SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor

Ron Brightwell and Kevin Pedretti
Scable System Software Department
Sandia National Laboratories
Albuquerque, New Mexico 81785–1319
{rbbrigh,ktpedre}@sandia.gov

Trammell Hudson
Operating Systems Research
1527 16th NW #5
Washington, DC 20036
hudson@osresearch.net

***Abstract*—This paper describes SMARTMAP, an operating system technique that implements fixed offset virtual memory addressing. SMARTMAP allows the application processes on a multi-core processor to directly access each other's memory without the overhead of kernel involvement. When used to implement MPI, SMARTMAP eliminates all extraneous memory-to-memory copies imposed by UNIX-based shared memory strategies. In addition, SMARTMAP can easily support operations that UNIX-based shared memory cannot, such as direct, in-place MPI reduction operations and one-sided get/put operations. We have implemented SMARTMAP in the Catamount lightweight kernel for the Cray XT and modified MPI and Cray SHMEM libraries to use it. Micro-benchmark performance results show that SMARTMAP allows for significant improvements in latency, bandwidth, and small message rate on a quad-core processor.**

I. Introduction

As the core count on processors used for high-performance computing continues to increase, the performance of the underlying memory subsystem becomes significantly more important. In order to make effective use of the available compute power, applications will likely have to become much more sensitive to the way in which they access memory. Applications that are memory bandwidth bound will need to avoid any extraneous memory-to-memory copies. For many applications, the memory bandwidth limitation is compounded by the fact that the most popular and effective parallel programming model, MPI, mandates copying of data between processes. MPI implementors have worked to make use of shared memory for communication between processes on the same node. Unfortunately, the current schemes for using shared memory for MPI can require either excessive memory-to-memory copies or potentially large overheads inflicted by the operating system (OS).

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.

In order to avoid the memory copy overhead of MPI altogether, more and more applications are exploring mixed-mode programming models where threads and/or compiler directives are used on-node and MPI is used off-node. Unfortunately, the complexity of shared memory programming using threads has hindered both the development of applications as well as the development of thread-safe and thread-aware MPI implementations. The initial attractiveness of mixed-mode programming was tempered by the additional complexity induced by finding multi-level parallelism and by initial disappointing performance results [1], [2], [3]. Recently, however, unpublished data on mixed-mode applications suggest more encouraging results on multi-core processors.

In this paper, we introduce a scheme for using fixed-offset virtual address mappings for the parallel processes within a node to enable efficient direct access shared memory. This scheme, called Simple Mapping of Address Region Tables for Multi-core Aware Programming, or SMARTMAP, achieves a significant performance increase for on-node MPI communications and eliminates all of the extraneous memory-to-memory copies that shared memory MPI implementations incur. SMARTMAP can also be used for more than MPI. It maps very well to the partitioned global address space (PGAS) programming model and can be used to implement one-sided get/put operations, such as those available in the Cray SHMEM model. This strategy can also be used directly by applications to eliminate the need for on-node memory-to-memory copying alto-

gether.

The main contributions of this paper are:

- an OS virtual memory mapping strategy that allows direct access shared memory between processes on a multi-core processor
- a description of how this strategy can be used for on-node data movement between processes on a multi-core processor
- a detailed analysis of the performance impacts of using this strategy for MPI peer communication, MPI collective communication, and Cray SHMEM data movement operations

The rest of this paper is organized as follows. The next section provides background on the current approaches to using shared memory for intra-node data movement. In Section III, we describe the implementation of the SMARTMAP and its advantages over other approaches. Section IV provides a detailed description of the enhancements that we have made to MPI and SHMEM implementations on the Cray XT to use it. Section V presents performance results using several micro-benchmarks. Relevant conclusions of this paper are summarized in Section VI, and we close by discussing possible avenues of future work in Section VII.

II. Background

POSIX-based operating systems generally support shared memory capability through two fundamental mechanisms: threads and memory mapping. Unlike processes, which allow for a single execution context inside an address space, threads allow for multiple execution contexts inside a single address space. When one thread updates a memory location, all of the threads sharing the same address space also see the update. A major drawback of threads is that great care must be taken to ensure that common library routines are reentrant, meaning that multiple threads could be executing the same piece of code simultaneously. For non-reentrant functions, some form of locking must be used to ensure atomic execution. The same is true for data accessed by multiple threads – updates must be atomic with respect to one another or else difficult to debug race conditions will occur. Race conditions and fundamentally non-deterministic behavior make threads difficult to use correctly.

In memory mapping, cooperating processes request a shared region of memory from the operating system and then map it into their private address space, possibly at a different virtual address in each process. Once initialized, a process may access the shared memory region in exactly the same way as any other memory in its private address space. As with threads, updates to shared data structures in this region must be atomic.

Explicit message passing is an alternative to shared memory for intra-node data sharing. In message passing, processes pass messages carrying data between one another. No data is shared directly, but rather is copied between processes on an as necessary basis. This eliminates the need for re-entrant coding practices and careful updates of shared data, since no data is shared. The main downside to this approach is the extra overhead involved in copying data between processes.

In order to accelerate message passing, memory mapping is often used as a high-performance mechanism for moving messages between processes [4]. Unfortunately, such approaches to using page remapping are not sufficient to support MPI semantics, and general-purpose operating systems lack the appropriate mechanisms. The sender must copy the message into a shared memory region and the receiver must copy it out – a minimum of two copies must occur. It would be ideal if messages could be moved directly between the two processes with a single copy. This would be possible if all processes operated entirely out of the shared memory region, but this would amount to the processes essentially becoming threads, with all of their inherit problems. Furthermore, message passing APIs such as MPI allow message buffers to be located anywhere in an address space, including the process's data, heap and stack.

As of MPI 2.0, MPI applications may make use of both threads and memory mapping, although few MPI implementations provide full support for threads. More commonly, MPI implementations utilize memory mapping internally to provide efficient intra-node communication. During MPI initialization, the processes on a node elect one process to create the shared memory region and then the elected process broadcasts the information about the region to the other processes on the node (e.g., via a file or the sockets API). The other processes on the node then "attach" to the shared memory region, by requesting that the OS map it into their respective address spaces.

Note that the approach of using shared memory for intra-node MPI messages only works for the point-to-point operations, collective communication operations, and a subset of the MPI-2 remote memory access operations. Copying mandates active participation of the two processes involved in the transfer. Single-sided put/get operations, such as those in the Cray SHMEM programming interface, cannot be implemented using POSIX shared memory.

A. Intra-Node MPI

There are several limitations in using regions of shared memory to support intra-node MPI [5], [6], [7]. First,

the MPI model doesn't allow applications to allocate memory out of this special shared region, so messages must first be copied into shared memory by the sender and then copied out of the shared region by the receiver. This copy overhead can be a significant performance issue. Typically there is a limitation on the amount of shared memory that a process can allocate, so the MPI implementation must make decisions about how to most effectively use this memory in terms of how many per-process messages to support relative to the size of the contents of each message. The overhead of copying messages using shared memory has led researchers to explore alternative single-copy strategies for intra-node MPI message passing.

One such strategy is to use the operating system to perform the copy between separate address spaces [8]. In this method, the kernel maps the user buffer into kernel space and does a single memory copy between user space and kernel space. The drawback of this approach is that the overhead of trapping to the kernel and manipulating memory maps can be expensive. Another limitation is that all transfers must be serialized through the operating system. As the number of cores on a node increases, serialization and management of shared kernel data structures for mapping is likely to be a significant performance limitation. Another important drawback of this approach is that there are two MPI receive queues – one in the MPI library and one in the kernel. When the application posts a non-specific receive using `MPI_ANY_SOURCE`, great care must be taken to insure that the atomicity and ordering semantics of MPI are preserved. There is a potential race for a non-specific receive request to be satisfied by both the MPI library and the operating system. Managing atomicity between events in kernel space and user space is non-trivial.

Another strategy for optimizing intra-node transfers is to use hardware assistance beyond the host processors. The most common approach is to use an intelligent or programmable network interface to perform the transfer. Rather than sending a local message out to the network and back, the network interface can simply use its DMA engines to do a single copy between the communicating processes. The major drawback of this approach is serialization through the network interface, which is typically much slower than the host processor(s). Also, large coherent shared memory machines typically have hardware support for creating a global shared memory environment. This hardware can also be used when running distributed memory programs to map arbitrary regions of memory to provide direct shared memory access between processes. SGI's NUMAlink hardware is one such example [9]. The obvious drawback of this approach is the additional cost of this hardware.

A comprehensive analysis of the different approaches for intra-node MPI communication was presented in [10]. More recently, a two-level protocol approach that uses shared memory regions for small messages and OS support for page remapping individual buffers for large messages was proposed and evaluated [11]. There has also been some recent work on optimizing MPI collective operations using shared memory for multi-core systems [12].

B. Intra-Node Communication on the Cray XT

All communication between processes on the Cray XT use the Portals [13] data movement layer. Two implementations of Portals are available for the SeaStar [14] network. The default implementation is interrupt driven and all Portals data structures are contained inside the operating system. When a message arrives at the SeaStar, it interrupts the Opteron host processor, which then inspects the message header, traverses the Portals data structures and programs the DMA engines on the SeaStar to deliver the message to the appropriate location in the application process' memory. This implementation is referred to as "Generic Portals" (GP) because it works for both Catamount on compute nodes and in Linux on service and I/O nodes. The other implementation supports a complete offload of Portals processes and uses no interrupts. When a message arrives at the SeaStar, all of the Portals processing occurs on the SeaStar itself. This implementation is known as "Accelerated Portals" (AP) and is available only on Catamount, largely due to the simplified address translation that Catamount offers.

For intra-node transfers, the Generic Portals implementation takes advantage of the fact that Portals structures for both the source and destination are in kernel space. The kernel is able to traverse the structures and perform a single memory copy to move data between processes, since all of user space is also mapped into kernel space. At large message sizes, it becomes more efficient for the kernel to use the DMA engines on the SeaStar to perform the copy, so there is a crossover point where it switches to using this approach. For the Accelerated Portals implementation, all Portals data structures are in SeaStar memory, so it must traverse these structures in the same way it does for incoming network messages, so there is little advantage to intra-node transfers. In fact, intra-node transfers are slower going through the SeaStar rather then the operating system, due to the speed of the host processor (2+ GHz) relative to the network processor (500 MHz).

III. SMARTMAP IMPLEMENTATION

SMARTMAP is a virtual memory mapping technique that allows for direct access shared memory between the processes running on a multi-core processor. This technique leverages many of the characteristics of a lightweight compute node kernel to achieve shared memory capability without the limitations of POSIX shared memory mapping or the complexity of multi-threading. SMARTMAP preserves the idea of running a single execution context within a separate address space, but also provides the ability to easily access the address space of the other execution contexts within the same parallel job on the same node. The following provides a description the implementation of SMARTMAP and its advantages over existing approaches for intra-node data movement.

A. Catamount

The Catamount lightweight kernel [15] is a third-generation compute node operating system developed by Sandia National Laboratories along with Cray, Inc., as part of the Red Storm project [16]. Red Storm is the prototype for what has become the commercially successful Cray XT line of massively parallel processing systems. Catamount has several unique features that are designed to optimize performance and scalability specifically for a distributed memory message passing-based parallel computing platform.

One such important feature is memory management. Unlike traditional UNIX-based operating systems, Catamount does not support demand-paged virtual memory and uses a linear mapping from virtual addresses to physical pages of memory. This approach can potentially have several advantages. For instance, there is no need to register memory or “lock” memory pages involved in network transfers to prevent the operating system from unmapping or remapping pages. The mapping in Catamount is done at process creation time and is never changed. This greatly simplifies translation and validation of virtual address for the network interface. Virtual address validation is a simple bounds check and translating virtual addresses to physical addresses is a simple offset calculation.

The SMARTMAP approach for direct access shared memory takes advantage of Catamount’s simple memory management model, specifically the fact that Catamount only uses a single entry in the top-level page table mapping structure (PML4) on each X86-64 (AMD Opteron or Intel EM64T) core. 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 X86-64 architecture supports a 48-bit address space, so there are 512 entries in the PML4.

```
1   static void initialize_shared_memory( void )
2   {
3     extern VA_PML4T_ENTRY *KN_pml4_table_cpu[];
4     int cpu;
5     for( cpu=0 ; cpu < MAX_NUM_CPUS ;cpu++ ) {
6       VA_PML4T_ENTRY * pml4 = KN_pml4_table_cpu[ cpu ];
7       if( !pml4 )
8         continue;
9       KERNEL_PCB_TYPE * kpcb = (KERNEL_PCB_TYPE*) KN_cur_kpcb_ptr[cpu];
10      if( !kpcb ) continue;
11      VA_PML4T_ENTRY dirbase_ptr = (VA_PML4T_ENTRY)
12        (KVTOP( (size_t) kpcb->kpcb_dirbase ) | PDE_P | PDE_W | PDE_U );
13      int other;
14      for( other=0 ; other<MAX_NUM_CPUS ; other++ ) {
15        VA_PML4T_ENTRY * other_pml4 = KN_pml4_table_cpu[other];
16        if( !other_pml4 ) continue;
17        other_pml4[ cpu+1 ] = dirbase_ptr;
18      }
19    }
20  }
```

Fig. 1: SMARTMAP kernel code

```
1   static inline void * remote_address( unsigned core,
2                                        volatile void * vaddr)
3   {
4     uintptr_t addr = (uintptr_t) vaddr;
5     addr |= ((uintptr_t) (core+1)) << 39;
6     return (void*) addr;
7   }
```

Fig. 2: User function for converting a local virtual address to a remote virtual address

Each core writes the pointer to its PML4 table into an array at core 0 when a new parallel job is started. Each time the kernel enters the routine to run the user-level process, it copies all of the PML4 entries from each core into the local core. This allows every core on a node to see every other core’s view of the virtual memory across the node, at a fixed offset into its own virtual address space. Figure 1 shows the 20 lines of kernel code that implement direct access shared memory in Catamount.

Another feature of Catamount is that the mapping of virtual addresses for the same executable image is identical across all of the processes on all of the nodes. The starting address of the data, stack, and heap is the same. This means that the virtual address of variables with global scope is the same everywhere. The Cray SHMEM environment refers to such addresses as *symmetric* addresses, whereas other addresses, such as those allocated off of the stack as the application is running, are termed to be *non-symmetric*. Figure 2 shows the user-level function for converting a *local* virtual address into a *remote* virtual address for a process on a different core. Symmetric addresses combined with this simple remote address translation function make it extremely easy for one process to read or write the corresponding

data structure in another process' address space running on a different core of the same processor.

Catamount's memory management design is much simpler than a general-purpose OS like Linux. Linux memory management is based on the principle that processes execute in different address spaces and threads execute in the same address space. Most architecture ports, x86 included, maintain a unique set of address translation structures (e.g., a page table tree on x86) for each process and a single set for each group of threads. SMARTMAP operates differently in that a process's address space and associated translation structures are neither fully-unique or fully-shared. For example, SMARTMAP on the x86 architecture maintains a unique top-level page table (the PML4) for each process; however, all processes share a common set of leaves linked from this top-level table. Linux memory management does not support this form of page-table sharing, so each process must be given a replicated copy of each shareable leaf. This results in more memory being wasted on page tables (2 MB per GB of address space on x86) and a larger cache footprint than necessary. Modifications to Linux to support sharing a single page table entry for shared memory mapped regions has been proposed, but the changes have not been accepted in the mainline kernel.

B. Limitations

SMARTMAP is currently limited to what the top-level X86-64 page table supports – 511 processes (one slot is needed for the local process) and 512 GB of memory per process. However, this will likely be sufficient for a typical compute node for the foreseeable future. Since Catamount only runs on X86-64 processors, SMARTMAP is currently limited to this processor family as well. However, the concepts are generally applicable to other architectures that support virtual memory. For example, even though the PowerPC uses an inverted page table scheme that is very different from x86-64, the hardware's support for segmentation can be used to implement SMARTMAP just as efficiently. On other architectures with software-based virtual memory support (i.e., a software managed translation look-aside buffer), SMARTMAP is straightforward to implement.

IV. Using SMARTMAP

We have used the SMARTMAP capability in Catamount to optimize intra-node data movement for the Cray SHMEM one-sided operations, as well as for MPI point-to-point and collective operations. This section describes the modifications to these libraries.

```
1  void shmem_putmem( void *target, void *source, size_t length, int pe )
2  {
3    int core;
4
5    if ( (core = smap_pe_is_local(pe)) != -1 ) {
6      void *target_r = (void *)remote_address( core, target );
7      memcpy( target_r, source, length );
8    } else {
9      pshmem_putmem( target, source, length, pe );
10   }
11 }
```

Fig. 3: SHMEM Put Function

A. Cray SHMEM

The Cray SHMEM library was first available on the Cray T3 series of machine circa 1994. It supports a variety of one-sided get/put data movement functions as well as collective reduction functions and remote atomic memory operations, such as atomic-swap and fetch-and-increment.

The existing implementation for Catamount on the Cray XT uses Portals for all data movement operations. Similar to MPI's profiling interface, Cray has implemented an alternative library interface to all SHMEM functions to support user-level redefinition of library routines. All functions are defined as weak symbols with a set of shadow functions whose names are prefaced by a 'p'. For example, the library defines `shmem_put()` as a weak symbol and defines `pshmem_put()` as the actual function. This makes it possible for an application to define it's own version of the function that in turn calls the underlying library function. This mechanism makes it easy to extend the implementation to use SMARTMAP for intra-node transfers.

At library initialization time, we determine which destination ranks are on the local node. We do this using information from the Catamount runtime system that conveys the rank, node id, and core of each process in the job. We actually use the SMARTMAP capability for each process on a node to determine a global rank to core rank mapping. Once this mapping is determined, we simply add logic to each function to determine whether the destination process is on-node or off-node. For on-node communications, we use the virtual address conversion function to determine the remote virtual address to use and then perform the appropriate operation. If the destination rank is off-node, we fall through to the actual function. Figure 3 shows the implementation of the `shmem_putmem()` routine using SMARTMAP. We have done this for the basic put and get operations in order to measure the performance gain from SMARTMAP. Implementations of the strided get/put operations as well as the atomic memory operations would be similarly straightforward.

Changes to the internal implementation of the collective operations would be needed to differentiate between on-node and off-node data movement.

B. MPI Point-to-Point Communication

We have modified the Open MPI implementation to make use of SMARTMAP. We chose Open MPI because it is the only open-source implementation that supports shared memory that already has support for the Cray XT. Recently, Cray has released an implementation of MPI for their compute node Linux environment that supports shared memory. However, this implementation is encumbered with SGI contributions and is not available outside of Cray. Cray is continuing to maintain a completely separate MPI implementation for Catamount, which is also not available as open source. The modular component-based architecture of Open MPI also simplifies the introduction of a new transport layer.

There are two different paths that Open MPI can use for MPI point-to-point communications using Portals. The default path is to use a PML module that implements MPI matching semantics inside the MPI library and uses the underlying Byte Transport Layer (BTL) to simply move bytes. This layer can make use of several BTL modules at one time, including shared memory or the network as appropriate for the destination. The second path is for the PML to use a Matching Transport Layer (MTL). This path assumes that the underlying module is responsible for implementing MPI matching semantics. Unlike the BTL, there can only be one of these modules in use at any given time. An important distinction between these two paths is the location of the MPI receive queue. For the BTLs, the MPI receive queue is inside the library, but for an MTL, the receive queue is managed outside of the MPI library.

We modified both the shared memory BTL in Open MPI as well as the Portals MTL to use SMARTMAP. This approach allows us to better quantify the advantage of avoiding an extra copy in the shared memory BTL.

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 required to enable the Portals MTL to use SMARTMAP. A detailed description of a prototype of this implementation can be found in [17]. The prototype only had support for intra-node transfers, but it has since been extended to support both on-node and off-node communication. This implementation has two posted receive queues – one inside Portals for off-node transfers and one inside the MPI library for on-node transfers – so it is subject to the same complexity that other such implementations are. In particular, non-specific receives are not currently fully supported. If a receive request using `MPI_ANY_SOURCE` cannot be immediately completed, a failure is returned. We are currently extending the implementation to handle this situation. We do not expect this extra logic to have a significant impact on performance, especially for communication micro-benchmarks and codes that do not employ a large number of wildcard receive requests.

A key difference between the BTL and MTL implementations is that the 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 transfered 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, short messages using SMARTMAP are synchronous – the data is only copied when the matching receive buffer has been posted.

C. MPI Collective Communication

We have also created an Open MPI collective communication module that uses SMARTMAP to implement the barrier, broadcast, reduce, allreduce, and alltoall collective operations. We briefly describe the implementation here.

The SMARTMAP collective module uses a structure containing the following information:

- `counter`
- `context`
- `address`
- `turn`
- `finished`

This structure is globally-scoped so that it is at the same memory location in all of the processes on a node. The first two items, `counter` and `context`, are specific to the MPI communicator involved in the collective operation. Since MPI collective operations are blocking, a process can be participating in at most one collective at a time. The communicator's `counter` is incremented each time a collective operation is started and the `context` is used to identify the specific communicator that is being used. This prevents sub-communicators in overlapping collective operations from interfering with each other.

When a process enters a collective operation, it first determines whether it is the root of the collective operation, For non-rooted operations, the root defaults to rank 0 within the communicator. Once the root is determined, the process determines the core on which the root process is running. If a process is not the root, it gets the remote address of the collective structure in the root process' address space and waits for the `counter` and `context` values to indicate that the root has entered the same collective operation.

For the barrier operation, the root process initializes the `finished` value to 1, sets the `counter` and `context` values appropriately, and then spins on this value waiting for it to be equal to the size of the communicator. Once the non-root processes enter the collective operation, they increment the `finished` value in the root's address space using an assembly language atomic increment operation. As with the root, they also spin waiting for this value to be equal to the size of the communicator.

For the broadcast operation, the root process again initializes the `finished` value to 1 and sets the `address` value to the location of the user buffer. It then waits for the other processes to increment the `finished` value. Once the non-root processes enter the collective operation, they read the `address` value in the other process' address space, convert this value to a remote address, and then copy the data from the source buffer directly to the destination buffer in its address space. When the copy is complete, the process atomically increments the `finished` value.

For the reduce operation, the root process first copies the send buffer to the receive buffer (provided the MPI_IN_PLACE flag is not used), initializes the `finished` value to 1, and initializes the `turn` value to 0. It sets the `context` and `counter` values, and then proceeds as the non-root processes do. Once in the collective operation, a non-root process recognizes the destination address and converts it to a remote address in the root core's address space. It then waits for the `turn` value to be equal to its rank. Once this occurs, the process performs the reduce operation with the its buffer and the root's buffer. When the reduce operation is complete, it atomically increments the `turn` value to let the next rank proceed, and atomically increments the `finished` value to indicate that it is done. When the root process' turn is up, it simply increments the counter to let the following rank proceed. As with the other SMARTMAP collectives, the root waits for the `finished` value to reach the size of the communicator.

Currently, the allreduce operation is implemented as a reduce followed by a broadcast. The alltoall operation is implemented as a broadcast with each process taking turns being the root. The current implementation of alltoall is cache friendly, since all cores are copying the same buffer at the same time. An alternative implementation could allow for each process to copy its chunk of data to the other processes.

V. Performance Evaluation

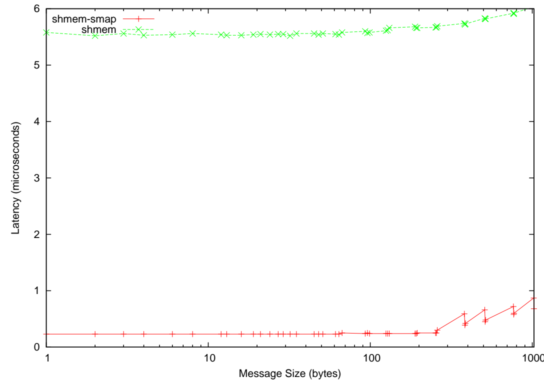
A. Test Environment

The platform used to gather our performance results is a Red Storm development system that contains four 2.2 GHz quad-core Opterons. We have added SMARTMAP capability to the Catamount N-Way (CNW) kernel version 2.0.41. Our changes to Open MPI were performed on the head of the development tree.

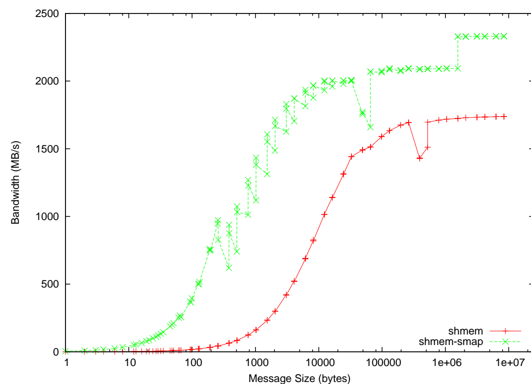
For intra-node results, we limited our results to the interrupt-driven version of Portals because it is more efficient at intra-node transfers. The ability to have the operating system perform a copy between processes outperforms having the SeaStar adapter do the copy. Due to limitations of the SeaStar, 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.

B. SHMEM

Figure 4 shows the ping-pong latency and bandwidth performance for a Cray SHMEM put operation using the default implementation and the SMARTMAP-enabled implementation as measured with the NetPIPE [18] benchmark. Single-byte latency for the default implementation is more than 5 μs, while the SMARTMAP latency is 230 ns. Bandwidth performance for the SMARTMAP-enabled SHMEM also significantly outperforms the default implementation, having a much steeper curve and achieving much higher asymptotic performance. The erratic nature of the bandwidth curve for the SMARTMAP-enabled SHMEM is due to the sensitivity of the memory sub-system to misalignment as a result of the various transfer lengths that NetPIPE uses. The dip at 32 KB is repeatable and is also likely due to the memory hierarchy, since this is half of the size of the first-level cache. We used a memory copy routine with non-temporal stores, which we believe is responsible for the jump in bandwidth performance at 2 MB. For SHMEM over Portals, the crossover point from using shared memory to using the network is clearly visible at 512 KB.



(a) Latency



(b) Bandwidth

Fig. 4: SHMEM Put Performance

C. *MPI Point-to-Point*

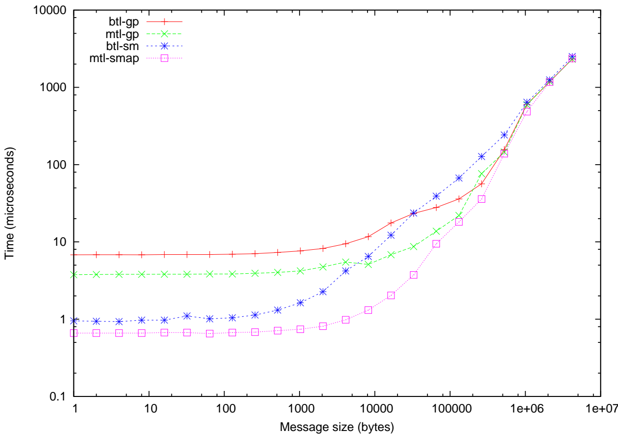
Figure 5 shows the performance of widely-used Intel MPI Benchmark suite version 2.3 for the point-to-point operations. We compare the default Portals BTL (btl-gp) and MTL (mtl-gp) with the shared memory BTL (btl-sm) using SMARTMAP and the SMARTMAP Portals MTL (mtl-smap).

Ping-pong latency performance is shown in Figure 5(a). The Portals MTL with SMARTMAP is able to achieve a zero-byte latency of 630 ns, with the shared memory BTL using SMARTMAP slightly higher at 830 ns. This is a significant improvement over the 3 μsPortals MTL, where the OS performs the memory copy between the processes. The difference in performance between the MTL and BTL is likely due to the additional memory operations needed by the BTL to enqueue a request in a shared data structure. For the MTL, each process has exclusive access to the data structures necessary for enqueing a request.

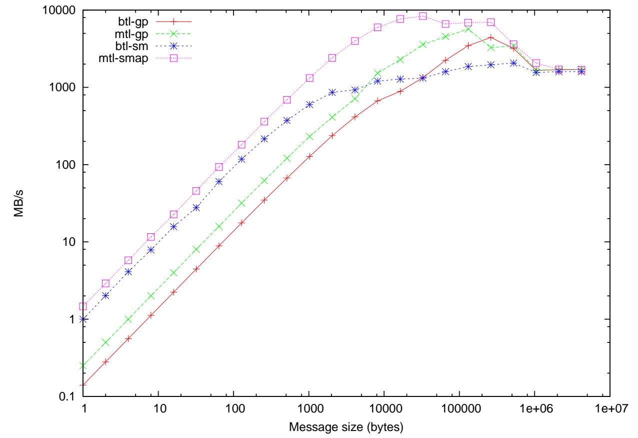
Ping-pong bandwidth performance is shown in Figure 5(b). Here again we see that the SMARTMAP-enabled MTL is able to outperform the others, peaking at 9.4 GB/s. This is significantly higher than the peak 5.7 GB/s of the Portals MTL without SMARTMAP. We can also see that the performance of the shared memory BTL starts to be affected by doing two memory copies rather than one. Unlike the previous SHMEM bandwidth test that uses NetPIPE, the IMB bandwidth test does not actually read the receive buffer, so the improved performance of MPI over SHMEM is due to cache effects.

Figures 5(c) and 5(d) show performance for the IMB Sendrecv and Exchange benchmarks. We chose these benchmarks to illustrate the capability of SMARTMAP to allow for simultaneous communications within a node. The Sendrecv benchmark measures performance between pairs of processes communicating with the `MPI_Sendrecv()` function, while the Exchange benchmark measures the performance of exchanging data with a pair of neighbor processes. The Portals BTL and MTL are limited by serialization through the OS, while with the shared memory based transports, the processes are able to communicate without any serialization. We can also see the penalty that the two-copy shared memory strategy has for these operations as well.

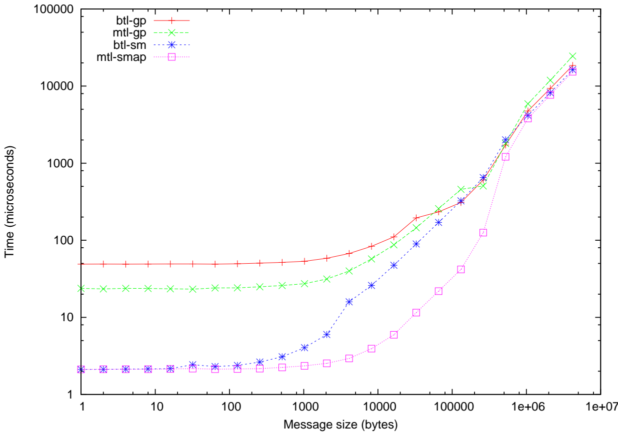
Another important measurement of MPI point-to-point performance is message rate. We used the PathScale (now QLogic) MPI message rate benchmark, which is a modified version of an MPI bandwidth benchmark from Ohio State University. The original benchmark was enhanced to support reporting message rate as well as bandwidth, to calculate and report the $N^{1/2}$ message size and rate, and to allow for running multiple processes per node to calculate aggregate performance. Figures 6(a) and 6(a) show the message rate for one pair of communicating processes and two pairs of processes respectively. For one pair, the shared memory BTL is able to achieve more than 3.5 million messages per second, while the Portals MTL with SMARTMAP achieves about 2.4 million message per second. The non-SMARTMAP layers achieve less than 300 thousand messages per second. The memory copies in the shared memory BTL allow for decoupling the sender and the receiver. For short messages, the BTL is able to copy the message into shared memory and, from the MPI perspective, the send is complete. However, since the SMARTMAP MTL is synchronous, it does not perform the memory copy until the receiver has posted a receive request. The overhead of this synchronization degrades message rate performance, but the single-copy ability of SMARTMAP eventually catches up at larger message



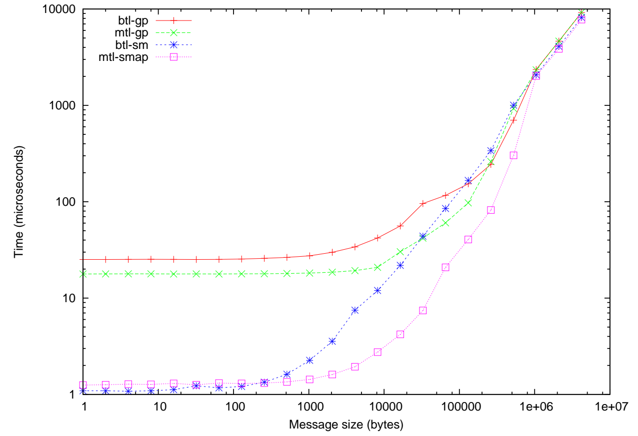
(a) MPI PingPong Latency



(b) MPI PingPong Bandwidth



(c) MPI Exchange



(d) MPI Sendrecv

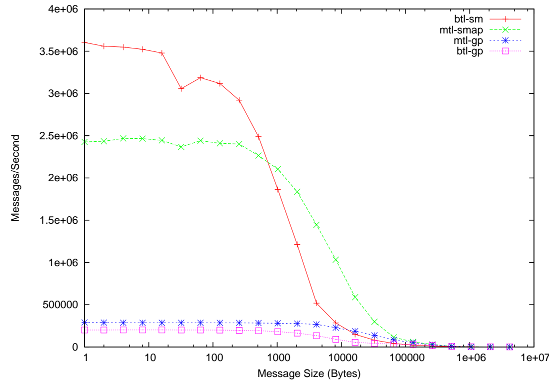
Fig. 5: IMB MPI Point-to-Point Results

sizes. For two pairs of processes, message rate for the MTL scales nearly linearly, almost doubling to 4.6 million messages per second, while the BTL rate remains constant. The message rate actually decreases slightly for the Portals-based transports.

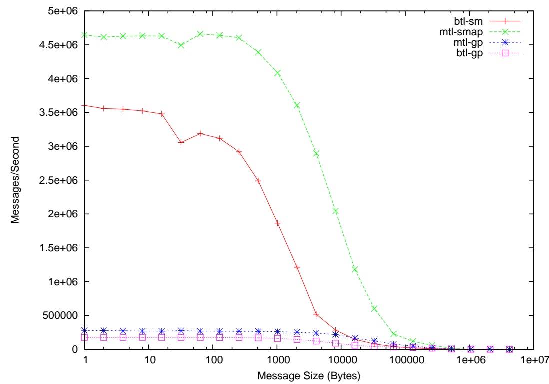
We finish our analysis of MPI point-to-point communication with a halo exchange benchmark from Argonne National Lab. We ran this benchmark across four quad-core nodes using sixteen processes. The results are shown in Figure 7. Unlike the intra-node performance results, this benchmark shows the advantage of the AP version of Portals. The Portals MTL with SMARTMAP enabled allows for efficient on-node transfers, while the AP implementation of Portals allows for more efficient off-node transfers.

D. MPI Collectives

Figure 8 shows performance for the broadcast, reduce, allreduce, alltoall, and barrier MPI collective operations on a single quad-core node. This graph also includes performance of the SMARTMAP collective module (smap-coll). As with the point-to-point operations, we can again see the significant performance gain for using SMARTMAP. For the broadcast and alltoall operations in Figures 8(a) and 8(d) respectively, we can also see the advantage that the single copy approach has for larger message sizes over the two-copy approach of the shared memory BTL. Barrier performance in Figure 8(e) demonstrates the advantage of using a counter in shared memory rather than using message passing in shared memory.



(a) One pair



(b) Two pairs

Fig. 6: MPI Message Rate

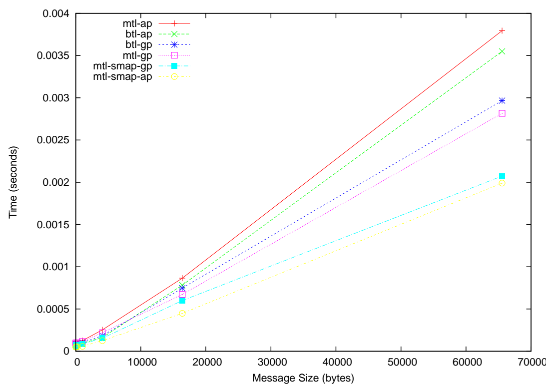


Fig. 7: MPI Halo Exchange Performance

VI. Conclusion

The SMARTMAP capability in Catamount 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 approaches for intra-node data movement using Portals on the Cray XT. We expect the shared memory BTL performance to be similar to what Cray's Compute Node Linux (CNL) environment could achieve using shared memory in Linux. However, we have also shown that the single-copy ability of SMARTMAP in Catamount is able to significantly outperform the mulitple-copy approach that must be used in a POSIX-based shared memory environment like Linux. Additionally, SMARTMAP can support operations that Linux shared memory cannot. First, SMARTMAP can eliminate *all* extraneous memory-to-memory copies for intra-node MPI communications. This is a significant advantage in light of the growing memory bandwidth limitation of multi-core processors. SMARTMAP can also support true one-sided get/put operations and extremely efficient collective operations, including the ability to perform reduction operations directly on the destination buffer.

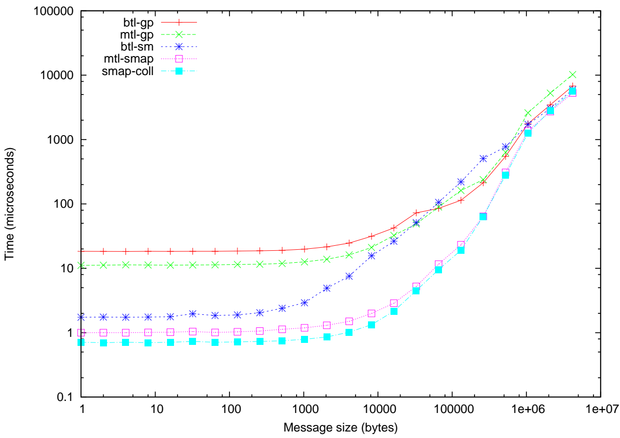
VII. Future Work

There is more work left to do to fully utilize the SMARTMAP capability for MPI. First, because the Portals data movement layer encapsulates the MPI posted receive queue, the complexity of handling `MPI_ANY_SOURCE` receives is significantly increased. The current implementation does not fully support non-specific receives, but we do not expect the logic needed to support them to significantly impact performance. We would also like to implement single-copy non-contiguous data transfers and MPI-2 remote memory access operations.

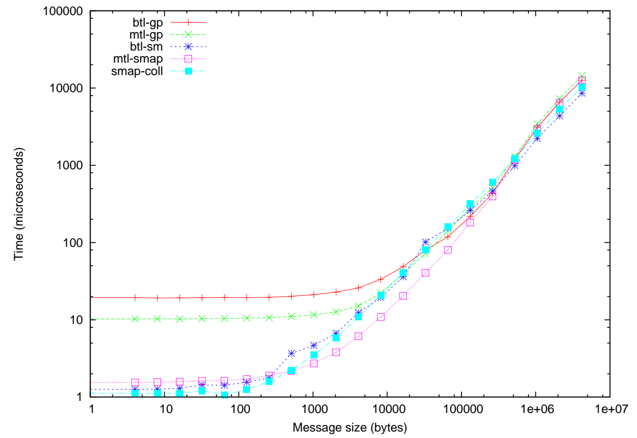
We are currently working on additional collective operations for the SMARTMAP collective module, specifically the gather operations. We would like to do an in-depth analysis of collective performance using Open MPI's hierarchical collective module, where on-node collectives would use the SMARTMAP module in combination with a network-based collective module.

With the recent release of a Cray implementation of MPI for CNL that supports shared memory transfers, we would like to do an in-depth analysis of on-node MPI communication performance between Catamount and CNL.

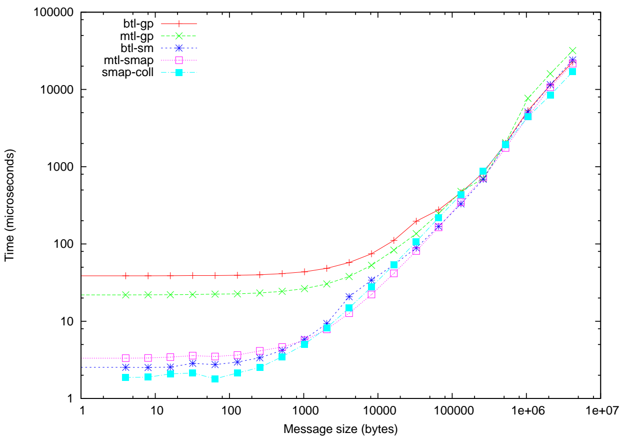
Once we have complete point-to-point and collective layers, we would also like to perform and in-depth analysis of application performance. Our current 4-node quad-core environment is not sufficient to analyze application



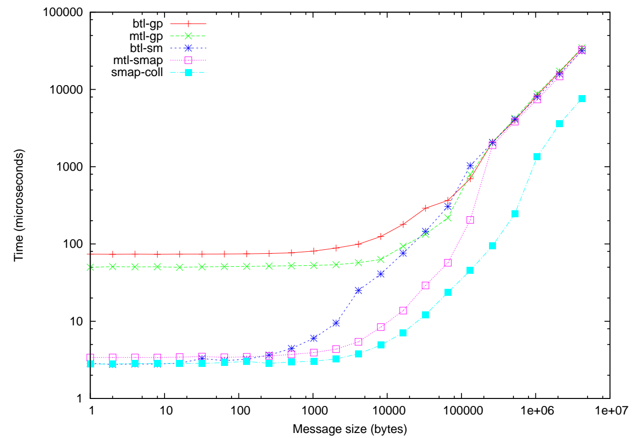
(a) MPI Broadcast



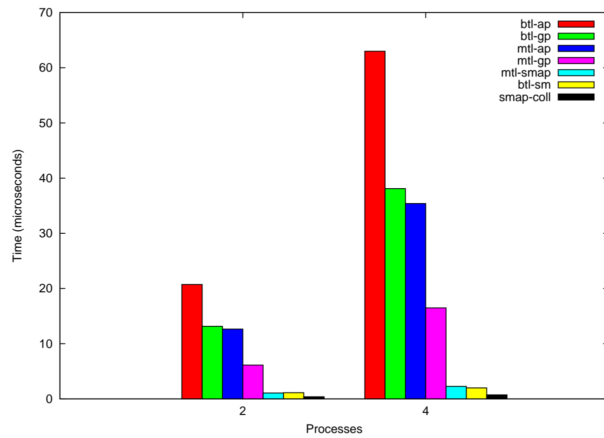
(b) MPI Reduce



(c) MPI Allreduce



(d) MPI Alltoall



(e) MPI Barrier

Fig. 8: IMB MPI Collective Performance

performance and scalability. The center section of the Red Storm system, approximately six thousand nodes, will soon be upgraded to quad-core processors, and we expect to perform an exhaustive analysis of applications as part of the upgrade. It would also be interesting to measure SMARTMAP performance on larger core counts, such as a dual-socket quad-core Cray XT5 node.

SMARTMAP is also a natural fit for implementation of the Partitioned Global Address Space (PGAS) Model. The implementations of Unified Parallel C, Co-Array Fortran, and Global Arrays could be enhanced to leverage SMARTMAP capabilities.

We are also exploring ways for applications to use the SMARTMAP capability directly, through library interfaces that allow processes to do direct remote loads and stores. We currently have MPI applications that are conducive to recoding pieces of them to use shared-memory style communications. The advantage of SMARTMAP for this is that we can avoid the memory copy overhead of using MPI and also avoid the complexity of mixing MPI with threads or OpenMP compiler directives.

Finally, we are also considering exposing the topology of the underlying machine to applications using MPI communicators. We can easily create communicators to be used for on-node or off-node communications (e.g, `MPI_COMM_NODE` and `MPI_COMM_NET`). Some applications may be able to decompose communication into two levels to better leverage the advantages of intra-node communication performance.

VIII. Acknowledgments

The implementation of SMARTMAP would not have been possible without the efforts of John Van Dyke, who is responsible for implementing virtual node mode support in Catamount. Kurt Ferreira also provided many useful discussions regarding virtual memory mapping and lightweight kernel memory management. The authors also gratefully acknowledge the assistance of Sue Kelly and the Cray support staff at Sandia with the Red Storm development systems. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

References

[1] F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks," in *Proceedings of the ACM/IEEE International Conference on High-Performance Computing and Networking (SC'00)*, November 2000.

[2] D. S. Henty, "Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling," in *Proceedings of the ACM/IEEE International Conference on High-Performance Computing and Networking (SC'00)*, November 2000.

[3] S. Dong and G. E. Karniadakis, "Dual-level parallelism for deterministic and stochastic CFD problems," in *Proceedings of the ACM/IEEE International Conference on High-Performance Computing and Networking (SC'02)*.

[4] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 189–202, December 1993.

[5] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem," *Parallel Computing*, vol. 33, no. 9, pp. 634–644, September 2007.

[6] ——, "Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem," in *Proceedings of the 2006 European PVM/MPI Users' Group Meeting*, September 2006.

[7] ——, "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*, May 2006.

[8] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Limic: Support for high-performance MPI intra-node communication on Linux," in *Proceedings of the 2005 Cluster International Conference on Parallel Processing*, June 2005.

[9] K. Feind and K. McMahon, "An ultrahigh performance MPI implementation on SGI ccNUMA Altix systems," in *Proceedings of the SGI Users' Group Technical Conference*, June 2006.

[10] D. Buntinas, G. Mercier, and W. Gropp, "Data transfers between processes in an smp system: Performance study and application to mpi," in *Proceedings of the 2006 International Conference on Parallel Processing*, August 2006.

[11] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda, "Designing an efficient kernel-level and user-level hybrid approach for MPI intra-node communication on multi-core systems," in *Proceedings of the International Conference on Parallel Processing*, September 2008.

[12] R. L. Graham and G. Shipman, "MPI support for multi-core architectures: Optimized shared memory collectives," in *Proceedings of the 15th European PVM/MPI Users' Group Conference*, September 2008.

[13] 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.

[14] R. Brightwell, T. Hudson, K. T. Pedretti, and K. D. Underwood, "SeaStar interconnect: Balanced bandwidth for scalable performance," *IEEE Micro*, vol. 26, no. 3, May/June 2006.

[15] S. M. Kelly and R. Brightwell, "Software architecture of the light weight kernel, Catamount," in *Proceedings of the 2005 Cray User Group Annual Technical Conference*, May 2005.

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

[17] R. Brightwell, "A prototype implementation of MPI for SMARTMAP," in *Proceedings of the 15th European PVM/MPI Users' Group Conference*, September 2008.

[18] Q. O. Snell, A. Mikler, and J. L. Gustafson, "NetPIPE: A network protocol independent performance evaluator," in *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, June 1996.