On the Current State of Open MPI on Cray Systems

Nathan T. Hjelm, Samuel K. Gutierrez
Los Alamos National Laboratory
Los Alamos, NM
{*hjelmn, samuel*}*@lanl.gov*

Manjunath Gorentla Venkata
Oak Ridge National Laboratory
Oak Ridge, TN
manjugv@ornl.gov

***Abstract*—Open MPI provides an implementation of the MPI standard supporting native communication over a range of high-performance network interfaces. Los Alamos National Laboratory (LANL) and Oak Ridge National Laboratory (ORNL) collaborated on creating a port for Cray XE and XK systems. That work has continued and with the release of version 1.8 Open MPI now conforms to MPI-2.2 and MPI-3.0 on Cray XE, XK, and XC systems. The features introduced with this work include dynamic process support (`MPI_Comm_spawn()`), important for implementing fault-tolerant MPI systems; improved collective operations required for scalability and performance of applications; and Aries support to enable running Open MPI on Cray XC systems. In this paper, we present an update on the design and implementation of Open MPI for Cray systems and evaluate the performance and scaling characteristics on both Gemini and Aries networks.**

***Keywords*-Open MPI; Cray; Gemini; Aries; uGNI; Generic Network Interface; XPMEM**

I. Introduction

Open MPI is a widely used open-source MPI implementation of the MPI-3.0 specification that is developed and maintained by collaborators from academia, industry, and national laboratories [1]. It supports a wide variety of high-performance network interfaces including InfiniBand, Cray SeaStar, and Myrinet. Past work [2] introduced support for *uGNI*, which provides support for both the Cray *Gemini* and *Aries* networks. *Open MPI* aims to provide thread safety and concurrency as well as network and process fault tolerance. Additionally, Open MPI supports multiple resource managers and can natively support both direct launch (e.g. srun, aprun) and mpirun.

This paper details the modifications and extensions in *Open MPI* to support Cray XE, XK, and XC systems. These changes and extensions are already a part of *Open MPI* and released in the 1.8 series. The *vader* and *uGNI Byte Transport Layer* (BTL) components provide byte-level data transfer support for intra- and inter-node communication, respectively. The *udreg Memory Pool* (mpool) provides support for caching memory registrations. The *Application Level Placement Scheduler* (ALPS) (*Resource Allocation Subsytem* (RAS)) and *Process lifetime management* (PLM), *Open Runtime* (ORTE) components support launching via mpirun. MPI-3 one-sided support is implement with platform independent components that provide optimizations for shared-memory windows. Dynamic process support is provided by ORTE and the *uGNI* BTL when processes are launched via mpirun.

The rest of the paper is organized as follows. Section II provides a brief description of the *Gemini* and *Aries* Network Interfaces and a high-level overview of *Open MPI*. Sections III and IV provide high-level overviews of the *Open MPI* components that support Cray systems and VII outlines the performance evaluation and presents some micro-benchmark performance results. Section IX concludes with future work.

II. Background

A. Gemini Network Interface

The *Gemini System Interconnect* is the network used by the Cray XE and XK system families and is the successor to the *SeaStar** network interconnect found in XT systems. A 3D torus network is built from Gemini application-specific integrated circuits (ASICs) that provide 2 network interface controllers (NICs) and a 48-port router [3]. Two Opteron nodes are connected to a Gemini that provides 10 torus connections - 8 divided evenly between *X* and *Z* and 2 in *Y*. Link bandwidths are 4.68 to 9.375 GB/s per direction [3].

The *Aries System Interconnect* is the network used by the Cray XC system family and is the successor to the *Gemini* network interconnect. XC systems build a dragonfly network from *Aries* ASICs that provide 4 NICs and a 48-port router. More details about the *Aries* interconnect can be found in [4].

The Generic Network Interface (GNI) [5] exposes low-level, user-space communication services through uGNI, which helps facilitate the effective utilization of the underlying *Gemini* or *Aries* hardware. In particular, GNI exposes an interface that provides two mechanisms for initiating remote direct memory access (RDMA) transactions: *Fast Memory Access* (FMA) and *Block Transfer Engine* (BTE).

FMA transactions come in several forms. Short message (SMSG) and *Shared Message Queue* (MSGQ) are both used to transfer point-to-point short messages, but differ in memory resource requirements and performance characteristics. In particular, SMSG provides the lowest latency and the highest short messaging rates, but suffers from higher memory requirements due to dedicated buffers, called

Mailboxes, which are allocated on a per-peer and per-connection basis. MSGQ uses SMSG facilities for message transfers, but shares the *Mailbox* information required for an SMSG connection with all job instances located within the same node [5]. Sharing resources in this manner allows MSGQ to scale in the number of nodes, rather than in the number of peers, but does, however, come at the cost of additional performance overhead [6].

BTE is best suited for large, asynchronous message transfers. Once the transfer is initiated, up to 4 GB of data can be transfered by the Gemini hardware without CPU involvement [3].

Detailed descriptions surrounding the usage and design of GNI can be found in [5] and [6].

B. Open MPI

Open MPI's design and implementation revolves around the concept of a modular component architecture (MCA) [7]. Within Open MPI, functionality is provided by self-contained software modules with well-defined interfaces. The communication infrastructure that we chose to leverage in this port comprises three major frameworks: the point-to-point management layer (PML), the BTL management layer (BML) and, the BTL. The PML layer provides MPI semantics, the BML layer is responsible for multiplexing MPI messages, and the BTL layer is responsible for transferring data between communication endpoints. More details regarding Open MPI's architecture can be found in [1].

C. Related Work

The uGNI BTL's design is very similar to that of MPICH2's uGNI network module, which also provides MPI support for Cray XE, XK, and XC systems. The module uses an eager protocol for small and medium message transfers and a rendezvous protocol for large message transfers [6]. For message sizes greater than the SMSG message limit, MPICH2 uses the BTE PUT and GET protocols. Open MPI, however, can make use of FMA PUT, which does not require memory registration, and BTE PUT, which does not have a 4-byte alignment restriction imposed by Gemini for data buffers, for message sizes greater than the SMSG message limit. Open MPI also supports packing additional data with the in an RDMA request to handle some of the alignment restrictions of the BTE get operation. In addition, unlike MPICH2's uGNI network module, the uGNI BTL is an open-source implementation that leverages uGNI to support the *Gemini* and *Aries* Network Interfaces.

III. Enhanced Shared-Memory BTL

This section describes the changes and extensions to *vader* BTL. First, we provide an overview of XPMEM, and then briefly describe the design and implementation of *vader*.

A. XPMEM

XPMEM is a Linux kernel module and user-level library that enables a process to map the memory of another process into its virtual address space [8]. XPMEM exposes a small application programming interface (API) that comprises 7 routines: *xpmem_version*, *xpmem_make*, *xpmem_remove*, *xpmem_get*, *xpmem_release*, *xpmem_attach*, and *xpmem_detach*. XPMEM setup is a three-phase process that requires process *A* to export a region of its virtual address space, via *xpmem_make*, to a cooperating process *B*. The cooperating process then attaches to the exported region by calling *xpmem_get* and then *xpmem_attach*. Once this process is complete, *A*'s exported memory region is directly accessible to *B*. That is, *B* can perform single-copy transfers within that region via direct loads and stores, thus avoiding costs related to more traditional copy-in/copy-out (CICO) schemes that require data associated with a transfer to be copied twice – a copy into a shared memory region by the sender and a copy out of the shared memory region by the receiver. Attached regions are permitted to contain "holes," that is, virtual memory regions that are not allocated. A segmentation fault will occur if a process mapping a region tries to access unallocated memory in that region.

B. Shared-Memory BTL – vader

vader is the default shared-memory BTL component within *Open MPI*. It provides mechanisms and protocols for intra-node data transport. *vader*s design and implementation is heavily influenced by the single-copy, RDMA-like capabilities provided by XPMEM. The first version of this component was introduced in [9]. The need for a higher bandwidth, lower latency, and more portable BTL for intra-node communication on XE/XK systems was the impetus behind the implementation of this kernel-assisted shared-memory BTL.

To improve performance, we modified both SEND and RDMA protocols. Small message (< 256 bytes) latencies are improved through the use of lock-free, per-peer receive queues. For larger, contiguous messages using either the SEND or RDMA protocol, only the pointer to the user buffer is passed to the receiving process. The receiving process uses XPMEM to map the necessary pages into its memory space, and the data is given directly to the receiving PML. Copy-in-copy out semantics are used only for sending non-contiguous data. The SEND protocol was patterned after the Nemesis protocol [10] used by MPICH. Besides these protocol changes, the data structures were made more cache-friendly, and the critical path was optimized by identifying and eliminating the unnecessary instructions.

To increase portability, protocols based on Linuxs *Cross Memory Attach* (CMA) were introduced. This enables applications using *Open MPI* on systems without XPMEM to take advantage of *vader*'s optimizations.

IV. *uGNI* BTL

The *uGNI* BTL provides inter-node communication through the *BTL Send()*, *Put()*, and *Get()* functions. This functionality is provided using three protocols: short message, eager get, and long message. A high-level overview of these protocols is provided in the following subsections.

A. Initialization and Connection Setup

The default behavior of the *ugni* BTL is to retreive endpoint information from the modex and bind *uGNI* endpoints and allocate reserved receive buffers (SMSG mailbox resources) on an on-demand basis. We choose this approach so that the memory overhead and modex characteristics would be representative of the communication characteristics of the application. This approach, however, comes at the cost of some additional overhead when first communicating with a peer. Due to limitations in registration resources, *Mailboxes* are allocated in 2 MiB blocks up to the maximum size needed. For large scale jobs the mailbox allocation size is adjusted to consume no more than 12.5% of available registration resources when fully connected.

B. Short Message Protocol

The short message protocol handles calls to *BTL Send()* with messages smaller than the SMSG send limit, which is configurable at invocation time by the $btl\_ugni\_smsg\_send\_limit$ MCA parameter. This parameter is set to $autoselect$ (0) by default, which sets the SMSG limit based on the number of MPI tasks. The default SMSG limits have been updated to improve the balance between memory usage and messaging performance at medium scales. A table of the *uGNI* BTL's current SMSG limits can be found in Table I.

Short messages that fall between the SMSG send limit and the eager limit specified by the $btl\_ugni\_eager\_limit$ MCA parameter use an eager get protocol. This protocol is described in detail in [2] and [9].

Number of MPI Tasks	Default SMSG Limit
$[2, 512)$	8192
$[512, 1024)$	2048
$[1024, 8192)$	1024
$[8192, 16384)$	512
16384+	256

Table I

C. Long Message Protocol

The *Gemini* hardware imposes a 8 byte alignment and size restriction on get operations that use the BTE. To improve the performance of the *uGNI* BTL we added support for Get operations between similarly-aligned buffers. This is done by passing the extra bytes as part of the RDMA protocol exchange in the *ob1 Point-to-point Messaging Layer* (PML). More detail on the long message protocols can be found in [9] and details about the *ob1* PML can be found in [11] and [12].

V. UDREG MPool - Memory Registration

To reduce the overhead associated with memory registration, the *ugni* BTL makes use of the new UDREG memory registration pool. This registration pool stores a cache of unused registrations in a least recently used (LRU) list. Cached registrations can either be reused for future transactions or released when resources are exhausted. The UDREG mpool provides multiple improvements over the RDMA mpool. Unlike the RDMA mpool, which uses ptmalloc2 to get memory deallocation notifications, the UDREG mpool uses the ureg library available on XE, XK, and XC systems. It also provides support for allocating huge page resources for SMSG mailboxes. By default, Open MPI will use 2 MiB huge pages for SMSG mailboxes but it is configurable using the $btl\_ugni\_smsg\_page\_size$ MCA parameter. To avoid deadlock due to resource starvation, we chose to limit the maximum number of registrations a process can hold in its LRU to a fraction of available registrations. By default the limit is based on the number of active MPI processes on a compute node, but is configurable using the $btl\_ugni\_max\_mem\_reg$ MCA parameter.

VI. Additional Features

Open MPI introduced many new features throughout the 1.7 feature release series that are available in Open MPI 1.8. These features include support for dynamic creation of processes using `MPI_Comm_spawn()` and MPI-3 one-sided support. Dynamic process support is currently supported by the Open RunTime Environment (ORTE) and requires the use of Open MPI's mpirun job launcher. Inter-node MPI-3 one-sided support is currently provided by the *rdma One-Sided Communication* (OSC) component which emulates one-sided operations using two-sided communication.

VII. Evaluation

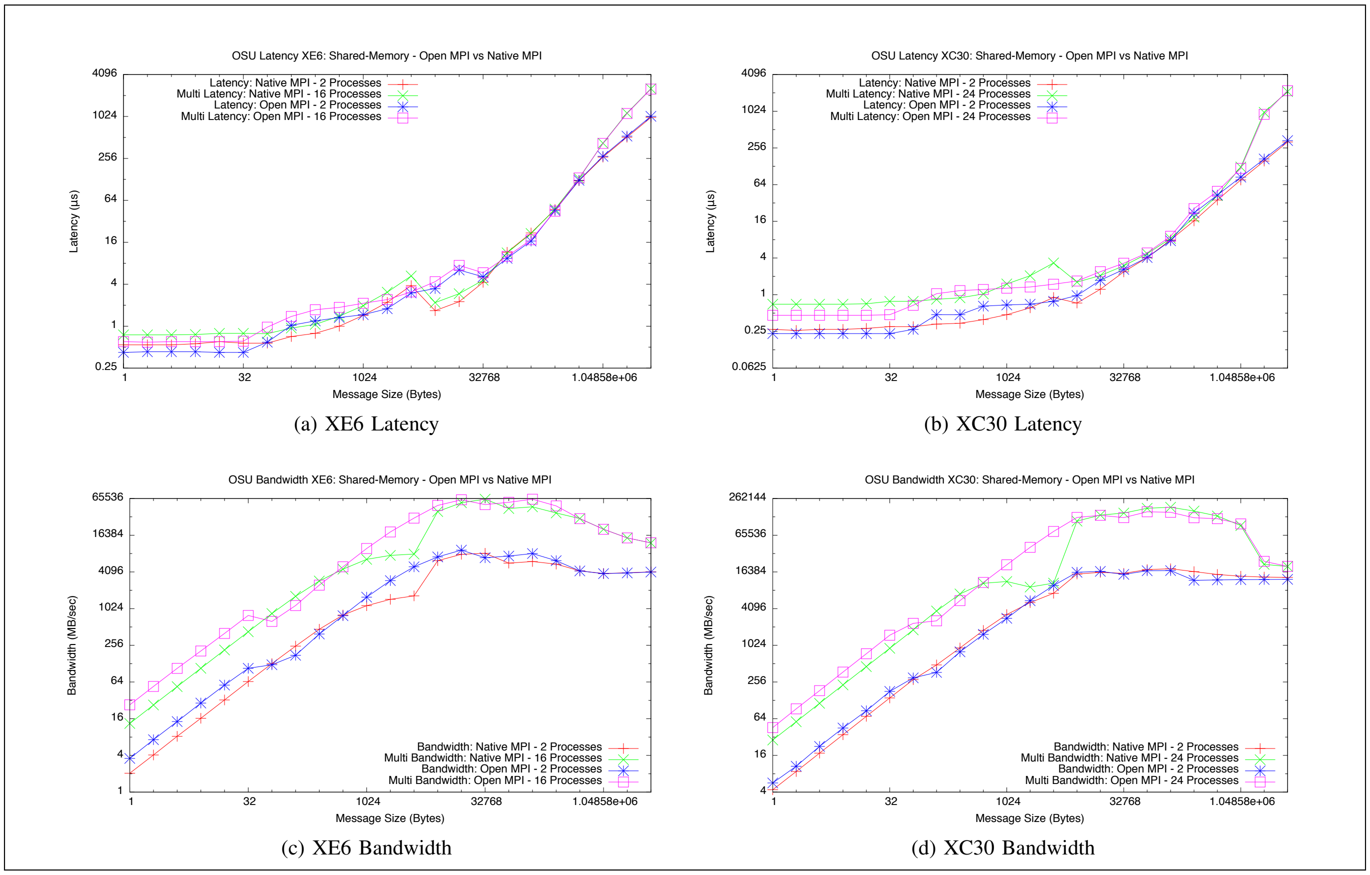
This section describes the test beds used for the evaluation of Open MPI on Cray systems. It then presents some point-to-point performance results for the *vader* and *ugni* BTLs and the point-to-point and shared memory one-sided components.

A. System Description

To evaluate the performance of the uGNI BTL, we used Cielo and Edison.

Cielo is a Cray XE6 located at LANL. The system has 322 service nodes and 8,894 compute nodes totaling 142,304 CPU cores. Each compute node has two 2.4 Ghz AMD Opteron Magny-Cours CPUs and 32 GiB memory. It uses the *Gemini* network interface for network communication.

Edison is a Cray XC30 located at the National Energy Research Scientific Computing Center (NERSC). It has



(a) XE6 Latency

(b) XC30 Latency

(c) XE6 Bandwidth

(d) XC30 Bandwidth

Figure 1: Log-log plots showing shared-memory latency and bi-directional bandwidth for Open MPI and the Native MPI on both XE6 and XC30. Results reported by OSU's MPI micro-benchmark suite. Latency measured with *osu_latency* and multi latency measured with *osu_multi_lat*. Bandwidth measured by *osu_bibw* and multi bandwidth measure with *osu_mbw_mr*.

5,576 compute nodes, each containing two 2.4 Ghz Intel Ivy Bridge CPUs and 64 GiB of memory. Edison uses the *Aries* network interface for network communication.

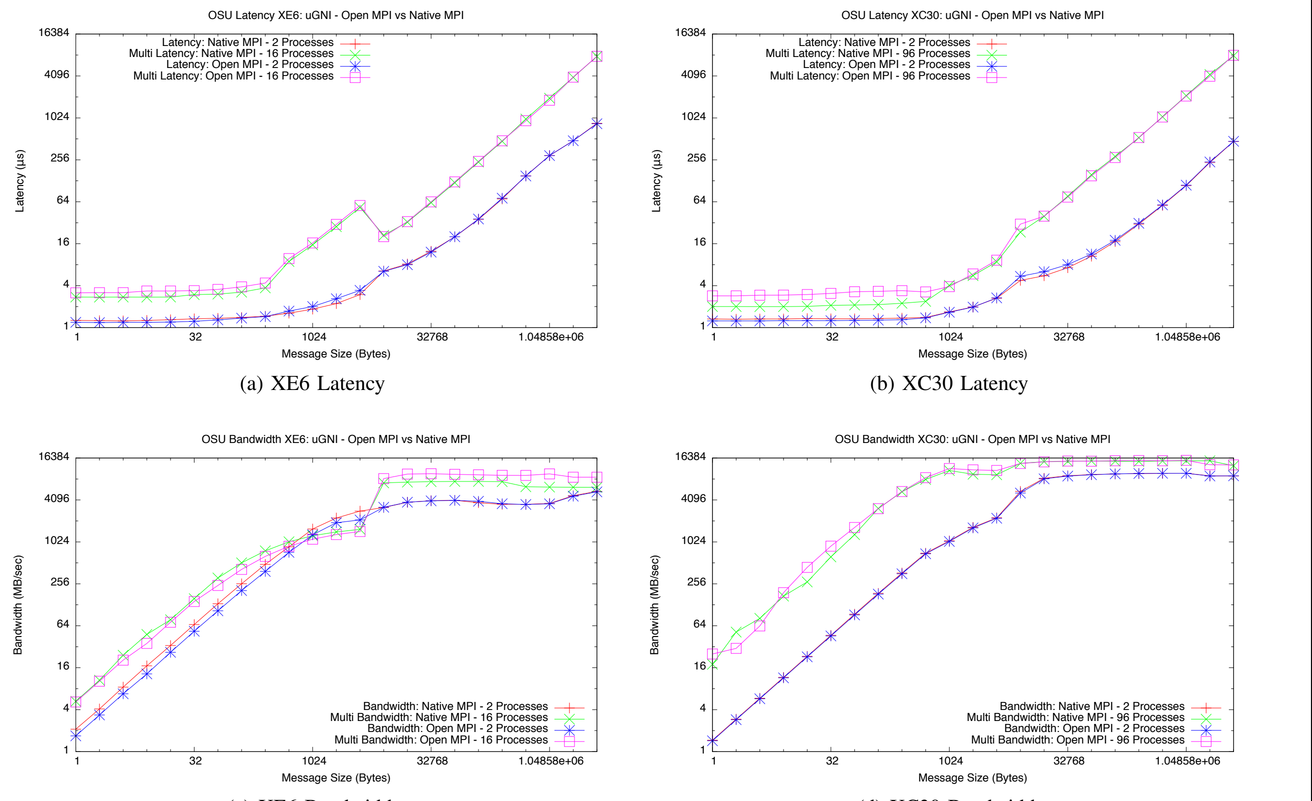
A general overview of the system software used for all tests is as follows. Cielo: CLE 4.1.40, XPMEM 0.1-2.04, *uGNI* 4.0-1.0401, udreg 2.3.2-1.0401, and gcc 4.7.2. Edison: CLE 5.1.29, XPMEM 0.1-2.0501, *uGNI* 5.0-1.0501, udreg 2.3.2-1.0501, and gcc 4.8.2. On both systems we used Open MPI 1.9 pre-release (development trunk) r31308 and Cray MPICH 6.3.0. All binaries were statically built and were run in a single allocation to minimize timing differences due to node placement. The default *uGNI* and *vader* BTL parameters will used for all tests. In addition all tests were run on live systems with active user jobs.

B. Benchmarks

Point-to-point latency: We used the *osu_latency* and *osu_multi_lat* micro-benchmarks from the OSU benchmark suite [13] to evaluate the latency characteristics of both *vader* and *ugni*. *osu_latency* measures message transfer latency by exchanging a ping-pong message between a pair of MPI processes and reports the average, one-way latency of a message transfer. *osu_multi_lat* measures the one-way latency of message transfers between a pair of MPI processes, while multiple pairs of MPI processes are exchanging ping-pong messages.

Point-to-point bandwidth: To evaluate the bandwidth characteristics of both the *vader* and *ugni* BTLs, we used the *osu_bibw* and *osu_mbw_mr* benchmarks from the OSU benchmark suite. *osu_bibw* measures the maximum aggregate bandwidth achieved by a pair of MPI processes. The processes here send a fixed number of messages and wait for the reply. The reported results are an average of multiple iterations of this exchange. *osu_mbw_mr* measures the maximum aggregate bandwidth achieved by a pair of MPI processes while multiple pairs of MPI processes in the network are doing a similar message exchange.

One-sided performance: To evaluate the performance of the MPI-3 one-sided code, we used the *osu_get_latency*, *osu_get_bw*, *osu_put_latency*, and *osu_put_bw* benchmarks from the OSU benchmarking suite. *osu_get_latency* amd *osu_put_latency* measure the one-sided latency of a pair of processes when using the `MPI_Get()` and `MPI_Put()` calls. *osu_get_bw*, and *osu_put_bw* mea-



(a) XE6 Latency

(b) XC30 Latency

(c) XE6 Bandwidth

(d) XC30 Bandwidth

Figure 2: Log-log plots showing uGNI latency and bi-directional bandwidth for Open MPI and the Native MPI on both XE6 and XC30. Results reported by OSU's MPI micro-benchmark suite. Latency measured with *osu_latency* and multi latency measured with *osu_multi_lat*. Bandwidth measured by *osu_bibw* and multi bandwidth measure with *osu_mbw_mr*.

sure the aggregate one-sided bandwidth between a pair of MPI processes. All four tests can be configured to use `MPI_Win_create()` or `MPI_Win_allocate()` to create the shared memory window. The tests can also be configured to use any of the MPI-2 or MPI-3 synchronization functions. For these tests we chose to run with the default mode which uses `MPI_Win_allocate()` and `MPI_Win_flush()`.

C. vader BTL - Point-to-Point Performance Characteristics

Intra-node Performance: Figures 1(a) - 1(d) show the intra-node latency and bandwidth characteristics of both Open MPI (*vader*) and the native MPI (Cray MPT). For this experiment, all MPI processes were configured to be on the same compute node with each process pinned to a single CPU core. For the 2 process tests the two processes were mapped to the same CPU socket.

Figures 1(a) and 1(b) show the latency of Open MPI and the native MPI when processes are participating in a ping-pong message exchange while message sizes are increased. On XE6 with the 2 process configuration, the reported 1 byte message latency for Open MPI is 0.42 μs and the native MPI is 0.53 μs . On XC30 these latencies are 0.23 μs and 0.28 μs respectively. At 1 kB, Open MPI's message latency is 1.46 μs and 0.65 μs which is 2% and 22% worse than the native MPI's latency. At a 4 MB message size, Open MPI's message latency is 1.029 ms and 0.338 ms compared to the native MPI's latency of 0.992 μs and 0.319 ms . Running *osu_multi_lat* with a rank for every core (16 process on XE6, 24 on XC30) Open MPI's message latency is 0.60 μs and 0.46 μs which is 20% and 35% better than the native MPI's 0.75 μs and 0.70 μs . For a 4 MiB message multi latency message exchange Open MPI's latency is 2.535 ms and 2.286 ms , which is similar to the native MPI's performance.

Figures 1(c) and 1(d) show the bandwidth of Open MPI and the native MPI reported by *osu_bibw* and *osu_mbw_mr*. At 2 processes, Open MPI achieves a maximum bidirectional bandwidth of 15 GB/sec and 33 GB/sec at 128 kiB. The native MPI achieves 13 GB/s at 32 kiB and 31 GB/sec at 128 kiB. Running *osu_mbw_mr* with one process per core Open MPI achieves a maximum bandwidth of 62.4 GB/sec at 64 kiB and 156 GB/sec at 64k compared to the native MPI's 62.4 GB/sec at 32 kiB and 184 GB/sec at 128 kiB.

D. uGNI BTL - Point-to-Point Performance Characteristics

Inter-node Performance: Figure 2 shows the inter-node latency and bandwidth characteristics of both Open MPI and the native MPI. For this experiment, each process is configured to talk to a process on another node. In the 2 process test all processes were placed on nodes connected to the same *Gemini* or *Aries* ASIC. The 64 and 96 process tests were run to test the total bandwidth between two pairs of nodes.

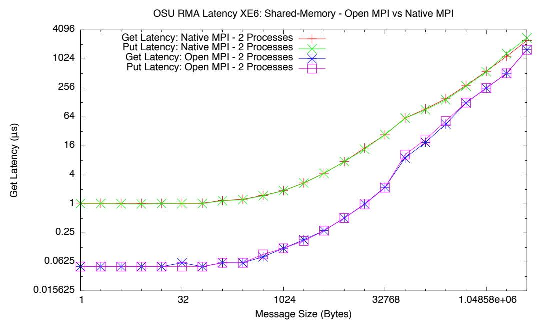
Figures 2(a) and 2(b) show the latency characteristics of Open MPI and the native MPI when processes are participating in a ping-pong message exchange while message sizes are increased. At 2 processes the 1 byte message latency Open Open MPI is 1.19 μs and 1.24 μs compared to the native MPI's 1.27 μs and 1.35 μs . At 1 kiB, the Open MPI message latency is 2.01 μs and 1.66 μs which is 9% worse and 3% better than the native MPI. At 4 MiBs, Open MPI's latency is 843 μs and 472 μs which is 1.5% and 2.3% better than the native MPI. At 64 processes on XE6 the 1 bye, 1 kiB, and 4 MiB message latencies are 3.16 μs , 16.41 μs , and 7.84 ms compared to the native MPI's 2.75 μs , 15.43 μs , and 7.84 ms . With 96 processes on XC30 the latencies are 2.88 μs , 3.88 μs , and 8.11 ms for Open MPI and 2.01 μs , 4.04 μs , and 8.02 ms for the native MPI.

Figures 2(c) and 2(d) show the bandwidth characteristics of processes participating in a message exchange while message sizes are increased. On XE6 with 2 processes, Open MPI and the native MPI achieve a maximum bandwidth of 9.0 GB/s and 8.6 GB/s, respectively, at a 4 MB message size. With 64 processes, Open MPI achieves a maximum bandwidth of 9.5 GB/s at 32 kiB and the native MPI reaches a maximum bandwidth of 7.4 GB/s at 128 kiB. On XC30 the maximum bandwidth for Open MPI with 2 and 96 processes are 14.5 GB/sec and 14.8 GB/sec at 1 MiB. The maximum bandwidth for the native MPI is 14.5 GB/sec and 14.6 GB/sec at 1 MiB respectively.

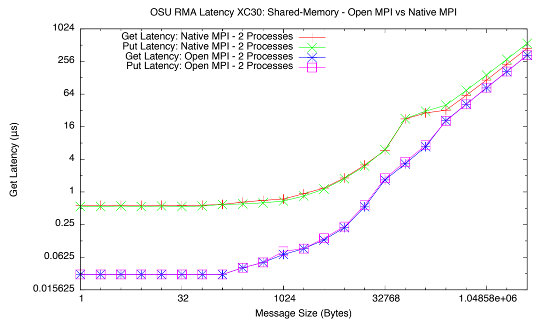
E. One-sided Performance Characteristics

One-sided Performance: Figures 3, 4, 5, and 6 show the latency and bandwidth of both Open MPI and the native MPI when using the `MPI_Get()` and `MPI_Put()` one-sided functions. These tests were run both with shared memory and over the *Gemini* or *Aries* networks. For the shared memory case both processes were bound to the same socket. For the uGNI tests, the two processes were bound to core 0 on socket 0 of nodes sharing the same ASIC. This layout was chosen to keep the processes close to the NIC.

Figures 3 and 4 show the one-sided characteristics for both Open MPI and the native MPI when both processes are on the same node. In this configuration Open MPI can take advantage of the process locality and use shared memory optimizations for all one-sided operations. This gives Open MPI a clear advantage over the native MPI when all processes are on the same node. The performance of Open MPI when this optimization is not possible will be similar to that of the native MPI.



(a) XE6 RMA Latency



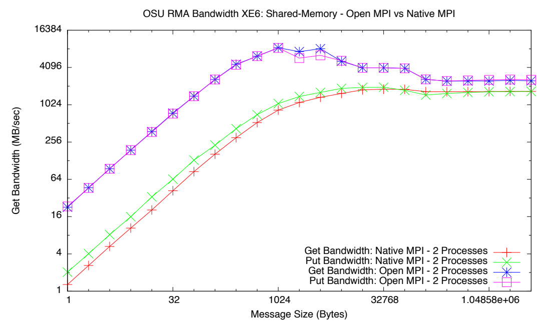
(b) XC30 RMA Latency

Figure 3: Log-log plots showing one-sided shared-memory latency of Open MPI and the Native MPI on both XE6 and XC30. Results reported by OSU's MPI micro-benchmark suite. Latency measured with *osu_get_latency* and *osu_put_latency*.

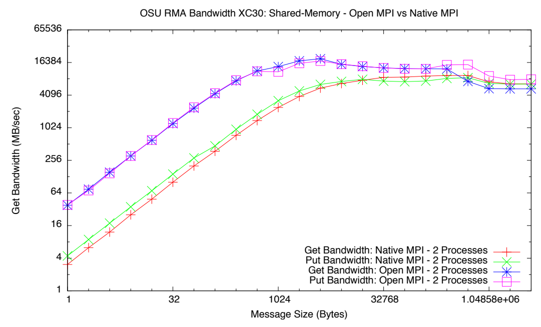
Figures 5 and 6 show the one-sided characteristics for both Open MPI and the native MPI when the processes are on different nodes. The latency of a 1 byte with Open MPI is 3.05 μs on XE6 and 2.86 μs on XC30 compared to the native MPI's 2.51 μs and 2.62 μs is 21% and 9% slower. The latency of a 1 byte get with Open MPI is 4.36 μs and 4.34 μs which is 61% and 65% slower than the native MPI. The differences in the performance characteristics of two implementations are consistent until the message size reaches 8 kib. With message sizes ranging from 8 kib to 4 MiB Open MPI is between 7% and 95% faster than the native MPI on XE6 and has similar performance to the native MPI on XC30.

VIII. Analysis

As reported in [2], the point-to-point performance characteristics of both implementations are very similar. Open MPI

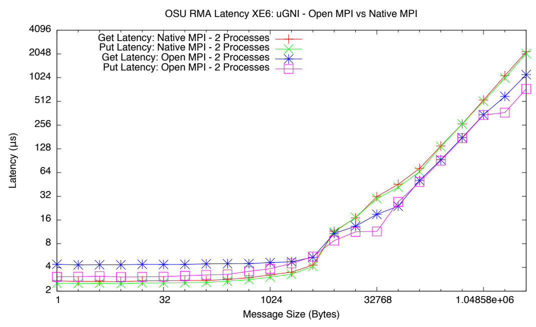


(a) XE6 RMA Bandwidth

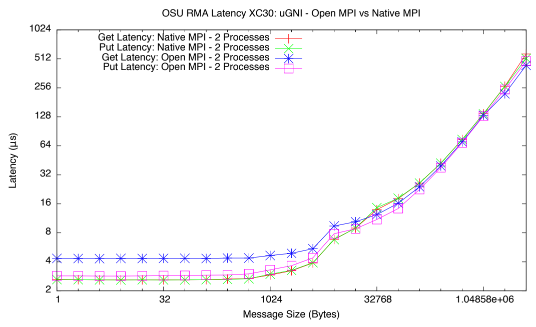


(b) XC30 RMA Bandwidth

Figure 4: Log-log plots showing one-sided shared-memory bandwidth for Open MPI and the Native MPI on both XE6 and XC30. Results reported by OSU's MPI micro-benchmark suite. Bandwidth measured with *osu_get_bw* and *osu_put_bw*.



(a) XE6 RMA Latency



(b) XC30 RMA Latency

Figure 5: Log-log plots showing one-sided uGNI latency of Open MPI and the Native MPI on both XE6 and XC30. Results reported by OSU's MPI micro-benchmark suite.

has improved on the intra-node latency for small messages and is now $18-22\%$ better that the native MPI for 1-32 byte messages on both XE and XK systems. For messages between 64 bytes and 64 kiB the latency of the native MPI is up to 280% better than Open MPI. In this range Open MPI's bandwidth is generally better than the native MPI's. The difference between the implementations in this range could be due to *vader*'s use of of a single copy eager send protocol over a rendezvous protocol or buffered eager protocol. The protocol used for this message range can be changed by reducing the *btl_vader_eager_limit* from the default of 64kiB. After 64 kiB both implementations have similar characteristics.

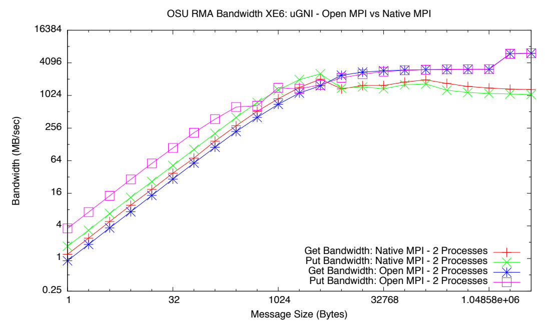
The inter-node communication latency of Open MPI with 1 byte messages is 6.7% better than the native MPI on XE6 and 8.9% better on XC30. For 1 kiB messages Open MPI's latency is worse that the native MPI's latency by 9% on XE6 but 3% better on XC30. For 4 MiB messages the latency of Open MPI is 1.5-2.3% better than the native MPI. Overall the performance of point-to-point inter-node communication with Open MPI is similar to the native MPI.

Open MPI's on-node one-sided performance is generally better than the native MPI implementation for both `MPI_Put()` and `MPI_Get()`. For small inter-node messages, Open MPI is 61-65% slower than the native MPI for `MPI_Get()` and 9-21% slower for `MPI_Put()`. The small-message latency in Open MPI is worse due to the current one-sided support which emulates small inter-node one-sided operations using buffered two-sided operations. The latency of large-message Gets are similar in both implementations, but lower latency is seen for Puts and higher bandwidth is seen for both Put and Get.

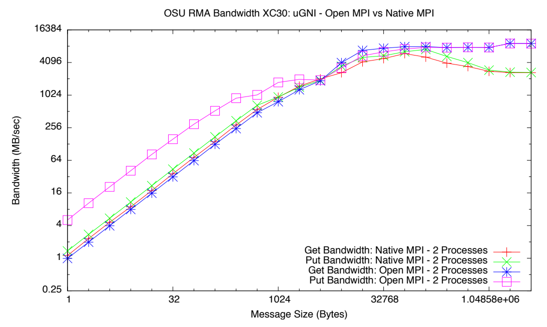
IX. Conclusion and Future Work

The micro-benchmark results demonstrate that Open MPI's implementation of its point-to-point communications for the *Gemini* and *Aries* network interfaces exhibit good performance characteristics. Point-to-point intra-node and inter-node latency characteristics are similar to the vendor's implementation. The intra-node bandwidth and latency, however, are better than the vendor MPI at many message size. The intra-node one-sided characteristics are better than the vendor MPI implementation for all message sizes.

In the future, we plan to investigate improvements in the scalability of both the MPI implementation and the ORTE process launcher. Improvements will be targeted at



(a) XE6 RMA Bandwidth



(b) XC30 RMA Bandwidth

Figure 6: Log-log plots showing one-sided uGNI bandwidth of Open MPI and the Native MPI on both XE6 and XC30. Results reported by OSU's MPI micro-benchmark suite.

both launch time and memory usage. Additionally, we plan to investigate improving the implementation of the one-sided code to directly use the atomic and RDMA operations supported by the *Gemini* and *Aries* hardware.

Acknowledgment

The authors would like to thank Alliance for Computing at Extreme Scale (ACES) management and staff for their support. Work supported by the Advanced Simulation and Computing program of the U.S. Department of Energy's NNSA. Los Alamos National Laboratory is operated by Los Alamos National Security, LLC for the NNSA. The authors would also like to thank the Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Additionally, the authors would like the thank National Energy Research Scientific Computing Center (NERSC) for use of their Edison system. LA-UR-14-22496

References

[1] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[2] M. G. Venkata, R. L. Graham, N. T. Hjelm, and S. K. Gutierrez, "Open mpi for cray xe/xk systems," 2012.

[3] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, Aug. 2010, pp. 83 –87.

[4] Cray Inc., "Cray xc series network," in *Cray Marketing Document*, vol. WP-Aries01-1112, 2012. [Online]. Available: http://www.cray.com/Assets/PDF/products/xc/CrayXC30Networking.pdf

[5] ——, "Using the gni and dmapp apis," in *Cray Software Document*, vol. S-2446-4002, Dec. 2011. [Online]. Available: http://docs.cray.com/books/S-2446-4002/S-2446-4002.pdf

[6] H. Pritchard, I. Gorodetsky, and D. Buntinas, "A ugni-based mpich2 nemesis network module for the cray xe," in *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 110–119. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042476.2042490

[7] J. M. Squyres and A. Lumsdaine, "The component architecture of open MPI: Enabling third-party collective algorithms," in *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, Eds. St. Malo, France: Springer, July 2004, pp. 167–185.

[8] (2011) XPMEM, cross-process memory mapping. [Online]. Available: http://code.google.com/p/xpmem/

[9] S. Gutierrez, N. Hjelm, M. Venkata, and R. Graham, "Performance evaluation of open mpi on cray xe/xk systems," in *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*, Aug 2012, pp. 40–47.

[10] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem," in *International Symposium on Cluster Computing and the Grid*, 2006, pp. 530–540.

[11] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges, "Infiniband scalability in open mpi," in *Proceedings of IEEE Parallel and Distributed Processing Symposium*, April 2006.

[12] "High performance RDMA protocols in HPC," in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science. Bonn, Germany: Springer-Verlag, September 2006.

[13] OSU micro-benchmarks. [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/