2012 IEEE 20th Annual Symposium on High-Performance Interconnects

Performance Evaluation of Open MPI on Cray XE/XK Systems

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

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

Open MPI is a widely used open-source implementation of the MPI-2 standard that supports a variety of platforms and interconnects. Current versions of Open MPI, however, lack support for the Cray XE6 and XK6 architectures – both of which use the Gemini System Interconnect. In this paper, we present extensions to natively support these architectures within Open MPI; describe and propose solutions for performance and scalability bottlenecks; and provide an extensive evaluation of our implementation, which is the first completely open-source MPI implementation for the Cray XE/XK system families used at 49,152 processes.

Application and micro-benchmark results show that the performance and scaling characteristics of our implementation are similar to the vendor-supplied MPI's. Micro-benchmark results show short-data 1-byte and 1,024-byte message latencies of 1.20 μs and 4.13 μs, which are 10.00% and 39.71% better than the vendor-supplied MPI's, respectively. Our implementation achieves a bandwidth of 5.32 GB/s at 8 MB, which is similar to the vendor-supplied MPI's bandwidth at the same message size. Two Sequoia benchmark applications, LAMMPS and AMG2006, were also chosen to evaluate our implementation at scales up to 49,152 cores – where we exhibited similar performance and scaling characteristics when compared to the vendor-supplied MPI implementation. LAMMPS achieved a parallel efficiency of 88.20% at 49,152 cores using Open MPI, which is on par with the vendor-supplied MPI's achieved parallel efficiency.

***Keywords*-Open MPI, Cray, Gemini, uGNI, XPMEM**

I. Introduction

The Cray XE6 and XK6 architectures are an important new class of systems for running demanding scientific simulations, as they have demonstrated the ability to achieve petascale performance [1]. A critical component in realizing this level of performance within a parallel scientific simulation is the system's network infrastructure.

The Cray XE6 and XK6 architectures include a new network infrastructure called the Gemini System Interconnect [2]. Cray provides two low-level interfaces for implementing communication libraries targeted for the Gemini: User Generic Network Interface (uGNI) and DMAPP. In particular, the User Generic Network Interface interface is geared towards message passing libraries and DMAPP towards global address space (GAS) libraries.

Most scientific applications running on these system architectures leverage a Message Passing Interface (MPI) library for communication and synchronization, and can spend a significant amount of simulation time in MPI routines. For example, the 32,768-core LAMMPS benchmark run presented in this paper spends approximately 20% of the its simulation time in MPI routines. Similarly, our AMG2006 benchmark problem spends approximately 64% of its simulation time in MPI routines. Despite the importance of MPI for these systems and applications, there are currently no completely open-source implementations of MPI that natively support the Cray XE6 and XK6.

In this work, we implement an open-source MPI implementation for Cray XE6 and XK6 systems by extending Open MPI [3], a production grade and widely used open-source implementation of MPI. Adding support for new network interfaces in Open MPI requires implementing a network-specific message transfer component (i.e., either a Byte Transfer Layer (BTL) or a Matching Transport Layer (MTL)); while other infrastructure provided within Open MPI, such as the runtime system, message multiplexing and demultiplexing layer, and the layer that provides MPI semantics, can be directly leveraged with minimal to no modifications.

To support the Cray XE/XK architecture families, we introduce two new BTLs, *vader* and *ugni*. *vader* provides protocols for intra-node communication and *ugni* provides protocols for inter-node communication. Besides implementing commonly known eager and rendezvous protocols for message transfers, we introduce new protocols and mechanisms to obtain desirable scalability and performance characteristics. Among them, a newly introduced Eager GET protocol, which provides better memory utilization characteristics than using Gemini's small message protocol for small data messages; and a Put Fallback protocol, which acts as an alternate protocol when the constraints of Gemini's GET protocol cannot be satisfied.

The rest of the paper is organized as follows. Section II provides background and Section III discusses related work. Section IV discusses the *vader* component and its protocols for intra-node communication. Secion V discusses the *ugni* component, its protocols for inter-node communication, scala-

978-0-7695-4831-9 2012

DOI 10.1109/HOTI.2012.11

bility bottlenecks, and optimizations. Section VI describes the experimental setup, evaluates the *vader* and *ugni* components, and presents micro-benchmark and application performance results. Section VII provides an analysis of the results and Section VIII concludes and discusses future work.

II. Background

Open MPI's Point-to-Point Architecture: The point-to-point infrastructure that we chose to leverage in this work consists of three major layers: the Point-to-Point Management Layer (PML), the BTL Management Layer (BML), and the Byte Transfer Layer (BTL).

The Point-to-Point Management Layer (PML) is responsible for implementing MPI-like semantics, including message buffering, message matching, and scheduling of message transfers [4]. The PML is also responsible for implementing the eager and rendezvous protocols, which leverage the underlying messaging services (e.g., send/receive and Remote Direct Memory Access (RDMA)) provided by either the MTL or BTLs. An eager protocol is typically used for short data transfers, and a rendezvous protocol is used for long data transfers. The BML layer is responsible for multiplexing MPI messages, and the BTL layer is responsible for transferring data between communication endpoints. More information about Open MPI's point-to-point architecture can be found in [4].

Gemini System Interconnect: The Gemini System Interconnect is Cray's network for current petascale system architectures such as *Cielo*, a 143,104-core Cray XE6 housed at Los Alamos National Laboratory, and *Titan*, Oak Ridge Leadership Computing Facility's (OLCF) 299,008 core XK6, and is the successor to the Cray SeaStar* network interconnect found in XT systems. Each Gemini provides 10 torus connections - 8 evenly divided up 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].

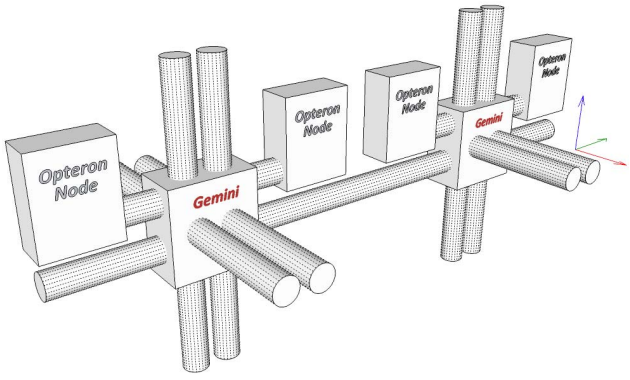

Fig. 1. High-level diagram showing 4 Opteron nodes connected by 2 Gemini ASICs. The X,Y, and Z axes are depicted as Red, Green, and Blue arrows, respectively.

Message passing libraries can effectively leverage Gemini hardware though a user-level interface provided by the User Generic Network Interface (uGNI) [2][5]. Furthermore, we anticipate Cray's next generation interconnect will expose a similar user-level application programming interface (API), which could potentially reduce porting efforts focused on supporting Cray's next generation network architecture. More information about the Gemini System Interconnect and the uGNI can be found in [2] and [5].

User Generic Network Interface: The GNI exposes low-level user-space communication services through uGNI. In particular, two mechanisms are provided for initiating RDMA transactions: Fast Memory Access (FMA) and Block Transfer Engine (BTE).

FMA: FMA provides in-order RDMA. FMA transactions come in a variety of forms. Short message (SMSG) and Shared Message Queue (MSGQ) provide send/receive semantics and are used to transfer short point-to-point messages. FMA Distributed Memory (FMA DM) is used to execute PUT, GET, and atomic memory operations (AMOs). FMA PUT transactions do not require the registration of send memory.

BTE: BTE is well-suited for large, asynchronous message transfers because once a transfer is initiated, up to 4 GB of data can be transferred by the Gemini hardware without CPU involvement [2].

XPMEM: XPMEM [6] is an open-source kernel module and user-space library that provides cross-process memory mapping capabilities that allow for single-copy address-space to address-space transfers between cooperating processes, thus avoiding costs associated with copy-in/copy-out (CICO) semantics that require data buffers associated with a transfer to be copied twice. That is, a copy of the buffer into a shared memory region by the sender and a copy out of the shared region by the receiver. XPMEM exposes a small API that is made up of 3 major functions: `xpmem_make()`, `xpmem_get()`, and `xpmem_attach()`.

At a high-level, XPMEM setup first requires process A to export a region of its virtual address space to a cooperating process B by calling `xpmem_make()`, which returns an XPMEM segment ID. The XPMEM segment ID is then shared with B via an out-of-band communication mechanism. Next, B calls `xpmem_get()` to obtain an access permit ID (apid) that is associated with A's shared segment information. Finally, B attaches to a subset of the exported region by calling `xpmem_attach()`. Once B is successfully attached, that subset of A's exported memory region is directly accessible to B. In particular, B can now perform single-copy transfers via direct loads and stores to within that region.

III. Related Work

Many design aspects of Cray's MPI implementation for the Cray XE/XK [7] are echoed within the open-source *ugni* BTL that was implemented within Open MPI to support the same target high-speed network. This is primarily due to a nice mapping between uGNI data transfer protocols and required Network Module (Netmod)/BTL functionality. For example, SMSG is a natural choice for implementing short message

support within the *ugni* BTL because of its low latency and high messaging rates, but is not well-suited for large message transfers because of its high memory requirements due to SMSG *Mailbox* resources.

Cray provides a high-performance MPI implementation based on MPICH2 [8], an open-source MPI implementation, within its Message Passing Toolkit (MPT) that targets Gemini through a closed-source uGNI-based Netmod, which directly interfaces with the uGNI [7]. Short and medium messages use an eager message protocol and large messages use a rendezvous protocol, which is also seen within the *ugni* BTL. A registration library leveraging Linux MMU Notifier, uDREG, was used to alleviate memory registration costs for large message transfers within Cray's MPI [7], while a user-level registration cache was leveraged by both the *vader* and *ugni* BTLs. Cray's MPI implementation also leverages XPMEM for intra-node communication, but published design and implementation details surrounding their usage of XPMEM could not be found. Therefore, we cannot compare and contrast design and implementation decisions regarding the two XPMEM drivers.

The Eager GET protocol presented in this paper is similar to the one presented in [7], but was independently discovered during our implementation of the *ugni* BTL – when the need for such a protocol became apparent.

IV. Intra-Node Communication Component

vader[1], a newly introduced BTL, is the component that provides protocols for intra-node communication for the Cray XE/XK system architectures[2]. *vader* leverages XPMEM and implements its queues using a similar lock-free scheme presented in [9].

A. Message Protocols

vader uses an eager protocol for short message transfers. Each process allocates space for 4 kB blocks of per-peer receive queues – which we will refer to as *fastboxes* in order to leverage already established terminology [9] – and a single shared receive queue. Each *fastbox* is broken up into 128-byte chunks - each reserving 2 bytes of storage for message header information.

Messages smaller than 126 bytes are preferentially placed in a *fastbox* when space permits; otherwise, they are placed on the target peer's shared receive queue. Short messages that are larger than the *fastbox* size limit but smaller than the single-copy threshold, are simply copied into the target's shared receive queue. Messages larger than the single-copy limit but smaller than the eager limit are handled differently. In particular, only the sender-side base pointer is sent eagerly. When the eager fragment is processed on the receiver side, the base-pointer is used to perform a direct copy from the sender's address space into the receiver's.

[1]Internal development name that has no meaning.

[2]*vader* can also be used on other Linux platforms when XPMEM support is present.

A rendezvous protocol is used for messages larger than the eager limit. PUT and GET semantics are facilitated by XPMEM. More details surrounding *vader*'s design and implementation are provided in [10].

V. Inter-Node Communication Component

This section describes the *ugni* BTL, which implements the protocols required for inter-node communication. The *ugni* BTL uses an eager protocol for short data messages and a rendezvous protocol for long data messages. Further in this section, we describe the bottlenecks of the current protocols for Gemini and introduce new protocols to overcome these bottlenecks.

A. Initialization and Connection Setup

Connection information must be exchanged before the short data SMSG-based protocol can be used. Connection establishment between two peers involves exchanging SMSG attributes and control information via `GNI_EpPostData()`. By default, connections are established lazily between peers on an on-demand basis, so memory resource overhead associated with connection establishment will be representative of the application's communication characteristics.

B. Short Message Protocol

ugni uses an eager protocol for short message transfers, where each process reserves and registers, per-peer destination buffers, called *Mailboxes*. During a message transfer, the sender directly writes data to its designated *Mailbox* on the receiver's side. An SMSG connection is established between two peers during their first short message exchange. Once a connection is established, the sending process sends SMSG messages through the established channel via `GNI_SmsgSendWTag()`. Short messages sent through this channel include a header that describes the message type and size. SMSG handles the delivery to the remote *Mailbox* and raises a local completion event on the sender's SMSG endpoint upon successful delivery.

C. Long Message Protocol

ugni uses a rendezvous protocol for long data message transfers. When a sending process is ready to send a long message, it first sends a ready-to-send (RTS) message to the receiving process and waits for the clear-to-send (CTS) message from the receiving process. The RTS message, in addition to expressing the intent to send the message, also provides the data buffer information to the receiver. On receiving the CTS from the receiving process, the sender directly writes data to the target receive buffer using PUT, or the receiver can initiate the data transfer using GET without sending the CTS. After completion, a finish (FIN) is sent to indicate the completion of message transfer.

The RTS and CTS messages are handled by the lower-latency short message protocol. Data transfer is handled by the BTE's PUT or GET operations. Data transfer using GET requires that the data buffer on both sides (sender and receiver)

be a multiple of 4 and 4-byte aligned. When buffer size and alignment restrictions are not met, PUT is used as a fallback protocol.

D. Scalability Bottlenecks and Optimizations

Using an SMSG-based eager message protocol for short message transfers requires a significant amount of registered memory resources, thus creating the potential for memory resource scarcity for other processes on the same node. To alleviate this problem, SMSG is used for communication up to a certain small message size, which is a configurable run-time option. Above the SMSG limit, but less than the rendezvous limit, *ugni* switches to the Eager GET protocol. This protocol uses a small shared pool of pre-registered buffers as opposed to the per-peer *Mailboxes* used by the SMSG protocol.

Eager GET Protocol: The sending process sends an `Eager_Get_Request` to the receiver, which indicates the intent to send a message and also provides data buffer information. Once the receiving side has processed the `Eager_Get_Request` and is ready to receive the message, the receiver reserves a preregistered buffer from a shared pool and uses GET to fetch the data specified within the `Eager_Get_Request` message. In the *ugni* BTL, the sender sends the `Eager_Get_Request` message along with header information indicating an Eager GET protocol using `GNI_SmsgSendWTag()`. Data transfer is completed with either BTE or FMA GET. Once the receiver process receives the message, it sends an *RDMA_COMPLETE* message to the sender, thus completing the message transfer. Upon receipt of the *RDMA_COMPLETE* message the sender marks the message as complete and frees any resources associated with the message. Figures 2 and 3 are control flow diagrams (sender- and receiver-side) for the Eager GET protocol.

Put Fallback Protocol: Gemini imposes a 4-byte data buffer alignment requirement on all FMA and BTE GET operations. If a GET operation cannot be completed due to buffer size or alignment violations, *ugni* will use the Put Fallback protocol.

The receiver sends a request-to-receive (RTR) message to the sender that includes receiver-side data buffer information. The sender uses the information stored in the RTR to send the data using either FMA or BTE PUT. Once the PUT operation is complete the sender sends a FIN to indicate the completion of the request.

VI. Evaluation

This section describes the experimental test bed, micro-benchmarks, and application benchmarks used for evaluating our implementation. To evaluate our extensions to Open MPI, we compare the performance and scaling characteristics of our implementation with the vendor-supplied MPI library – referred to as "Vendor MPI" in the following sections. First, we compare Open MPI's latency and bandwidth characteristics to Vendor MPI's at small scale. Then, we evaluate our implementation's performance and scaling characteristics

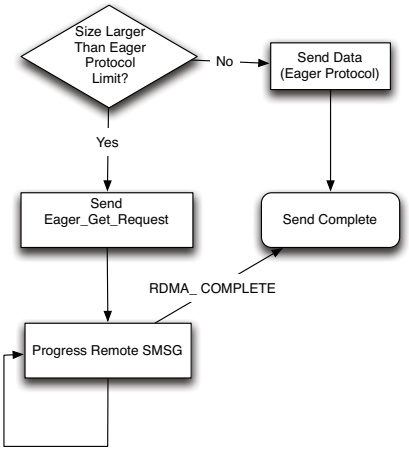

Fig. 2. Figure showing the steps involved in the Eager GET protocol (sender-side).

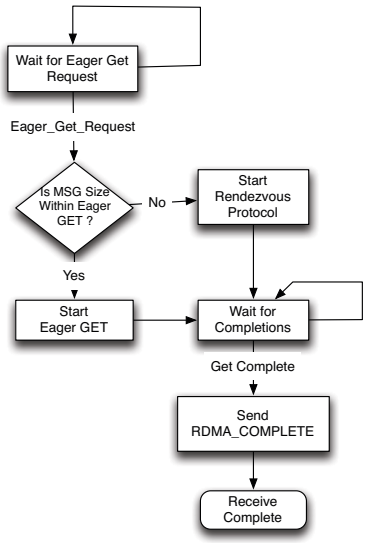

Fig. 3. Figure showing the steps involved in the Eager GET protocol (receiver-side).

beyond micro-benchmark evaluation using two application benchmarks: LAMMPS and AMG2006.

A. Test Platform

Performance results were gathered on *Cielo*, a capability-class Cray XE6 housed at Los Alamos National Laboratory and operated by Advanced Computing at Extreme Scale (ACES). *Cielo*'s Gemini System Interconnect has a 16x12x24 (XYZ) 3-dimensional torus topology built from Gemini Application-Specific Integrated Circuits (ASICs) that provide 2 Network Interface Controllers (NICs) and a 48-port router. Each Gemini connects two 16-core (2.4 Ghz 8-core AMD Opteron Magny-Cours) nodes, each with 32 GB memory. *Cielo* has 8,944 compute nodes totaling 143,104

compute cores and 272 6-core AMD Opteron Istanbul service nodes. Compute nodes run Cray Linux Environment (CLE), a Linux-based operating system. Cluster Compatibility Mode (CCM) was not used for any of the performance evaluation runs presented in this paper. Data were collected during regular operating hours, so the system was servicing other workloads alongside the performance evaluation runs.

A general overview of the system software used for all tests is as follows: CLE 4.0.up02, XPMEM 0.1-2.04, uGNI 2.3-1.0400, Open MPI 1.7 pre-release (development trunk) compiled with GCC 4.6.2, and Cray MPT (mpich2/5.4.2). All benchmark binaries were statically built and reported data for a given application and core count were sequentially run within a single interactive compute resource allocation to minimize timing differences due to node placement.

B. Benchmarks and Applications

1) Micro-Benchmarks: Small-scale micro-benchmark performance results were gathered using NetPIPE [11], a protocol independent performance tool. NetPIPE performs simple ping-pong tests at various message sizes. Open MPI was configured to switch from the SMSG protocol to the Eager GET protocol at 1 kB and the rendezvous protocol at 8 kB, as was Vendor MPI. This was done to highlight the effects of the Eager GET protocol.

2) Applications: Two tier 1 Advanced Simulation and Computing (ASC) Sequoia benchmark applications [12] were used to evaluate the performance characteristics of Open MPI and Vendor MPI on *Cielo*.

AMG2006 is a parallel algebraic multi-grid solver for linear systems written in International Organization for Standardization (ISO) C. AMG2006 was compiled with the HYPRE_NO_GLOBAL_PARTITION option, as recommended by the benchmark documentation.

LAMMPS [13] is an open-source classical molecular dynamics code written in C++, whose communication is primarily nearest neighbor.

All tests, including the micro-benchmarks, were compiled with GCC 4.6.2.

C. Results

1) Micro-benchmark Latency: Figure 4 shows message exchange latencies between two MPI processes using Open MPI and Vendor MPI, as reported by the NetPIPE benchmark, where MPI processes were configured to be on a different node. The 1-byte messages latencies are 1.20 μs and 1.32 μs for Open MPI and Vendor MPI, respectively. The 1 MB message latencies of Open MPI and Vendor MPI are 189.54 μs and 192.07 μs, respectively. Open MPI and Vendor MPI message latencies are very similar for most message sizes, except for medium message sizes, where the Eager GET protocol is used. In particular, at the 1 kB message size, Open MPI's message latency is 4.13 μs – 39.71% better than Vendor MPI's.

Open MPI uses the Put Fallback protocol for all messages that are not a multiple of 4 and not 4-byte aligned. Open MPI's message latencies are better than Vendor MPI's for message sizes that fall within this regime. In Figure 4, we can observe message latency spikes for messages that fall within this regime. At 8,195 B, the reported message latencies of Open MPI and Vendor MPI are 7.74 μs and 8.33 μs, respectively. For message transfers beyond 512 KB, this performance advantage disappears.

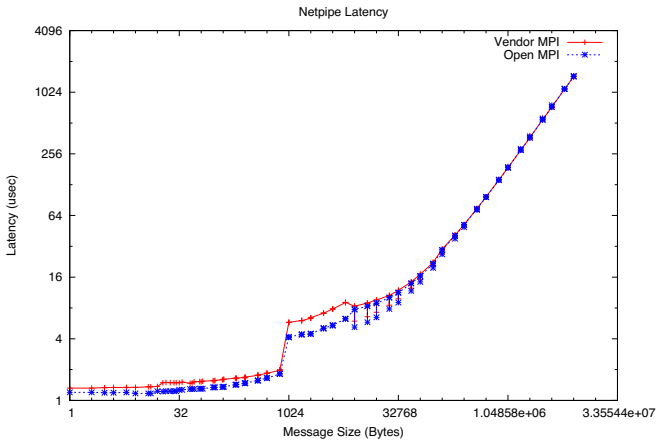

Fig. 4. Latencies on Cielo as reported by NetPIPE (log-log plot).

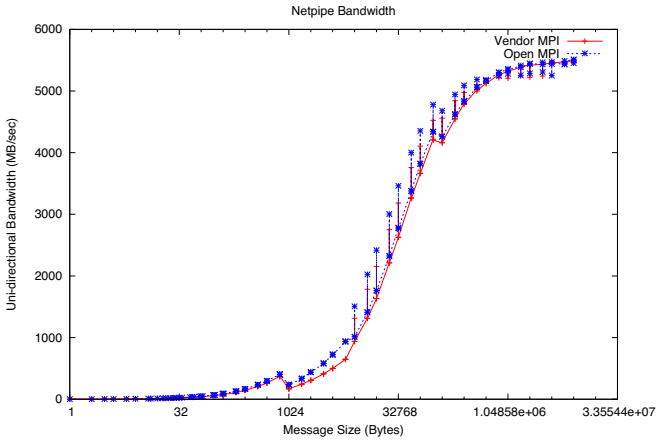

Fig. 5. Bandwidth on Cielo as reported by NetPIPE (log-log plot).

2) Micro-benchmark Bandwidth: Figure 5 shows the uni-directional bandwidth of Open MPI compared to Vendor MPI, as measured by the NetPIPE benchmark. The bandwidths achieved by both implementations are very similar. Open MPI and Vendor MPI achieve 5.32 GB/s and 5.34 GB/s of message bandwidth for an 8 MB message, respectively.

For messages that are not a multiple of four and not 4-byte aligned, Open MPI achieves better bandwidths than Vendor MPI. Within this message regime, Open MPI uses the Put Fallback protocol. The effects of the protocol can be seen as dips in bandwidth in Figure 5. For 8,195 B messages, the reported bandwidths for Open MPI and Vendor MPI are 1,010.22 MB/s and 937.81 MB/s, respectively.

To demonstrate the performance of our implementation outside of micro-benchmarks, we ran two applications with different communication characteristics: LAMMPS and AMG2006.

Figure 6 shows the loop time of 100 steps of LAMMPS for the weak-scaling Lennard-Jones Liquid problem for both Open MPI and Vendor MPI. The local size for each simulation was fixed at 256,000 atoms. At 49,152 processes, LAMMPS using Open MPI completed 100 steps in 95.65 seconds, and LAMMPS using Vendor MPI completes 100 steps in 95.88 seconds. At some problem sizes, LAMMPS using Open MPI slightly outperforms Vendor MPI – particularly at 32, 64, 1,024, 2,048, 8,192, and 49,152 processes. However, the general trend is that both Open MPI and Vendor MPI exhibit very similar performance and scaling characteristics within LAMMPS.

Figure 7 shows the ideal efficiency for LAMMPS for the weak-scaling Lennard-Jones Liquid problem compared with calculated efficiencies using Open MPI and Vendor MPI, as the number of cores are increased. This graph shows that Open MPI and Vendor MPI have similar scaling characteristics. At 49,152 processes, the LAMMPS run using Open MPI achieves 88.2% parallel efficiency, and LAMMPS run using Vendor MPI achieves 87.4% parallel efficiency.

Figure 8 shows the figure of merit (FOM) for AMG2006 as a function of core count, per the provided benchmark instructions, for Open MPI and Vendor MPI. We ran AMG2006 in a weak-scaling mode where the problem size per node is kept constant. The benchmark was configured to solve a 3D 7-point Laplace problem on a cube. The recommended FOM is *System Size * (Number of Iterations / Solve Time)*. We can observe that up to 16,384-cores, Open MPI performs similarly to Vendor MPI. Above 32,768-cores, Vendor MPI outperforms Open MPI. We suspect that this is primarily due to an increasing amount of time spent in collective routines, as the AMG2006 simulation scales in core count. In particular, mpiP [14] reported that approximately 20% of the total simulation time was spent in `MPI_Allreduce()`, which was up from the 15% that it spent in the same routine at 16,348 cores. Unfortunately, due to system time limitations, we were unable to profile AMG2006 at 49,152 cores.

VII. Analysis

3) Application Benchmarks: With our extensions to support the Cray XE/XK system architectures, Open MPI and the vendor-supplied MPI both exhibit similar micro-benchmark latency and bandwidth performance characteristics. Furthermore, application benchmark results show both implementa-

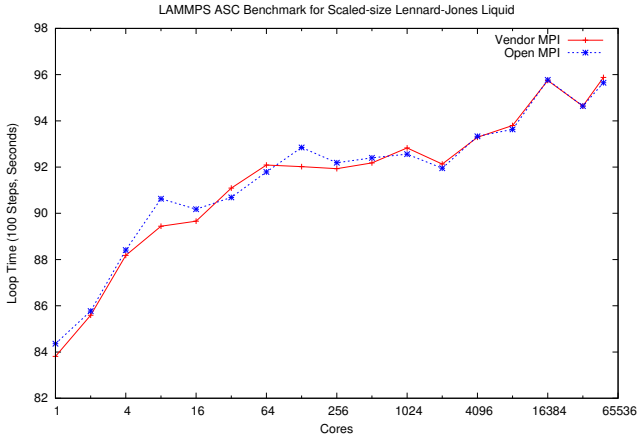

Fig. 6. Reported LAMMPS loop time in seconds for 100 iterations of the weak-scaling Lennard-Jones Liquid problem (lower is better).

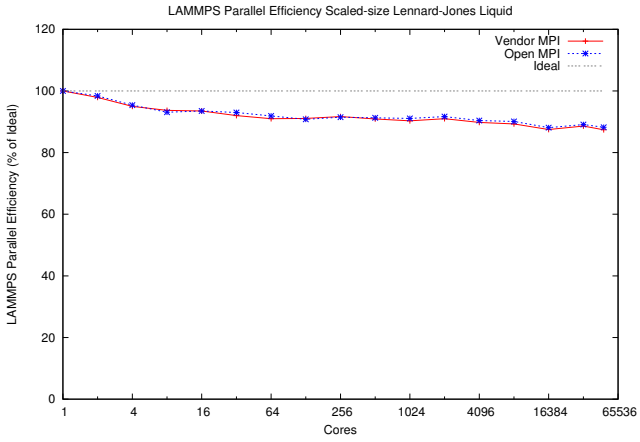

Fig. 7. Calculated LAMMPS parallel efficiency for 100 iterations of the weak-scaling Lennard-Jones Liquid problem (higher is better).

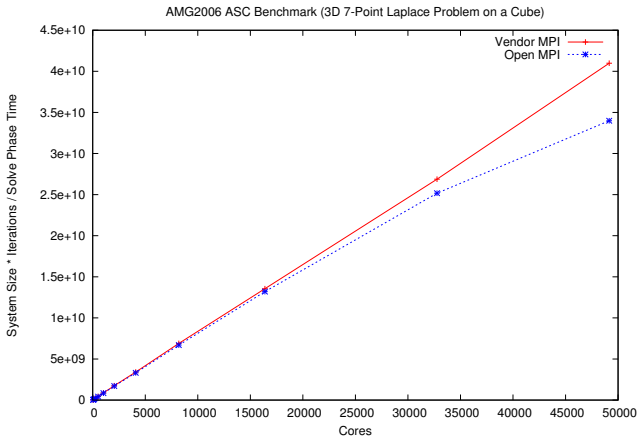

Fig. 8. Recommended figure of merit versus core count for a 3D 7-Point Laplace problem on a cube (higher is better).

tions exhibit similar performance characteristics when communication isn't dominated by collective operations.

Figures 4 and 5 establish our implementation's basic performance characteristics. In Figure 4 we can observe that Open MPI's short data latency is better than Vendor MPI's by 39.71% at 1,024 bytes. For large data, however, the performance is very similar.

Open MPI outperforms Vendor MPI within the 1 kB to 8 kB message size range – where both implementations were configured to use the newly introduced Eager GET protocol.

In Figure 5, we can observe that Open MPI and Vendor MPI have similar bandwidth characteristics.

For messages that are not a multiple of four and not 4-byte aligned, our Open MPI implementation uses the Put Fallback protocol because GET cannot be used. This protocol was introduced because the sender has insufficient information to detect when GET cannot be used. The Put Fallback protocol has higher latency due to the overhead of an additional small message. Figures 4 and 5 show the effect of the Put Fallback protocol (e.g., latency spikes). Open MPI has a latency of 5.12 μs for 8,192-byte messages and a latency of 7.69 μs for 8,195-byte messages. For comparison with 8,195-byte messages, Open MPI performs 7.62% better than Vendor MPI.

After establishing basic performance characteristics, we measure the impact of our MPI implementation on application performance. We ran our MPI implementation against LAMMPS and AMG2006 – both having very different communication characteristics. From the data presented in Figures 6, 7, and 8, we can observe that both Vendor MPI and our implementation have very similar performance and scaling characteristics.

In order to understand the application benchmark performance results, we profiled the applications using mpiP and determined approximately how much time was spent in MPI routines. Figure 9 shows the percentage of time spent in MPI routines. LAMMPS spends a significant time in `MPI_Send()`, `MPI_Wait()`, and `MPI_Waitall()`. As seen in the Figure 4 and 5, the message transfer performance characteristics of Open MPI and Vendor MPI are very similar, resulting in similar performance characteristics between both MPI implementations. Figure 9 shows that AMG2006 spends a significant amount of time in `MPI_Allreduce()` – an increase from 15% to 20% when the system size is increased from 16,384 to 32,768 cores. In addition, our measurements show that Open MPI's default 8-byte `MPI_Allreduce()` is approximately 50% worse than Vendor MPI's at 16,384 cores. We believe that AMG2006 spending a significant time in `MPI_Allreduce()`, coupled with the degraded performance of Open MPI's `MPI_Allreduce()`, is resulting in the growing performance disparity between the two MPI implementations above 16,384 cores.

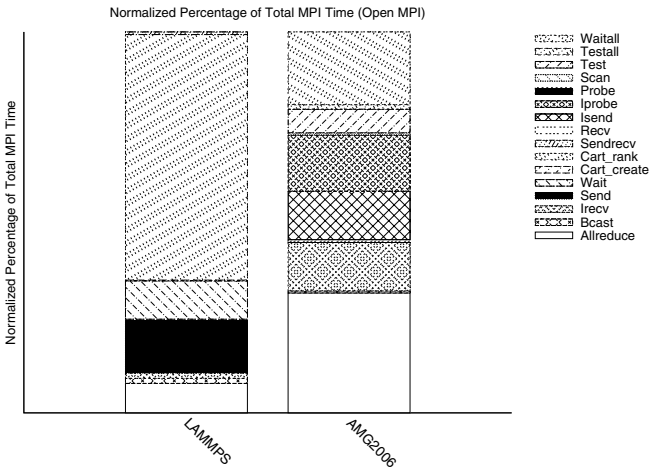

Fig. 9. Normalized percentage of time spent by LAMMPS and AMG2006 in MPI routines.

VIII. Conclusion and Future Work

An open-source MPI implementation for Cray XE/XK systems was implemented by extending Open MPI. Besides typically used short and long data protocols, new protocols were implemented to achieve desirable scaling and performance characteristics. Both performance and scaling characteristics were evaluated. Micro-benchmark results show that Open MPI's short data message latency characteristics were better than Vendor MPI's, and are very similar for long data messages. Application performance evaluation shows that Open MPI and Vendor MPI have very similar performance and scaling characteristics.

In the future, we plan to update the implementation to use the MSGQ protocol for point-to-point communication at large scale, and evaluate its effect on performance and scalability.

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-12-21530.

References

[1] TOP500.org. (2012) Top500 supercomputing sites. http://www.top500.org/. [Online]. Available: http://www.top500.org/

[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] 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.

[4] R. L. Graham, R. Brightwell, B. Barrett, G. Bosilca, and Pješivac-Grbović, "An evaluation of open mpi's matching transport layer on the cray xt," Oct 2007.

[5] Cray Inc., "Using the gni and dmapp apis," in *Cray Software Document*, vol. S-2446-4003, Dec. 2011. [Online]. Available: http://docs.cray.com/books/S-2446-4003/S-2446-4003.pdf

[6] (2011) XPMEM, cross-process memory mapping. http://code.google.com/p/xpmem/. [Online]. Available: http://code.google.com/p/xpmem/

[7] 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

[8] Argonne National Laboratory. MPICH2 : High-performance and Widely Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/.

[9] 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.

[10] M. G. Venkata, R. L. Graham, N. T. Hjelm, and S. K. Gutierrez, "Open mpi for cray xe/xk systems," in *Proceedings of the 2012 Cray User Group Annual Technical Conference*, May 2012.

[11] Ames Laboratory. (2012) NetPIPE - a network protocol independent performance evaluator. http://www.scl.ameslab.gov/netpipe/. [Online]. Available: http://www.scl.ameslab.gov/netpipe/

[12] Lawrence Livermore National Laboratory. ASC Sequoia Benchmark Codes. https://asc.llnl.gov/sequoia/benchmarks/.

[13] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1 – 19, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002199918571039X

[14] Lawrence Livermore National Laboratory. (2010) mpiP: lightweight, scalable mpi profiling. http://mpip.sourceforge.net/. [Online]. Available: http://mpip.sourceforge.net/