# Optimizing Multi-Core MPI Collectives with SMARTMAP

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

***Abstract*—We recently enhanced a lightweight compute node operating system with a simple page table mapping strategy that allows parallel application processes within a node to share effectively a single address space. This strategy allows for each process to maintain a normal, private address space, but also allows for processes to behave like threads that can freely read and write the memory of all processes on a node. In this paper, we demonstrate the benefits of this approach for improving the performance of MPI collective operations. In particular, we describe a new multi-threaded MPI reduce algorithm that outperforms existing approaches by as much as a factor of seven on a quad-core processor.**

## I. Introduction

The increasing core count on commodity processors used in high-performance computing (HPC) creates several challenges for massively parallel systems and applications. One of the most significant challenges is dealing with the decreasing amount of memory bandwidth per core. The amount of effective memory bandwidth available to each core is declining, and this problem is exacerbated by the fact that the most popular parallel programming model, message passing using the Message Passing Interface (MPI), requires explicit copying of data between application processes. As such, a range of different approaches are being explored to address this issue. Some application developers are turning to mixed-mode programming, where MPI is used for communication between nodes, and an alternative strategy – typically multi-threading with pthreads or OpenMP – is used for parallel computations within a node. At the same time, system software developers are exploring alternative strategies for intra-node data movement that attempt to minimize the impact on memory bandwidth.

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 this paper, we continue to explore the benefits of a simple operating system page table mapping strategy for multicore processors that allows the parallel processes within a node to behave effectively as threads within a single address space. We have called this strategy SMARTMAP, Simple Mapping of Address Region Tables for Multicore-Aware Programming. With SMARTMAP, each process maintains its own private address space, but is also able to read and write easily the memory of the other processes on a node. In previous work [1], we introduced SMARTMAP and described its implementation in a lightweight operating system designed for massively parallel distributed memory systems. Subsequently, we described an initial prototype implementation of MPI point-to-point operations [2] as well as an initial implementation of MPI collective operations [3] using SMARTMAP. While our previous implementation of MPI collective operations demonstrated significant performance improvement by minimizing on-node memory-to-memory copies, we did not fully exploit the available parallelism. This study describes subsequent enhancements to two important MPI collective operations, global reduction and global exchange, further emphasizing the benefits that a simple OS page table mapping strategy for multicore processors can have on MPI performance.

The rest of this paper is organized as follows. The next section provides background on the current approaches for using shared memory for intra-node MPI communication. In Section III, we provide details on SMARTMAP and its implementation in two different lightweight compute node operating systems. Section IV describes the implementation of several MPI collective operations and how they have been optimized using SMARTMAP, and Section V contains detailed performance results and analysis. We conclude in Section VI with a summary of this study and our plans for future

work with SMARTMAP.

## II. Background

In order to accelerate intra-node data transfer, 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. For MPI point-to-point operations, the sending process must copy the message into a shared memory region and the receiving process must copy it out – a minimum of two copies must occur. Ideally, messages could be moved directly between the two processes with just 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. Furthermore, message passing programming interfaces like MPI do not place any restriction on the location of the source or destination address, allowing message buffers to be located anywhere in an address space, including the process's data, heap and stack.

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 a 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 use most effectively 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 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].

## III. SMARTMAP

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

### A. Catamount

Catamount [13] is a third-generation compute node operating system developed by Sandia National Laboratories with Cray, Inc., as part of the Red Storm project [14]. 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 full-featured 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 during the life of a process.

SMARTMAP 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.

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.

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 a variable with global scope is the same in every process. A "local" virtual addresses can be converted to a "remote" virtual address by simply flipping a few bits at the upper part of the address. This makes it extremely easy for one process to read and write the corresponding data in another process's 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, X86-64 included, maintain a unique set of address translation structures (e.g., a page table tree on X86-64) for each process and a single set for each group of threads. Our mapping strategy operates differently in that a process's address space and associated translation structures are neither fully-unique or fully-shared. For example, our map on the X86-64 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 X86-64) and a larger cache footprint than necessary. Modifications to Linux to support sharing a single page table entry for shared memory mapped regions have 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, our mapping strategy 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 our strategy just as efficiently. On other architectures with software-based virtual memory support (i.e., a software managed translation look-aside buffer), our strategy is straightforward to implement.

### *C. MPI*

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 using shared memory for intra-node transfers that also has support for the Cray XT. We have added point-to-point and collective modules to Open MPI that use SMARTMAP directly. SMARTMAP is also able to emulate POSIX shared memory regions, which allows for using the shared memory modules in Open MPI as well. In previous work [2], we described the implementation of these modules and showed the performance benefit that SMARTMAP provides. The Open MPI collective modules allow for implementing collective operations using MPI point-to-point operations or directly using an alternative transport.

## IV. SMARTMAP Collective Optimizations

In this section, we describe how we have optimized several MPI collective operations using SMARTMAP. In contrast to point-to-point operations, collective communication involves a group of processes that all must participate and cooperate to complete the operation. Broadcast and barrier are two common collective operations. MPI also defines several other collective operations, including a global reduction operation (MPI_Reduce), a complete global reduction (MPI_Allreduce) and a complete exchange operation (MPI_Alltoall). We begin by describing the basic infrastructure for implementing collective operations using SMARTMAP.

We have implemented a collective module in Open MPI that is enabled when all of the cooperating processes are running on the same node. Each process defines the following globally-scoped data structure, placing it at the same virtual address in each process's address space.

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

The first two elements, `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 being used. This prevents sub-communicators in overlapping collective operations from interfering with each other.

MPI collectives fall into two categories: those that are rooted and those that are not. In a rooted collective operation, all processes participate, but only one process receives the result. In a non-rooted collective, all processes participate and all processes receive the result.

When a process enters a rooted collective operation, it first determines whether it is the root. If it is the root process, it initializes the other elements in the `coll_info_t` structure with the appropriate information and then sets `counter` and `context` to signal that the root has joined the collective. If a process is not the root, it determines the address of the collective info structure at the root and spins waiting for the root to enter the collective operation. Some collective operations require that each process initialize its local collective info structure with local data before synchronizing with the root. This is true in the case of the threaded MPI_Reduce operation described below.

### A. Reduce

The MPI_Reduce function combines the elements provided by each process using an operation and returns the combined values across all processes to the root process. The routine must be called by all group members with input buffers of the same length and with elements of the same type. The root must call the operation with an output buffer that is the same length and same type as the input buffer. MPI has several predefined operations for reductions, such as sum, product, minimum and maximum, and also allows user-defined operations.

Our initial implementation of reduce took advantage of the global address space capability of the OS to eliminate any extra memory-to-memory copies. The reduce operation was performed in-place using the root process's receive buffer directly. The following describes this initial implementation.

To begin the reduce, the root process first copies the send buffer to the receive buffer, initializes the `finished` value to one, and initializes the `turn` value to zero. It sets the `context` and `counter` values, and then proceeds as the non-root processes do. A non-root process reads the root's receive buffer address and converts it to a remote address in the local process'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 using its local send buffer and the root's receive 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. The root then waits for its `finished` value to reach the size of the communicator, at which point the root's receive buffer now contains the correct result. Once a non-root process has completed its part of the reduction, it can exit the operation.

We have since optimized the reduction operation to be "threaded". Instead of serializing access to the receive buffer by each process, we have implemented a new algorithm that divides up the work evenly among all of the processes and lets them proceed in parallel, acting as threads rather than processes.

In this new algorithm, each process fills in its local `send_buff` and `recv_buff` information in the collective info structure and initializes `finished` to

zero. Each process then determines an offset and length based on its rank. On a quad-core processor, for example, each process is responsible for processing a quarter of the data. When each process has finished initializing its local collective info structure, it atomically increments the `turn` counter at the root. All process then wait for the `turn` counter at the root to be equal to the number of processes participating in the collective. Each process then loops over all of the ranks, converting the send buffer address of each rank to a remote address and performing the operation on their chunk of the data at the root. For example, if the rank zero process is the root, it copies a piece of the send buffer into the receive buffer and then proceeds to perform the reduce operation using the remaining rank's send buffers. It converts the address of rank one's send buffer to a remote address and then performs the reduce operation on its piece of the receive buffer. It continues by converting rank two's send buffer to a remote address and so on. After each step, it atomically increments the `finished` counter in the corresponding rank's collective info structure. Once it has processed its chunk for every rank, the process waits for its local `finished` counter to be equal to the number of participating processes so that it knows it is safe to leave the reduce function.

One limitation of this algorithm is that it does not support operations where the local rank is part of the computation. For example, MPI has two predefined operations, MPI_MAX_LOC and MPI_MIN_LOC, that not only return the arithmetic result, but also return the rank of the process containing the maximum or minimum value. Since the process performing the computation may not be the actual process that contains the value, we use the serial algorithm for these operators and for any user-defined operator.

### B. 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.

### C. 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.

### D. 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.

## V. Performance

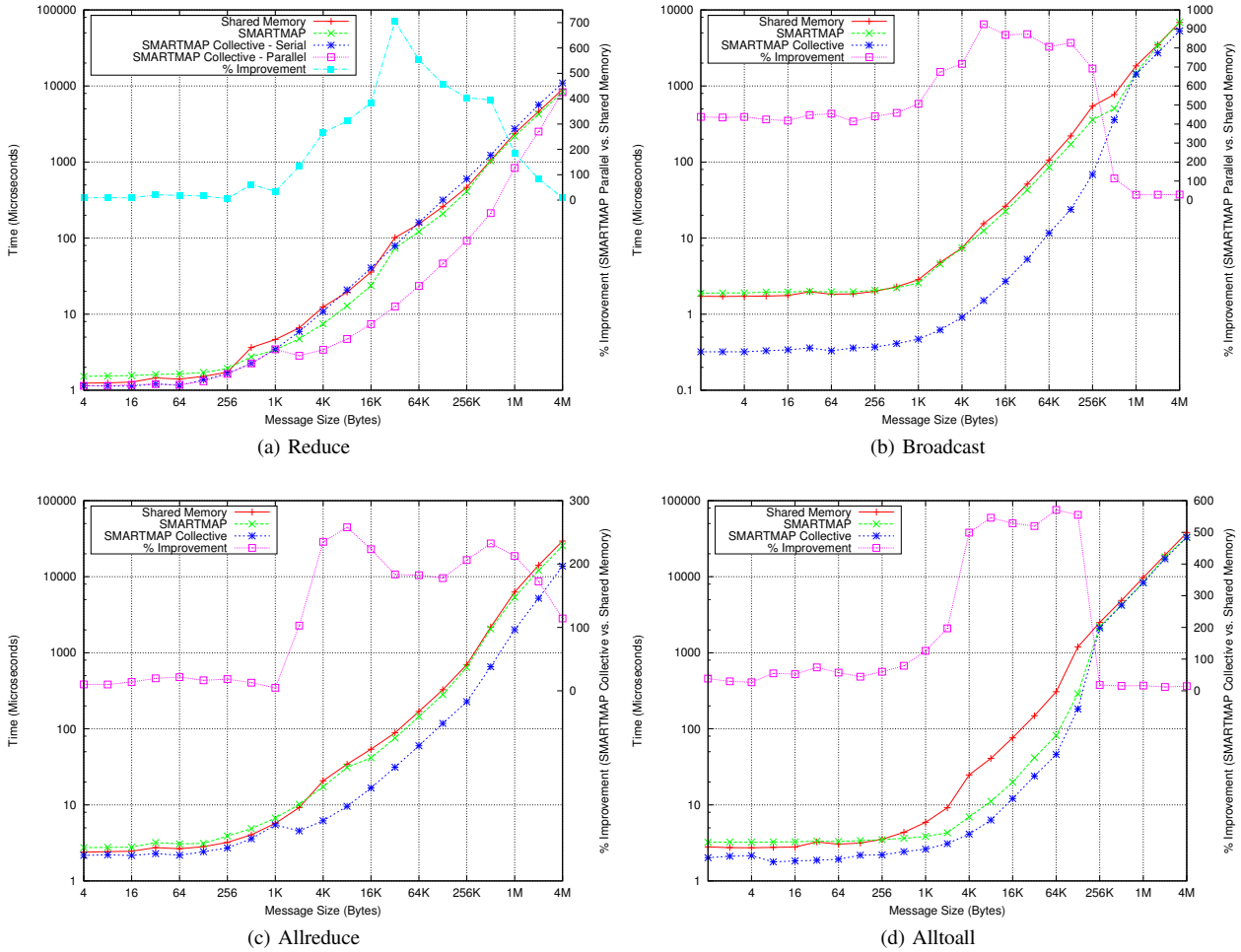
### A. Test Environment

The platform used to gather our performance results is a Red Storm development system that contains 44 2.2 GHz quad-core Opterons. The compute nodes were running Catamount version 2.0.41 which has been enhanced with SMARTMAP. Our changes to Open MPI were performed on the head of the development tree. We used the Intel MPI Benchmark suite, version 2.3 to measure performance.

### B. Intra-node Collectives

Figure 1(a) shows the performance of MPI_Reduce on a single quad-core processor for increasing message sizes. It compares the implementation layered on MPI point-to-point operations using SMARTMAP-emulated POSIX shared memory (Shared Memory) and SMARTMAP point-to-point operations (SMARTMAP), the original serial direct SMARTMAP implementation (SMARTMAP Serial) and the new parallel direct SMARTMAP implementation (SMARTMAP Parallel). The parallel algorithm is not active until the buffer is at least 1024 bytes, so the SMARTMAP direct implementation defaults to the serial algorithm.

From the graph, we can see that the serial and parallel algorithms have identical performance up to 1024 bytes, at which point the parallel algorithm significantly outperforms the other approaches. The implementation layered on SMARTMAP point-to-point (SMARTMAP) starts out poorly because the data transfer is synchronous




(a) Reduce (b) Broadcast

(c) Allreduce (d) Alltoall

Fig. 1: Quad-core MPI collective performance

– a message is not delivered until the receiver has requested it. In contrast, the shared memory implementation (Shared Memory) is asynchronous – message delivery is complete at the sender as soon as the message is copied into shared memory. Copying short messages is a win up to about 1024 bytes, where the overhead of two copies becomes significant and the single-copy approach of SMARTMAP is faster.

We have also plotted the percentage of improvement that the SMARTMAP parallel algorithm has over the layered shared memory approach in Figure 1(a). At a message size of 32 KB, the SMARTMAP parallel approach is an improvement over the shared memory approach of more than 700%.

Figure 1(b) shows the performance of MPI_Bcast on a single quad-core processor for increasing message sizes. The SMARTMAP collective module shows a significant performance increase over both of the layered implementations. At the 8 KB message size, SMARTMAP is nearly 10x faster.

Figure 1(c) shows the performance of the MPI_Allreduce operation on a single quad-core processor. The SMARTMAP collective module again shows a significant performance increase of 2-3x over the shared memory implementation for messages from 1 KB to nearly 1 MB.

Figure 1(d) shows the performance of Alltoall for layered shared memory (Shared Memory), layered SMARTMAP (SMARTMAP), and direct SMARTMAP (SMARTMAP Collective). We can see that the SMARTMAP collective module significantly outperforms the others for all message sizes. We can also see that the asynchronous behavior of the shared memory implementation again has an advantage out to a size

of a few hundred bytes, at which point the single-copy strategy of the SMARTMAP point-to-point module has better performance. At medium message sizes, the advantage of the single-copy strategies using SMARTMAP is clearly evident. From message sizes of 4 KB to 128 KB, the SMARTMAP collective is more than a 500% improvement over using shared memory.

### C. Hierarchical Collectives

Since we are mostly concerned with parallel application performance at large scale, we are interested in whether the optimization of on-node collective operations leads to a performance increase at larger scales. The Open MPI implementation contains a hierarchical collective module that accounts for the locality of the processes involved in the operation. It breaks the collective operation down into on-node and off-node components. For example, a broadcast operation is implemented by first sending the data to a local leader process on each node using a network transport and then broadcasting the data among the processes within a node.

We have modified the Open MPI hierarchical collective module to be able to use the SMARTMAP collective module for intra-node operations. Figure V-C compares the performance of the hierarchical collective implementations of reduce, broadcast, and allreduce using SMARTMAP and shared memory on 128 processes (32 quad-core processors). Currently there is no hierarchical implementation of alltoall.

Figure V-C(a) shows the performance of the hierarchical reduce operation. The SMARTMAP collective module is able to outperform the baseline shared memory implementation for nearly all messages sizes, peaking at almost a 30% improvement for 2 MB messages. Figure V-C(b) shows the performance of the hierarchical broadcast operation. SMARTMAP again achieves better performance than the shared memory baseline getting a 5-20% performance increase for messages from 4-512 KB. Finally, Figure V-C(c) shows the performance of the hierarchical allreduce. As expected, SMARTMAP shows a similar level of performance increase beyond shared memory for this operation.

## VI. Conclusion and Future Work

In this paper, we have described SMARTMAP, a simple page table mapping strategy for multicore processors that provides the ability for parallel application processes within a node to read and write easily each others' memory. We have implemented SMARTMAP in two lightweight compute node operating systems designed for massively parallel distributed memory systems. We described how SMARTMAP can be used to implement MPI collective communication operations more efficiently than traditional POSX shared memory approaches. SMARTMAP not only provides the ability to do single-copy MPI message within a node, but also allows for "threading" a collective reduction operation to achieve an even greater level of performance. We presented a performance comparison using a well-known MPI communication benchmark that demonstrates the advantage of the SMARTMAP approach for several collective operations. We have also shown that the intra-node optimization provided by SMARTMAP translates to a performance increase for hierarchical collective operations when using up to 32 quad-core nodes.

For future work, we are planning to do an in-depth analysis of the performance benefit of SMARTMAP-enabled MPI for real applications at scale. We are in the process of obtaining dual-socket quad-core boards from Cray to explore the scalability of SMARTMAP on an eight-core system. We expect the performance improvements offered by SMARTMAP to be even more significant as core count increases.

We are exploring more ways to exploit the SMARTMAP capability within a lightweight kernel beyond MPI. 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.

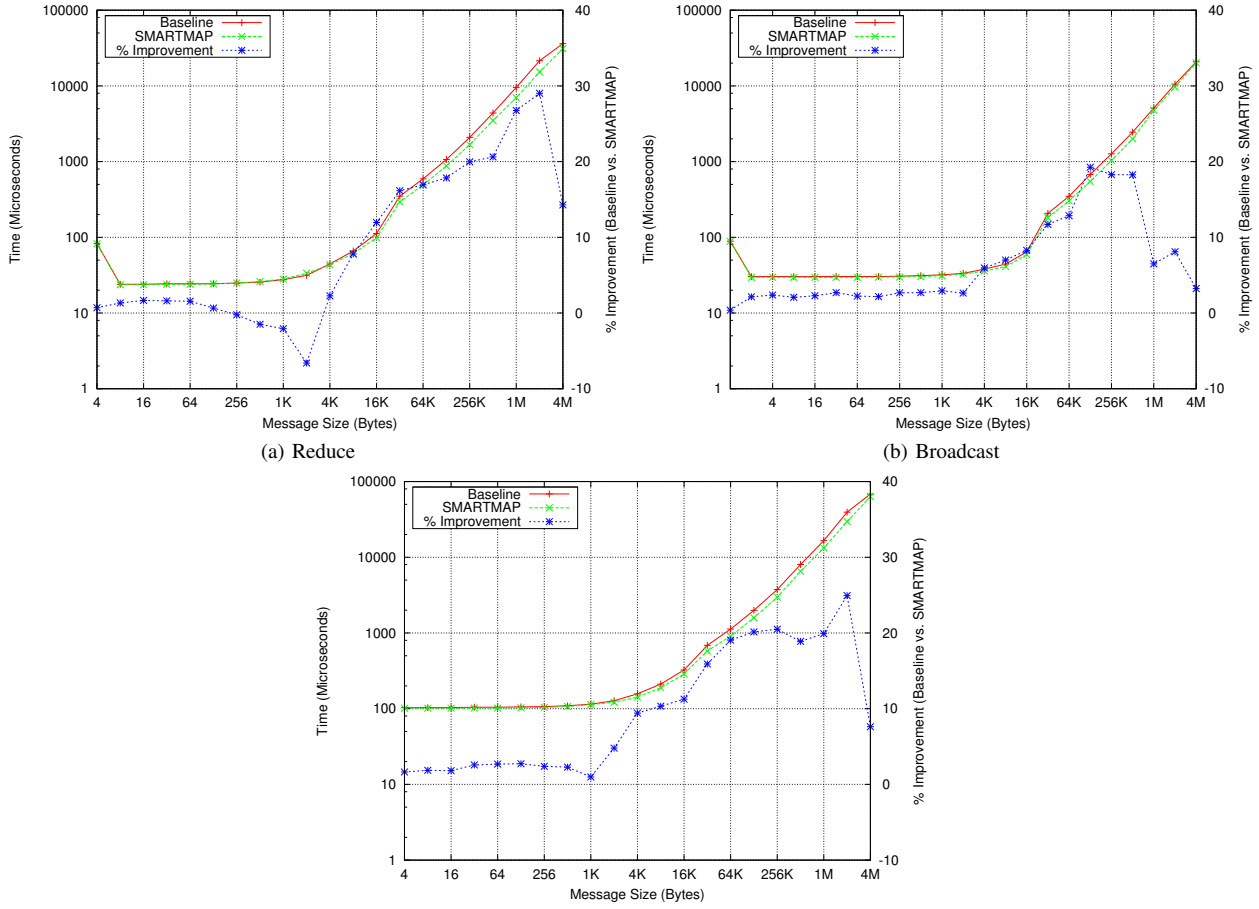
## References


[1] R. Brightwell, "Lightweight kernel support for direct shared memory access on a multi-core processor," in *Proceedings of the First Workshop on Managed Many-Core Systems*, June 2008.

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

[3] R. Brightwell, T. Hudson, and K. Pedretti, "SMARTMAP: Operating system support for efficient data sharing among processes on a multi-core processor," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'08)*, November 2008.

[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.




(a) Reduce

(b) Broadcast

(c) Allreduce

Fig. 2: Hierarchical MPI collective performance


[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] 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.

[14] 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.