Message Passing Toolkit (MPT) Software on XT3

Howard Pritchard, Doug Gilmore, Monika ten Bruggencate, David Knaak, and Mark Pagel, *Cray Inc.*

ABSTRACT: *This paper describes the implementations of the main components of the Message Passing Toolkit (MPT) on XT3 - MPI and SHMEM. Approaches to tuning of applications using MPT components are discussed. Current and upcoming feature work for both packages is also described.*

KEYWORDS: CRAY XT3, MPI, Portals, SHMEM,MPICH2

1. Introduction

The Message Passing Toolkit is available on a range of current and legacy CRAY systems. On most of these platforms, the Toolkit has featured implementations of MPI and SHMEM.

The XT3 MPT package includes an MPI implementation based on MPICH2 1.0.2 with some updates from the MPICH2 1.0.3 release. Most MPI-2 functionality available in MPICH2 is available on XT3 including remote memory access (RMA), MPI I/O, collective operations using inter-communicators, etc. MPI-2 process creation and management features, described in Chapter 3 of [1] are not currently supported.

The SHMEM implementation is all-new for XT3, partly owing to the significant differences between remote memory access methods available for SHMEM on XT3 and those available on most legacy and current CRAY platforms

Both the MPI and SHMEM implementations utilize the Portals interface to the XT3 network [2,3]. This interface places an emphasis on scalability and offload of communication-related work from the host processor. This has a significant impact on how applications can best use MPI and SHMEM on XT3.

Portals is an API which encapsulates many elements important to MPI-1 messaging, in particular MPI message matching order semantics. It presents an asynchronous interface to the application. An application submits requests that involve data motion – *PtlGet, PtlPut, etc.* to Portals. An application can also submit requests to match application receive buffers with the data motion operations from other processes. Portals indicates the progress of these various requests to an application via entries deposited in *event queues(EQ)* that have been associated with these requests. Cray has also ensured that, to some extent, progress of Portals may also be inferred by the visibility of updates in host memory on a node.

The rest of this document will be organized as follows. Section 2 will describe essentials of the MPI implementation on XT3. Section 3 will likewise give an overview of the SHMEM implementation on XT3. These descriptions should help application programmers better determine how to optimize their MPI or SHMEM codes for XT3. Section 4 will consider performance tuning and pitfalls to avoid for MPI and SHMEM applications. Section 5 will consider some outstanding issues with XT3 MPI, as well as future work planned for MPT components.

2. XT3 MPICH2 Implementation

The XT3 MPICH2 uses a custom abstract device interface (ADI3) based on a combination of the approach used in an earlier Portals-based MPICH1 implementation [4] and elements of the MPICH2 CH3 device [5].

As with a number of other MPI implementations, an *eager* protocol is used for sending short messages. By default, short in the context of XT3 MPICH2 means messages with up to 128000 bytes of data. A sender assumes that the receiver has sufficient buffer space (either using MPI internal buffers or pre-posted application buffers) and other necessary resources to manage short messages at the receiver. If the message to send is very short (1024 or fewer bytes by default), the sender first allocates a local buffer and copies the application data into this local, MPI internal buffer. The data is then transferred in manner

similar to that used for longer short messages. This approach allows blocking sends to return more quickly.

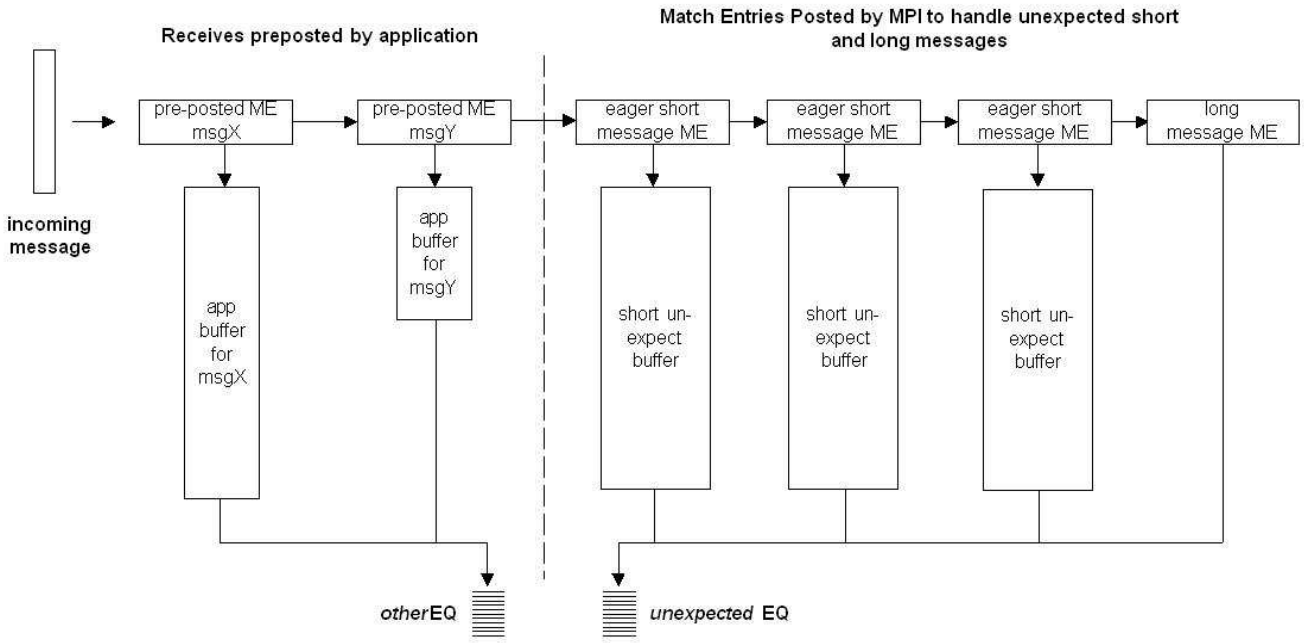

Figure 1 MPI Configuration of Portals Match Entries (MEs) for handling pre-posted receives and unexpected long and short messages.

Starting with the MPT 1.3 release, a sender can use one of two procedures for handling longer messages. The default is for the sender to build a small 8-byte message, which is sent to the receiver. This message has sufficient information for the receiver to pull the message data using a Portals *PtlGet*, when a match is found with a posted receive. In previous releases, the default was for the sender to send the message data much as if it were delivering a short message using the *eager* protocol. The former procedure is better when it is likely that the receiver is unable to pre-post matching receives prior to arrival of the data, but may be trying to do so shortly thereafter. This series of events occurs frequently in MPI benchmark tests, as well as for certain collective operations. The latter method is better when application logic insures that matching receives are often pre-posted. Using the MPICH_PTL_EAGER_LONG environment variable, the latter, *eager* procedure can be recovered. Note that the long protocol also assumes there are sufficient resources (space in the *unexpected EQ*) for the receiver to process the message.

It is on the receiver side of the ADI3 device that some of the special features of Portals are more evident. MPI is responsible for inserting *Match Entries (ME)* into a linked list managed by the Portals layer of the network protocol stack. MEs can be used to encapsulate MPI message matching criteria – a message tag, a communicator context, source process, and the wildcards MPI_ANY_TAG and MPI_ANY_SOURCE. In the XT3 Portals implementation, a *memory descriptor (MD)* can be associated with a ME. When an incoming message is received off of the network, Portals extracts information in the message header (the first packet of the message) and traverses this linked list of MEs looking for a match between the message header and an ME. This information can include an MPI tag, context from a MPI communicator, and sender. When a match is found, Portals delivers the data into the buffer described by the MD. An application can request that Portals manage delivery of data into this buffer – a *receive-side managed* buffer. The application can also specify that a ME be removed from the linked list if a match is found. The application can further specify the maximum amount of data, which can be received from each incoming message, or optionally, that it is okay to truncate the message. Even if there is a match between the message header and the ME, Portals will continue traversing the linked list looking for a match, if the message does not meet these length requirements (maximum allowed message size and available space in the MD if truncation is not allowed).

XT3 MPICH2 uses these Portals features as follows. At job start-up within *MPI_Init*, three buffers for *eager* messages are allocated and special MEs for these buffers are initialized and inserted at the start of the linked list of MEs

for the MPI library. The buffers are marked for *receive-side management* by Portals, that the maximum size message which can be received is 128000 bytes, and that truncation is not allowed. Portals is responsible for delivering unexpected messages with 128000 or fewer bytes of data into these buffers. Behind these three MEs in the linked list, a ME for long messages is inserted. This ME will truncate the message data to 0. Data in the event queue entries generated by matches to this ME provide MPI with enough information to *pull* the data from the sender side using a *PtlGet*. MPI is now ready to handle the application's MPI communication requirements.

At some point, a process will probably post a receive to the MPI library using one of the variants of the *MPI_Recv* function. The library first checks the receive against any unexpected messages which have already been received. If the receive matches a previously arrived short, unexpected message, the data is copied from the part of one of the three short unexpected message buffers holding the data and the receive is marked complete. If the posted receive matches an unexpected long message, the data is pulled from the sender node via a *PtlGet*. When a *get end event* is received, the receive is marked complete. If no matches are found in the unexpected queue, MPI builds an ME and MD to describe the receive request. Through a series of operations which are somewhat involved, owing to potential race conditions between incoming unexpected messages and receipt of the ME/MD by Portals, MPI inserts the ME into the linked list of requests ahead of the three MEs for unexpected short messages, but behind any MEs inserted into the linked list for previously posted receive requests. This arrangement is depicted in Figure 1.

At some point, a buffer for short, unexpected messages may get filled up with unexpected message data. The Portals layer informs MPI via an attribute in an event queue entry (EQE) deposited into the *unexpected EQ* that the buffer has become inactive. MPI must then scan the buffer for any data which has not yet been matched by a receive, and copy this data into heap memory. A ME is then reconstituted for this short, unexpected message buffer. It is then reinserted into the linked list behind the two other short, unexpected message MEs.

MPI uses two Event Queues to monitor progress of the requests submitted to Portals. An *unexpected* EQ is used to track incoming unexpected messages. Arrival of an unexpected message, whether short or long, will generate two EQEs. A *put start event* will be generated as the data begins to arrive. A *put end event* will be generated when the data has been deposited in host memory and can be used. It is possible that a flood of unexpected messages can overrun this EQ. The *other* EQ is used for tracking of *other* events. These include sending of data from the send side. The sender needs to know when it is safe to reuse a buffer. This is indicated by receipt of a *send end event* for short messages. For long messages utilizing the default GET protocol (receiver pulls the data), the buffer can only be reused upon receipt of a *reply end event*. The *other* EQ is also used to handle pre-posted receives and MPI-2 RMA requests. EQs will be discussed further in Section 5.

XT3 MPI-2 RMA Implementation

The XT3 MPICH2 uses a Portals-based variant of the approach taken in the CH3 ADI3 device for supporting MPI-2 remote memory access (RMA). In this approach, one-sided operations are simulated using two-sided send/receive messaging. To avoid interference with the MPI-1 machinery described in the previous section, a duplicate set of MEs for unexpected short RMA messages is at job start-up. There is no eager long protocol for MPI-2 RMA. The intent of the implementation is to provide MPI-2 RMA functionality.

MPI-2 RMA accesses are organized around exposure epochs and windows[1]. An application marks the beginning of an exposure epoch using one of the RMA synchronization functions: *MPI_Win_fence*, *MPI_Win_start/MPI_Win_post*, or *MPI_Win_lock*. The application can then begin using RMA access calls: *MPI_Put*, *MPI_Get*, *MPI_Accumulate*. In the XT3 MPICH2 implementation, no data motion actually occurs in these calls. Almost all data motion occurs at the end of an epoch, *e.g.* in a *MPI_Win_fence* call. At this point all processes involved in the epoch determine how many updates from calls to RMA access functions by the other processes need to be processed. Each process then examines the list of RMA access requests posted locally by the application and begins building RMA messages. For accesses that involve derived data types, this includes packing information about the data type into the message. If the access request is a PUT or ACCUMULATE, the application data is also packed into the message. The message is then sent using approaches similar to that for MPI-1 send/receive operations described in the previous section. Receivers unpack these messages and for PUT or ACCUMULATE operations, update the target region of the window. For GETs, the message is unpacked to determine which region(s) of the window are being accessed. Any derived data information is unpacked and used to reconstitute the data type used in the original *MPI_Get* call. The appropriate region(s) of the window are then packed into a buffer using this data type information. This buffer is then delivered as a message back to the original requestor.

The performance of MPI-2 RMA on XT3 should be similar to that provided by other MPICH2 channels implementations, though better efficiency results from more

efficient allocation of some of the internal data structures specific to the Portals implementation.

XT3 MPI-IO(ROMIO)

The *liblustre* implementation provides fairly strong file system consistency semantics. This allows a straightforward MPICH-2 ROMIO implementation on Catamount. Since record locking is available, non-contiguous I/O optimizations are possible (this is not possible for example, when PVFS is used).

3. XT3 SHMEM Implementation

As with the MPI implementation on XT3, the SHMEM implementation relies on the Portals network protocol stack to transfer data between processes. The SHMEM implementation on XT3 has been presented previously [6]. An overview of the implementation is given here.

SHMEM uses Portals in a more static manner than MPI. At job start-up, the SHMEM library examines segments of the application process: data segment, symmetric heap, private heap, and stack. The base address and lengths of all segments are defined at job startup, and are the same across all processes running the same executable. The data segment and symmetric heap are the only remotely accessible program segments supported currently with XT3 SHMEM. Only processes running the same executable in a MPMD job can use SHMEM for interprocess communication.

Three *memory descriptors* (MDs) are used for the data segment and three for the symmetric heap. Two of these MDs are associated with event queues, one for GET and one for PUT operations. The third is not associated with any EQ, but is bound to a *match entry* (ME). It is also configured to generate acknowledgements when a transfer has completed, and is globally visible in the target node memory. An initiator's Portals PUT and GET requests target this ME at a remote node. The three MDs for the data segment and the three for the symmetric heaps are marked as persistent and *source-side* managed. By not associating the target ME/MD with an event queue, generation of events at the target is avoided. This fits with the one-sided program model SHMEM supports. Two MDs are associated with the private heap and stack. Again, this is to allow for separate tracking of PUT and GET requests, which in turned facilitates implementation of non-blocking SHMEM puts and gets in subsequent releases. The

Figure 2. Layout of SHMEM processes and Portals Match Entry (MEs), Memory Descriptors (MDs), and Event Queues.

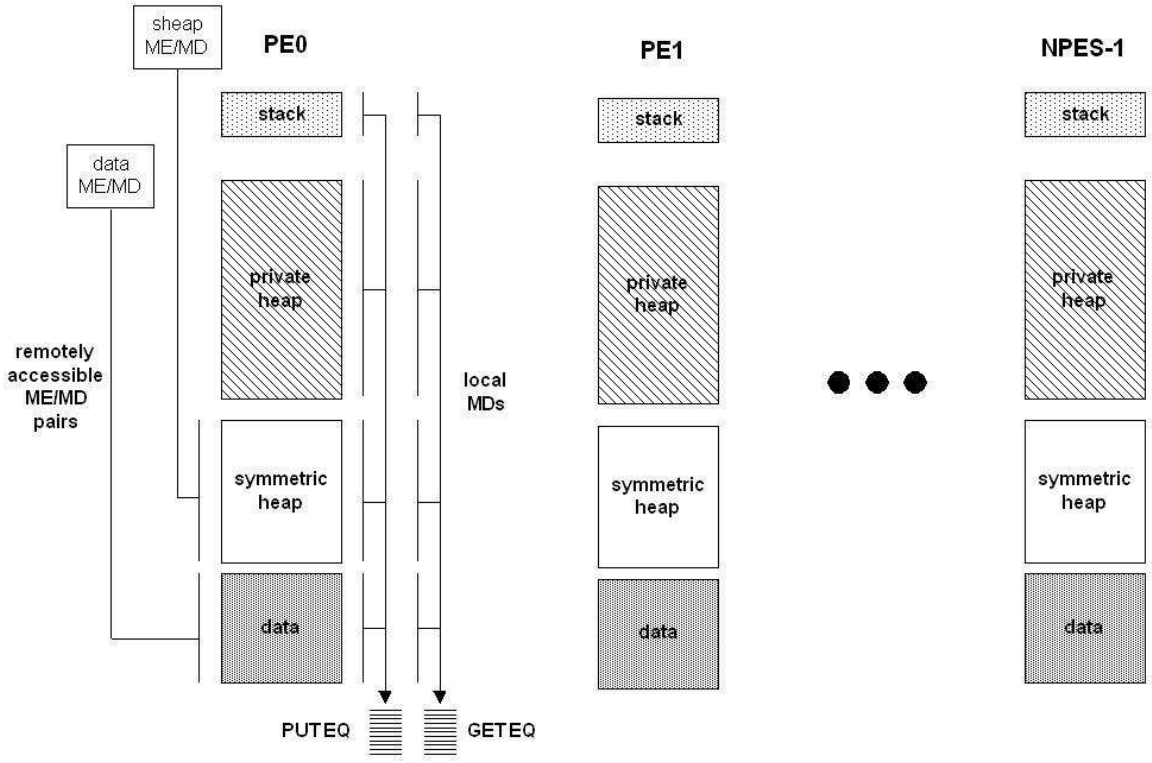

Table 1. Important XT3 MPI environment variables in XT3 MPT 1.3 and 1.4 releases.

EnvironmentVariable	Description	Default
MPICH_MAX_SHORT_MSG_SIZE	Sets the maximum size of a message in bytes that can be sent via the short (eager) protocol.	128000 bytes
MPICH_MAX_VSHORT_MSG_SIZE	Specifies in bytes the maximum size message to be considered for the vshort path.	1024 bytes
MPICH_PTL_UNEX_EVENTS	Sets the number of event queue entries for unexpected MPI point-to-point messages.	20480 entries
MPICH_PTL_OTHER_EVENTS	Sets the number of entries in the event queue that is used to receive all MPI-related Portals events other than for unexpected messages.	2048 entries
MPICH_UNEX_BUFFER_SIZE	Overrides the size of the buffers allocated to the MPI unexpected receive queue. This value is interpreted as the number of bytes or megabytes, if the string ends in M, to be allocated to buffers associated with this queue.	60M bytes
MPICH_VSHORT_BUFFERS	Specifies the number of 16384 byte buffers to be pre-allocated for the send side buffering of messages for the vshort protocol.	32 buffers

acknowledgments mentioned above are used by the initiator to track outstanding PUT and GET requests. The job layout is depicted in Figure 2.

In sharp contrast to MPI, SHMEM only uses event queues on the initiating side of a PUT or GET operation. In principle this can lead to fewer difficulties with scalability issues concerning event queues at high process counts, since there is no generation of events at remote nodes owing to PUT or GET activity at an initiator.

Compared to other CRAY platforms, the Portals network stack does allow for significant offloading of data motion from the host processors. To leverage this capability, a non-blocking variant of PUTs has been introduced into XT3 SHMEM. Future releases will include support for non-blocking GETS as well.

4. Performance Tuning and Pitfalls to Avoid for MPI and SHMEM on XT3

MPI Performance Tuning and Pitfall Avoidance

One of the main goals of the Portals network protocol stack is to allow for progress of any state associated with MPI requests made by an application as independent of application activity as possible. So typically, structuring an application to allow for maximum progress of this MPI state without the need for additional MPI calls by the application can lead to better performance.

Pre-posting Receives

One way for applications to insure better independent progress of MPI-related state is to structure algorithms so that a receive is guaranteed to be posted before the matching message arrive. This allows for the Portals stack to match the message with the application buffer associated with the message, and deliver the incoming message directly into this buffer. Among other things, this reduces the need for additional memory copies at the receiver side. Applications, which are structured in this manner, should also be run with the MPICH_PTL_EAGER_LONG environment variable set.

In general non-blocking sends and receives are preferred on XT3. There are no particular performance advantages for persistent send or receive requests.

For the same reason that the Portals protocol stack delivers best performance when receives are pre-posted, it is generally not desirable to use *MPI_Probe* or *MPI_Iprobe* in performance critical sections of an application, as this approach to handling messages basically eliminates much of the benefits of overlapping communication with computation for which Portals is optimized.

Note that pre-posting of receives does consume certain limited Portals resources (MEs). There is currently a hard limit of 2048 active MEs on Catamount nodes. Trying to pre-post more receives than this ME resource limitation allows will result in the job aborting. Thus, although it is

Table 2 Environment Variables for tuning MPICH2 Collectives. Available in XT3 MPT release 1.4.

EnvironmentVariable	Description	Default
MPICH_ALLTOALL_SHORT_MSG	Adjusts the cut-off point for which the store and forward Alltoall algorithm is used for short messages.	512 bytes
MPICH_BCAST_ONLY_TREE	Setting to 1 or 0, respectively disables or enables the ring algorithm in the implementation for MPI_Bcast for communicators of nonpower of two size.	1
MPICH_REDUCE_SHORT_MSG	Adjusts the cut-off point for which a reduce-scatter algorithm is used. A binomial tree algorithm is used for smaller values.	64K bytes
MPICH_ALLTOALLVW_SENDRECV	Disables the flow-controlled Alltoall algorithm. When disabled, the pairwise sendrecv algorithm is used which is the default for messages larger than 32K bytes. Setting this variable may avoid situations where the flow-controlled Alltoall algorithm causes event queue overflow.	not enabled

good to pre-post receives in general, care should also be taken not to pre-post too many of them.

Table 1 gives XT3 MPI environment variables that most typically require tuning for applications. These environment variables are defined for both the 1.3 and 1.4 releases of XT3 MPT.

Derived Datatypes

Compared to the handling of derived datatypes in MPICH1, techniques employed in MPICH2 for derived types are much improved. Nonetheless, usage of derived types does result in extra overhead on both the send and receive sides. On the send side, a temporary buffer must be allocated and the message data packed into this buffer before being delivered to the Portals network stack. Similarly at the receive end, if the message is not found in the unexpected queue, a temporary buffer must be allocated into which the incoming message data is delivered. The data then has to be copied out of this temporary buffer into the application buffer. There is no special hardware support on XT3 for handling derived types. In general it is better to avoid derived types on XT3 if possible.

Usage of Collective Operations

Optimization of MPI collectives in XT3 MPICH2 is a work in progress (see Section 5). It is expected that the performance of commonly used collectives will improve in future releases of XT3 MPT.

That having been said, it should be noted that to some degree, frequently observed usages of collective operations conflict with one of the design goals of Portals, the independent progress of MPI state and the application. If analysis of an application indicates substantial time being spent in a collective communication operation that is essentially used for moving data following a computational cycle, it may be worth investigating if the computational cycle could be changed to allow for point-to-point communication calls to be embedded within the computational cycle. This approach has a chance of allowing for overlap of communication with computation.

MPI-2 RMA

As described in Section 2, the MPI-2 RMA implementation in the XT3 MPICH2 library is intended to provide functionality, not performance. Usage of MPI-2 RMA is currently discouraged on XT3.

Debugging Related Environment Variable

The MPICH_DBMASK environment variable (MSMDB_MASK in XT3 MPT 1.3 and earlier releases) can be used to assist in debugging MPI runtime related problems such as argument checking, EQ overflow, etc. Setting this environment variable to 0x200 will, for example, cause the application to abort and give a coredump and traceback when an MPI–related error occurs. Such errors include invalid arguments to MPI function calls, exhaustion of Portals or MPI internal resources, etc.

SHMEM Performance Tuning and Pitfall Avoidance

For XT3 the most important point with respect to SHMEM is probably whether or not to use it. Applications being ported from legacy CRAY platforms may have been using SHMEM owing to its significantly lower overhead than MPI for many types of data exchange scenarios. On the XT3 however, the number of scenarios where SHMEM may offer lower overhead than MPI is significantly reduced. Thus if both an MPI and SHMEM version of an application exist, it may be more worthwhile to first port the MPI version to XT3.

Usage of SHMEM Barrier

The SHMEM barrier functionality is implemented in software on XT3, and has relatively high overhead. Usage of barriers should be kept to a minimum. Figure 2 shows the performance of *shmem_barrier_all* as a function of process count when using 1 and 2 processes per node.

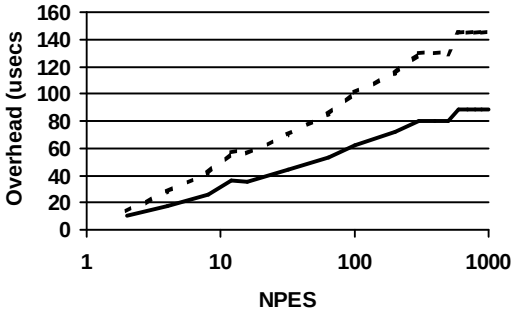

Figure 2 Shmem_barrier_all overhead when running two processes per node(dashed line) and one process per node (solid line).

Using non-blocking Puts

As mentioned in Section 3, non-blocking SHMEM functions are being introduced in the XT3 SHMEM implementation to take advantage of the Portals network protocol stack offload capability. In general, better performance will be obtained by using non-blocking SHMEM_PUTs. The current mechanisms for insuring completion of a non-blocking put operation are either by invoking the *shmem_quiet* function, one of the various SHMEM atomic functions, or one of the SHMEM barrier functions. To avoid exhaustion of the PUT event queue in SHMEM, one of these functions should be called periodically when invoking many non-blocking SHMEM_PUTs.

Strided Gets and Puts

The Portals network protocol stack on XT3 is optimized for block transfers. However, support for generalized gather/scatter remote memory operations is very poor, being over two orders of magnitude slower than block transfers of the same amount of data. Hence performance for strided SHMEM GET and PUT operations is poor on XT3. These functions should not be used in performance critical parts of SHMEM applications on XT3.

Spin Waiting on Remote Variables

One occasionally observes constructs like the following in SHMEM codes ported from other CRAY platforms:

```
while(remval != 0) {
shmem_get64(&remval,&rem_flag,1,pe);
}
```

This type of construct can severely tax the Portals network protocol stack, particularly if many processes are spinning on a variable at a single target process (PE). If possible, other synchronization mechanisms relying on spinning on local memory should be employed [7].

5. Issues and Future Work

MPI Flow Control-Related Issues

A significant number of MPI applications are experiencing resource exhaustion issues (particularly the *unexpected EQ*) as job sizes increase. Applications employing techniques such as dynamic load balancing which generate many-to-many communication patterns involving small messages appear to be especially susceptible to this issue. The problem stems from the assumptions at the send side concerning resources at the receive side for short and long messages described in Section 2. CRAY is actively investigating solutions to this problem that will allow for such applications to run satisfactorily without exhausting Portals resources

Collective Communications

Optimization of frequently used MPI collective operations on XT3 is a major priority for the MPT group. Efforts are currently focused on

- tuning of the existing MPICH2 algorithms which exhibit problematic performance issues at higher process counts
- SMP aware algorithms to deliver better performance when using multi-core processors
- developing better algorithms for latency dominated *alltoall* and *allgather* operations involving small messages
- topology-aware algorithms for better scaling of bandwidth intensive operations like *alltoall* at high process counts

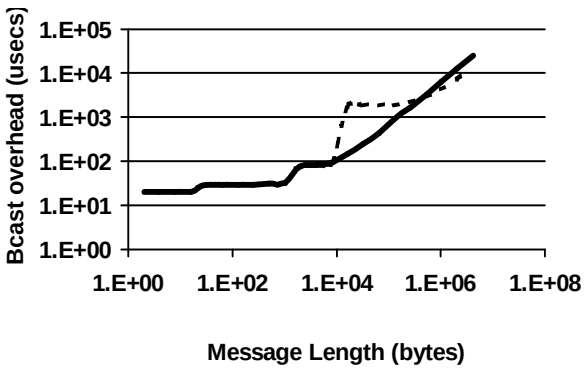

Figure 3 MPI_Bcast overhead using the default MPICH2 algorithms(dashed line) and a binomial tree only algorithm (solid line). Results from Pallas benchmark on 100 processors.

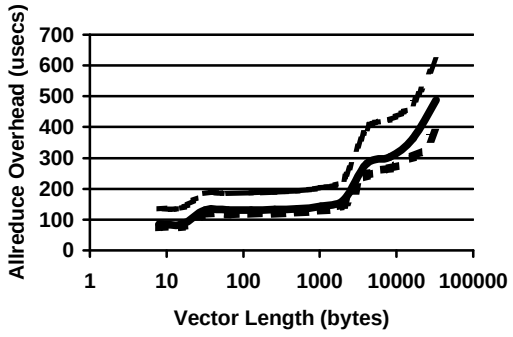

Figure 4 MPI_Allreduce using SMP aware (solid line) and default MPICH2 allreduce algorithm at 2 processes per node using 128 nodes(long dashed line). Results for 1 process per node are given by the short dashed line. Results of PMB Allreduce test.

Some of the work involving tuning of existing MPICH2 algorithms will be available in the upcoming 1.4 release of MPT. In particular, the performance of MPI_Bcast and MPI_Reduce for larger message sizes is improved over that realized in the 1.3 release for certain message lengths. See Figure 3. Table 2 gives a list of new environment variables to assist in tuning the current MPICH2 collective communication algorithms for XT3.

SMP-aware algorithms for collective operations are also under investigation and will be available in an upcoming release of XT3 MPT. These optimizations will be further improved by the availability of a *cut-through* path in Portals when exchanging data between processes on the same node. See Figure 4.

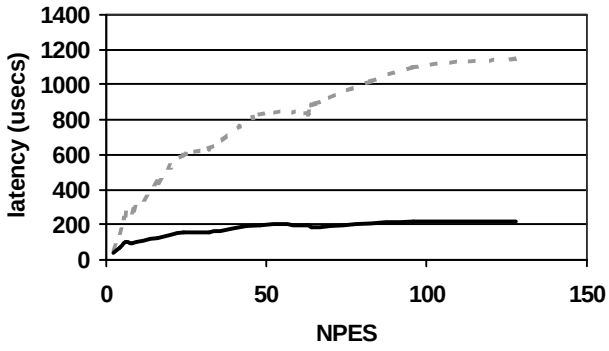

Figure 5 Comparison of latency for shmem_int_sum_to_all in current 1.3 release(dashed line) and 1.5 pre-release (solid line).

Work is also on-going in optimizing SHMEM collective operations. Performance of the *shmem_X_sum_to_all* will be significantly improved in the 1.5 release. See Figure 5.

SHMEM Atomics

SHMEM atomic operations are fairly widely used in more complex SHMEM applications. Their functionality is hard to replace with other constructs. Some SHMEM atomic functions supported on other current CRAY platforms will be available in the XT3 MPT 1.4 release.

The performance of the various atomic operations are similar in terms of latency and repetition rate. Since these operations are implemented in software – the Portals network protocol stack – rather than hardware, latencies are substantially higher, and the repetition rates significantly lower, than on other CRAY platforms. Figure 6 shows the latency for a SHMEM_FADD operation in which increasing numbers of processes are trying to update the same variable on a given target node.

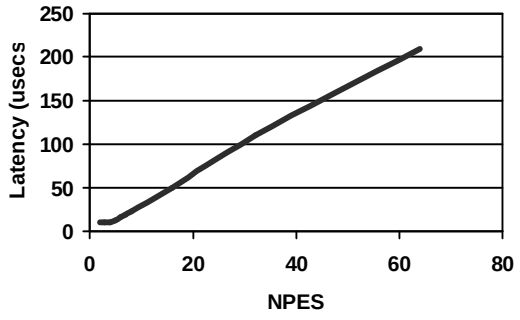

Figure 6 Latency of shmem_fadd as a function of PEs updating one variable.

Accelerated Portals

The MPT components will be modified to best make use of the accelerated path through the Portals network protocol stack on Catamount compute nodes when it becomes available. Concurrent with any such effort, the MPT group will also pursue reducing the MPICH2 component to short message latency.

About the Authors

The authors are members of the Message Passing Toolkit group at Cray. Howard Pritchard can be reached at howardp@cray.com.

References

[1] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. MPI – The Complete Reference, vol. 2, The MPI Extensions, MIT Press, 2nd edition, 1998.

[2] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol Building Blocks for Low Overhead Communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters,* April 2002.

[3] R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K. Underwood. Implementation and performance of Portals 3.3 on the Cray XT3. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing,* September 2005.

[4] R. Brightwell, A. B. Maccabe, and R. Riesen. Design, Implementation, and Performance of MPI on Portals 3.0. In *Internationl Journal of High Performance Computing Applications.* Vol. 17, No. 1 (2003).

[5] W. Gropp and E. Lusk. MPICH2: A High Performance Portable Implementation of MPI, Clusterworld 2004. http://www.clusterworld.com/CWCE2004/William_Gropp_presentation.pdf

[6] M. ten Bruggencate. Cray SHMEM on XT3. In *CUG Summit 2005 Proceedings.*

[7] J. H. Mellor-Crummey, and M. L. Scott. Algorithms for scalable synchronization on shared memory multiprocessors. In *ACM Trans. Computer Systems.* Vol. 9, No. 1 (1991).