Evaluation of Remote Memory Access Communication on the Cray XT3

V. Tipparaju A. Kot J. Nieplocha
Pacific Northwest National Laboratory

M. ten Bruggencate
Cray, Inc.

N. Chrisochoides
College of William & Mary

Abstract

This paper evaluates remote memory access (RMA) communication capabilities and performance on the Cray XT3. We discuss properties of the network hardware and Portals networking software layer and corresponding implementation issues for SHMEM and ARMCI portable RMA interfaces. The performance of these interfaces is studied and compared to MPI performance.

1. Introduction

Remote memory access (RMA) communication is an important communication paradigm for modern systems. RMA operations offer support for an intermediate programming model between message passing and shared memory. RMA model combines some advantages of shared memory, such as direct access to shared/global data, and the message-passing model, namely the control over locality and data distribution. RMA is sometimes considered a form of message passing; however, an important difference over the MPI-1 message-passing model is that RMA does not require explicit receive operation and thus offers increased asynchrony of data transfers. RMA has been implemented by system vendors along with the traditional message passing interfaces of MPI. Perhaps the most well known of the vendor devised RMA interfaces is the Cray SHMEM that was introduced on the Cray T3D 12 years ago and since then implemented by multiple vendors including Cray, SGI, IBM, and Quadrics. A special form of RMA has been added to the MPI-2 standard. However, the MPI-2 interface introduces more synchronization than any other existing RMA interfaces and still has not been widely adopted by applications. In addition to the direct use of RMA libraries in applications, RMA is used to implement global address space programming models such as Co-Array Fortran, UPC, Global Arrays or X10 language under development by IBM. These programming models are critically dependent on the quality of RMA implementation on the parallel systems.

The current paper discusses implementation issues of RMA communication over Portals layer and evaluates performance of RMA interfaces on the Cray XT3: the Cray SHMEM, MPI-2 and ARMCI which were implemented directly over Portals. We also include in our performance evaluation study MPI-2 one-sided operations; however, we note that it is not recommended by Cray to use MPI-2 on XT3. We used a set of basic microbenchmarks to evaluate the data transfers operations provided by these interfaces.

Since the RMA model promises performance advantages due to its higher level of asynchrony over message passing, we also evaluate whether this feature enables effective overlapping of communication with computation on the XT3. The availability of nonblocking RMA operations presents additional opportunities for overlapping data transfers and computations. Although prefetching and poststoring instructions are often supported by the shared memory h/w and are exploited by compilers to overlap computations with data movement, a scientific programmer on shared memory systems typically faces difficulties when attempting to explicitly overlap computations and communication due to the lack of precise APIs. Such explicit nonblocking APIs are present in the most RMA interfaces.

Finally, we use NAS MG to establish whether RMA implementation of this benchmark can be competitive to the standard MPI-1 version distributed by NAS.

2. Cray XT3 Network hardware and software

2.1 Seastar Network

The Cray XT3 uses the Seastar network interconnect. It is a full system-on-chip design that integrates high-speed serial links, a 3-D router, with network interface functionality. The network interconnect includes an embedded PowerPC processor, in a single chip. The Seastar, initially, was designed specifically to support the ASCI applications on the Sandia Red Strom system. On the Seastar network, there are 2 DMA engines, one for

1-4244-0910-1/07/$20.00 © 2007 IEEE.

sending and the other for receiving, that interact with a router that supports a 3-D torus interconnect and the HyperTransport (HT) cave that provides an interface to the AMD Opteron processor(s) and the host memory. The embedded processor is provided to program the DMA engines and assist with other network-level processing needs. The DMA engines provide robust support for transferring data between the network and host memory by providing hardware support for breaking each outgoing message into 64 byte packets and re-assembling incoming messages. One reason behind the packetization is to allow incoming packets from several different messages from distinct sources to be interleaved. The Seastar has a hardware mechanism to match incoming packets to their appropriate message stream, but the number of concurrent streams that can be processed is limited to 256. This is a clear limitation but the Portals implementation tries to address it for this network.

2.2 Portals network Interface

The Portals network programming interface is intended to allow scalable, high-performance network communication between nodes of a parallel computing system [1]. The Portals API was adopted by Cray for their XT3 supercomputer and forms the lowest programming API for the Seastar network. MPI-2, ARMCI and CRAY SHMEM have been implemented on top of Portals which is the high-performance low level network programming interface provided by Cray for their SeaStar interconnect on the XT3 platform.

The following are the properties of the Portals network programming interface that were envisioned for highly scalable network architecture: 1) connectionless to maximize scalability 2) independent of network h/w to achieve portability 3) user-level flow control and OS bypass for low latency communication, 4) receiver managed message passing to minimize memory consumption, 5) ability to handle unexpected messages to support MPI.

The Portals layer provides RMA data movement operations, but unlike other RMA programming interfaces, the target of a remote operation is not a virtual address. Instead, each message contains a set of match bits that allow the receiver to determine where incoming messages should be placed. This flexibility allows Portals to support both traditional RMA operations and two-sided send/receive operations. A target process can choose to accept message operations from any specific process or can choose to ignore message operations from any specific process.

Data movement. A Portal represents an opening in the address space of a process. Other processes can use a Portal to read (get), write (put), or atomically swap (getput) the memory associated with the portal. Every data movement operation involves two processes, the *initiator* and the *target*. The initiator is the process that initiates the data movement operation. The target is the process that responds to the operation by either accepting the data for a put operation, replying with the data for a get operation, or both for a swap operation.

In addition to the standard address components (process id, memory buffer id, and offset), a Portal address includes a set of match bits. This addressing model is appropriate for supporting one-sided operations as well as traditional two-sided message passing operations. Specifically, the Portals API provides the flexibility needed for an efficient implementation of MPI-1, which defines two-sided operations with one-sided completion semantics.

Figure 1 presents a graphical representation of the structures used by a target in the interpretation of a Portal address. The process id is used to route the message to the appropriate node. The memory buffer id, called the *portal id*, is used as an index into the Portal table. Each element of the Portal table identifies a match list. Each element of the match list specifies two bit patterns: a set of "don't care" bits, and a set of "must match" bits. In addition to the two sets of match bits, each match list element has at most one memory descriptor (MD). Each MD identifies a memory region and an optional event queue (EQ). The memory region specifies the memory to be used in the operation and the EQ is used to record information about these operations.

Figure 2 illustrates the steps involved in translating a Portal address, starting from the first element in a match list. If the match criteria specified in the match list entry are met and the MD list accepts the operation, the operation (put, get, or swap) is performed using the memory region specified in the MD. If the MD specifies that it is to be unlinked when a threshold has been exceeded, the match list entry is removed from the match list and the resources associated with the MD and match list entry are reclaimed. Finally, if there is an EQ specified in the MD and the MD accepts the event, the operation is logged in the EQ. Associating a MD with an EQ is optional.

Access Control. A process can control access to its

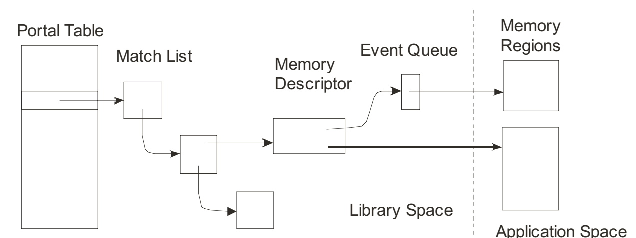

Figure 1: Portal Addressing Structures

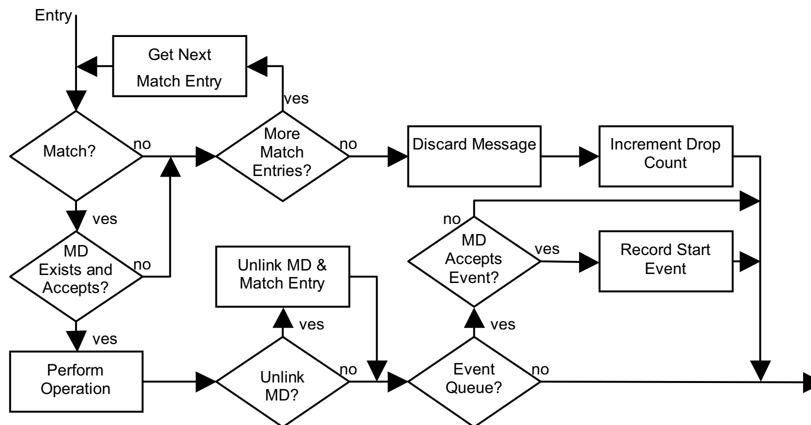

Figure 2: Portal Address Translation

Portals using an access control list. Each entry in the access control list specifies a process id, possibly a job id, a user id, and a Portal table index. The access control list is actually an array of entries. Each incoming request includes an index into the access control list (i.e., a "cookie" or hint). If the id of the process issuing the request doesn't match the id specified in the access control list entry or the Portal table index specified in the request doesn't match the Portal table index specified in the access control list entry, the request is rejected. Process identifiers, job identifiers, user identifiers, and Portal table indexes may include wildcard values to increase the flexibility of this mechanism.

Receiving messages. The Portals API uses four types of messages: put requests, acknowledgements, get requests, and replies. When an incoming message arrives on a network interface, the communication system first checks that the target process identified in the request is a valid process that has initialized the network interface (i.e., that the target process has a valid Portal table). If this test fails, the communication system discards the message and increments the dropped message count for the interface. The remainder of the processing depends on the type of the incoming message. Put and get messages are subject to access control checks and translation (searching a match list), while acknowledgement and reply messages bypass the access control checks and the translation step.

Acknowledgement messages include a handle for the MD used in the original *PtlPut* or *PtlPutRegion* operation. This MD will identify the EQ where the event should be recorded. Upon receipt of an acknowledgement, the runtime system only needs to confirm that the MD and EQ still exist and that there is space for another event. Should the any of these conditions fail, the message is simply discarded and the dropped message count for the interface is incremented. Otherwise, the system builds an acknowledgement event from the information in the acknowledgement message and adds it to the EQ.

3. SHMEM

Cray Research Inc. introduced the Cray SHMEM library in 1993 to support the global address space programming model of the Cray T3D massively parallel system. This was the first scalable system with hardware that allowed any process was able to read or write data from any other process at any time. Since then, Cray SHMEM has been carried forward to all subsequent Cray platforms. The Cray SHMEM library supports a wide set of functionality in the area of RMA communication, including

1. Data transfer operations e.g., *shmem_put* and atomic operations e.g., *shmem_swap*
2. Collective data transfer operations e.g., *shmem_sum_to_all*
3. Initialization and information operations e.g., *shmem_init, shmem_my_pe*
4. Synchronization operations e.g., *shmem_barrier, shmem_quiet*

On Cray XT3 systems, the Cray SHMEM library is implemented on top of the Portals, using Version 3.3 of the Portals API [1]. At start-up Cray SHMEM has to prepare memory for communication. To minimize overhead on the critical data transmit path, Cray SHMEM sets up all Portals resources, which it will use throughout the execution of an application at this time. The resources are set up statically and are reused for the duration of the job run, thereby avoiding unnecessary system calls on the transmit path. Data accessed by a Cray SHMEM data transfer operation can reside in any of the following four memory segments: data segment, private heap, symmetric heap and stack. Each of these memory segments is bound to the Portals device by a distinct MD to enable access to the memory segment via Cray SHMEM data transfer calls. The start address and length of each memory segment are supplied by the Catamount loader. Only two of the four

memory segments are symmetric on Cray XT3 and thus remotely accessible in the context of SHMEM, namely the data segment and the symmetric heap. One match list element per symmetric memory segment is allocated so that the memory segment becomes remotely accessible via an associated Portals Index. Cray SHMEM reserves two Portals Indices, one for the data segment and one for the symmetric heap, in order to minimize the time spent searching through match lists during data transfers. The MDs describing the stack and private heap remain free-floating. EQs are allocated to allow monitoring the completion of transfers. Specifically, separate EQs are allocated for Cray SHMEM Put and Cray SHMEM Get operations. This approach allows for easy separation of and monitoring for Put and Get related events.

Cray SHMEM uses Portals communication calls to implement data transfer and synchronization operations. Once the application initiates a Cray SHMEM data transfer operation, the source and target addresses and the length of the data transfer supplied to the call determine which MD and associated EQ, local and remote offsets and Portals Index to supply to the appropriate Portals routine. Also, the target PE number is translated into the target Portals Process ID. After returning from the Portals routine called, Cray SHMEM monitors EQs to determine when it is safe to return control to the application. Strided Get and Put operations are implemented as a series of contiguous Cray SHMEM Put or Get operations. All Cray SHMEM Put calls request acknowledgement events to support global synchronization. Throughout the execution of an application Cray SHMEM keeps track of the number of outstanding Put operations. Global synchronization is achieved by waiting for acknowledgement events to arrive on the initiator for all outstanding Put operations.

4. ARMCI

Aggregate Remote Memory Copy Interface (ARMCI) [11] is a portable RMA communication library that can be used in MPI applications. It was also used to implement several parallel programming models such as CAF, Global Arrays or GPSHMEM [12]. ARMCI offers an extensive set of functionality in the area of RMA communication: 1) data transfer operations 2) atomic operations 3) memory management and synchronization operations, and 4) locks. In scientific computing, applications often require transfers of noncontiguous data that corresponds to fragments of multidimensional arrays, sparse matrices, or other more complex data structures. The noncontiguous interfaces are available in ARMCI to address these needs. ARMCI offers blocking and nonblocking data transfer operations.

ARMCI implementation of Portals has to: 1) prepare the memory for communication, 2) use Portals communication calls to implement ARMCI put, get and accumulate functionality for contiguous, strided and vector data types and 3) implement atomic read modify write operations

One of the fundamental tasks for the ARMCI implementation over Portals involves preparation for the local and remote memory used in communication. ARMCI communication library requires all remote memory used for communication to be allocated with ARMCI memory allocator. This enables ARMCI library to prepare for remote communication to and from that memory. For supporting communication to and from a remote memory, ARMCI implementation over Portals relies on a "wild card" MD. Portals message passing interface requires a corresponding descriptor on the remote end for every communication call. Because most ARMCI calls are one-sided in nature and don't involve explicit remote process involvement, ARMCI implementation on top of the Portals message passing layer post an anticipatory wild card remote MD. These descriptors are posted such that they don't have any association to any EQs (See section 2.2).

For the local memory used in communication operations, there could be three kinds of memory: 1) ARMCI global memory allocated with the collective ARMCI_Malloc call 2) ARMCI local memory allocated with ARMCI_Malloc_local call and 3) user memory which could either be static or allocated with malloc. In order to allow communication from these three kinds of memory, we use different sets of MD. A Portals MD can optionally be either retained or unlinked upon the completion of the communication call. Retaining a MD allows us to quicken communication calls by merely modifying a MD for all future communications. Allowing it to be unlinked frees up NIC resources but requires the MD to be bound to the EQ for every communication call. In ARMCI implementation, MDs describing Global memory are retained when possible and local MDs are allowed to be unlinked upon completion. This is primarily because we don't have a way to determine how many *malloc* calls a user might do or how many static memory segments the user might use. All descriptors posted for local memory are associated with EQs to enable the ARMCI library for tracking completion of communication operations that use these MDs.

First, the ARMCI implementation tries to find a corresponding MD for the memory used by the initiator of the call in Get or a Put operation to check if it is already retained (see section 2.2). If the MD is retained, it checks to ensure the descriptor doesn't have a pending operation on it and subsequently updates the descriptor. In the case that the descriptor is not retained, it binds the descriptor. After this a Portals memory transfer operation is initiated (via a PtlGetRegion or a PtlPutRegion call). Vector and strided Get or Put operations are implemented as a series

of contiguous Portals Put or Get calls. The Catamount Kernel is the only supported kernel even as to date on the Cray XT3 platform. This kernel doesn't support user level threads. ARMCI uses either of the Owner Computes or Caller Computes [14] methods. The Owner computes method requires either explicit support from the network layer or the ability to run a user level thread. Since both these options are not available, ARMCI Accumulate operation currently uses the caller computer mode.

5. MPI-2 One-sided

RMA extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. Message-passing communication achieves two effects: communication of data from sender to receiver; and synchronization of sender with receiver. The RMA design separates these two functions. Three communication calls are provided: MPI_PUT, MPI_GET and MPI_ACCUMULATE (remote update). RMA communications fall in two categories: active target communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication and passive target communication, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer.

The Cray XT3 MPICH2 uses a Portals-based variant of the approach taken in the MPICH2 CH3 ADI3 device to support MPI-2 remote memory access (RMA) [5]. One-sided operations are simulated using two-sided send/receive messaging. MPI-2 RMA accesses are organized around exposure epochs and windows [6]. 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 Cray XT3 MPICH2 implementation, no data transfer actually occurs in these calls. Almost all data transfer occurs at the end of an epoch, e.g. in an 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 examines the list of RMA access requests posted locally by the application and begins to build 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 send/receive operations in the Cray XT3 MPICH2 [7]. 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.

6. Performance Evaluation

We performed our experiments on a Cray XT3 supercomputer with 2.6 GHz dual-core AMD Opteron nodes with 4GB of memory. Each node is connected to a Cray Seastar router through Hypertransport, and the Seastar NICs are all interconnected in a 3D-torus topology. The system is located at the National Center for Computational Sciences, Oak Ridge. It uses UNICOS/lc 1.5 operating system (includes SHMEM), Portals library version 3.3 and MPICH2 version 1.0.2. The experiments included several micro benchmarks to evaluate different parameters of the communication operations. In addition, the MG NAS benchmark was used to evaluate performance in of RMA in application context on the Cray XT3.

6.1 Micro benchmarks

The motivation for the experiments described in this section was to demonstrate the performance of the implementation at the system level. The next section shows how much of these gains can be leveraged at the application level.

Bandwidth test. The first experiment shows the effective bandwidth of blocking get and put operations for all three discussed communication libraries: ARMCI (over Portals), SHMEM and MPI. We measured execution time of ARMCI_Get, ARMCI_Put, shmem_getmem,

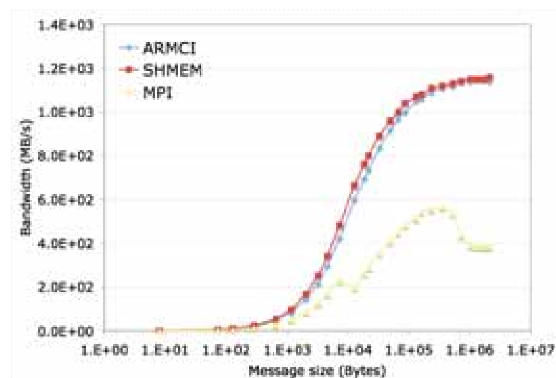

Figure 3. Effective Get bandwidth

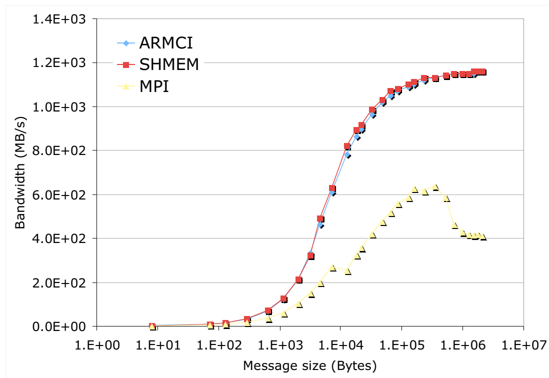

Figure 4. Effective Put bandwidth

shmem_putmem, MPI_Get and MPI_Put (see 3 and 4). For comparison we also show the performance of two-sided MPI communication using the ring benchmark (see Figure 5). The timings have been averaged over 1000 iterations. They show substantial difference between RMA performance of MPI vs. ARMCI and SHMEM. Figure 5 shows that performance of MPI one-sided operations are lagging performance of MPI two sided. The performance levels of MPI-2 active and passive models are equivalent. Moreover, the MPI-2 performance presented above is the upper bound of applications would experience. We found that the bandwidth results can deteriorate if remote process does not make MPI calls which indicates the progress engine is not implemented as truly one-sided.

Overlap test. The next experiment deals with overlapping communication with computation, and it was performed in the context of ARMCI and MPI-2 (Figure 6). Note that MPI_Get is not a part of MPI-1 standard. This is included in MPI-2 standard. This test is irrelevant for Cray SHMEM since it does not support a non-blocking get operation. The computation is incorporated in the program in the form of a delay. Increasing computation is gradually inserted between the initiating non-blocking get call and the wait completion call. As we keep increasing the computation, at some point the sum of the non-blocking

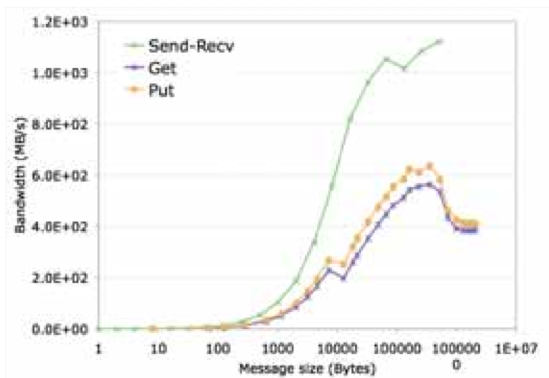

Figure 5. Effective MPI bandwidth

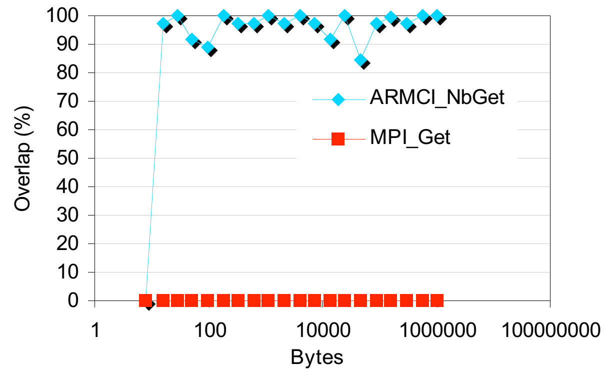

Figure 6. Maximum potential overlap

call issue overhead and computation would exceed the idle CPU time, so the total benchmark running time would increase. This point gives us the maximum possible overlap. We performed this experiment on two nodes, with one node issuing the non-blocking get for data located on the other and then waiting for the transfer to be completed in the ARMCI_Wait or the corresponding MPI Fence or Unlock calls (depending on active or passive mode). The experiment indicates that MPI implementation (active and passive models have same behavior) there is no overlap. The overlap potential in ARMCI is very good. These results are consistent with Cray's recommendation for not using MPI-2 interface in performance critical code sections. Fluctuations on ARMCI Get overlap percentage is primarily because of the fluctuations in the measured time for the computation incorporated in the overlap test as mentioned above.

6.2 MG Benchmark

The NAS-MG multigrid benchmark solves Poisson's equation in 3D using a multigrid V-cycle. MG benchmark carries out computation at a series of levels and each level of the V-cycle defines a grid at a successively coarser resolution [15]. The NPB 2.4 code uses a three-step

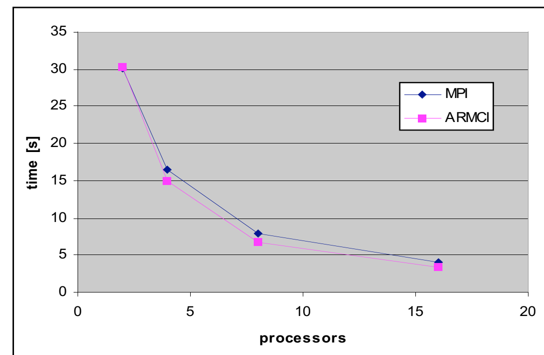

Figure 7. Comparing MPI-1 and ARMCI RMA implementation of NAS MG benchmark

dimensional exchange algorithm to satisfy boundary conditions. This is implemented with point-to-point message passing communication. In addition to this, point-to-point communication is used in the parallel implementation of these stencils to update every processors boundary values for each dimension that is distributed.

Our primary modification involved replacing these point-to-point communications with ARMCI PUT operations. Figure 7 shows that RMA communication can be effective for applications on the Cray XT3. The graph in Figure 7 compares the performance of the NAS benchmarks implemented by NASA AMES based on the MPI-1 message passing standard with the ARMCI implementation for Class B [15] problem size of 256X256X256.

7. Conclusions and Future work

We have evaluated the implementation of three implementations of RMA communication libraries (ARMCI, SHMEM, MPI-2) on the Portals network interface on the Cray XT3. We have shown the effectiveness of the RMA model through microbenchmarks and application benchmarks. ARMCI offers superset of all the one-sided communication operations in the MPI-2 and SHMEM standard. Performance of SHMEM and ARMCI are equivalent and significantly better than MPI-2 one-sided operations. We are investigating extentions to the Portals network layer to make it more general and better support RMA models. ARMCI shows over 80% overlap for all the message sizes tested. Performance of ARMCI and SHMEM implementations are comparable. MPI-2 RMA implementation, as recommended by Cray, is not yet ready to be used in performance critical sections of the code on the XT3 platform that the tests were performed on.

References

[1] Brightwell, Ron, Trammell Hudson, Kevin T. Pedretti, Rolf Riesen, Keith D. Underwood, "Portals 3.3 on the Sandia/Cray Red Storm System," Cray User Group, May 2005.

[2] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. June, 1995.

[3] Message Passing Interface Forum. MPI2: Extensions to the Message Passing Interface. July, 1997.

[4] R. Bariuso, Allan Knies, SHMEM's User's Guide, Cray Research, Inc., SN-2516, rev. 2.2, 1994.

[5] R. Thakur, W. Gropp, B. Toonen. Optimizing the Sychronization Operations in MPI one-sided communication. In International Journal of High Performance Computing Applications. Volume 19 , No.2 (2005).

[6] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. MPI - The Complete Reference, Vol2, The MPI Extensions, MIT Press, 2nd edition, 1998.

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

[8] R. Numrich, J.K. Reid, Co-Array Fortran for parallel programming. ACM Fortran Forum, 17(2):1-31, 1998.

[9] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Tech Report CCS-TR-99-157, Center for Computing Sciences, 1999.

[10] K. Parzyszek, J. Nieplocha and R. Kendall, A Generalized Portable SHMEM Library for High Performance Computing, Proc PDCS-2000, 2000.

[11] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. Panda. High Performance Remote Memory Access Comunications: The ARMCI Approach. International Journal of High Performance Computing and Applications, 20(2), 2006.

[12] J. Nieplocha, R.J. Harrison, R.J. Littlefield, “Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance computers”, The Journal of Supercomputing, vol 10, pp. 197-220, 1996.

[13] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit." International Journal of High Performance Computing and Applications, 20(2), 2006

[14] Jarek Nieplocha, Vinod Tipparaju, and Edoardo Apra, An Evaluation of Two Implementation Strategies for Optimizing One-Sided Atomic Reduction, in proc. of Communication Architecture for Clusters 2005 workshop of International Parallel and Distributed Processing symposium 2005, 2005

[15] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, The NAS parallel benchmarks, RNR-94-007, NASA 1994.