# An Intra-Node Implementation of OpenSHMEM Using Virtual Address Space Mapping

Ron Brightwell Kevin Pedretti

Sandia National Laboratories *
PO Box 5800
Albuquerque, NM 87185-1319
{rbbrigh,ktpedre}@sandia.gov

## Abstract

The recent OpenSHMEM effort has generated renewed interest in developing a portable, high-performance implementation of the SHMEM programming interface. One advantage of SHMEM is the simplified one-sided communication model, but the traditional UNIX shared memory model does not support the single-copy semantics offered by SHMEM for exchanging data between processes on the same processor or within the same memory coherency domain. In this paper, we describe an initial implementation of the OpenSHMEM programming interface that uses operating system virtual address space mapping capabilities to provide efficient intra-node operations. We describe the details of the implementation for two different operating systems, Linux and the Kitten lightweight kernel, and we use micro-benchmarks to compare the performance of our implementation to the SHMEM implementation available on a Cray XE6 platform.

## 1. Introduction

The recent OpenSHMEM effort [4] has generated renewed interest in developing a portable, high-performance implementation of the SHMEM [6] Partitioned Global Address Space (PGAS) library. An important part of facilitating the use of OpenSHMEM on current systems is providing high-performance for processes running on the same multi-core processor or in the same memory coherency domain. Sandia National Laboratories recently developed an implementation of OpenSHMEM for the latest-generation Portals network programming interface [1], and we used this implementation as a basis for developing an implementation focused on efficient intra-node operations.

The main issue with implementing OpenSHMEM or any one-sided communication library is that the traditional UNIX shared memory model does not fully support single-copy semantics for exchanging data between processes. Tradtional shared memory can be used to implement the symmetric heap in OpenSHMEM, but is not able to support one-sided transfers for symmetric data objects outside the symmetric heap. Similar to two-sided message passing, there are multiple approaches that can be used to implement operations between processes on the same node.

Intra-node OpenSHMEM operations can be implemented using an area of memory that is shared among all of the processes on a node. This is how most MPI libraries implement intra-node message passing. For two-sided send-receive operations, the sender copies data into the shared memory region, and the receive copies it out. The drawback of this approach for one-sided operations is that some asynchronous progress agent must exist at the target, since the target does not explicitly participate in the operation. The most straightforward progress agent would be a thread for each process that recognizes incoming requests or data in the shared memory region and reacts to it. The major drawbacks of using a thread per process are the overhead of consuming extra processor cycles and the complexity of managing responsiveness.

Intra-node operations could also be implemented via the network interface, just like off-node transfers. The network interface could recognize on-node operations and perform them directly through a memory-to-memory copy mechanism. The major drawback of this approach is that all requests are serialized through the network interface, which creates a bottleneck for all on-node operations and potentially interferes with off-node operations as well.

Finally, the operating system could assist in on-node one-sided operations, either by performing the operations via a

* Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.



*PGAS '11* Galveston, TX

kernel thread or by re-mapping address spaces appropriately. In the former case, the operating system could be extended to provide the support necessary to copy data between the address spaces of the communicating processes. Two drawbacks of using the operating system are the overhead of making a system call to transfer data and serialization of requests. Ideally a one-side put operation would have very low latency, and operating system traps are generally expensive. The alternative, which we explore in this paper, is to use a mechanism where the operating system provides direct load-/store access to the memory of another process using virtual addressing support. Every process on the node maps the address space of the other processes into its own address space so that it can easily read and write the memory of the other processes.

We have implemented this memory mapping capability in a new lightweight kernel and have developed an implementation of OpenSHMEM to make use of it. We also show how this implementation can work for a similar memory mapping capability provided for the Linux operating system. We compare the performance of our implementation of OpenSHMEM against the implementation of SHMEM provided by Cray on their XE6 platform. The rest of this paper is organized as follows. The following section provides background on the SHMEM PGAS library and discusses related work. Section 3 describes the implementation of the virtual memory mapping capabilities of the two operating systems. We present the details of the OpenSHMEM implementation in Section 4, and Section 5 analyzes performance of the implementation for several micro-benchmarks. We summarize the results of this paper and describe a plan for future activities in Section 6.

## 2. Background

The SHMEM library was first available on the Cray T3 [14, 15] series of machines in the early 1990's. It provided lightweight one-sided peer communication, collective communication, and synchronization operations. Implementations for other platforms and networks have been developed, including the series of networks from Quadrics [12]. There was also a previous effort to develop a general portable implementation of SHMEM for clusters [11]. Currently, SGI holds the copyright on the SHMEM programming interface [16].

The SHMEM programming interface is an attractive alternative to traditional two-sided message passing with message selection capabilities because it eliminates the complex matching operations required at the target of the operation and avoids the implicit need to synchronize the movement of data between the initiator and target. Because data movement is one-sided, the initiator of a peer communication operation (put or get) knows the destination of the data at the target, which can eliminate the need for any intermediate buffers. Unlike MPI two-sided message passing where the receiver must determine where the data ends up by matching the contents of the message and potentially buffer the message until the receiver provides a matching buffer, SHMEM semantics encourage and support moving data directly from the initiator to the target without any intermediate buffering. Eliminating the implicit synchronization between sender and receiver supports applications that have highly unstructured communication patterns. The value of such remote memory access (RMA) operations was recognized by the MPI community, motivating the addition of RMA operations to the MPI-2 [10] standard. The emergence of networks with remote DMA capability, such as InfiniBand, have arguably been influenced by the capabilities first provided by the one-sided put and get operations first available in SHMEM.

## 3. Virtual Address Space Mapping

We recently demonstrated a simple address space mapping capability in the Catamount lightweight kernel called SMARTMAP [3], which uses top-level page table entries of a processor core to map the address space of all of the cooperating processes in the same parallel job on the same multi-core processor. We demonstrated how this mechanism could be used to implement single-copy MPI message passing and efficient on-node MPI collective operations. We have implemented this capability in our latest open-source lightweight kernel, Kitten [13], and we describe that implementation here. A Linux kernel module has also recently been released that enables a similar capability, although it does so in a manner typical of a virtual-memory-based UNIX operating system. This capability was initially developed by SGI for their Altix system to enable "cross-partition memory mapping" and is known as XPMEM [18]. We describe the details of these two approaches below.

### 3.1 Kitten SMARTMAP

Kitten is an open-source compute node operating system designed specifically for high-performance computing. It employs the same philosophy of previous lightweight kernels from Sandia and the University of New Mexico. Kitten also addresses several of the limitations of these previous operating systems. Kitten provides partial Linux binary compatibility so that standard tool chains and systems libraries, including the GNU standard C library, can be used. The resulting binary executable can be run on either Linux or Kitten. Kitten currently targets the x86_64 architecture, but could easily be ported to other architectures. The code leverages Linux for bootstrap and initialization, but subsystems that are critical for scalability and performance, such as scheduling and memory management, have been replaced. Unlike previous lightweight kernels that were export controlled, Kitten has been released under version two of the GNU Public License.

The SMARTMAP implementation in Kitten is essentially identical to the previous implementation in Catamount. Each

process is assigned to a single core of a multi-core processor, and one top-level page table entry is used to map the virtual address space of that process. The other processes on the node are mapped using the remaining entries in the top-level page table. These duplicate mappings offset by the top-level entry provide the ability to take a virtual address in the address space of a process, set the proper bits high in the address to get to the appropriate top-level page table entry, and access that same virtual address in the address space of another process running on a different core. Replicating the top-level page table entry for all processes on all cores takes less than twenty lines of code in Kitten. More details on the SMARTMAP implementation in Catamount can be found in [3].

### 3.2 Linux XPMEM

SGI developed a similar capability for mapping the virtual address space of processes in their IRIX operating system. IRIX provided a lightweight process called an *sproc* that had separate data, heap, and stack segments like a process, but *sprocs* were able to attach segments of other *sprocs* to their own address space. These segments of other *sprocs* were mapped at different virtual addresses in each *sproc*, so a virtual address offset translation was needed to be able to access the memory of other *sprocs*. This mechanism was used to implement SHMEM and explore optimized MPI communications [17].

SGI subsequently provided this capability in Linux using a kernel module called XPMEM on their Altix platform. An Altix running multiple OS images was identified as "partitioned" and the term "cross-partition" was used to refer to functionality that spanned partitions. The "cross-partition memory" (XPMEM) module allowed a process in one partition to identify portions of its address space that other processes running in the same or different partition can map into their own address spaces. The XPMEM kernel module is approximately 2200 lines of source code.

In addition to a kernel module, there is a user-level library that provides access to the XPMEM capabilities. Our OpenSHMEM implementation uses three of these functions. The `xpmem_make()` routine returns a unique handle that can be used to identify the segment of the address space of a process that it has made available and to which other processes can attach. A process that obtains this handle can then call `xpmem_get()`, which returns a handle that can be used to access the mapped segment from the other process. The `xpmem_attach()` routine then takes this handle and maps the identified memory segment of the other process into the calling process at a virtual address determined by the operating system.

## 4. Implementation

In this section, we describe the implementation of the various operations in OpenSHMEM. For most of the operations, the implementation using SMARTMAP in Kitten and XPMEM in Linux is identical. They only differ in initialization and how a remote virtual address is calculated.

```
static inline
void *
remote_address(unsigned rank,const void *vaddr)
{
  uintptr_t addr=(uintptr_t)vaddr;

  addr |= ((uintptr_t)(rank+1))<<39;

  return addr;
}
```

**Figure 1.** Kitten code for converting a local address to a remote address.

### 4.1 Initialization

When an application calls the OpenSHMEM library initialization function (`start_pes()`) in the Kitten implementation, the library determines the number of processing elements (PEs) participating in the job and the PE rank of the calling process. Kitten provides this information by using standard Linux system calls, namely `getpid()` and `sched_getaffinity()`. The lower sixteen bits of the process identifier are used to represent the rank of the process, and the non-zero bits in the CPU set returned from the processor's affinity mask represent the number of active processes in the job.

Following this, the process allocates memory for the symmetric heap out of its normal heap. The size of the symmetric heap is 64 MB by default and can be overridden by an environment variable. Kitten guarantees that this address will be identical across all processes provided that all processes call `malloc()` the same number of times with the same arguments before initializing the OpenSHMEM library. Our implementation uses the `dlmalloc` library from Doug Lea [8] to implement the allocation routines. We configure `dlmalloc` to never use `mmap()` for allocation.

The function for converting an address in the address space of the local process to an address that can be used to read or write the same location in the address space of another process is called `remote_address()`. It takes an address and a PE rank and returns an address. In Kitten, this routine simply takes the input address and combines it with a mask that sets the appropriate bits based on the destination PE rank. This code is shown in Figure 1.

For the Linux implementation, we use the SLURM resource manger to launch the processes on a node. We use the SLURM environment variables `SLURM_PROCID` and `SLURM_NPROCS` for each process to determine the number of processes in the job and its PE rank. We allocate the symmetric heap in the same manner as Kitten. We use the Linux symbols `data_start` and `end` to determine the starting address and length of the data section. The `xpmem_make()`

```
static inline void *
remote_address( unsigned rank, const void *vaddr )
{
  uintptr_t addr = (uintptr_t) vaddr;
  uintptr_t base;
  uintptr_t rbase;
  uintptr_t offset;
  uintptr_t raddr;

  if (((char *)addr > (char *)shmem_internal_data_base)&&
      ((char *)addr < (char *)shmem_internal_data_end)){

    base = (uintptr_t)shmem_internal_data_base;
    rbase = (uintptr_t)shmem_internal_data_rbase[rank];

  } else

  if (((char *)addr > (char *)shmem_internal_heap_base)&&
      ((char *)addr < (char *)shmem_internal_heap_end)){

    base = (uintptr_t)shmem_internal_heap_base;
    rbase = (uintptr_t)shmem_internal_heap_rbase[rank];

  }

  offset = addr - base;

  raddr = rbase + offset;

  return raddr;
}
```

**Figure 2.** Linux XPMEM code for converting a local address to a remote address.

routine will only accept regions that are aligned to the start of a memory page, so we call `getpagesize()` and align the starting addresses of the data section and symmetric heap appropriately. Each process then calls `xpmem_make()` twice, once for the data section and once for the symmetric heap, obtaining two segment ids. Every process then opens a file in `/tmp` and writes its segment ids. Then each process opens the files written by the other processes, and calls `xpmem_get()` to obtain a handle for each region in the other process. Finally, the process calls `xpmem_attach()` to map those regions into its own address space. Each process ends up with an array that contains two virtual address entries for all other processes, one for the start of the data section and one for the start of the symmetric heap.

The function for converting a symmetric address to a remote address for XPMEM, shown in Figure 2, simply uses a bounds check to determine whether the address is a data segment address or a symmetric heap address, and then calculates the offset from the beginning of the appropriate region. It then uses this offset from the start of the corresponding PE's virtual address to determine the correct remote address.

### 4.2 Peer Communication

The block and strided put and get operations for the different supported types are all implemented using two inline functions. The put routine converts the target of the put operation on a remote PE to a remote address and then calls `memcpy()` to move the data. The get operation does essentially the same thing, but copies from the remote address into the local address. Code for these two operations is shown in Figure 3. Since the put operation always copies the data into the target buffer, there is no issue with ordering, and hence `shmem_fence()` is a no-op. The `shmem_quiet()` routine requires a memory store fence to ensure store operations are visible to subsequent load operations on other cores.

```
static inline
int
shmem_internal_put(void *target,const void *source,
                   size_t len,int pe)
{
  void *target_r = remote_address(pe,(void*)target);

  memcpy(target_r,source,len);

  return 0;
}


static inline
void
shmem_internal_get(void *target,const void *source,
                   size_t len,int pe)
{
  void *source_r = remote_address(pe, (void*)source);

  memcpy(target, source_r, len);

}
```

**Figure 3.** Inline functions for performing put and get operations.

### 4.3 Collective Communication

The various OpenSHMEM broadcast operations are all implemented by the same algorithm. Each of the PEs in the active set copy the contents of the root PE's source buffer into its target buffer.

The symmetric work array at the root of the broadcast contains two flags, `start` and `end`, and a counter. When the root of the broadcast enters the operation, it first initializes the flags and the counter. It sets the `start` flag to a non-zero value to signal the other PEs that the broadcast operation has begun. The root then performs its local operation, copying the source buffer into the target buffer. Once the local copy is complete, the root waits for the other PEs in the active set to increment the counter, indicating that all of them have entered the broadcast and copied the root's source buffer into their target buffers. Once all PEs have incremented the counter, the root re-initializes the `start` flag, resets the counter, and sets the `end` flag to a non-zero value. The root then waits for all PEs to increment the counter flag again before exiting the broadcast.

Each non-root PE in the active set first converts the address of its symmetric work array and its source address to a remote addresses in the root PE's address space. It then blocks waiting for the `start` flag to become non-zero. Once the flag is set, it copies the contents of the root PE's source buffer into its target buffer. When the copy is complete, it

atomically increments the counter, and then waits for the root to set the `end` flag to a non-zero value. When this flag is set, the PE atomically increments the counter again and exits the broadcast operation.

This broadcast algorithm is similar to the algorithm used for on-node MPI broadcasts using SMARTMAP [3]. However, the MPI broadcast is slightly less complex because it is easier for an MPI implementation to keep track of the particular broadcast operation that is being performed. Since all ranks in the same communicator must call the MPI broadcast routine in the same order, each rank can keep a counter associated with the communicator to track the current round. Because OpenSHMEM has no structure that represents an active PE set, care must be taken to ensure that the root does not re-enter a broadcast operation and re-initialize flags before non-root PEs have exited the previous barrier operation. This extra synchronization is the reason that each PE must increment the counter twice – once to indicate they have copied the data from the root and once more to indicate that they have left the operation. The root must ensure that it is the last PE to leave the operation.

The algorithm for the reduction operations works similarly to broadcast, but uses two counters, `pe` and `done`. When the root enters the reduction, it copies the contents of the source buffer into the target buffer. It then increments `pe` with the appropriate PE stride value to let the next PE in the active set make its contribution. The root then waits for all PEs in the active set to increment the `done` flag.

The non-root PEs in the active set convert the address of the symmetric work array and target array to remote addresses in the root PE's address space. Each non-root PE then blocks waiting for the value of `pe` to become its PE rank, at which point it performs the appropriate operation on the target buffer using its source buffer. It then increments `pe` with the appropriate PE stride value to let the next PE go. Each PE then waits for the value of `pe` to be greater than the size of the active set, indicating that all PEs have made their contribution. At this point, all PEs then copy the remote target buffer into their local target buffer and atomically increment the `done` flag.

The collect algorithm for OpenSHMEM works nearly in the same way as the reduction algorithm, but in addition to keeping a counter that indicates which PE should go next, the algorithm also keeps an offset counter so the non-root PEs are able to keep track of the offset where their contribution must be copied. Since the fcollect routines have an identical length for all PEs, every PE can copy its contribution into the target array once the operation has begun. Every PE waits for the operation to start, copies its contribution, then waits for all PEs to contribute. Each PE then copies the entire result, atomically increments a counter indicating it has completed the copy, and then atomically increments another counter as it exits the function.

### 4.4 Atomic Operations

The OpenSHMEM atomic operations are implemented using x86-64 assembly instructions to ensure atomicity, since all of these operations are being performed on shared memory. The `shmem_*_add` operations use the lock assembly instruction (LOCK) followed by an add instruction. The `shmem_*_fadd` operations use the lock instruction followed by an exchange-and-add (XADD) operation. This is also used to implement the `shmem_*_finc` operations. The `shmem_*_swap()` and `shmem_*_cswap` operations use the exchange (XCHG) and compare and exchange (CMPXCHG) instructions.

### 4.5 Synchronization

The `shmem_barrier` and `shmem_barrier_all` routines are implemented using an algorithm that is similar to the broadcast algorithm described above. The lowest PE number in the active set, or PE zero in the case of the global barrier, is chosen as the root. All PEs wait for the root to signal the start of the barrier and atomically increment a counter to indicate their participation in the barrier. They must also atomically increment a counter on the way out of the barrier so that the root is the last PE to leave the operation, allowing the root to safely re-initialize flags for the next round.

The `shmem_*_wait` and `shmem_*_wait_until` routines spin waiting for the value to change. Care must be taken to implement these functions using the `volatile` keyword to ensure that compiler optimization does not remove the spin loop.

### 4.6 Lock

OpenSHMEM provides a distributed lock using a single word on each PE as the lock. The lock is FIFO ordered and guarantees fairness. Our implementation uses a Mellor-Crummey and Scott (MCS) lock [9] algorithm to implement this distributed lock. MCS locks require two data elements at each PE representing a *signal* field and a *next* pointer, and a *last* pointer at PE 0, which is used to indicated whether or not the lock is free. Our implementation uses a single symmetric 64-bit word on each PE to represent the lock. The 64-bit word is broken down into four 16-bit components, representing the *last*, *next*, and *signal* portions of the lock, plus an unused 16 bits of pad. This algorithm is straightforward to implement using 16-bit atomic compare-and-swap and atomic swap operations.

## 5. Performance Evaluation

### 5.1 Test Platforms

We compare the performance of our OpenSHMEM implementation using XPMEM and SMARTMAP to the OpenSHMEM implementation available on the Cray XE6, which also uses XPMEM for some intra-node operations. Since our XE6 was not available for testing with a custom operating system, we evaluate our implementation using a commodity

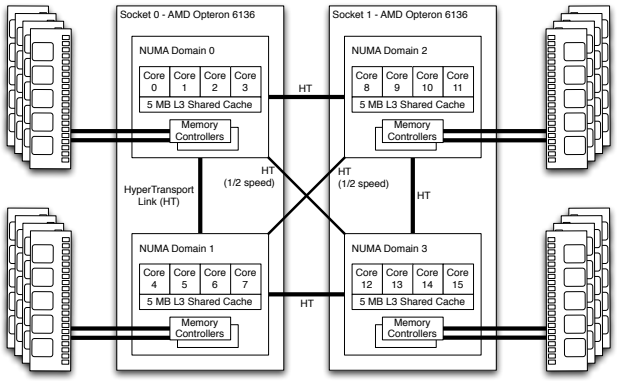


**Figure 4.** Node architecture of test systems.

HP blade system that is nearly identical to the XE6 compute nodes. The general architecture of both systems is shown in Figure 4.

Each node contains two 2.4 GHz AMD "Magny-Cours" 8-core processors, consisting of two 4-core dies each. This results in an aggregate of 16 cores per node, split into four NUMA domains. Each NUMA domain (die) has two dedicated memory controllers connected to 8 GB of DDR3-1333 memory, providing an aggregate of 32 GB of memory and 85.3 GB/s of memory bandwidth per node. The cores in each NUMA domain share a common 5 MB L3 cache, potentially accelerating intra-NUMA domain communication. All caches in the system are kept coherent via a coherency protocol operating over the HyperTransport links. Additional details on the Magny-Cours processor and system architecture can be found in [5].

The XE6 software environment consisted of the Cray Linux Environment (CLE) version 3.1.61 and Message Passing Toolkit (MPT) version 5.2.3. The default compilation environment is PGI version 11.6. The HP blade system was tested in two software configurations. In the first, a custom lightweight Linux image modeled on CLE was used to evaluate our OpenSHMEM implementation using XPMEM for intra-node communication. This image used the Linux kernel version 2.6.35.7 configured similarly to the CLE compute node kernel. In the second configuration, the Kitten lightweight kernel was used to evaluate our OpenSHMEM implementation using SMARTMAP for intra-node communication. We used a development version of Kitten for this testing, which can be obtained by checking out SHA1 ID: 384dcf6453b9 from the Kitten repository at Google Code. The compiler used for both of these configurations was gcc version 4.1.2.

### 5.2 Micro-Benchmarks

An OpenSHMEM ping-pong micro-benchmark was used to measure the latency and bandwidth between two cores. PEs were pinned to specific cores and memory affinity was used to ensure that only PE-local memory was allocated. In the test, PE0 issues a `shmem_putmem` to PE1 and then calls `shmem_wait` to wait for the reply. PE1 calls `shmem_wait` to wait for PE1's message, then sends a reply with `shmem_putmem`. Each message size was tested for 1000 trials, and the average, minimum, and maximum one-way latency and bandwidth were recorded.

The Sandia Micro-Benchmarks (SMB) [2] message rate test was ported from MPI to OpenSHMEM and used to measure the aggregate intra-node message rate achievable for an entire node. The test was configured to use all 16 cores, running one PE per core. For each iteration of the test, each PE sends 128 messages to each of six peers. After sending its messages, each PE uses `shmem_wait` to wait for the last message it is expecting to arrive. The total time for these operations is recorded and used to calculate the message rate (messages per second) achieved by the PE. The test is performed for 4096 iterations and the average per PE message rate is reported. To produce more realistic results, the caches are invalidated between each iteration, and this time is excluded from the measurement.

### 5.3 Results

In all figures in this section, "CrayXE/XPMEM" refers to a compute node of our Cray XE6 system, and "Linux/XPMEM" and "Kitten/SMARTMAP" refer to our HP blade test system running Linux and Kitten, respectively.

The measured ping-pong latency between two cores in the same NUMA domain for each OpenSHMEM implementation is shown in Figure 5. The line in each plot indicates the average latency (half round-trip time) and the error bars indicate the minimum and maximum values recorded. In general, all three OpenSHMEM implementations have comparable average latencies, with an average 16-byte latency of around 200 ns. In comparison, MPI ping-pong latency on Cray XE6 for two cores on the same NUMA node is approximately 600 ns for 16-byte messages. We also measured the latency between all combinations of two cores in different NUMA domains and the results were similar, with slightly higher latency due to the off-die communication.

The maximum latency values recorded for Cray XE6 and the HP blade system running Linux are observed to be much higher than the HP blade system running Kitten. This demonstrates a key strength of Kitten – its ability to control maximum latency, in addition to providing good average latency. This is enabled by the low OS noise environment that Kitten provides to applications. It should be noted that we made significant effort to ensure that a fair comparison was being made, and are confident that is the case. For example, the same compiler, optimization level, and system libraries were used in both environments (Kitten executes Linux binaries). We also verified the accuracy of the timing function in both OS environments and that the same physical cores were being used in all cases.

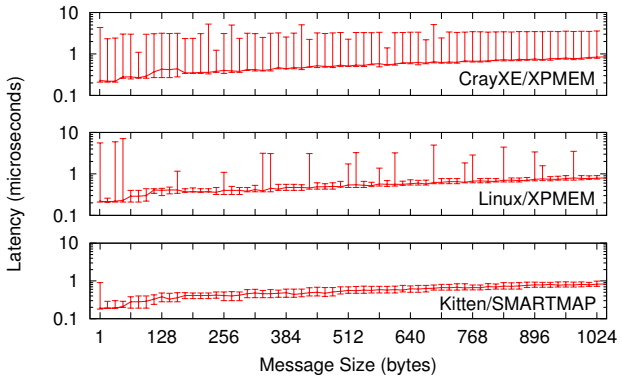


**Figure 5.** Ping-pong latency performance.

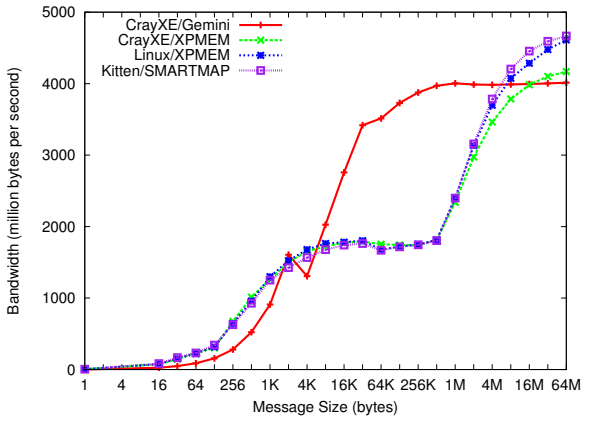


**Figure 6.** Ping-pong bandwidth performance.

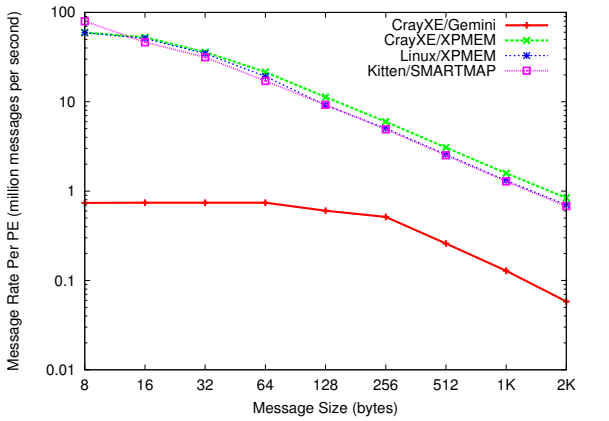


**Figure 7.** Message rate benchmark performance.

Figure 6 shows the average ping-pong bandwidth measured between two cores in the same NUMA domain. The minimum and maximum values were very close to the average in all cases, so error bars are not shown. Again, all three OpenSHMEM implementations show similar behavior. The shape of the bandwidth curve was surprising. Typically, there will be a gradually increasing bandwidth curve up to the bandwidth limit. Instead, the OpenSHMEM bandwidth curves are nearly flat from 4 KB to 512 KB messages, before starting to ramp up again. We initially theorized that this may have something to do with the `memcpy()` implementation, so we tried two other versions– a version using non-temporal store instructions and a version optimized by Cray for previous-generation Opteron processors. Neither resolved the issue. Ultimately, we determined the cause to be the system's cache coherency protocol interacting badly with our benchmark's buffer management scheme. The result was many cache-to-cache transfers between the two cores participating in the test. As others have observed, cache to cache transfers on Magny-Cours generation Opteron processors are much slower than expected [7]. We reran our benchmark on an Intel Nehalem processor, which resulted in a bandwidth curve with the expected shape.

For the Cray XE6, we also measured the ping-pong bandwidth between two cores in the same NUMA domain when using the Gemini network interface to move intra-node messages (labeled "CrayXE/Gemini"). As can be seen in the figure, this produces a radically different result. For messages from 4 KB to 2 MB, the Gemini is able to move messages much more efficiently than `memcpy()`. This is because the data path of the message data does not result in cache-to-cache transfers. Instead, cache lines move directly to and from the Gemini network interface.

The per-PE message rate achieved for the SMB message rate benchmark running on 16 cores is shown in Figure 7. The three OpenSHMEM implementations show roughly the same behavior. The differences were confirmed to be repeatable from run-to-run, and are likely due to slightly different system tuning between the Cray and HP systems. Kitten was observed to have a repeatable 35% higher message rate than the other systems for 8-byte messages, but this advantage disappears for larger messages.

We were initially surprised by the high message rates achieved by the OpenSHMEM implementations. MPI typically achieves a few million messages per second for similar tests. However, after converting message rates to bandwidths the results are reasonable. For example, the per-PE 80 million messages per second achieved by Kitten for 8-byte messages represents a system aggregate bandwidth of 9.8 GB/s. This is well within the system's peak capability of 85.3 GB/s. The difference between the observed and peak bandwidths is due primarily to the overhead of making OpenSHMEM library calls and the library-internal address translation that must be performed. In contrast, configuring Cray's SHMEM to use the Gemini for intra-node communication results in a much lower system aggregate of 90.2 MB/s. This demonstrates the negative effect of serializing all communication through the network interface.

## 6. Summary and Future Work

This paper has described the details of an implementation of the OpenSHMEM PGAS data movement library for on-node communication using operating system virtual memory mapping techniques. This implementation currently supports the Kitten lightweight kernel and Linux on modern x86-64 multi-core processors. We described the implementation using Kitten's SMARTMAP capability as well as a similar virtual address mapping capability provided by the XPMEM Linux kernel module and library. Performance results from communication micro-benchmarks were used to compare the implementation on Kitten and Linux with a production implementation of SHMEM available on the Cray XE6 platform. Results show that the OpenSHMEM implementation provides comparable performance to the Cray implementation and also indicates that Kitten latency performance is much more predictable than Linux. As part of the performance evaluation, we also identified several characteristics of a multi-socket AMD Magny-Cours multi-core processor system that can impact intra-node OpenSHMEM performance.

This paper has described our initial implementation effort, and provided some early performance results, but it has also identified several avenues of future work that are important for helping to strengthen the viability of OpenSHMEM as part of an environment for extreme-scale computing.

There are several activities related to the current implementation that we plan to pursue. Our implementation has provided the base capability for several operations but there is a significant amount of performance tuning that needs to be done, particularly with respect to collective operations and factors impacting real applications. For example, we have done an initial evaluation of using a non-temporal memory copy mechanism that showed little benefit for micro-benchmarks, but real applications that are sensitive to cache effects may behave differently. One important outcome of the OpenSHMEM effort will be to develop a broader set of application benchmarks that can be used to provide a more complete perspective on key performance issues, as few benchmarks are currently available.

We plan to integrate this on-node implementation with the Portals implementation [1] to provide a complete solution for distributed memory platforms. The implementation of the collective and atomic operations must be modified in order to consider both on-node and off-node communications. In particular, there are situations where spin waiting needs to be replaced or combined with the appropriate calls to make progress on network transfers. We recognize that integration of these two implementations may occur in an official OpenSHMEM code base.

We are also considering other ways of using the virtual address mapping capability provided by SMARTMAP and XPMEM. For example, rather than having each process map all of the other processes on a node, an approach where only a single non-application process maps in all of the application processes and coordinates moving data between them may have some benefit, such as mitigating some of the cache pollution side affects of memory-to-memory copies. Should OpenSHMEM be extended to include non-blocking operations, this approach could potentially support better overlap of communication and computation as well. This method could also support an implementation approach where a dedicated process interacts with the network and handles all requests for on-node as well as off-node communication. There are some application scenarios and networks where this approach may provide a significant performance benefit.

## References


[1] B. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, and K. Underwood. Enhanced support for OpenSHMEM communication in Portals. In *Proceedings of the 19th IEEE Annual Symposium on High Performance Interconnects*, August 2011.

[2] B. W. Barrett and K. S. Hemmert. An application based MPI message throughput benchmark. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing*, September 2009.

[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] B. Chapman, T. Curtis, C. Koelbel, J. Kuehn, S. Poole, and L. Smith. Introducing OpenSHMEM. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Models*, October 2010. (Poster).

[5] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30:16–29, March 2010.

[6] Cray Research, Inc. *SHMEM Technical Note for C, SG-2516 2.3*, October 1994.

[7] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2009.

[8] D. Lea. *A Memory Allocator*. `http://g.oswego.edu/dl/html/malloc.html`.

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[10] *MPI-2: Extensions to the Message-Passing Interface*. Message Passing Interface Forum, July 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[11] K. Parzyszek, J. Nieplocha, and R. A. Kendall. A generalized portable SHMEM library for high performance computing. In M. Guizani and X. Shen, editors, *Proceedings of the IASTED*

*Parallel and Distributed Computing and Systems*, pages 401–406. IASTED, November 2000.

[12] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.

[13] *The Kitten Lightweight Kernel*. Sandia National Laboratories. `https://software.sandia.gov/trac/kitten`.

[14] S. L. Scott and G. Thorson. Optimized routing in the Cray T3D. In *PCRCW '94: Proceedings of the First International Workshop on Parallel Computer Routing and Communication*, pages 281–294, London, UK, 1994. Springer-Verlag. ISBN 3-540-58429-3.

[15] S. L. Scott and G. M. Thorson. The Cray T3E network: Adaptive routing in a high performance 3D torus. In *Fourth IEEE Symposium on High-Performance Interconnects (HotI'96)*, August 1996.

[16] *SHMEM API for Parallel Programming*. Silicon Graphics International. `http://www.shmem.org/`.

[17] D. G. Solt. *Cooperative Data Transfer: Fast Message-Passing on Shared-Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[18] M. Woodacre, D. Robb, D. Roe, and K. Feind. *The SGI Altix 3000 Global Shared-Memory Architecture*.