Open MPI for Cray XE/XK Systems

Manjunath Gorentla Venkata, Richard L. Graham
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN
{manjugv, rlgraham}@ornl.gov

Nathan T. Hjelm, Samuel K. Gutierrez
High Performance Computing Division, HPC-3
Los Alamos National Laboratory
Los Alamos, NM
{hjelmn, samuel}@lanl.gov

***Abstract*—Open MPI provides an implementation of the MPI standard supporting communication over a range of high-performance network interfaces. Recently, Oak Ridge National Laboratory (ORNL) and Los Alamos National Laboratory (LANL) collaborated on creating a port of Open MPI for Gemini, the network interface for Cray XE and XK systems. In this paper, we present our design and implementation of Open MPI's point-to-point and collective operations for Gemini, and techniques we employ to provide good scaling and performance characteristics. The point-to-point operations are implemented within a new byte transfer layer (BTL) component that provides protocols that are optimized for small, medium, and long messages. An XPMEM BTL was also implemented for use in all intra-node communication. The collective operations are implemented within a new basic collectives (BCOL) module within Cheetah, a framework in Open MPI for implementing hierarchical collectives.**

The micro-benchmark results show that Open MPI's point-to-point performance characteristics are similar to that of the native MPI's. The collective operation evaluation results show that the atomic operation-based Barrier outperforms the message-based Barrier by 39%, thus demonstrating the potential of using atomic operations for implementing further collective operations.

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

I. Introduction

Open MPI is an open-source MPI implementation of the MPI-2 specification that is developed and maintained by collaborators from academia, industry, and national laboratories [1]. Open MPI provides thread safety and concurrency as well as network and process fault tolerance. It also supports network heterogeneity and various high-performance network interfaces, including InfiniBand, Cray SeaStar, and Myrinet. Current versions of Open MPI, however, lack support for the *Gemini System Interconnect* [2], which is a newly introduced network interface for the Cray XE and XK system families.

This paper presents extensions for Open MPI to support *Cielo*, a 142,304 core XE6 capability-class platform for the Advanced Simulation and Computing (ASC) Program, and *Titan*, a 299,008 core XK6 that is slated to be Oak Ridge Leadership Computing Facility's next flagship system. Two new BTLs were created for the XE/XK port: *ugni*, which leverages the uGNI interface, and *vader*, which leverages XPMEM [3] - a Linux kernel module that enables a process to map the memory of another process into its virtual address space. Currently, *vader* is used exclusively for intra-node message transfers and *ugni* is used exclusively for inter-node message transfers. An extension was added to the Cheetah framework, a hierarchical collectives framework in Open MPI, to support collectives for the Gemini network interface [4]. Only minor modifications were needed in the Open Run-Time Environment (ORTE) to complete this port.

The rest of the paper is organized as follows. Section II provides a brief description of the Gemini Network Interface and Open MPI. Sections III and IV provide a detailed description of the enhanced shared-memory BTL and the uGNI BTL, respectively. Section V provides a description of the Cheetah framework and the extensions to support collectives for the Gemini network. Section VI outlines the performance evaluation and presents some micro-benchmark performance results. Section VII provides an analysis of the performance characterists of this port. Section VIII 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 [2]. Two Opteron nodes are connected to a Gemini that provides 10 torus connections - 8 divvied evenly between *X* and *Z* and 2 in *Y*, as shown in Figure 1. Link bandwidths are 4.68 to 9.375 GB/s per direction [2].

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 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]. *FMA DM* (Distributed Memory) is used to execute PUT, GET, and atomic memory operations (AMOs).

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 [2].

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). 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 is comprised of 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 and XK 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, uses FMA PUT, which does not require memory registration, and BTE PUT, which does not have a 4-byte alignment restriction for data buffers, for message sizes greater than the SMSG message limit. Futhermore, Open MPI uses AMOs for implementing Barrier collective operations. At this time, we are not aware of MPICH2 using AMOs for any of its collective operations. Open MPI and MPICH2 both use shared-memory-based eager protocols for small, intra-node message transfers and a rendezvous protocol based on SGI's XPMEM for large message transfers [6]. In addition, unlike MPICH2's uGNI network module, the uGNI BTL is an open-source implementation that leverages the Gemini Network Interface.

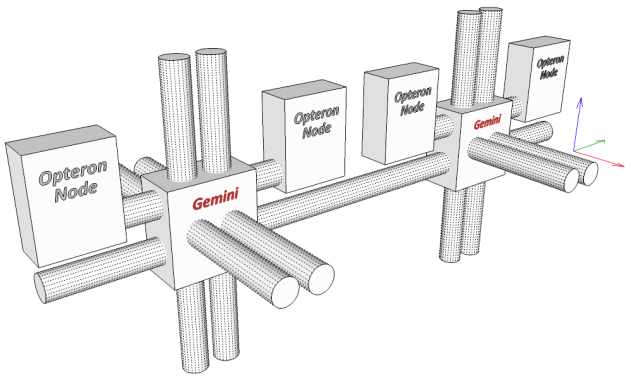

Fig. 1. High-level cartoon showing 4 Opteron nodes connected by 2 Gemini ASICs. The X,Y, and Z axes are depicted as Red, Green, and Blue arrows, respectively.

III. Enhanced Shared-Memory BTL

In this section we will outline the design and implementation of *vader*, the enhanced shared-memory BTL that was implemented for this port. We start with an overview of XPMEM, and conclude with a high-level discussion surrounding 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 [3]. XPMEM exposes a small application programming interface (API) that is comprised of 7 routines: *xpmem_version*, *xpmem_make*, *xpmem_remove*, *xpmem_get*, *xpmem_release*, *xpmem_attach*, and *xpmem_detach*. XPMEM setup is a relatively simple, 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. XPMEM BTL – Vader

vader is implemented as a new BTL component within Open MPI. *vader*'s design and implementation is heavily influenced by the single-copy, RDMA-like capabilities provided by XPMEM. The need for a higher bandwidth, lower latency BTL for intra-node communication on XE/XK systems was the impetus behind the implementation of this kernel-assisted shared-memory BTL.

vader implements both SEND and RDMA transfer protocols. The SEND protocol is patterned after the Nemesis protocol [7] used by MPICH. 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.

Due to the costly nature of *xpmem_attach* and *xpmem_make*, special design considerations were made in order to reduce the amount of times these routines were invoked. During *vader*'s component initialization, all participating processes export their entire address space via exactly one call to *xpmem_make*. Each process then makes a call to *xpmem_get* to obtain an *access permit ID* (apid) for each local peer. *vader* makes calls to *xpmem_attach* to access memory regions in peer processes, as needed. To reduce the overhead associated with the *xpmem_attach* call, *vader* attaches to a minimum of 2 MB at a time and caches the attached regions in a registration cache for reuse in subsequent transfers.

IV. uGNI BTL

The uGNI BTL provides point-to-point 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. 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. As is the case with the MPICH2 implementation [6], this parameter is set to $autoselect$ (0) by default, which sets to the SMSG limit based on the number of MPI tasks. A table of the uGNI BTL's current SMSG limits can be found in Table I.

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

TABLE I

When using the short message protocol, *BTL Send()* transfers the message header/data using *GNI_SmsgSendWTag()* with a tag of MCA_BTL_UGNI_TAG_SEND. SMSG handles the delivery of the message to the remote mailbox and, on successful completion, notifies the uGNI BTL through the endpoint's local completion queue.

B. Initialization and Connection Setup

The default behavior of the uGNI BTL is to bind uGNI endpoints and allocate SMSG mailbox resources on an on-demand basis. We chose this approach so that the memory overhead 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 MB blocks up to the maximum size needed.

C. Eager Get Protocol

The eager get protocol handles calls to *BTL Send()* with messages larger than the SMSG limit, but smaller than the eager limit specified by the $btl\_ugni\_eager\_limit$ MCA parameter. This protocol requires that the message is first copied into a registered buffer. By default, the uGNI BTL allocates a pool of 16 send/receive buffers in increments of 16 up to a maximum of 64. The default behavior can be changed by setting the $btl\_ugni\_eager\_num$, $btl\_ugni\_eager\_inc$, and $btl\_ugni\_eager\_max$ parameters at invocation.

When using the eager get protocol, the sending process sends all the information necessary to complete an RDMA transaction using *GNI_SmsgSendWTag()* with a tag of MCA_BTL_UGNI_INIT_GET. The receiving process allocates a registered buffer from its receive pool and starts either an FMA or BTE GET transaction. On completion, the receiving process notifies the sender by calling *GNI_SmsgSendWTag()* with a tag of MCA_BTL_UGNI_TAG_RDMA_COMPLETE.

D. Long Message Protocol

Long message support is provided through the *BTL Put() and Get()* functions. These functions mostly correspond to *GNI_PostRdma()* with post type GNI_POST_RDMA_PUT or GNI_POST_RDMA_GET, respectively (or their FMA equivalents). The only exception is the case where a BTE/FMA GET operation can not be completed due to size or alignment restrictions.

When a BTE/FMA GET operation can not be completed, we fallback on BTE/FMA PUT. In this case, the receiving process signals the sending process to switch to RDMA PUT by calling *GNI_SmsgSendWTag()* with a tag of MCA_BTL_UGNI_TAG_PUT_INIT. The sending process follows the normal RDMA PUT path. On completion, the sender notifies the receiver by calling *GNI_SmsgSendWTag()* with a tag of MCA_BTL_UGNI_TAG_RDMA_COMPLETE. The current implementation of the PUT fallback path requires the overhead of two extra SMSG messages.

E. Memory Registration

To reduce the overhead associated with memory registration, the uGNI BTL makes use of an RDMA registration pool provided within Open MPI. This registration cache stores unused registrations in a least recently used (LRU) list. Cached registrations can either be reused for future transactions or released when resources are exhausted. 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. The limit is based on the number of active MPI processes on a compute node.

V. Collective Operations for Gemini

A. Cheetah

Cheetah is a framework for building hierarchical collectives and is implemented as a framework within Open MPI. In Cheetah, the hierarchical collective operations are expressed

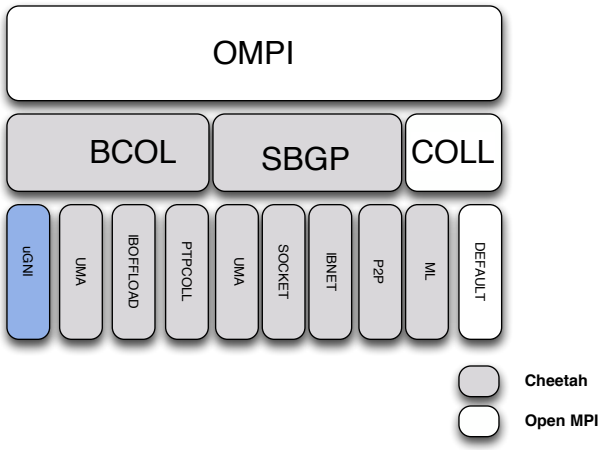

Fig. 2. Cheetah frameworks and components within Open MPI.

as a group of independently progressing collective primitives, where each collective primitive is optimized for a specific communication hierarchy.

The design of the framework is driven by the goal to provide collective operation implementations tailored to specific communication hardware, while also retaining the re-usability of the collective implementation. In Cheetah, this is achieved by decoupling the collective operation implementation from the topological organization of the processes. A brief description of the framework is provided in this section and a more detailed description of the framework and its design principles can be found in [4].

Figure 2 shows Cheetah's frameworks and its components. The Cheetah framework is implemented within Open MPI and is comprised of two component frameworks and a single COLL component. In particular, the Basic Collectives (BCOL) and subgrouping (SBGP) frameworks, and the messaging layer (ML) component. BCOL is a component framework for implementing collective primitives that are specialized for communication hierarchies. The current implementation supports BASESMUMA for shared memory, IBOFFLOAD for Mellanox's CORE-*Direct*, and P2P for other networks that are supported by Open MPI. SBGP is a framework for implementing subgroups, i.e., partitioning processes into subgroups based on the communication hierarchy shared between them. Currently, the Cheetah framework supports subgrouping over CPU sockets (SOCKET subgroup), shared memory (UMA subgroup), Mellanox's InfiniBand CORE-*Direct* networks (IBNET), and other networks (P2P subgroup). ML components combine collective primitives to implement an MPI collective operation. For example, an *MPI_Broadcast()* operation can be implemented by combining a shared-memory broadcast primitive (BASESMUMA BCOL and UMA subgroup) and a network broadcast primitive (PTPCOLL BCOL and P2P subgroup).

uGNI BCOL: To take advantage of Gemini's atomic operations for collectives, we implemented uGNI BCOL and took advantage of the P2P subgroup in Cheetah. The P2P subgroup groups all processes sharing the Gemini communication domain into a single P2P subgroup. The collective primitives in the uGNI BCOL are defined over this subgroup. Currently, this BCOL supports only the Barrier collective primitive.

B. Barrier Collective Operation

MPI_Barrier() is a collective operation that synchronizes all processes in a given communicator. The uGNI Cheetah Barrier, which implements *MPI_Barrier()*, is implemented using the uGNI BCOL collective primitive over the P2P subgroup. The uGNI BCOL Barrier collective primitive, called atomic Barrier hereafter, uses a Fan-in/Fan-out algorithm and leverages the atomic operations provided by the uGNI library. For both the fan-in and fan-out phase, we use an *n-ary* tree, where the radix of tree can be varied. The processes participating in the atomic Barrier are either designated as a root process, interior process, or a leaf process.

In the fan-in phase, the leaf processes update a counter on their interior, or parent process, using an atomic add operation. An atomic operation is invoked on the Gemini network by posting a descriptor using the *GNI_PostFMA()* primitive. The interior processes, after receiving updates from all of their respective children, update their respective parent processes. The root process, after receiving updates from its children, then switches to the fan-out phase. In the fan-out phase of the algorithm, the root process updates it children's counters and exits, thus completing the barrier. The interior processes, after receiving the update from their respective parents, update their children's counters, and exit the Barrier. The leaf processes, after receiving the update from their respective parents, exit the Barrier.

VI. Evaluation

This section describes the test beds used for the evaluation of our work. It then presents some point-to-point performance results for the *vader* and *ugni* BTLs and preliminary Barrier collective performance of the uGNI BCOL.

A. System Description

To evaluate the performance of the uGNI BTL and the uGNI BCOL, we used Cielo and Enhanced Jaguar (Jaguar with Interlagos processors and the Gemini Network Interface).

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 GB memory. It uses the Gemini network interface for network communication.

Enhanced Jaguar is a Cray XK6 located in the National Center for Computational Sciences (NCCS) at ORNL. It has 18,688 compute nodes, each containing one 2.2 GHz AMD Opteron Interlagos processor along with 32 GB of memory. Each AMD Opteron processor has 16 compute cores and 3 levels of cache memory. Out of the 18,688 compute nodes, 960 nodes also have a single NVIDIA graphical processing unit

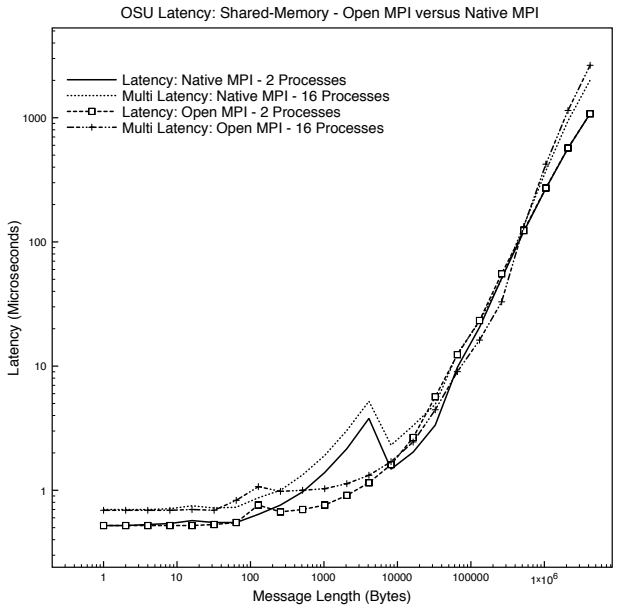

Fig. 3. Log-log plot showing shared-memory latencies for 2 and 16 processes reported by OSU's MPI micro-benchmark suite. Latency measured with *osu_latency* and multi latency measured with *osu_multi_lat*.

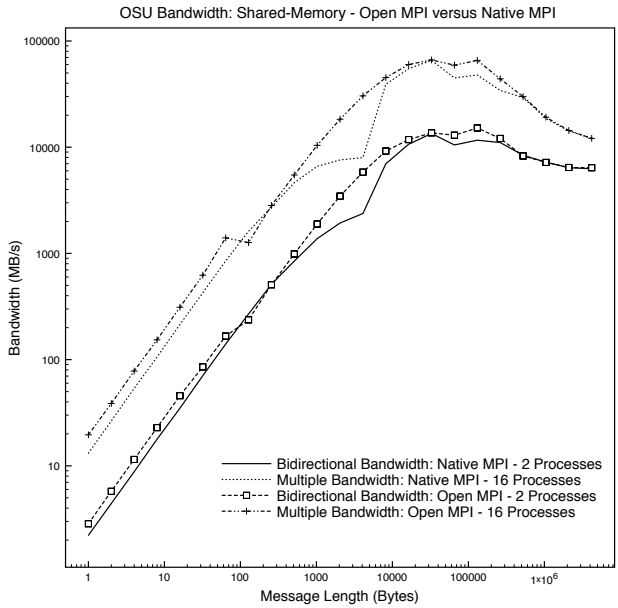

Fig. 4. Log-log plot showing shared-memory bandwidths for 2 and 16 processes reported by OSU's MPI micro-benchmark suite. Bidirectional bandwidth measured with *osu_bibw* and multiple bandwidth measured with *osu_mbw_mr*.

(GPU). Like Cielo, this system also uses the Gemini network interface for network communication.

B. Benchmarks

Point-to-point latency: We used the $osu\_latency$ and osu_multi_lat micro-benchmarks from the OSU benchmark suite [8] 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 *vader* and *ugni*, 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.

Barrier latency: To evaluate the performance of the Barrier operation, we ran *MPI_Barrier()* in a tight loop and measured its execution time. The performance reported is an average latency.

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

Intra-node Performance: Figure 3 and 4 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.

Figure 3 shows the latency of Open MPI and the native MPI when 2 and 16 processes are participating in a ping-pong message exchange while message sizes are increased. In the 2 process configuration, the reported 1 byte message latency for both MPI implementations is 0.52 $usecs$. At 1 kB, Open MPI's message latency is 0.76 $usecs$, which is 82% better than the native MPI's latency. At a 4 MB message size, Open MPI's message latency is 1.076 $msecs$ compared to the native MPI's latency of 1.079 $msecs$. At 16 processes, the 1 byte message latency of Open MPI is 0.69 $usecs$ compared to native MPI's 0.7 $usecs$. At 1 kB, Open MPI's message latency is 1.03 $usecs$ compared to native MPI's 1.91 $usecs$. For a 4 MB message exchange, Open MPI's message latency is 2.6 $msecs$, which is 31% worse than native MPI's performance.

Figure 4 shows the bandwidth of Open MPI and the native MPI when 2 and 16 processes are participating in a message exchange. At 2 processes, Open MPI achieves a maximum bidirectional bandwidth of 15 GB/s at 100 kB and the native MPI achieves a maximum bidirectional bandwidth of 13 GB/sec at 32 kB. At 16 processes, Open MPI and the native MPI achieve a maximum bidirectional bandwidth of 66.3 GB/s and 65.7 GB/s, respectively, for 32 kB messages.

Inter-node Performance: Figure 5 and 6 show the inter-node latency and bandwidth characteristics of both Open MPI and the native MPI. For this experiment, each MPI process is configured to be on a different node.

Figure 5 shows the latency characteristics of Open MPI

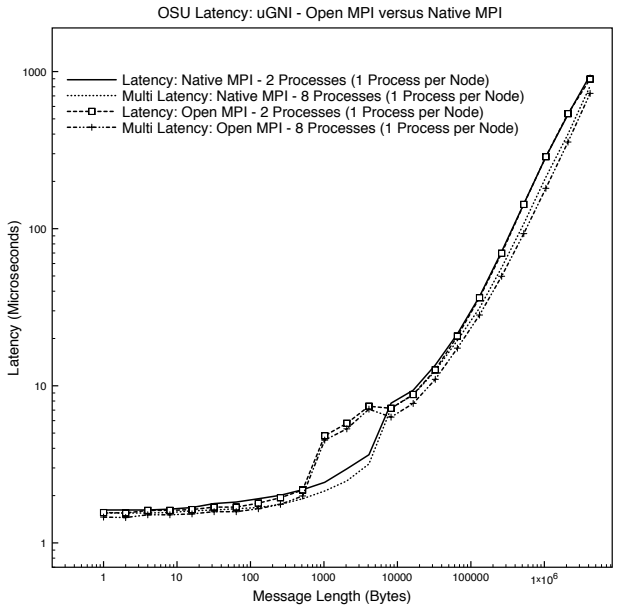

Fig. 5. Log-log plot showing uGNI latencies for 2 and 8 processes reported by OSU's MPI micro-benchmark suite. Latency measured with *osu_latency* and multi latency measured with *osu_multi_lat*.

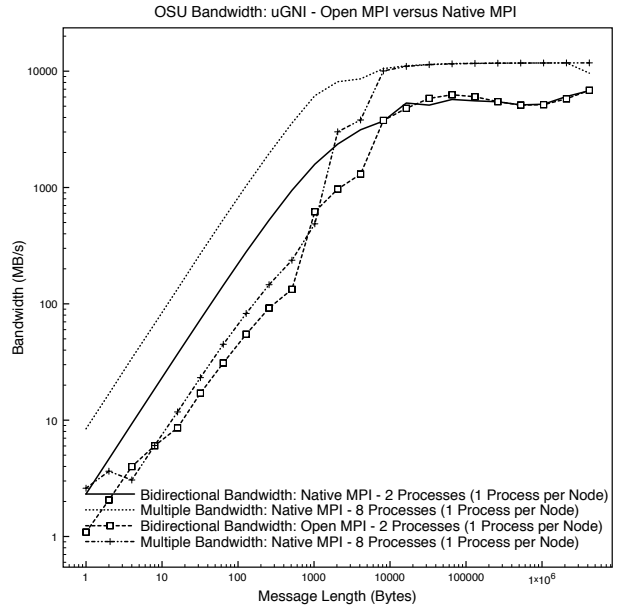

Fig. 6. Log-log plot showing uGNI bandwidths for 2 and 8 processes reported by OSU's MPI micro-benchmark suite. Bidirectional bandwidth measured with *osu_bibw* and multiple bandwidth measured with *osu_mbw_mr*.

and the native MPI when 2 and 8 processes are participating in the ping-pong message exchange while message sizes are increased. At 2 processes, the 1 byte message latency of Open MPI is 1.56 $usecs$ compared to the native MPI's 1.52 $usecs$. At the 1 kB message size, Open MPI message latency is 4.81 $usecs$ which is 47% worse than the native MPI's latency. At 4 MBs, Open MPI's message latency is 896 $usecs$, which is 5%

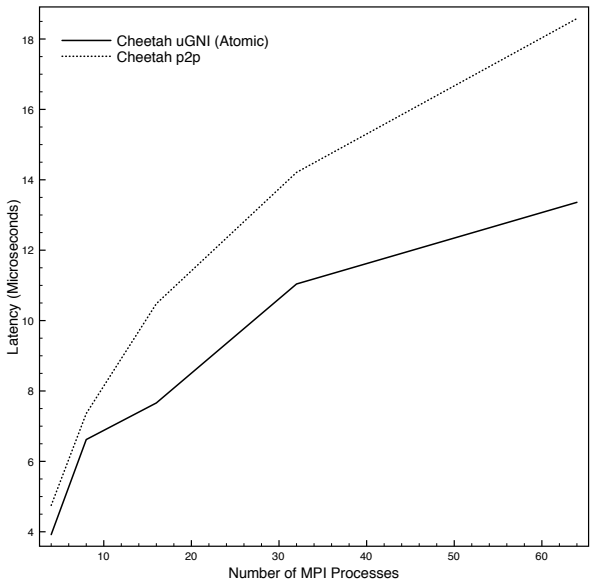

Fig. 7. Graph showing the performance of the Cheetah uGNI Barrier compared to the Cheetah p2p Barrier as a function of problem size.

better than the native MPI's message latency. At 8 processes, 1 byte, 1kB, and 4 MB message latencies of Open MPI are 1.46 $usecs$, 4.51 $usecs$, and 726.37 $usecs$, respectively. The 1 byte and 4 MB message latencies of the native MPI are 6% and 11 % worse than Open MPI's message latency. The 1 kB message latency of the native MPI is 47% better than Open MPI's.

Figure 6 shows the bandwidth characteristics of both MPI implementations when 2 and 8 processes are participating in a message exchange while message sizes are increased. With 2 processes, Open MPI and the native MPI achieve a maximum bandwidth of 6.8 GB/s and 6.7 GB/s, respectively, at a 4 MB message size. At 8 processes, Open MPI achieves a maximum bandwidth of 11.7 GB/s at a 4 MB message size and the native MPI reaches a maximum bandwidth of 11.7 GB/s at a 2 MB message size and drops to 9.6 GB/s at 4 MBs.

D. uGNI BCOL - Barrier Performance Characteristics

Figure 7 shows the performance of Barrier as the number of MPI processes are increased. For this experiment, each MPI process was configured to be on a different node. The figure shows the performance of two Barrier implementations. The first is Cheetah uGNI's Barrier performance curve using a Fan-in/Fan-out algorithm with a radix of 4 implemented using atomic operations. The second is Cheetah p2p's Barrier performance curve using the uGNI BTL.

At 64 processes, the latency of Cheetah uGNI's (atomic) Barrier is 13.36 $usecs$, which is better than the Cheetah p2p Barrier performance by 39%.

VII. Analysis

The point-to-point performance characteristics of both implementations are very similar. The intra-node latency of Open

MPI and the native MPI for 1 byte messages is similar, but for 1 kB messages, Open MPI's latency is better than native MPI's latency by 82%. At a 4 MB message size, however, the native MPI's latency is better than Open MPI's by 31%. This latency trend also holds for the multi-process message exchange. At 2 processes, Open MPI's intra-node bandwidth is 15 % better than the native MPI's bandwidth.

The 1 byte, inter-node latency of both implementations is similar, but for 1 kB messages, Open MPI's latency is worse than native MPI's latency by 47%. Furthermore, Open MPI's 4 MB latency is better than the native MPI's latency by 5%. This latency trend also holds for multi-process message exchanges.

The uGNI BCOL's Barrier performance clearly shows the advantage of using atomic operations for synchronization and small data collective operations. It outperforms Cheetah's p2p Barrier by 39%, which uses the SMSG protocol for exchanging synchronization information.

VIII. Conclusion and Future Work

The micro-benchmark results demonstrate that Open MPI's implementation of its point-to-point communications for the Gemini network interface has good performance characteristics. Point-to-point intra-node and inter-node latency characteristics are similar in both implementations. The intra-node bandwidth characteristics, however, are better than the native MPI's bandwidth characteristics.

Also, the Barrier performance results demonstrate the potential advantages of using atomic operations for implementing some collective operations. In the future, we plan to use the MSGQ protocol for point-to-point communication and evaluate its effect on performance and scalability. We also plan to provide better support for switching from GET to PUT, which will eliminate the need for an extra message - as required in the current implementation. We plan to evaluate the performance and scalability characteristics of the atomic collective operations at higher scale, and also its potential in implementing other collective operations such as small-data *MPI_Reduce()* and *MPI_Allreduce()*.

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. In addition, the authors would also like to thank the Office of Advanced Scientific Computing Research's FASTOS program and the Math/CS Institute EASI!; U.S. Department of Energy, and partial work was performed at ORNL, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. This research used resources of 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. LA-UR-12-20472.

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] 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.

[3] (2011) XPMEM, cross-process memory mapping. [Online]. Available: http://code.google.com/p/xpmem/

[4] R. Graham, M. G. Venkata, J. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer, "Cheetah: A framework for scalable hierarchical collective operations," *CCGRID 2011*, 2011.

[5] Cray Inc., "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] 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.

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