Cray X1 MPI Implementation

Jeff Nicholson and **Tom Goozen,** *Cray Inc.*

ABSTRACT: *For the MPI implementation on the Cray X1 architecture, a number of changes had to be made to the MPI algorithms ported from SGI in 2000. These changes were required to take full advantage of the high bandwidth between the Cray X1 processors: several key routines were completely rewritten to take advantage of the Remote Translation Table (RTT) memory access feature; the mechanism used to do basic sends and receives was changed to a "hot path" to reduce latencies; and a new algorithm to reduce barrier timings was written that rivals the T3E hardware barrier. These changes, among others discussed here, have resulted in the very fast, very efficient movement of large amounts of data between processors.*

1. Introduction

It became clear after Cray ported MPI from SGI's implementation in 2000 that the SGI MPI algorithms could not accommodate the high bandwidth between processors in the Cray X1 system. This is in part because the SGI MPI implementation was written to be cluster-aware more so than for performance. Our task, therefore, was to streamline, simplify, and in some cases replace the SGI algorithms and routines with an eye toward optimized performance. A number of subroutines were deleted, and smaller routines were replaced with inlined code and macros. Cluster-aware code was removed and replaced with memory access schemes – these allowed us to take advantage of both the Cray X1 system's high memory bandwidth and its vector gather/scatter hardware. In addition to these changes, Fortran bindings were incorporated in the same source files as C routines, thereby eliminating additional calling overhead.

2. Cray's Commitment to MPI

A large number of Cray customers run parallel processing applications developed under MPI, the Message Passing Interface programming model. MPI has been implemented across the Cray SV1, SV1ex, Cray T3E systems, and now the Cray X1 system, to take advantage of each platform's unique processing characteristics. The MPI implementation for the Cray X1 complies fully with the MPI 1.2 Standard and selected portions of the MPI-2 Standard.

MPI 1.2 and MPI-2 Support

The full functionality of the MPI 1.2 standard has been ported to the Cray X1. Several of its core routines, mentioned later in this paper, include new algorithms that replace previous versions. These new algorithms take advantage of the hardware performance characteristics of the Cray X1. As we implemented these algorithms, we also removed areas of code that did not support our single system image (SSI) strategy for the Cray X1.

The MPI-2 features that were ported include Remote Memory Access (RMA – also called "one-sided") and MPI-I/O. MPI-I/O was ported from ROMIO version 1.2.4. Several of the same macros and inlined routines that were implemented in the core group of MPI routines have been implemented in the MPI-I/O routines as well to provide more performance enhancements.

The extended MPI-2 collective operations will be ported and available in a future release of Cray MPT.

MPI-2 Porting Exceptions

Due to a limitation in the operating system, the MPI-2 Dynamic Process Management (DPM) feature was not ported and the Generalized Requests functionality has not been ported to the Cray X1.

Mixing Programming Models

In addition to MPI, we have implemented the following parallel programming models: SHMEM, Co-Array Fortran (CAF), Unified Parallel C (UPC), and the soon to be available OpenMP. To date, we have successfully mixed MPI with CAF/UPC and with OpenMP. Applications developers will be interested to know that one-way message

passing in Co-Array Fortran and Unified Parallel C takes advantage of our compilers and low calling sequence overhead by doing data puts or gets from other processes within their parallel application program. These models do not require the data type conversion overhead needed in MPI.

3. Hardware Advantages

Single System Image

Early in the development cycle of the Cray X1 MPI implementation, we determined that to get the best performance, and thereby use the full hardware resources of the Cray X1, we would restrict MPI to a Single System Image (SSI) model. We decided this because, under normal operations, the scheduling scheme dedicates processor and memory resources to the application, but these resources are not relinquished until the application terminates. This meant that, had we decided to support a clustered implementation, valuable processor and memory resources would sit idle, waiting for the relatively slow network communication to complete.

Distributed Memory Architecture

The Cray X1 system is a NUMA architecture. With NUMA, memory is globally accessible, but it can be accessed faster locally than it can be accessed remotely. To help balance the access times to remote memory, a new facility, the Remote Translation Table (RTT), was introduced. The RTT allows parallel programming models to quickly determine remote memory addresses by managing the node and memory address fields and issuing a fetch or store to that address. The address is a 64-bit word, its upper-most bits reserved for the NODE-ID and other address information. Using the RTT removed the need for the regular Translation table code.

SSP and MSP modes (available in the PE 5.0 release)

To accommodate the range of demands of parallel programming applications, the Cray X1 system uses two processing modes: Single Streamed Processor (SSP) mode and Multiple Streaming Processors (MSP) mode, and provides a set of MPI libraries for each. MSP mode is for MPI applications that require large amounts of data to pass between processes. Here, all four SSPs on a MSP are called upon to help transfer portions of the data to the other processes. On the other hand, if an application is particularly compute-intensive, SSP mode enlists the other SSPs on the MCM to perform computations. An optimisation guide is currently being built that will better explain the different ways an application may provide better performance.

Scaling

Parallel programming models lose efficient use of computer resources as processes are added. This is because the more processes there are, the more overhead required in the parallel libraries to keep track of data movement between these processes. To compensate for this, we implemented additional parallel algorithms to reduce the library overhead needed to handle the message passing required by a large number of processes. These new algorithms allow for fewer serialized data handling schemes, plus they use the Cray X1 vector instructions. These algorithms, described later, rely less on the root process to do the majority of the work. They also use fewer global synchronization words, thereby reducing the likelihood of memory collisions.

32 and 64 Bit Library Support

Fortran programmers can take advantage of either the default 32-bit word size or use the 64 bit word size library versions. C programmers, of course, must specify `int` for the default 32 bit value, and `long` for the 64 bit word size.

4. Performance enhancements to MPI

This section describes the performance enhancements made to the core set of MPI message passing routines. Many other MPI routines are based on these routines and they have been enhanced as a consequence of the work done in the core set.

The MPI collective routines (reduce, gather, scatter, and so on) have been modified to enhance performance. The general types of modifications are as follows:

- Moving the Fortran entry points into the same file as the C entry points, and forcing inlining of the code into the Fortran version. Some routines also contained calls to internal versions of the code once tracing and error checking had been completed. Many of these have also been inlined.

- Convert calls that were to short external routines to macros that contain the code for these routines. These include routines such as `MPI_CRAY_type_is_bad`, `MPI_CRAY_comm_is_bad`, `MPI_CRAY_comm_rank`, and `MPI_CRAY_comm_size`, among others.

- The collective routines relied almost exclusively on the basic Send/Recv/Wait mechanisms in MPI. Since all of these routines effectively blocked at execution, and since they separated their communications from normal user Send/Recv traffic (they used a separate set of tags), it did not appear necessary to use the normal communication mechanisms. Since their

communication followed simple, well-behaved patterns, it was clear that they could benefit from a different data transfer mechanism.

The new basic data transfer mechanism used by the collective routines uses symmetrically allocated data structures that are referenced from the *gps* structure. Given the rank of a process, the *gps* structure can be referenced to determine the location of data structures located within that process. The data structures used by the new collective routines consist of:

- An array of size num_ranks that contains the address of the data to be transferred.
- An array of size num_ranks that contains the count of the data to be transferred.
- An array of size num_ranks that contains the data type of the data to be transferred.
- An array of size num_ranks that contains a start flag that triggers the transfers.
- An array of size num_ranks that contains a done flag (used only by the barrier routine) that indicates that the barrier has been processed by rank 0.
- A check-in counter that counts the number of processes that have processed a transfer request. This counter indicates that previous collective operations by the current process have completed.

The addresses of these structures are translated and placed into the *gps* structure so that any process can locate one of these data structures in another process.

Arrays for buffer addresses, counts, data types, and start flags are used so that multiple transfers can be initiated at the same time. If two collective operations should share some processes but have different communicators and be issued at the same time, using the arrays will keep the necessary information separate. The root or sending process rank, when referenced in the non-root or receiving process, typically indexes the arrays.

SPECIFIC CHANGES FOR COLLECTIVE ROUTINES

`MPI_Allgather`, `MPI_Allgatherv` and `MPI_Allreduce` Routines:

These three routines are basically algorithmically unchanged from the originals. They simply call the appropriate *Gather*, *Gatherv* and *Reduce* functions and then broadcast the results to the other processes. There are no significant performance advantages to rewriting these functions to combine the operation and the broadcast.

`MPI_Alltoall` and `Alltoallv` Routines:

In the old MPI implementation, the algorithm was to loop over all processes performing receive operations, then to loop performing send operations and finally to loop waiting for the sends and receives to complete.

The new algorithm uses the collective structures to broadcast addresses, counts and datatypes. If the send buffer is not contiguous, it is packed before any transfers are attempted. This allows the other processors to use a simpler algorithm to perform the data transfers – in most cases it allows for a simple *bcopy* to move the data. Initially, each processor places the address, count, and data type of the portion of the send buffer to be moved into each process, into the collective structures in the remote process. The location in the remote process is indexed by the sending process global rank. As mentioned above, this is why the address, count, data type and start flags are allocated as arrays. The check-in counter is set to the number of processes and used to indicate that all processes have received their data. Once the address, count and datatype information has been written to the receiving process, then the start flag is set and the remote process can read the data and can begin the transfer.

Each process enters a loop that reads the start flags being set by the remote senders. Any time a start flag is found, the address, count and datatype information associated with that start flag is read. The *type2type* routine is then called to move the data from the remote process to the local process. The *type2type* routine will handle any data conversion and reformatting issues in the transfer. The receiver then decrements the check-in flag for the sender.

The loop that scans for start flags is executed until all remote processes have been heard from and all data has been transferred. The use of the scan loop allows for receives to be processed out of order. Once all transfers are complete the code waits until the check-in counter goes to zero, indicating that all sends are complete.

`MPI_Barrier`:

The algorithm for barriers on the Cray X1 system is based on a four-way merge of incoming processes and a single broadcast to signal a resume. Both the shmem barrier

and the MPI barrier use the same algorithm, with minor differences to account for the calling parameters.

The basic structure used to signal arrival at the barrier is a four-way tree. At each level in the tree, up to four processes synchronize using a single word. The depth of the tree is the log(base 4) of the total number of processes participating in the barrier. As each process enters the barrier it uses an atomic memory operation (fadd) to increment a word shared by up to three other processes. Until the last process arrives, the processes before it enter a wait loop that spins on a resume word in local memory.

The last process to arrive continues to the next level in the tree and again increments a word shared by the next three groups of processes. If at any time a process is not the last process at that level then it enters the spin wait loop that waits on its local resume word.

The last process, at the bottom level, starts the waiting processes. Rather than transiting up through the tree again, activating each level, the final process uses a vector scatter store to set all the resume words in all the waiting processes. When the store is complete then the process is free to exit the barrier. All waiting processes exit when they see their resume word set. All processes clear their resume word before exit.

There are several important features of this algorithm:

- The number of processes that will access a single word at a single level is limited to four. This prevents large "convoy" times where large numbers of processes are attempting to access the same word on the same node. This also aids in the scalability of the algorithm.
- Because the algorithm uses a four-way tree instead of a typical binary tree, there are fewer levels needed for large numbers of processes. This also means that for a full barrier, the first level of synchronization takes place entirely on a node.
- At each level a new synchronization word is used which is in a separate memory bank. This prevents lower- and higher-level updates from colliding with each other.
- There is no "master" process, and the last arriving process triggers the resume.
- Since all updates to the level synchronization words are through atomic memory operations, and all updates of the resume words are through normal memory operations, no gsync instructions are needed by the algorithm. Only a single gsync is placed at the start of the routine to ensure that user memory operations are complete.
- Waiting processes spin on a resume word in their local memory. After the first pass through the wait loop, this word is loaded into the data cache, and no memory traffic leaves the processor chip from a waiting process.
- When the last process triggers the resume we do not have to wake up processes going back up the tree. With the vector hardware the algorithm can make effective use of the memory bandwidth of the machine. The last process can also exit when the last resume word is written. No gsync is necessary, and the resume write can still be in the memory network while the process is exiting the barrier routine.
- The resume words are skewed across the processes so that each resume word is in a separate memory bank on the node. This ensures each process will get its resume signal from a unique memory bank, and it prevents collisions in writing the resume word.
- This algorithm remains unchanged whether running with MSPs or SSPs. The only difference is that the first two levels of the tree take place on a node. The skewing for the synchronization and resume words already allows for the possibility of SSP mode operation. No attempt is made to use msync instructions since not all P chips within an MCM may be participating in the barrier and, unlike MSP mode, SSPs may not be assigned in sequence into a job.

NPES	IBM Power 4	Cray SX-6	Cray X1
2	6.7	5	3.25
4	12.1	7.1	6.45
8	19.8	22	7.80

Figure 1. ***Times are in microseconds***

MPI_Bcast:

The old algorithm used a power-of-2 type explosion from the root. This involved log(base2) levels of broadcasting. The even processors at each level would perform a heavy weight send operation to the odd processors for the same level. This "even/odd"ness was determined by checking the 2**level bit in the rank. Since the checking started at the high bit of the rank, this involved initially sending to distant processors, and ended with send/recv pairs to nearest neighbours. The algorithm had one drawback: if the root was not zero, the processes were effectively wrapped with the non-zero root being used as a base. This could lead to even more communication between distant processors.

In the new algorithm, the non-root processes pull the data from the root process. The root process moves the data to a preallocated buffer – if it is small enough (< 32 bytes). This allows the root process to exit quickly, even before the non-root processors have transferred the data. If the amount of data is large and non-contiguous, it is packed into a contiguous buffer. The root process then broadcasts the address of the appropriate buffer along with the count and the datatype for the transfer. The root process then sets the start flag to trigger the non-root processes.

The non-root processes loops to wait for the start trigger, then read the address, count and datatype. (They use the *type2type* routine to transfer data and perform any data conversion and reformatting.) Once the non-root processes have transferred the data, they decrement the check-in flag on the root process. As each non-root process completes its transfer, it exits the broadcast routine.

MPI_Gather and MPI_Gatherv:

In the old algorithm, the root process issued Irecv requests for all the non-root processes to place the data into the sections of the receive buffer. It used *type2type* to transfer its own data from its send buffer to the receive buffer. It then waited for all Irecv requests to complete. The non-root processes performed a Send operation to send their data to the root process. This resulted in the root process performing all of the actual work to copy all of the data segments to the receive buffer.

In the new algorithm the root process determines the locations for each process to place its data, and then uses the collective structures to transmit the address, count and datatype to the non-root processes. It sets the check-in flag to the number of processes minus 1, then transfers its own data from the send buffer to the receive-buffer using *type2type*. The root process then loops on the check-in count to wait for all non-root processes to transfer their data. This effectively involves a "push" from all non-root processes.

The non-root processes loop to wait for the start flag, and then read the destination address count and datatype. These processes use the *type2type* routine to transfer the data from their local storage to the root receive buffer. On completion of the transfer they use atomic memory ops to decrement the check-in count in the root process.

By using a "push" type algorithm, the new gather routines can perform the actual data transfer in parallel. Again, all looping on wait flags is performed on data words in the local memory space, thereby eliminating system memory traffic. Broadcasts of values and trigger flags could be done in vector mode in an application being run in RTT mode.

MPI_Reduce:

The old algorithm used a power of 2 folding on the processes involving log(base2) levels of operations. The odd processes used Send to transmit their data to the even processes. Where "even/odd"ness was determined at each level by the process number ANDed with 2^{level}. The even processes performed receive operations, which include performing the actual data transfer, and invoked the reduction function. Since the MPI standard requires that the reduction functions have only two input buffer arguments, with the second serving as the output buffer as well, the algorithm established two sets of buffers and carefully arranged the buffer so that the final operation placed the result in the receive buffer for the reduce. If the root for the reduction was not rank zero, a final copy was often needed to transfer the data to the root. (This only happened if the operation is non-commutative.)

The new algorithm still uses a power of 2 folding on the processes. The odd processes place the address of a buffer they wish to send into the collective structure and trigger the start flag. The even processes start the first level by transferring data from the send buffer to the receive buffer (Note: This is due to the problem with the MPI definition of the reduction function. A method is being investigated to eliminate this initial copy for the "built-in" MPI reduction types.). The even processes then wait for a buffer address to arrive from the odd processes. Once the address arrives, the reduction function is invoked and the data from the remote processor is read directly by the reduction function, combined with the local data, and output into the local buffer. The remote processor is signalled that the transfer is complete by decrementing the check-in flag in the remote process via an atomic memory operation.

Based on the commutatively of the operation being performed (all built-in MPI reduction functions are commutative), the new algorithm pulls in data from processors to the right (that is, higher rank processes). This results in good performance for reductions to the rank zero process, and requires no further transfers once the reduction is complete. For reductions to a non-zero rank process, the process numbers are wrapped by adding the root rank modulo the number of processes. This can lead to the need for transfers from some more distant processes, but it is better than having to add another copy after the final reduction.

For non-commutative operations, the algorithm remaps the process numbers such that the pull of the data takes place from the left (that is, lower rank processes). This

results in the final reduction being placed in the highest numbered process and almost certainly means that the result will have to be copied to some other process (probably rank zero). This is not desirable but seems to be necessary due to the non-commutative nature of the operation and the MPI standard definition of the reduction functions.

MPI_Reduce_scatter:

This routine is effectively unchanged from the original. It simply calls the Reduce and then the Scatterv functions. No significant performance advantage appears to result from attempting to combine these operations since the reduction must be completed before the scatter can begin.

MPI_Scan:

The old algorithm involved a Receive from the next lower ranked process, a copy of the send data to the receive buffer, and the invocation of the MPI reduction function. The process then sent the result to the next higher ranked process. Each process was free to continue as soon as it sent its data on to the next process in rank.

The new algorithm has each process use the collective structures to wait for the address, count, and data type from the lower ranked process. Because variations in data layout and type are permitted between processes, the process must use *type2type* to transfer the data into its local memory. The MPI reduction function is then invoked with the result being placed in the receive buffer. For non-commutative (and therefore non-built-in) functions, the algorithm performs a copy of the initial data from the send buffer to the receive buffer before calling the MPI reduction function. The process then passes on the address, count and data type for its receive buffer to the next higher process, and waits until the higher ranked process has picked up the data.

MPI_Scatter, MPI_Scatterv:

The old algorithm had the root loop over an Isend to each of the remote processors with the appropriate portion of the send buffer being transmitted. It then received a "send_to_self" in order to handle datatype conversion problems. It then performed a loop over request_wait to wait for the sends to complete. The non-root processes simply performed a receive from the root and transferred the data into their receive buffers.

The new algorithm has the root process loop over all of the other processes, and transmit the proper address offset within the send buffer along with the count and data type. It sets its check-in value to the number of processes (minus 1) and it then sets the start flag to trigger the other processes. The root process then calls *type2type* directly to perform the data transfer from its send buffer into the receive buffer with the proper data type conversion. The root waits until the check-in count goes to zero before returning.

The non-root processes wait until they are triggered by the start flag, and then pick up the address, count, and data type. They perform the data transfer in parallel using the *type2type* routine. When they complete the transfer, they decrement the check-in flag on the root process and exit.

SPECIFIC CHANGES TO SEND - RECEIVE

MPI_Send, MPI_Receive:

The MPI_Send process begins by acquiring a packet in the destination process memory. This packet is used to store information about the sending process context, tag, and source rank number. It also contains information about the type and length for the data to be transferred. The packet optionally contains either a pointer to the data in the sending processes memory and an acknowledgement flag word, or it can contain the data itself if it is less than a fixed payload size (currently 64 words).

If the amount of data to be sent is less than 64 words, then the data is copied directly into the packet on the remote destination process. The packet address is added to the incoming list queue on the remote process; and the send operation is considered to be complete and returns to the caller.

If the amount of data to be sent is greater than the payload size, a pointer to the data is placed in the packet along with the address of a word to be set when the transfer is complete. The packet is added to the incoming list queue of the destination process; and the sending process waits to be signalled that the destination process has transferred the data.

If buffering is turned on in MPI, the data to be sent is packed into buffer space; and then the address of the buffer space is passed in the packet along with a pointer to an acknowledge list entry. The packet is then added to the incoming list queue of the destination process; and the sender can return to the caller.

When MPI_Recv is called by the destination process, the receive request list is checked to see if there are other outstanding receive requests that have been generated by calls to MPI_Irecv. If there are none, then the receiving process simply loops to wait for a packet to show up in the incoming queue. Once a packet appears in the incoming list, a match is attempted against the context, source, and tag of the current receive request. The incoming list is continually

scanned until a matching entry appears in the list. Once the match is found, the data is transferred, and if necessary, a completion flag is set or an acknowledgement is flagged in the sending process.

If, when MPI_Recv is called, there are other receive requests already in the list as a result of prior calls to MPI_Irecv, then the current request must be added to the list in the proper order. This requires that a request entry be acquired from the free list and filled with the information specified by the request. Once the current request has been added to the list, the entire list is matched, in order, against the incoming list.

Paired receive requests and incoming list packets are processed by moving the data from the packet, for short sends, or from the sending process memory, for long or buffered sends. Completion flags and acknowledgement flags are set in the sending process as required by the incoming packet.

This process of matching the receive requests with the incoming list continues until the request associated with the MPI_Recv is matched and the data transferred. Once the request has been satisfied, then the MPI_Recv routine is allowed to return to the caller.

5. Coding and Algorithmic Considerations

Much performance testing has been done, and the results indicate that we have made great progress in most areas relative to these same tests run on the T3E.

Graphing the data show that larger data sizes (128KB – 2MB) that get transferred between Cray X1 processors have better throughput than the same data sizes on either the IBM Power-4 or SX-6, as shown here:

NPES=2	128KB	512KB	2MB
IBM Power4	1760 MB/s	1863 MB/s	131 MB/s7
Cray SX-6	6211 MB/s	8266 MB/s	958 MB/s0
Cray X1	10288 MB/s	14148 MB/s	1981 MB/s2

Figure 2. [1]

Note that we are still working to improve our performance.

[1] Figure 1 Data taken from Table 4 in the paper "*Performance Evaluation of the SX6 Vector Architecture for Scientific Computations"* by Leonid Oliker, Andrew Canning, Jonathan Carter, John Shalf, David Skinner CRD/NERSC, LBNL; Stephane Ethier Princeton Plasma Physics Lab; Rupak Biswas, Jahed Djomehri, Rob Van der Wijngaart NAS Div. NASA;
Tom Goozen, Cray Inc collected the Cray X1 data.

Review Application Design Assumptions

If code designs have not been looked at for four to five years, a review of the design assumptions should be made in light of current architectural designs in both hardware and software technologies.

Replace simple Send/Receive with CAF/UPC

In some instances replacing simple sends and receives with Co-Array Fortran (CAF) or Unified Parallel C (UPC) will boost performance. This is true in cases where the basic datatype is known and arrays of similar data are to be moved from one process to another. The overhead is eliminated that would normally be needed for MPI to handle requests, as is the overhead for the checking that normally happens for data conversion.

About the Authors

Jeff Nicholson is the lead software engineer for MPI with Cray Inc. He has broad background in compilers, libraries and provided input in the instruction set design of the Cray X1 system. Jeff has been with Cray for over 20 years.

Tom Goozen manages the Programming Environment Libraries group that includes Craylibs, MPT and X-window libraries.