EXPLOITING DIRECT ACCESS SHARED MEMORY FOR MPI ON MULTI-CORE PROCESSORS

Ron Brightwell

SANDIA NATIONAL LABORATORIES, PO BOX 5800,
ALBUQUERQUE, NM 87185-1319, USA
(RBBRIGH@SANDIA.GOV)

Abstract

In 2008, the Catamount lightweight kernel was extended to support direct access shared memory between processes running on the same compute node. This extension, called SMARTMAP, allows each process read/write access to another process' memory by extending the virtual address of each process. Simple virtual address bit manipulation can be used to access the same virtual address in a different process' address space. In this paper we describe an implementation of the Message Passing Interface (MPI) that uses SMARTMAP for intra-node message passing. SMARTMAP has several advantages over POSIX shared memory techniques for implementing MPI. We present performance results comparing MPI using SMARTMAP to the existing MPI transport layer on a quad-core Cray XT platform.

Key words: MPI, shared memory, multi-core, lightweight kernel, operating system

1 Introduction

Catamount (Kelly and Brightwell, 2005) is a third-generation lightweight kernel developed by Sandia National Laboratories and Cray, Inc., as part of the Sandia/Cray Red Storm project (Camp and Tomkins, 2002). Red Storm is the prototype of the Cray XT series of massively parallel machines. Recently, Catamount was enhanced using a technique called SMARTMAP (Simple Memory of Address Region Tables for Multi-core Aware Programming). SMARTMAP allows the processes running on a compute node as part of the same parallel job to efficiently read and write each other's memory. Unlike POSIX shared memory, SMARTMAP allows a process to access another process' memory by simply manipulating a few bits in a virtual address. This mechanism has several advantages for efficiently implementing MPI for intra-node communication.

We have developed a prototype MPI implementation using Open MPI that is able to use SMARTMAP for intra-node communication. Initial performance results show that SMARTMAP is able to achieve significant improvement for intra-node point-to-point and collective communication operations. The following section describes the advantages of SMARTMAP compared with existing approaches for intra-node MPI. Section 3 provides a detailed description of the implementation of MPI communication using SMARTMAP, which is followed by a performance comparison between SMARTMAP and the existing transport for Red Storm. Relevant conclusions and an outline of future work are presented in Section 5.

2 Background

SMARTMAP takes advantage of the fact that Catamount only uses a single entry in the top-level page table mapping structure (PML4) on each core of a multi-core AMD Opteron processor. Each PML4 slot covers 39 bits of address space, or 512 GB of memory. Normally, Catamount only uses the first entry covering physical addresses in the range `0x0` to `0x007FFFFFFFFF`. The Opteron supports a 48-bit address space, so there are 512 entries in the PML4.

Each core writes the pointer to its PML4 table into an array at core 0 startup. Each time the kernel enters the routine to start a new context, the kernel copies all of the PML4 entries from every core into every other core. This allows every process on a node to see every other process' view of the virtual memory at an easily computed offset in its own virtual address space. The following routine can be used by a process to manipulate a "local" virtual address into a "remote" virtual address on a different core:

The International Journal of High Performance Computing Applications,
Volume 24, No. 1, Spring 2010, pp. 69–77
DOI: 10.1177/1094342009359014

Figures 1–3 appear in color online: http://hpc.sagepub.com

```
static inline void *remote_address
  ( unsigned core, void *vaddr )
{
  uintptr_t addr = ((uintptr_t) vaddr)
    & ( (1UL<<39) - 1);
  addr |= ((uintptr_t) (core+1)) << 39;
  return (void*) addr;
}
```

SMARTMAP also takes advantage of Catamount's physically contiguous address space mapping and the fact that the address mappings are static. Unlike traditional demand-paged operating systems, Catamount determines the mapping from virtual to physical addresses when a process is created and the mapping is never changed. Because each process from the same executable will have the same virtual address mapping, the location of variables with global scope will be identical across all of the processes, both on node and off node.

There is much previous work on using shared memory for intra-node MPI communications (Buntinas et al., 2006c,b, 2007). The traditional approach is to use a POSIX shared memory to allocate a region of memory that is shared between communication processes on a node. This memory is divided up among the processes and message queues are built inside the region. In order to send a message, the sender copies data into the shared region and the receiver copies it out. Other approaches that use only a single copy have been implemented and studied. One such implementation is a Linux kernel module that handles re-mapping of user memory pages into kernel space so that the operating system can do a single memory copy between processes (Jin et al., 2005). Another approach is to have an intelligent or programmable network interface perform a single copy between processes on the same node. A comprehensive analysis of the different approaches for intra-node MPI communication was presented in Buntinas et al. (2006a).

Both of these approaches are currently used for the MPI implementation on Red Storm using the Portals data movement layer (Brightwell et al., 2005). There are two different implementations of Portals available on Red Storm. The default implementation interrupts the operating system to service the network. For intra-node transfers, the operating system simply copies data between the processes. This approach is much like the Linux kernel module, except that Catamount's static memory mapping avoids having to do any re-mapping of pages. For larger messages, it is more efficient for the OS to use the SeaStar (Brightwell et al., 2006) network interface to perform the copy. In the second implementation of Portals, all network processing is performed on the SeaStar. Therefore, all intra-node transfers must go through the network interface.

The simplified memory model of Catamount means that there is no memory registration overhead, no system call, and no setup or teardown time necessary for one process to directly move data into another process. In addition, there is no serialization of processes through the operating system. Processes are free to move data without any OS involvement. Relative to using an intelligent network adapter, there is no need to have data traverse a I/O bus and there is no serialization of requests through the network interface. There is no synchronization mechanism needed to transfer large messages. Our current implementation has a single protocol for moving any amount of data. Since data only moves when both sender and receiver have initiated communication, there are no unexpected messages and no protocols needed to distinguish between the various MPI point-to-point send modes. We are able to achieve low latency, high throughput, significantly increased small message rate, overlap of computation and communication, and we are able to support both the active and passive MPI-2 one-sided functionality. Our approach is much simpler in terms of resource allocation and management. For example, we are not constrained by a shared resource, such as how to best divide up shared memory regions between control and data.

3 MPI Implementation

In this section, we provide a detailed discussion of the implementation of MPI point-to-point and collective communication operations. We chose the Open MPI (Gabriel et al., 2004) implementation since it is the only open-source implementation that supports shared memory and has support for Portals on the Cray XT platform. The existence of shared memory support allows for direct comparisons between using traditional shared memory and SMARTMAP using the same code base on the same platform.

Open MPI supports two different paths for peer communications using Portals. The default path is the a module called the PML (Point-to-point Messaging Layer). Several different PMLs have been implemented since the inception of Open MPI. The PML used for these experiments is called OB1. The PML is layered on top of a Byte Transport Layer (BTL), which is simply used to move bytes. The PML encapsulates all of the MPI functionality, including MPI communicator and tag matching semantics, and also manages flow control and scheduling of multiple underlying BTLs. The Portals BTL does not take advantage of the fact that Portals can perform MPI matching semantics. See Graham et al. (2007) for a more complete discussion of these two approaches.

The second path for peer communications uses a PML called CM. This PML is essentially a pass-through layer

that assumes that the underlying transport will handle all MPI semantics. The layer under CM is referred to as an MTL (Matching Transport Layer). Since the MTL is expected to encapsulate all MPI transport semantics, including the MPI posted receive queue, only a single MTL can be in use at a time.

An important distinction between the two peer communication paths in Open MPI is that the OB1/BTL path must implement the MPI posted receive queue in user-space inside the MPI library. In contrast, the CM/MTL path moves the posted receive queue into the underlying transport layer, which allows the posted receive queue to be implemented in user-space, kernel-space, or even on the network interface (as is the case with SeaStar-based implementation of Portals).

3.1 Shared Memory BTL

In order to compare directly with a shared memory approach, we modified the shared memory BTL in Open MPI to SMARTMAP. Relatively few changes were necessary to allow the shared memory BTL to use SMARTMAP. Rather than having the individual processes use `mmap()` to map the same block of shared memory, the core 0 process on a node simply publishes the location of the block of memory that it has allocated from its local heap. Using SMARTMAP, the other processes read this location from core 0's memory and convert it to the appropriate remote address. More extensive changes were need to support SMARTMAP within the Portals MTL. These changes are described in detail below.

3.2 Point-to-Point Communication

3.2.1 Initialization Catamount supports multiple cores by running in virtual node mode, where each core is treated as a node that runs a process in the parallel job. The memory on a single physical node is divided evenly among the available cores. When a process is started, the parallel runtime system is responsible for setting the following values in the process' address space:

- number of processes in the job (`_my_nnodes`);
- global rank of the process (`_my_rank`);
- number of active cores on the node (`_my_vnm_degree`);
- core rank on the node (`_my_core`).

We then use these values to determine a global rank to core rank mapping and a core rank to global rank mapping. First, we allocate an array for the global rank to core rank mapping and initialize all entries to –1. We then allocate an array for the core rank to global rank mapping. We take advantage of the SMARTMAP capability to determine the core rank to global rank mapping. Each core loops from 0 to `_my_vnm_degree`. If the loop variable is equal to the local core's rank, it fills in the core rank to global rank mapping with its global rank. If the loop variable is not the process' core rank, it accesses the `_my_rank` value on the other core and fills in the array with this value. At this point, we can use this array to set the corresponding values in the global rank to core rank array to the appropriate values.

3.2.2 Sending Each process has a single queue for posted receives and a queue per core for posted sends. A queue is implemented as a circular array with a head and tail entry. Each queue element contains: buffer address, buffer length, local source rank, context id, tag, request address, and completion flag.

In order to send a message to another core on the same node, we first check the global rank to core rank mapping to discover whether the destination rank is a local process. If it is, we check the destination core to see whether the message is being sent to the sending process, e.g. a send-to-self message. If so, we traverse the posted receive queue looking for a match.

If the message is destined for a different core on the same node, we simply pop a queue element off of a stack, fill in the contents based on the send request, and add it to the send queue for that particular core in that core's address space. If the send is blocking, we then call into the progress routine, which is described in detail below.

Because the arrays for message queues are declared as globals, they are at the same virtual address in every process. To send a message, the sending core first obtains the address of its send queue in its local address space. It then converts this address to a remote address in the receiving core's address space. The sending core then converts the address of the send request structure from a local address to a remote address in its own address space. This way, when the receiving core uses the address, it will point to the correct request in the sending core's address space. The sending core maintains the index of the next slot to be used to send a message to the destination core. To send the message, the sending core writes the send request to the next slot in its queue in the receiving core's address space, and then increments a last slot pointer in its address space. The sending core then increments a next slot counter for sends to that core in its own address space. In order to avoid overrunning the destination send queue, the sender first checks to ensure that the next send queue slot is empty. If the slot still contains a non-NULL pointer, the current implementation aborts due to resource exhaustion. A more elegant approach would be to keep a local queue of pending sends and drain the queue as slots became available.

3.2.3 Receiving When posting a receive, we check to see whether the posted receive queue is currently empty. If it is, we can check for a match right away. If the source of posted receive is not `MPI_ANY_SOURCE`, we loop through the array of send requests in the local address space, comparing the contents of the send request with that of the posted receive. If a match is found, we copy the contents of the message from the source rank into the destination buffer and mark the send request as completed. The receiver also zeroes out this array entry in the send queue.

Because the sender has already converted the starting address of the send buffer and address of the send request to remote addresses in the local address space, there is no need for the receiving process to convert them. The receiving process simply uses these pointers directly. The receiving process then checks to determine whether the receive matched the first item in the send queue. If it did, it increments the start counter that indicates where the first message in the send queue is located.

If the posted receive queue is not empty, we simply pop a queue element off of a stack, fill in the contents, and add it to the local posted receive queue. We then call into the progress function to see whether any outstanding requests have been or can be completed. In order to probe for an incoming messages, we do the same steps as posting a receive; however, when a match is found, we simply fill in the appropriate status information and do not perform the data copy.

3.2.4 Wildcard Source Receives If the source of the receive is `MPI_ANY_SOURCE`, the implementation must ensure atomicity between inter- and intra-node messages. Because Portals encapsulates an MPI posted receive queue for incoming network messages and another posted receive queue is used for incoming shared memory messages, the implementation must be careful to ensure that a wildcard source receive is only satisfied by either transport, but not both.

When a wildcard source receive is posted, the implementation first checks the communicator to determine whether all of the ranks in the communicator are completely on-node or completely off-node. If all of the ranks are on-node, then the implementation tries to satisfy the receive using only shared memory, which is described below. If all of the ranks are off-node, the a receive can be posted to the network using Portals.

If the communicator spans ranks that are only on-node, then the implementation loops over all possible send queues rather than traversing a single send queue. The current implementation always begins looking for messages from core 0, so there is a potential fairness issue in that messages from lower-ranked cores will receive preference for matching wildcard source receives. This can be easily corrected by keeping track of which rank had the last match and starting the search at the next highest rank (modulo the number of ranks in the communicator).

If the communicator spans ranks that are both on-node and off-node, then the implementation enqueues a posted receive on the shared memory queue and posts a receive to the network using Portals. However, it creates a link between the two receive requests so that the shared memory request contains the Portals request and vice versa. If the Portals request is satisfied first, then the shared memory request will be removed when the Portals request is completed. If the shared memory request is matched, the implementation must first remove the Portals request. If the Portals request is removed successfully, then the shared memory request can be satisfied. If the Portals request cannot be removed, then it has been satisfied, and the shared memory request must be removed.

3.2.5 Progress The progress function first checks to see whether any outstanding posted receives can be completed. It does this in the same way as was just described in the previous section: traversing each send queue in looking for a match. After traversing the posted receive queue, the progress function traverses all of the outstanding send requests to see whether any have been marked as completed by the other cores. Any that have been marked as completed by the receiving core are dequeued and any cleanup routine for the request is run.

3.2.6 Synchronization In this strategy, the sending core manages insertion of a send request in the destination core's address space. Once the receiver has completed the request, the receiver marks the send request as completed in the sending core's address space. The sending core increments the tail pointer of the send queue in the receiver's address space, while the receiver keeps track of where the head of the queue is. The sending core is responsible for removing elements from a queue of outstanding send requests, while the receiver is responsible for removing elements from the posted receive queue. This approach removes any synchronization between the sender and receiver and eliminates any need for atomic shared memory operations.

3.2.7 Synchronous Data Movement A key difference between the Portals MTL with SMARTMAP and the shared memory BTL with SMARTMAP is that the MTL only moves messages when a matching receive has been posted. The shared memory BTL is able to copy user data along with the MPI envelope information, allowing for short send operations to complete before the data has actually been transferred to the receiver's buffer. Given that the focus of SMARTMAP is to decrease the number of memory-to-memory copies, we chose not to employ

this optimization for the MTL. Therefore, messages using SMARTMAP are synchronous.

3.3 Collective Communication

In contrast to point-to-point operations, collective communication involves a group of processes that all must participate and cooperate to complete the operation. In addition to a modular design for peer communications, the Open MPI implementation also has modules for implementing MPI collective operations. In this section, we describe how we have optimized several MPI collective operations using SMARTMAP.

3.3.1 Initialization The SMARTMAP collective module is enabled when all ranks on a given communicator on are the same node. Within each process, the following global data structure appears at the same virtual address:

```
typedef struct {
  int   counter;
  int   context;
  void *send_buff;
  void *recv_buff;
  int   turn;
  int   finished;
} coll_info_t;
coll_info_t coll_info;
```

Since MPI collective operations are blocking, there is only ever one outstanding collective operation per process. The first two elements of the data structure, `counter` and `context`, are used to record in which collective operation the process is currently participating. Each communicator has an associated counter that is incremented each time a collective operation is entered. Each communicator also has an associated context identifier, which is essentially a system-level tag. Since MPI collective operations are not synchronous (processes can exit the operation when their participation is done), the `context` and `counter` values are used to synchronize processes participating in the same collective operation, preventing overlapping collective operations from interfering with each other.

MPI collective operations can be characterized as *r*ooted, meaning that all processes participate but only one process receives the result, or *n*on-rooted, where all processes participate and all processes receive the result. Upon entering a rooted collective, a process determines whether it is the root. If so, it fills in the fields in the `coll_info_t` structure appropriately and then sets the `context` and `counter` values to indicate that the root has joined the collective. A memory barrier is issued before these two values are set to ensure that the other values in the structure are valid when they are read by the other processes on the other cores. Processes that are not the root of the collective operation convert the local address of the structure to a remote address on the root's core and then wait for the root to signal that it has entered the operation. Some collective operations also require that each process initialize its own structure before synchronizing with the root.

3.4 Reduce

We have implemented two different versions of the reduce operation: one that is serial and one that is parallel. Both of these implementations take advantage of SMARTMAP's ability to operate on the destination buffer in-place. In the serial version, the root copies the source buffer into the destination buffer and then each rank proceeds, in rank order, to operate on the destination buffer directly. In the parallel version, we take advantage of the fact that each process can also read every other process' source buffer. Owing to this, we can divide the problem up among all of the processes and have them operate simultaneously on the individual pieces of the buffer. In the initial copy phase, each process can copy a piece of the root's source buffer into the destination buffer, thus parallelizing the data copy. After the copy, each rank can then take a piece of the next rank's buffer and perform the reduce operation on the destination buffer. In this way, all cores are actively participating in the reduce, eliminating synchronization overhead and potentially making better use of all of the cores. We have called this a "threaded" reduction, since each process essentially acts like an independent thread working on its own piece of the operation.

3.5 Broadcast

For the broadcast operation, the root process again initializes the `finished` value to one and sets the `send_buff` to the location of the user buffer. It then waits for the other processes to increment the `finished` value. When non-root processes enter the collective operation, they read the `send_buff` value in the root process's address space, convert it to a remote address, and then copy the data directly from the source buffer to the destination buffer in its address space. When the copy is complete, the process atomically increments the `finished` value.

3.6 Allreduce

Semantically, an allreduce operation is simply a reduce followed by a broadcast so that all processes receive the result rather than just the root process. The SMARTMAP

implementation simply calls the SMARTMAP reduce and broadcast functions directly to implement allreduce.

3.7 Alltoall

For the MPI_Alltoall operation, each process sends a distinct set of data to every other node. The *j* th block sent from process *i* is received by process *j* at the *i* th block of its receive buffer. The routine must be called by all group members with input and output buffers of the same length and same type.

The implementation of this algorithm is more straightforward than the reduce operation. As with the reduce, each process fills in its local collective info structure appropriately and waits for all of the other processes to enter the collective. After all of the processes have arrived, each process loops through all of the ranks of the participating processes and copies its piece of its send buffer directly into the corresponding piece at the root.

3.8 Native Shared Memory Collectives

At the time of this writing, there was no implementation of native shared memory collective operations in Open MPI, despite the fact that an initial prototype implementation had been evaluated (Graham and Shipman, 2008). Therefore, the performance comparisons between the shared memory BTL and the native SMARTMAP collective operations are somewhat unfair because the comparison is incomplete. The shared memory BTL only uses shared memory as an MPI transport. We hope to compare native SMARTMAP collective operations to native shared memory collective operations when the latter is available in Open MPI.

4 Performance Evaluation

The platform used to gather our performance results is a Red Storm development system that contains 2.2 GHz quad-core Opterons. We used the Intel MPI Benchmark Suite (IMB) version 2.3 to measure point-to-point and collective communication performance. In order to characterize small message rate, we used the Ohio State message rate benchmark that has been modified by PathScale (now Qlogic).

Since we are interested in intra-node communication, all of our results are from a single quad-core node. We limited our measurements to the interrupt-driven version of Portals because it is more efficient at intra-node transfers. The ability to have the operating system perform a copy between processes is more efficient than having the SeaStar adapter perform the copy. Due to limitations of the SeaStar, send operations must go through the OS, so in addition to serializing requests through a slower network interface, requests must also be serialized through the OS.

Figures 1(a) and (b) show ping-pong latency and bandwidth, respectively. SMARTMAP is able to achieve a zero-byte latency of 520 ns, while Cray's MPI achieves 2.98 µs. We also plot the improvement of SMARTMAP compared to shared memory on the *y*-axis. For 32 kB messages, SMARTMAP outperforms shared memory by more than a factor of five. This performance increase is largely due to SMARTMAP's ability to support single-copy transfers.

SMARTMAP's bandwidth peaks at more than 9.3 GB/s, while the MTL in Open MPI is able to achieve nearly 5.8 GB/s. Again, we also show the improvement of SMARTMAP versus shared memory on the *y*-axis. Initially, the bandwidth advantage for SMARTMAP is around 30% and grows to more than 80% for larger message sizes. The Portals implementation uses a single copy for messages less than 512 kB, so the difference in bandwidth performance is largely due to the serialization through the OS and overhead incurred by having the OS perform the memory copy.

Figures 1(c) and (d) show the performance of four-process exchange and sendrecv operations, respectively. These results again show that the impact of serialization worsens as all four cores are attempting to exchange message simultaneously. For the exchange benchmark, SMARTMAP outperforms shared memory by a factor of seven for 16 kB messages, while the improvement for the sendrecv benchmark reaches nearly 500% for 32 kB messages.

Figures 2(a) and (b) show message rates for two process and four processes, respectively. SMARTMAP is able to achieve nearly 2.5 million messages per second for two processes and over 4.5 million messages per second when two pairs of processes are exchanging messages. Shared memory performance peaks at just over 3.5 million messages per second for both the two-process and four-process results. It is not clear that the shared memory performance would increase with more cores. Performance for the Portals-based transports actually decreases when increasing from one pair to two pairs of communicating processes.

Figures 3(a) and (b) show the performance of four-process broadcast and reduction collective operations. The SMARTMAP broadcast is less than 1 µs out to a 4 kB message size, while the shared memory broadcast start out at just under 2 µs. We again plot the improvement of SMARTMAP compared with shared memory. For broadcast, the SMARTMAP improvement is nearly an order of magnitude for 8 kB messages. For the reduce operation, we show the improvement of the parallel algorithm as compared to the shared memory implementation. We again see a very large improvement, more than 700% for 32 kB messages.

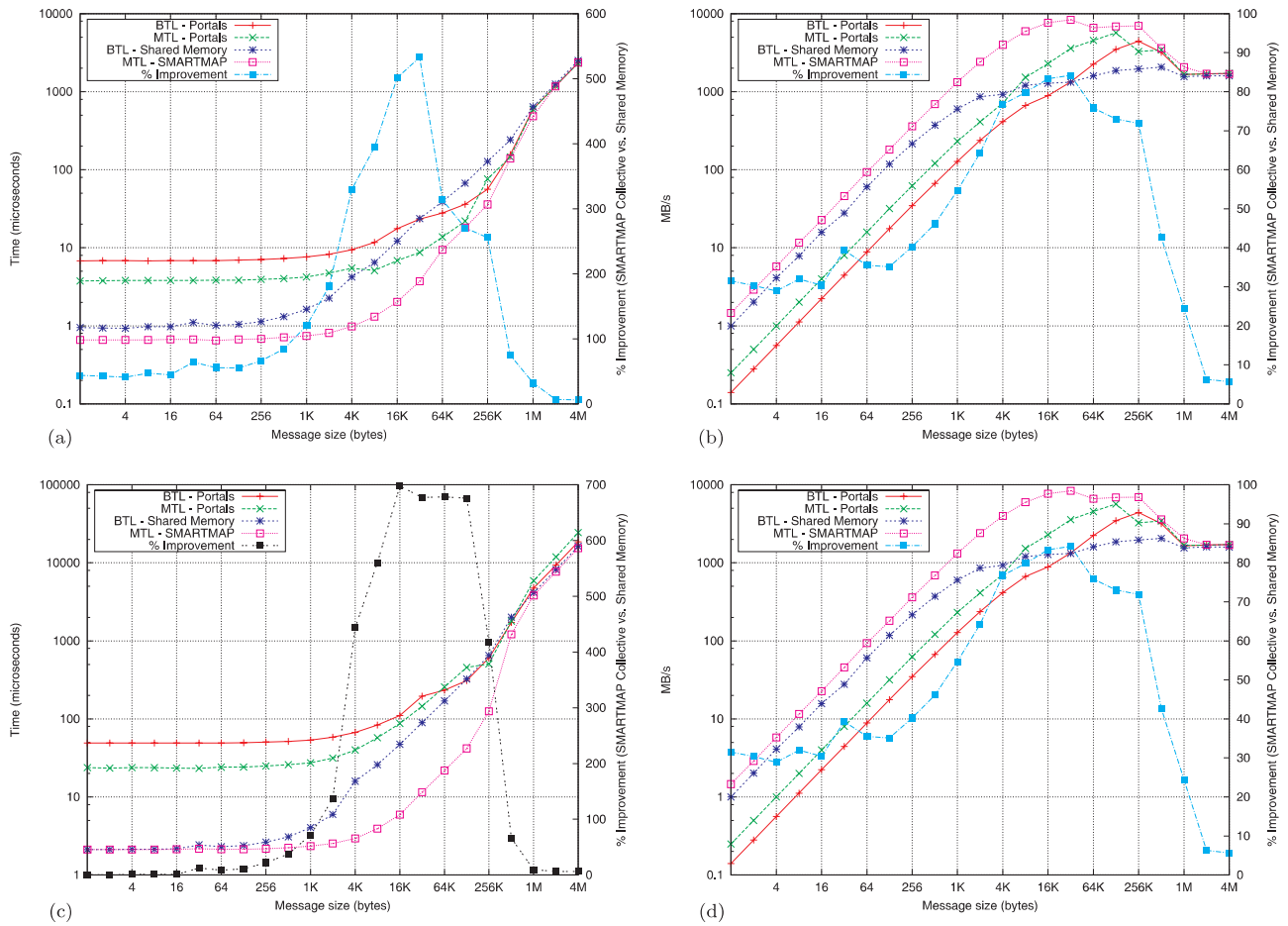

Fig. 1 IMB point-to-point performance: (a) PingPong latency; (b) PingPong bandwidth; (c) Exchange; (d) Sendrecv.

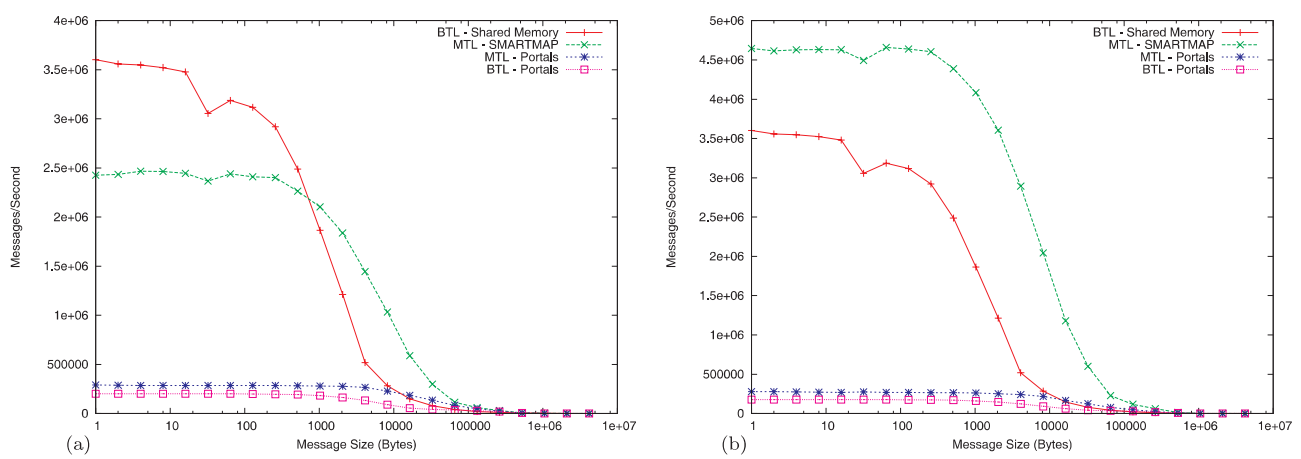

Fig. 2 MPI message rate performance: (a) one pair; (b) two pairs.

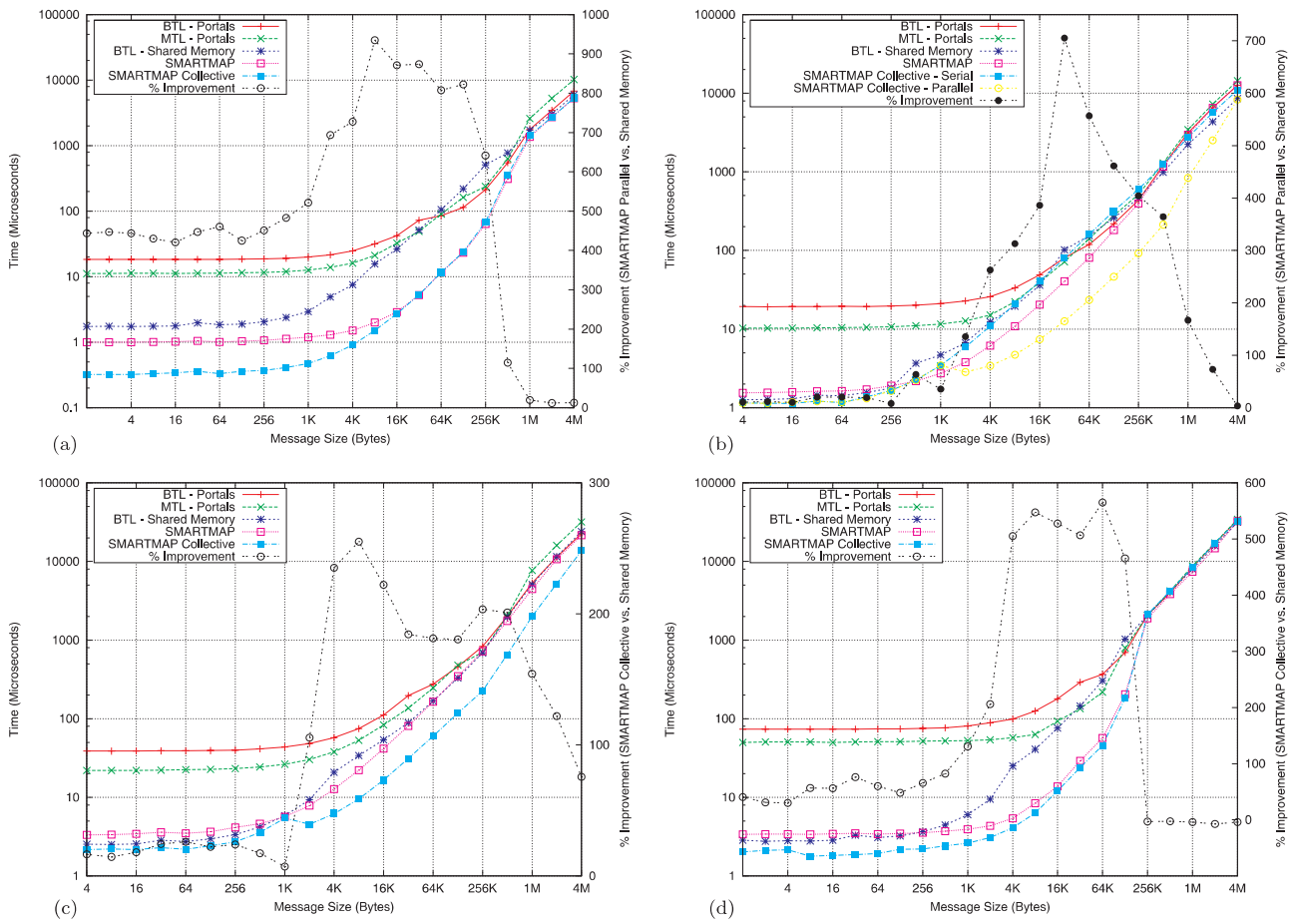

Fig. 3 IMB collective performance: (a) broadcast; (b) reduce; (c) allreduce; (d) alltoall.

We conclude collective performance results with allreduce and alltoall performance in Figures 3(c) and (d). These results again demonstrate the effectiveness of using shared memory when all processes are communicating simultaneously. The alltoall results are particularly dramatic, especially at larger message sizes. The improvement factor for the SMARTMAP parallel allreduce is more than 2.5 for an 8 kB message, while the improvement factor for the SMARTMAP alltoall peaks at nearly six for a 64 kB message.

5 Conclusion

The SMARTMAP capability in the Catamount lightweight kernel is able to deliver significant performance improvements for intra-node MPI point-to-point and collective operations. It is able to dramatically outperform the current shared memory approach for intra-node MPI data movement on the Cray XT. Because SMARTMAP allows processes to communicate directly at the user level, it eliminates the need to serialize intra-node communication through the operating system or network interface. SMARTMAP also eliminates all of the extraneous memory-to-memory copies that POSIX-style shared memory imposes. These overheads are shown to be significant on a quad-core processor, and the benefits of SMARTMAP are likely to continue to grow as core counts increase.

From a complexity standpoint, it is much easier to enhance MPI collective operations to use the SMARTMAP capability. It is not necessary to handle non-specific receive operations or non-blocking operations. We are currently adapting the hierarchical collective communication module in Open MPI to be able to use the SMARTMAP collective module.

We are also enhancing the production Cray-developed MPI implementation for Red Storm, which is based

on MPICH2, to have hierarchical collectives using SMARTMAP. We believe that this approach will likely result in the largest increase in application performance for quad-core processors. Since the performance of most large-scale parallel applications is determined by the longest latency, we do not think the decrease in on-node point-to-point latency with SMARTMAP is likely to result in a significant performance increase for a majority of applications. Once this implementation is complete, we expect to be able to do a more in-depth application performance evaluation on the full Red Storm system. We are also currently working on a port of the Catamount lightweight kernel to a dual-socket, quad-core XT5 system, so we hope to do similar evaluations and analyze the performance of eight cores on a node.

As part of future MPI development, we would like to use SMARTMAP to handle on-node non-contiguous data transfers with no intermediate buffering, and there is an opportunity to enhance the on-node MPI-2 one-sided operations using SMARTMAP as well. We are also exploring whether SMARTMAP can be used directly by application libraries, possibly avoiding the overhead of memory copies altogether.

Acknowledgments

Trammell Hudson is responsible for the implementation of SMARTMAP in Catamount, which was an outcome of discussions between the author, Kevin Pedretti, and Kurt Ferreira. This work would also not have been possible without John Van Dyke, who is responsible for implementing virtual node mode support in Catamount. The author would also like to thank Sue Kelly and the Cray support staff at Sandia for assistance with the Red Storm development systems. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Author Biography

Ron Brightwell received the B.S. degree in mathematics in 1991 and the M.S. degree in computer science in 1994 from Mississippi State University. He is currently a principal member of technical staff at Sandia National Laboratories. His research interests include high-performance, scalable communication interfaces and protocols for system area networks, operating systems for massively parallel processing machines, and parallel program performance analysis libraries and tools.

References

Brightwell, R. et al. (2005). Implementation and performance of Portals 3.3 on the Cray XT3. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*.

Brightwell, R. et al. (2006). SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro* **26**(3): 41–57.

Buntinas, D., Mercier, G. and Gropp, W. (2006a). Data transfers between processes in an SMP system: Performance study and application to MPI. In *Proceedings of the 2006 International Conference on Parallel Processing*.

Buntinas, D., Mercier, G. and Gropp, W. (2006b). Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proceedings of the 2006 International Symposium on Cluster Computing and the Grid*.

Buntinas, D., Mercier, G. and Gropp, W. (2006c). Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In *Proceedings of the 2006 European PVM/MPI Users' Group Meeting*.

Buntinas, D., Mercier, G., and Gropp, W. (2007). Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. *Parallel Computing* **33**(9): 634–644.

Camp, W. J. and Tomkins, J. L. (2002). Thor's hammer: The first version of the Red Storm MPP architecture. In *Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD.

Gabriel, E. et al. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, pp. 97–104.

Graham, R. L. et al. (2007). An evaluation of Open MPI's matching transport layer on the Cray XT. In *Proceedings of the 14th European PVM/MPI Users' Group Conference*.

Graham, R. L. and Shipman, G. (2008). MPI support for multi-core architectures: Optimized shared memory collectives. In *Proceedings of the 15th European PVM/MPI Users' Group Conference*.

Jin, H.-W. et al. (2005). Limic: Support for high-performance MPI intra-node communication on Linux. In *Proceedings of the 2005 Cluster International Conference on Parallel Processing*.

Kelly, S. M. and Brightwell, R. (2005). Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*.