the store is executed.

These low latencies are only achieved if one keeps the memory bank/network architecture of the machine in mind when laying out data structures to support this type of operation. Using the same strided vector store experiment, Figure 3 compares the execution time with and without padding of the data structure to accommodate the memory bank layout.

For short messages, the *collopt* buffer space associated with the MPI communicator is treated as a vector of length equal to the number of ranks in the communicator. Each vector element is 32 bytes in size to avoid memory bank conflicts. As an example, an alltoall broadcast (MPI_Allgather) of a 8 byte scalar quantity simply involves at most two vector stores with a stride consisting of a rank offset and a local offset:

```
for(dpe=0;dpe<nranks-rank;dpe++) {

   base[4*dpe*vma_offset/8 +(rank+dpe)*4]=
        scalar;

}

for(dpe=nranks-rank;dpe<nranks;dpe++) {

   base[4*dpe*vma_offset/8 +(rank-nranks+dpe)*4] =
        scalar;

}
```

`base` is the address of the symmetric *collopt* buffer on rank 0 of the communicator. The `vma_offset` is the byte offset between address spaces in the application team. The factor of 4 comes from the cacheline size (32 bytes) divided by the `sizeof(base)` which is 8 in this case. After the vector stores, a barrier is invoked, followed by a copy of the data from the *collopt* buffer into the application receiver buffer (two vector operations). Two vector operations are used to allow all process to concurrently execute optimal memory strides across the buffers. For more general communicators, the `vma_offset*dpe` is replaced with a table lookup since the *collopt* buffers are not symmetric for these communicators.

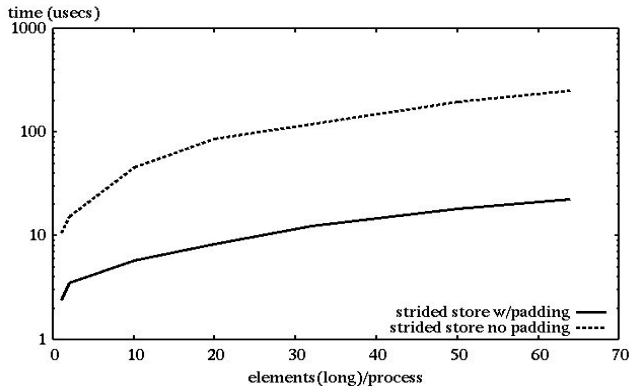

Figure 3. Comparison of the overhead for multiple vector store/gsync/flag stores with and without padding to avoid memory bank conflicts. Measurements were made for vector store across 128 MPI processes.

We found that for collectives involving short message exchanges, strided/scatter stores(puts) were more effective than the corresponding vector load(get) operation. Puts are more effective at masking latency to remote memory. Note that the use of puts works well on X1 owing to its caching policy and the large number (100s) of outstanding loads and stores allowable. Remote memory is not cached, but is directly written back to memory. Were remote memory cacheable, the result of a vector store operation would be many exclusively owned cachelines in the source processor's cache, which would create severe memory hot spot problems in a globally cache coherent system as the target processors attempted to change the cachelines' states from exclusive to shared.

The MPI_Alltoall and MPI_Allgather were optimized for short messages using this two vector store procedure. MPI_Barrier, MPI_Bcast, MPI_Allreduce, and MPI_Reduce were optimized using similar strategies. An internal version of the barrier, without argument checking and other MPI overhead was implemented for use within collectives. Internal versions of the all-to-all and allgather operations were also implemented to allow for efficient exchange of pointers and other data for use with medium and long length messages for MPI_Alltoall, MPI_Allgather, MPI_Allgatherv, and MPI_Alltoallv routines.

Medium and long message length optimizations

The AMO performance bottleneck that sometimes showed up for medium length data transfers was eliminated by replacing the AMO synchronization approach with an efficient gather of pointers and any other data using either the fast internal allgather or all-to-all mentioned above. The same approach is used for long messages.

For medium length messages, bandwidth was improved by using vector stores rather than loads of the remote data (put vs. get). In addition, for MSP mode, explicit streaming directives were used to stream outer loops over process count when possible.

In the case of the MPI_Allreduce operation, algorithms from MPICH2 were adopted. For the all-to-all operation, an edgecolor algorithm was used to pair up processes. These algorithms will be discussed in Section 4.

Release strategy

In order to avoid disruption to the production version of the X1 MPI (distributed as part of MPT 2.3), a strategy was adopted for gradually introducing these changes into the MPI software.

An independent staging library (MPTDEV) is used for initial development work. This library implements only a subset of the MPI collectives, invoking the corresponding profiling MPI interface function when necessary. An applications analyst or benchmarker only needs to link this library in ahead of the standard MPI library to pick up the optimized routines. This simpler library is easier to analyze from a performance perspective and easier to debug as no changes are made to the production MPI library to which the application must also be linked. After sufficient testing exposure, changes from the staging library are integrated into the development MPI (MPT 2.4) tree. After additional testing by applications analysts and benchmarkers, selected mods are being pushed back into the current MPT 2.3 production release.

The full set of collective optimizations will be incorporated into the MPT 2.4 release scheduled for the second half of 2004. A subset of these optimizations are being released in an upcoming MPT 2.3 update.

4. Specific MPI Collectives

MPI_Barrier

Although explicit barrier operations are normally not necessary in MPI-1 send/recv style programs, the barrier functionality is very important for internal use in some of the other collective optimizations. In addition the MPI-2 MPI_Win_fence operation can involve a barrier. For these reasons, the barrier function was the first target in the effort to optimize MPI collectives.

Initial investigations showed that the performance of AMO's under contention would require either fairly deep fan-in trees (log4), or a dissemination style barrier (log2). The barrier algorithm used in the current MPT 2.3 release uses the former type of tree barrier [11]. Neither of these approaches appeared capable of achieving a barrier time under 10 microseconds on 500 MSPs.

Given the performance improvements seen with some CAF optimizations and results from specialized algorithms developed by one of us which worked up to 64 processes, two algorithms were selected for further investigation. The first is a vectorized form of a tree barrier described in detail by Mellor-Crummey and Scott [15]. This MCS tree barrier does not rely on atomic memory operations. For the scalar processor based systems on which the authors did most of their measurements, small 2 or 4 way fan-in/fan-out ratios worked best. For the X1 hardware, it was found that a 64 way fan-in/fan-out was optimal over a wide range MPI job sizes. The key to its good performance on X1 is the high rate at which a vector of local values can be polled by the master process(es), and the efficiency of the strided vector store operation (the release phase of this barrier algorithm) described in Section 3. A second approach (JS tree barrier) uses an identical fan-in procedure for the first level of the tree, but in the second level of the tree each master process broadcasts a release flag to every other master process in an all-to-all broadcast pattern like that described in Section 3. The first approach involves less code, can work for arbitrary process counts, and is not as sensitive to whether or not symmetric arrays can be used. The second approach involves fewer trips through the memory network. It would need to be recoded as a recursive algorithm for more than two levels for machine configurations with more than 4096 processors.

Figure 4 compares the performance of **MPI_Barrier** using the MCS tree algorithm (MPT 2.4) with the AMO based barrier employed in MPT 2.3. Results for the JS tree barrier used within the MPTDEV library are also shown. It can be seen that in MSP mode, the newer algorithms are somewhat faster than the older AMO based approach. The JS tree barrier performs best for most process counts. Both the MCS and JS tree barriers take less than 10 microseconds on 500 MSPs.

Figure 5 compares the performance of the MCS tree algorithm (MPT 2.4) with the AMO based barrier in MPT 2.3 in SSP mode. At higher process counts, the MCS tree barrier results are much better. Work is ongoing to determine an optimal choice of tree barrier algorithm to incorporate in MPT 2.4.

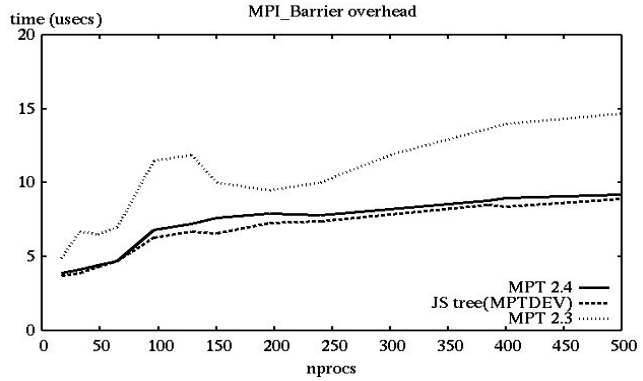

Figure 4. MPI_Barrier overhead comparison for MPT 2.3 and MPT 2.4 using the MSP mode.

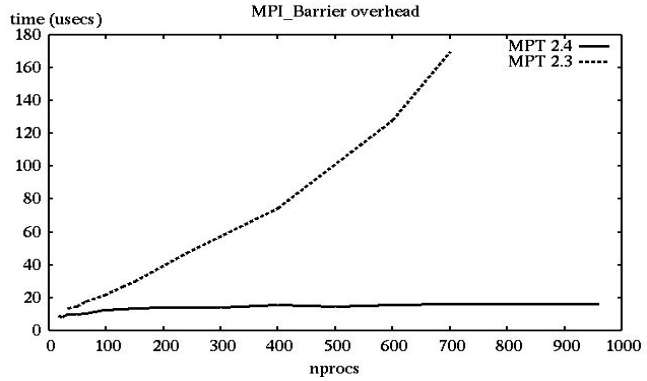

Figure 5. MPI_Barrier overhead comparison for MPT 2.3 and MPT 2.4 using the SSP mode.

MPI_Allreduce

The MPI_Allreduce function in the MPT 2.3 MPI library is implemented as a reduction followed by a broadcast. This approach does not scale very well largely due to network hot spots in the broadcast phase.

The new approach to the allreduce operation depends on the amount of data per process. For scalar quantities and short vector lengths (32 bytes or less) an algorithm adopted from some CAF optimization techniques is employed for predefined reduction operations. The CAF algorithm was modified to work in stages to allow for scaling to high process counts. The ranks in a communicator are grouped into blocks of 64. The first rank in each block is designated

a master rank. In the first stage of the reduction operation, all processes within each block store the contents of their source buffer into a location in the *collopt* buffer of the master rank for their group. Each then does a *gsync* followed by a scalar store of a flag value into a *collopt* flag buffer of the master rank process. The master rank does a vector poll on the flag array waiting for the other ranks to indicate their contributions to the sum have been stored. The master rank then does a vector reduction operation on the values (depending on the reduction operation specified by the application). If there are 64 or fewer processors in the communicator, the master rank then does a strided vector store of the global sum across the other ranks in the communicator, using locations in the ranks' *collopt* buffers. It then executes a similar vector store of a flag value into the ranks' *collopt* flag buffers. The other ranks, which had been spin waiting to see an update of their respective flags, then copy the resulting sum from the *collopt* buffer into the application receive buffer and return to the application. If more than 64 processes are involved in the operation, the master ranks in each of the groups participate in a second, inner series of operations analogous to those for the single level case.

This approach yields much better results than the reduction/broadcast method used in the MPT 2.3 library. Figure 6 compares the overhead for doing a global sum on a single MPI_DOUBLE quantity using the current reduce/broadcast method verses the approach described above. For most process counts, the improvement is about an order of magnitude.

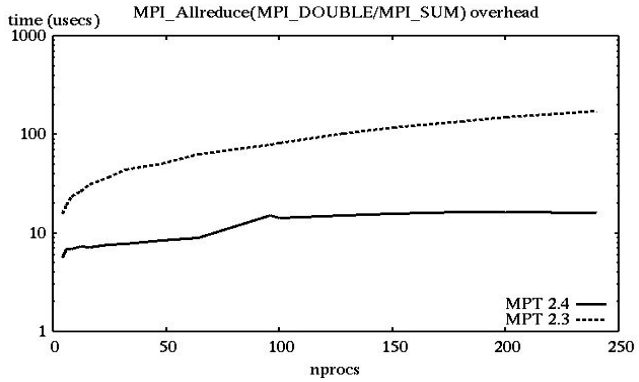

Figure 6. MPI_Allreduce overhead comparison for MPT 2.3 and MPT 2.4 using MSP mode.

For vector quantities longer than 32 bytes but fewer than 128 elements, a more familiar binary tree approach is taken. The *collopt* buffer in this case is organized into *n* segments, where *n* is the number of phases in the reduction process. This scales as log2 of the number of ranks in the communicator. Using the *collopt* buffers rather than the application buffers directly reduces the amount of synchronization required for these relatively short vector lengths. In theory, this method is more efficient than the reduce/broadcast algorithm by a factor of 2.

For long vector length reductions a check is made to see if the vector length divided by the number of ranks in the communicator is less than or equal to 64 elements. If this is the case, a binary tree method is again used, but this time employing the application buffers directly. If the vector length is greater, a reduce-scatter-gather algorithm [3,16] is employed. This is the algorithm employed in MPICH2 for longer vectors. It has significantly better scaling properties for long messages and high process counts.

Figure 7 compares the overhead for the allreduce operation for different vector lengths using the current reduce/broadcast algorithm in the MPT 2.3 library with the new allreduce operation currently in the MPTDEV library. The times were measured for a 128 MSP job executing an MPI_SUM on a vector of MPI_DOUBLE's.

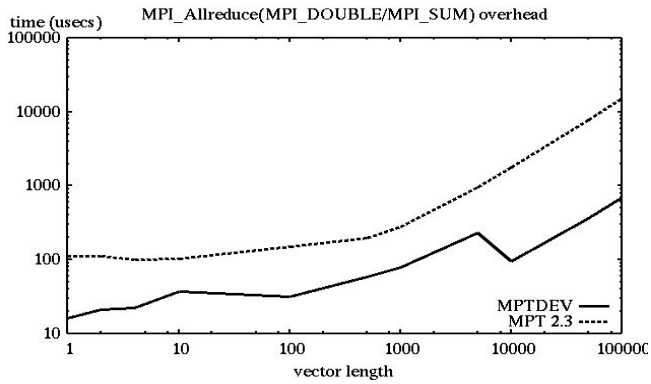

Figure 7. MPI_Allreduce overhead comparison for MPT 2.3 and MPTDEV for different vector lengths. Tests were run on 128 processors (MSP mode).

The drop in time from 5000 to 10000 elements for the MPTDEV library results owes to the switch from the binary tree to the reduce-scatter-gather method. The cutover criteria will be investigated further during integration of the method into the MPT 2.4 software.

User defined reduction operations employ the medium length binary tree approach.

MPI_Allgather/MPI_Alltoall

The MPI_Allgather and MPI_Alltoall functions involve a complete exchange of data between the ranks in a communicator. For the all-to-all exchange, different data is sent to every rank while for the allgather exchange, the same data is sent to all ranks.

The approach taken in the MPT 2.3 library is essentially the same as that used in the original SGI/CRAY MPI library, but with send/recv operations replaced by AMO based synchronization and exchange of pointers and datatypes. For short messages, application profiles have

shown that this AMO based synchronization can result in considerable network contention.

The new approach uses three algorithms depending on the amount of data to be exchanged with the other ranks. For messages between 0 and 64 bytes in size (depending on the amount of buffer space allowed per communicator and the number of ranks in the communicator) each process executes a series of strided puts of the source data into the *collopt* buffers of the other processes similar to the code shown in section 3. All processes then execute an internal barrier. The data is then copied from the *collopt* buffer into the application receive buffer.

Figure 8 compares the performance of the MPT 2.3 implementation of MPI_Alltoall with that in the MPT 2.4 library for a short message transfer (8 bytes) as a function of process count. Results are shown for an MSP mode job.

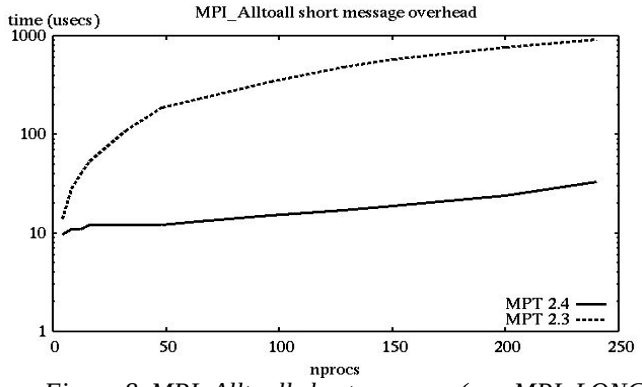

Figure 8. MPI_Alltoall short message (one MPI_LONG per process) overhead comparison for MPT 2.3 and MPT 2.4

For medium length messages, an internal allgather routine is used to collect pointers to the user receive buffers. This also serves as an effective barrier operation. Each process then executes an outer loop over processes and an inner loop over message length storing the appropriate segment of the application send buffer into the output buffer of the remote process. For the MSP mode library, streaming directives are used to insure that the outer loop is streamed and the inner loop over message length is vectorized.

For very long length messages it is more effective to stream over the message length rather than communicator ranks, so *bcopy* is employed. A put rather than get model is still used as this allows for higher aggregate bandwidth. Processes are paired using an edge coloring algorithm [17].

Figure 9 compares the performance of the MPT 2.3 implementation for MPI_Alltoall with that in the MPTDEV library for various message lengths at a fixed 48 MSP size job. The message length here is the amount of data exchanged with every other processor.

Results for MPI_Allgather are similar.

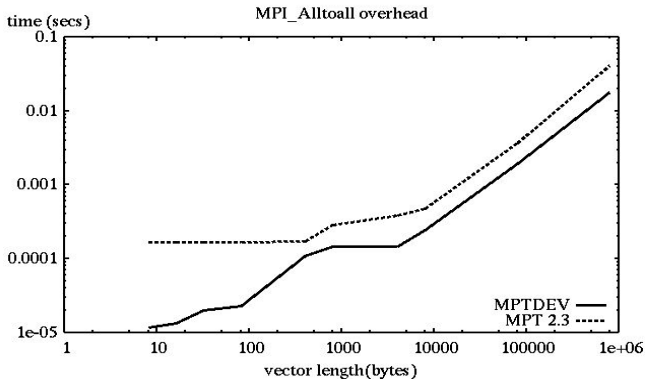

Figure 9. MPI_Alltoall overhead comparison for MPT 2.3 and MPT 2.4 for different message sizes. This test was run on 48 processors in MSP mode.

The aggregate bandwidth with the new algorithm using a put rather than get copy is substantially better than in the MPT 2.3 library - 2 GB/sec/process compared to 900 MB/sec/process. Work is ongoing to see if this aggregate bandwidth can be further improved using hand coded CAL copy routines optimized for remote stores.

MPI_Alltoallv/MPI_Allgatherv

MPI_Alltoallv and MPI_Allgatherv functions are generalizations of the MPI_Alltoall and MPI_Allgather functions. With alltoallv and allgatherv, each rank can send/recv different amounts of data to/from other ranks. In addition, arbitrary offsets in terms of the receive data types can be used to specify at which points to receive data from each rank in the application receive buffer. For **MPI_Alltoallv**, arbitrary displacements into the send buffer in terms of the send data type can also be specified.

In the MPT 2.3 library, AMO synchronization methods are used to gather pointers and data types in a manner similar to that done for MPI_Alltoall and MPI_Allgather. For shorter message lengths, this synchronization approach generates excessive network contention. Longer message size transfers exhibit rather low bandwidths owing to a get rather than put paradigm.

The new approach uses a similar algorithm to that employed in the current library, except that a optimized internal all-to-all algorithm is used to exchange pointers to the application receive buffers and data types. Also, a put rather than get approach is taken for moving the user data. For the MSP version of the library streaming directives are used to enable streaming over target rank, which is especially helpful for medium length messages.

To measure the performance of alltoallv and allgatherv, a test was devised where for a given median message length,

messages sizes were adjusted randomly to be between 0 and twice the median message length. Figure 10 compares the performance of the MPT 2.3 version of MPI_Alltoallv in MSP mode with the results from the MPTDEV staging library. The test was run on 48 msps.

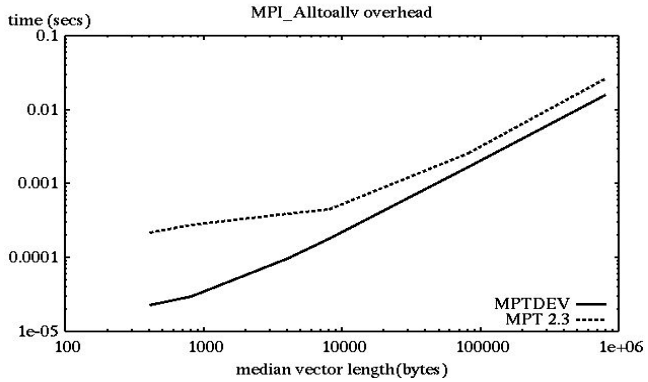

Figure 10. MPI_Alltoallv overhead comparison for MPT 2.3 and MPT 2.4 for different median message sizes. This test was run on 48 processors in MSP mode.

MPI_Bcast

The broadcast algorithm in the MPT 2.3 library uses a simple scheme in which the root process notifies all of the other processes in the communicator when data is ready to be received. The non-root processes then copy the data directly from the root's buffer into their respective receive buffer. This is an efficient algorithm for small system configurations, but exhibits very poor scaling behavior for small and large messages on larger systems owing to network contention.

The broadcast routine has been rewritten to use one of two algorithms depending on the message size. For messages between 0 and 64 bytes in size (depending on the amount of buffer space allowed per communicator and the number of ranks in the communicator) the root process executes a series of strided puts of the source data into the *collopt* buffers of the other processes similar to the code shown in Section 3. It then updates a status flag via a similar vector store. After accepting the data, the non-root processes update a flag in the root process' data structure. Some state information is recorded in the *collopt* data structure of the root process to indicate that a vector poll of its flags for the previous collective operation must be done in the next collective call.

Figure 11 compares the overhead for broadcast of a single MPI_DOUBLE quantity for the MPT 2.3 and MPT 2.4 versions of MPI as a function of number of MSPs.

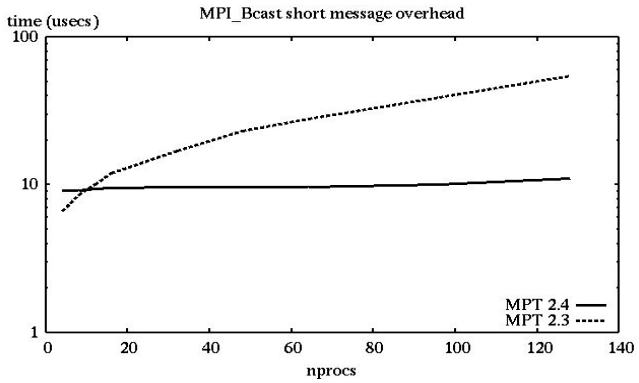

Figure 11. MPI_Bcast short message (one MPI_LONG) overhead comparison for MPT 2.3 and MPT 2.4. The test was run in MSP mode.

For long message lengths the original SGI/CRAY point-to-point binary tree was reintroduced for broadcast operations involving 32 or more ranks. This removed the network bottleneck exhibited by the current broadcast algorithm. Figure 12 shows the overhead for broadcast of 32000 bytes as a function of MSPs involved in the broadcast. Even with the overhead associated with point-to-point messages, it is much more efficient to use a binary tree operation as opposed to the original algorithm. Medium length messages will be handled using a buffering technique similar to that described for medium length MPI_Allreduce operations.

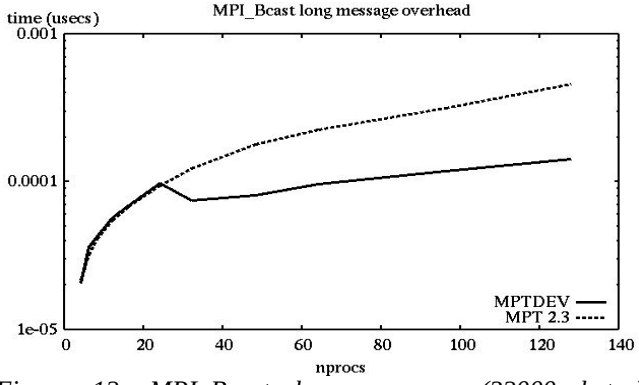

Figure 12. MPI_Bcast long message (32000 bytes) overhead comparison for MPT 2.3 and MPTDEV. The test was run in MSP mode.

A scatter/allgather is under investigation for use with long messages.

Other Collective Operations

The remaining MPI collective operations tend to be less commonly used than those previously discussed. Some are essentially specializations of other operations: MPI_Scatter is a variant of MPI_Bcast, MPI_Gather(v) is an MPI_Allgather(v) but with only one rank receiving the data, and MPI_Reduce is a reduction to a root rank rather than every rank in the communicator

The MPI_Reduce function is optimized similarly to MPI_Allreduce. The MPI_Scatter(v) and MPI_Gather(v) will be optimized as specialized versions of the algorithms described previously for MPI_Bcast and MPI_Allgather(v). The MPI-2 MPI_Alltoallw function will be optimized along lines similar to MPI_Alltoallv. However, since the main purpose of this routine is to allow for the use of derived data types, its performance will be constrained by the performance of derived types in X1 MPI.

5. Application Results

The scalability of MPI applications depends on many factors, so improvements to the performance of the MPI implementation for an architecture may not necessarily lead to an improvement in the performance of a particular application. Application developers porting and optimizing MPI applications for the X1 are encouraged to make use of available profiling tools including PAT and more MPI specific tools to isolate performance and scaling bottlenecks. In this section, results are shown for a couple of real applications for which problems in the MPI library limited application performance.

POP

The POP ocean model [18] includes a barotropic solver which is very sensitive to the performance of global reduction operations for certain problem sizes. The standard benchmark problem (320x384x40 grid) scales poorly on the X1 using the MPT 2.3 release. Figure 13 compares the performance of the application's MPI solver using MPT 2.3, a pre-release MPT 2.4, and a version of the code using a CAF -based solver. The performance of the solver is measured in simulated years per day of computer time. The MPT 2.4 and CAF versions are using essentially the same algorithm up to 64 processes.

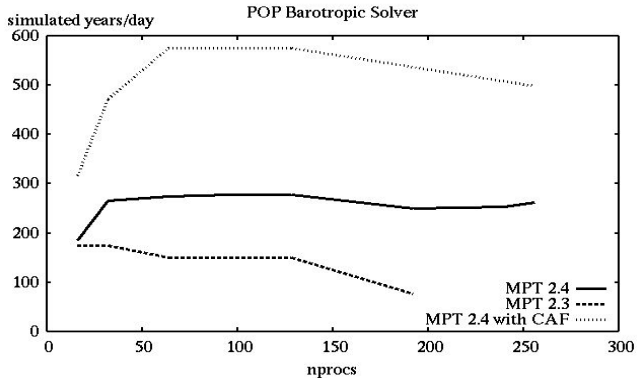

Figure 13. Comparison of the performance of the POP barotropic solver using MPI in MPT 2.3, MPT 2.4, and using CAF. Times are in simulation years/compute day.

Analysis showed that the CAF version benefits significantly from the compiler's ability to inline the CAF global reduction routine into the solver.

Community Atmospheric Model

The Community Atmospheric Model(CAM) does not show the same sensitivity to a particular MPI collective operation as POP. However, profiling indicated that it makes frequent use of MPI_Alltoallv and MPI_Allgatherv with medium length messages that were too short to stream effectively using the MPT 2.3 software. CAF has been used to eliminate this bottleneck. The MPTDEV library was linked in to the pure MPI verson of the application to see if the improvements to the MPI algorithms could achieve similar results. Figure 14 compares the performance of the application using MPT 2.3, MPTDEV, and CAF. Times are those reported by the application for the simulated years assuming a 30 day run. The CAF and MPTDEV versions yield similar results, which at the highest process counts are about 7% better than for MPT 2.3.

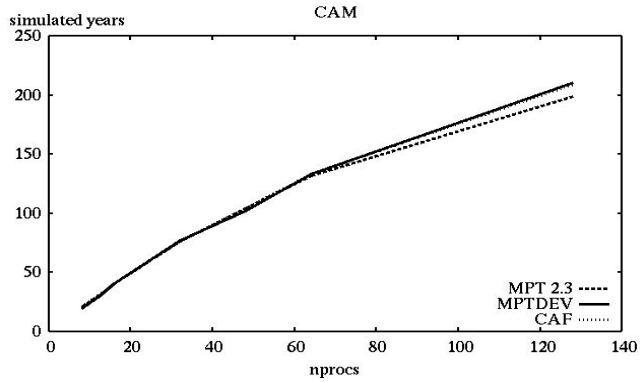

Figure 14. Comparison of the performance of CAM using MPI from MPT 2.3, MPTDEV, and CAF. Times are in simulation years assuming a 30 compute day run.

6. Conclusions

The CRAY X1 architecture is well suited to the implementation of efficient MPI collective communications, although non-standard algorithms need to be employed for short messages to make effective use of the vector processors, memory/interconnect organization, and the application team virtual memory layout. The next major release of CRAY's MPT package (MPT 2.4) will feature MPI collective functions which incorporate optimizations based on these short message algorithms, as well as more efficient traditional algorithms for longer message lengths.

Acknowledgements

The authors wish to thank Steve Scott for help with the barrier algorithm and John Levesque for help with some CAF kernels. Mark Pagel assisted in providing timing data for the POP application.

About the Authors

Howard Pritchard is a software engineer in the MPT group with Cray Inc. He can be reached at howardp@cray.com. **Jeff Nicholson** is a software engineer in the Programming Environment group at Ciray. He can be reached at jmn@cray.com. **Jim Schwarzmeier** works on performance optimization and as an interface between the hardware and software organizations. He can be reached at jads@cray.com.

References

[1] J. S. Vetter and F. Mueller. Communication characteristics of large scale scientific applications for contemporary cluster architectures. Int't Parallel and Distributed Processing Symposium (IPDPS'02), April 2002.

[2] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. MPI – The Complete Reference, vol. 1, The MPI Core, MIT Press, 2nd edition, 1998.

[3] M. Barnett, L. Shuler, S. Gupta, D. Payne, R. van de Geijn, and J. Watts. Building a high-performance collective communication library. Proceedings of Supercomputing 1994, pp. 107-116.

[4] S. Sistare, R. van de Vaart, and E. Low. Optimization of MPI collectives on clusters of large-scale SMP's. Proceedings of Supercomputing 1999.

[5] S. Sistare and C. Jackson. Ultra-high perfomance communication with MPI and the Sunfire link interconnect. Proceedings of Supercomputing 2002.

[6] K. Feind. SGI message passing status and plans. CUG Summit 2001.

[7] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Hardware and software-based collective communication on the quadrics network. Proceedings of the 2001 IEEE International Symposium on Network Computing and Applications (NCA 2001) Cambridge, Massachusetts, October 8-10,2001.
http://www.c3.lanl.gov/~fabrizio/papers/nca01.pdf

[8] R. Gupta, P. Balaji, D. Panda, and J. Nieplocha. Efficient collective operations using remote memory operations on VIA-based clusters. Int'l Parallel and Distributed Processing Symposium (IPDPS'03), April 2003.

[9] M. Golebiewski, H. Ritzdorf, J. Traeff, and F. Zimmermann. The MPI/SX implementation of MPI for NEC's SX-6 and other NEC platforms. NEC Research and Develop., Vol. 44(1), pp. 69-73, January 2003.

[10] G. Almasi, *et al.* MPI on BlueGene/L: designing an efficient general purpose messaging solution for a large cellular system. Proceedings of the 10th European PVM/MPI Users' Group Conference, Venice, Italy, September 2003.

[11] J. Nicholson and T. Goozen. Cray X1 MPI implementation. CUG Summit 2003 Proceedings.

[12] J. Schwarzmeier. Cray X1 architecture and hardware overview. CUG Summit Proceedings 2003.

[13] W. Gropp and E. Lusk. MPICH2: A high performance portable implementation of MPI, Clusterworld 2004.
http://www.clusterworld.com/CWCE2004/William_Gropp_presentation.pdf

[14] J. Levesque. The Cray X1: balanced supercomputers. CUG Summit 2003.

[15] J. H. Mellor-Crummey, and M. L. Scott. Algorithms for scalable synchronization on shared memory multiprocessors. ACM Trans. Computer Systems. Vol. 9 (1), pp. 21-65. 1991.

[16] R. Rabenseifner. A new optimized MPI reduce algorithm. High-Performance Computing-Center, Univ. of Stuttgart, Nov. 1997.
http://www.hlrs.de/mpi/myreduce.html.

[17] A. T. C. Tam and C. Wang. Efficient scheduling of complete exchange on clusters. The ISCA 13th International Conference on Parallel and Distributed Computing Systems (PDCS-2000), August 2000.

[18] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel ocean general circulation modeling. Physica D, 60, pp. 38-61, 1992.