Open MPI for HPE Cray EX Systems

Howard Pritchard
Los Alamos National Laboratory
Los Alamos, NM, USA
howardp@lanl.gov

Thomas Naughton, Amir Shehata, David Bernholdt
Oak Ridge National Laboratory
Oak Ridge, TN, USA
{naughtont,shehataa,bernholdtde}@ornl.gov

***Abstract*—Open MPI is an open-source implementation of the MPI-3 standard that is developed and maintained by collaborators from academia, industry, and national laboratories.**

Oak Ridge National Laboratory (ORNL) and Los Alamos National Laboratory (LANL) are collaborating on porting and optimizing Open MPI and related components for use on HPE Cray EX systems, with a focus on the DOE Frontier and Aurora exa-scale systems.

A key component of this effort involves development of a new LinkX Open Fabrics Interface (OFI) provider. In this paper, we describe enhancements to Open MPI, OpenPMIx runtime components, and the LinkX OFI provider. Performance results are presented for point to point and collective communication operations using both the vendor CXI provider and the LinkX provider, including results obtained using GPU accelerators. Recommended deployment options for EX systems will be discussed, along with future work.

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Open MPI is an open-source implementation of the MPI-3 standard that is developed and maintained by collaborators from academia, industry, and national laboratories [4]. It can make use of both UCX [7] and OFI libfabrics [5] to effectively utilize a wide variety of underlying network fabrics. Additionally, Open MPI's use of OpenPMIx [2] allows for interoperability with multiple resource managers and can support both direct launch for systems supporting OpenPMIx as well as Open MPI's internal mpirun mechanism.

Oak Ridge National Laboratory (ORNL) and Los Alamos National Laboratory (LANL) are collaborating on porting and optimizing Open MPI and related components for use on HPE Cray EX systems, with a focus on the ECP Frontier and Aurora exa-scale systems. In this paper, we highlight the design and development of necessary infrastructure that enables Open MPI to efficiently support both intra-node and inter-node communication on these new Slingshot 11 based systems.

This paper describes the modifications and extensions to Open MPI and components of the OpenPMIx Reference Runtime Environment (PRRTE) [1] to enable deployment on HPE's Slingshot 11 (SS11) network. In order to address significant performance issues encountered using the vendor's OFI libfabric provider - CXI - a new libfabric is being developed as part of the effort, and will also be described.

The rest of this paper is organized as follows. Section II-A describes the structure of Open MPI, with a focus on the components that use the OFI libfabric API. Changes made to the software to make use of the CXI and LINKx providers is discussed, changes made to improve support for systems with multiple GPU accelerators per node, and changes to PRRTE to support the vendor's PALS application launch system. The performance issues observed using the CXI provider directly will also be presented. Section III describes the LINKx provider being developed to address these performance issues. Section IV presents performance results using the CXI and LINKx providers. In Section V future work is discussed. A description of best practices for deploying Open MPI on HPE EX systems is provided in an appendix.

II. Open MPI

A. Background

Open MPI is structured around a modular component architecture (MCA), consisting of a set of frameworks that provide abstract interfaces used by upper layers to implement the MPI functionality needed by an application [8]. A simplified depiction of this structure is presented in Figure 1. Each of these frameworks includes one or more components which actually implement the abstract interfaces. Some frameworks allow for multiple components to be active, while others only allow a single component to be active while an application is using MPI. The frameworks of particular interest in this paper are the message transport layer (MTL) and the byte transport layer (BTL). The MTL framework only allows a single component to be active, while the BTL allows multiple components to be active concurrently. The MTL makes use of network transport layers that support MPI tag matching semantics, while the OB1 component of the point-to-point management layer framework (PML) makes use of multiple BTLs and handles MPI tag matching internally. Open MPI can also make use of BTLs to support MPI-RMA (one-sided) operations.

Open MPI has supported use of OFI libfabric via the OFI MTL and BTL for many years and is widely used on AWS EC2 instances (EFA provider), Cornelis networks (OPX provider), and Cisco USNIC networks. MCA parameters may

be used to specify that other providers, in particular layered providers, are to be used.

B. Modifications for CXI and LINKx Providers

Relatively few modifications were needed to the Open MPI OFI components to function over SS11.

For the OFI MTL, corrections were made to the way `fi_getname` was being invoked in order to support the longer libfabric endpoint(EP) names used by the LINKx provider, as well as modifications to the way registration of GPU memory was being done, as the CXI provider handles GPU memory registration internally. An optimization was also added to improve selection of SS11 devices to use based on node-level locality information.

More extensive changes were required to the OFI BTL. In addition to the aforementioned change to the use of `fi_getname`, changes were made in the BTL's memory registration path to handle the CXI provider's `FI_MR_ENDPOINT` memory registration mode requirement. Support was added to use the `fi_inject` method for short messages.

Note that all of these changes are present in the Open MPI main and v5.0.x branches. The community does not plan to support older releases of Open MPI for use with the CXI provider. In addition, almost all changes to the OFI layers of Open MPI to support GPU direct transfers for AMD accelerators are only present in these two branches of Open MPI.

In addition to changes in Open MPI, enhancements to the PRRTE job launch system were required for the ALCF Aurora systems. In particular, a PALS aware launcher (PLM) and environment specific services (ESS) were implemented to use the PALS launch mechanism for starting the PRRTE daemons on the nodes used by an application. Without this, Open MPI would not be able to use the CXI provider if the system has VNI enforcement enabled because the VNI key that allows clients to access the fabric must be setup by the privileged system resource manager. Note on Aurora, support for native launch of Open MPI applications using `aprun` is not planned as this launch mechanism lacks support for PMIx.

C. Performance problems using the CXI Provider

Performance testing following these modifications revealed significant issues owing to the design of the CXI provider. Unlike other OFI providers used by the OFI MTL, the CXI provider lacks any special pathway for intra-node messaging. Consequently, when using the CXI provider alone, intra-node latency and bandwidth realized by Open MPI was poor, particularly when using many MPI processes per node. Reworking the MTL framework to enable use of multiple components concurrently would require extensive redesign and buy-in from multiple stake-holders. In principle, the OB1 path through the OFI and shared memory BTL's could provide a way to work around this issue with the CXI provider. However, a performance analysis of this pathway indicated that, even with the addition of `fi_inject` support to the OFI BTL, it would require extensive refactoring to approach the performance of the OFI MTL for inter-node messaging.

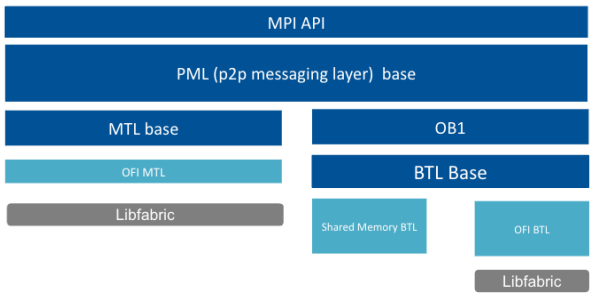


Fig. 1: Simplified depiction of Open MPI modular component architecture. Shown are the frameworks relevant to MPI message passing. Only components interfacing to OFI libfabric or shared memory are presented.

III. LINKx

A. Slingshot 11 Driver

The CXI provider supports the following paths for communication over the SS11 network hardware:

- Host to Host memory over SS11,
- GPU Device to Device memory over SS11,
- GPU Device to Host memory over SS11,
- Host to GPU Device memory over SS11.

The CXI provider does not provide a solution for on-node communication. On-node communication can be done via the libfabric CXI provider, however, messages will egress the node and ingress again. This will be subject to bottlenecks on performance that would not be there if the proper shared memory mechanism is used; xpmem, cma, or GPU IPC mechanisms.

The libfabric LINKx provider was created to allow both inter- and intra-node communication via a single libfabric provider. This design allows for the Open MPI OFI MTL to achieve good performance for both inter and intra-node messages II-A.

B. Design Overview

Libfabric provides a list of available interfaces to the application to select from. For example if there are both Ethernet and CXI interfaces available, libfabric via the `fi_getinfo()` API will provide a list of all possible methods that can be used for inter-process communication. These are represented as an ordered list of `fi_info` structures. Moreover, libfabric orders this list such that the most performant methods are listed first.

The application has the ability to then select one or more of these communication methods. Open MPI, and other applications, typically only select one of these methods per process.

This presents a problem when wanting to use libfabric for both inter- and intra-node communication. Each application will then need to implement logic to use libfabric's shared memory provider for intra-node communication, and another device for inter-node communication. In fact, Open MPI, does this very thing when using the OFI PML and its associated

BTLS, where the best method between each communicating peer is selected. On-node peers use the BTL shared memory (`sm`) module and off-node peers will use a different BTL component.

However, each application that wants to do both inter- and intra-process communication will need to implement the same logic. Therefore, it can be helpful to push this logic into the libfabric layer to avoid duplication of code by all applications needing this functionality.

LINKx is a new libfabric provider being developed that allows the application to link multiple libfabric providers together. For the purposes of this paper we will focus on one particular use case: linking SHM & CXI providers. This allows an application seamless access to both on and off node communication via a single libfabric provider. Instead, the LINKx layer is responsible for determining how best to access a given target and uses the correct provider for communication.

1) LINKx Flow: Figure 2 illustrates use of LINKx within Open MPI. The actions associated with this diagram are outlined below and show the basic steps for an application (e.g., Open MPI) when using LINKx.

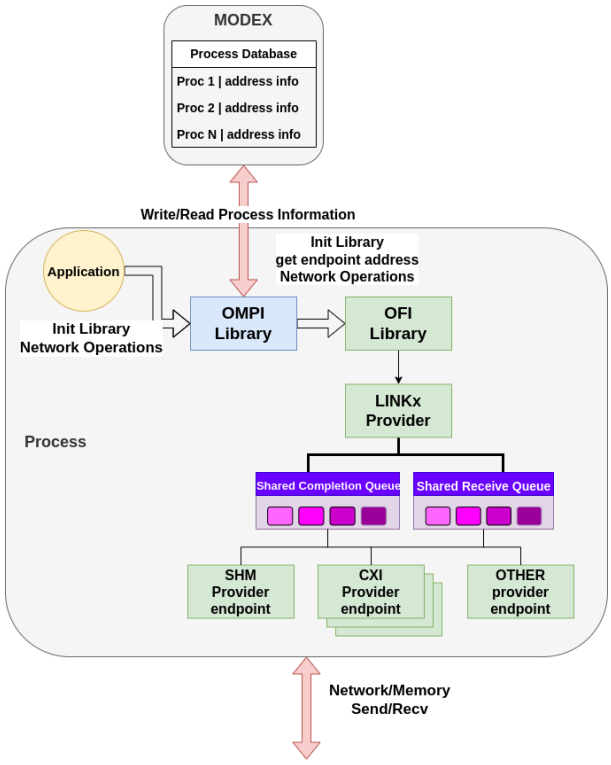


Fig. 2: Illustration of OFI libfabric LINKx provider joining SHM and CXI core providers for use by Open MPI library, which uses new shared queue capability managed by LinkX.

i. Initialization calls `fi_getinfo()` to get list of `fi_info` structures ordered from most to least performant; each structure describes one interface that can be used for communication. The LINKx `fi_info` structure returned in the list joins `SHM` with `CXI` as single provider
ii. Application selects the LINKx `fi_info` structure & fully initializes provider via typical procedures: `fi_fabric()`, `fi_domain()`, `fi_endpoint()`, etc.
iii. During setup, LINKx internally builds structures to track different `SHM` & `CXI` core providers included in the link.
iv. Before MODEX exchange, Open MPI requests the address of the provider, LINKx concatenates the addresses of all core providers in the link and Open MPI then publishes the data to MODEX.
v. After MODEX exchange, Open MPI reads all peer addresses that should also be using LINKx from MODEX and passes data to LINKx. LINKx parses the addresses and inserts them into the corresponding core providers within the link.
vi. Open MPI can then proceed to use the libfabric memory operation APIs, e.g, `fi_tsenddata()` to communicate with peers; LINKx examines the peer and based on locality decides which provider to use.

C. LINKx Shared Data Structures

With multiple core providers abstracted under one single LINKx provider, it becomes necessary to unify both the completion and receive queues.

1) Shared Completion Queues: When an operation completes a provider pushes it on the completion queue. libfabric provides APIs that can then be used to pull events off the completion queues. In essence, LINKx becomes the application to the core providers in the link. Instead of each core provider having its own completion queue, and then LINKx reading the completions off each core provider and inserting it into its own completion queue, which the application can query, on initialization the LINKx provider exports its completion queue to be used by the core providers. The owner of the shared completion queue, LINKx in this example, initializes callbacks on completion queue creation.

```
struct fi_ops_cq_owner {
   size_t size;
   ssize_t (*write)(struct fid_peer_cq *cq, void *context,
        uint64_t flags,
      size_t len, void *buf, uint64_t data, uint64_t tag,
          fi_addr_t src);
   ssize_t (*writeerr)(struct fid_peer_cq *cq,
      const struct fi_cq_err_entry *err_entry);
};
```

Listing 1: LINKx completion queue callbacks

The core providers use these callbacks to insert directly into LINKx's completion queue. This method is efficient as it adds no overhead to access the shared completion queue.

2) Shared Receive Queues: When an application wants to receive a message, it posts a receive request with the buffer(s) to add the requested data into the libfabric provider, in this case LINKx. The receive request can identify a specific peer to receive from or it can be a receive request for data arriving from any peer. The receive request can also have a tag associated with it, if it's targeting a tagged message. The receive request is inserted on the receive request queue via the libfabric receive APIs. Since LINKx manages multiple core

providers, it is more efficient to tell the core providers to use the LINKx receive queue for matching incoming messages. This requires the use of a new data structure; the shared receive queue. When LINKx initializes the core providers it registers a shared receive queue with each core provider. The shared receive queue has two sets of callbacks associated with it. Callbacks which belong to the owner of the queue and callbacks which the core provider, dubbed peer provider, fills in.

```
struct fi_ops_srx_owner {
   size_t size;
   int (*get_msg)(struct fid_peer_srx *srx, fi_addr_t addr,
               size_t size, struct fi_peer_rx_entry
                    **entry);
   int (*get_tag)(struct fid_peer_srx *srx, fi_addr_t addr,
               uint64_t tag, struct fi_peer_rx_entry
                    **entry);
   int (*queue_msg)(struct fi_peer_rx_entry *entry);
   int (*queue_tag)(struct fi_peer_rx_entry *entry);
   void (*free_entry)(struct fi_peer_rx_entry *entry);
};

struct fi_ops_srx_peer {
   size_t size;
   int (*start_msg)(struct fi_peer_rx_entry *entry);
   int (*start_tag)(struct fi_peer_rx_entry *entry);
   int (*discard_msg)(struct fi_peer_rx_entry *entry);
   int (*discard_tag)(struct fi_peer_rx_entry *entry);
};
```

Listing 2: LINKx shared receive queue callbacks

The owner of the shared receive queue provides callback to get and queue receive requests, tagged or otherwise, off the shared receive queue. The core provider registers callbacks to start message processing or discard messages, tagged or otherwise.

For example, when using shared receive queues the workflow for expected messages is as follows:

i. Application posts a receive request to LINKx queue
ii. The core provider receives a message on the wire and queries the shared receive queue for a receive request which matches the message. It uses the `get_msg()` or `get_tag()` callbacks for that.
iii. Since a matching receive request has been posted already, a match is successful and is returned to the core provider.
iv. The core provider can then receive the message data into the buffers identified in the shared receive queue.
v. Once it completes copying the data into the application provided buffers, it will post a completion event on the shared completion queue.

When using shared receive queues the workflow for unexpected messages is as follows:

i. The core provider receives a message on the wire and queries the shared receive queue for a receive request which matches the message. It uses the `get_msg()` or `get_tag()` callbacks for that.
ii. Since no receive request has been posted by the application, no match is returned.
iii. The core provider uses the `queue_msg()` or `queue_tag()` callbacks to create a temporary receive request entry which gets added to the shared receive queue.
iv. The application then attempts to post a receive request on the shared receive queue
v. LINKx will attempt to find if there is an unexpected message which matches this receive request.
vi. LINKx will match the unexpected message and will use the core provider registered `start_msg()` or `start_tag()` callbacks to tell the core provider to receive the message data into the associated buffers
vii. When the core provider finishes copying the data into the application provided buffers it posts a completion event into the shared completion queue.

3) Hardware Assisted tag matching: The shared receive queue precludes the core providers from using any hardware assisted tag matching. LINKx will need to perform software tag matching because it needs to maintain the integrity of the shared receive queue.

Unspecified receive requests present a potential race condition that must be explicitly managed by LINKx. For example, a receive request using `MPI_ANY_SOURCE` could be fulfilled by either the SHM or CXI provider in the link. This ambiguity is resolved by having both the CXI and SHM providers call into the shared receive request queue to get a matching request. The first provider that gets that request is able to operate on it. Only after this request is returned is it safe to copy the data into the associated buffers. The tag matching is done in the `get_tag()` owner callback. By necessity this logic has to be done by the LINKx provider. Otherwise, it is possible for the CXI provider to use HW assisted tag matching and then copies the data into the buffers, while the SHM provider does the same, leading to data corruption.

IV. Experiments

A set of experiments were run using the early-access Cray XE system *Crusher* at OLCF (Figure 3) [3]. The nodes in Crusher contain 1 64-core AMD EPYC CPU with 512 GB of memory. There are 4 AMD MI250X GPUs per node, with each GPU having 2 graphic compute dies (GCDs) that present as 8 distinct GPUs. The GPUs have 64 GB of high-bandwidth memory. The CPU to GPU peak bandwidth is 36+36 GB/s over the Infinity Fabric. The bandwidth between 2 GCDs on the same GPU have a peak bandwidth of 200 GB/s, and peak bandwidth between 2 GPUs ranges from 50-100 GB/s depending on the number of connections over the the Infinity Fabric between the two GPUs. There are 4 Slingshot 11 network interfaces (hsn0-3) that are attached to the GPUs. The CPUs have greater affinity to specific GPUs and network interfaces as shown in Figure 3. This plays a crucial role in achieving proper performance and is often the first place to look when seeing unexpected communication performance.

At the time of our experiments, Crusher was running SUSE Linux Enterprise Server 15 SP4 and Linux kernel 5.14.21 for the cray-shasta environment. SLURM was version 22.05.7 with core specialization enabled, with the defaults used so that 8 core per node (1 per l3cache) is reserved for system services. The software configuration used ROCm v5.3.0, CCE 15.0.0, xpmem v2.5.2-2.4_3.30__gd0f7936.shasta. CrayMPICH was

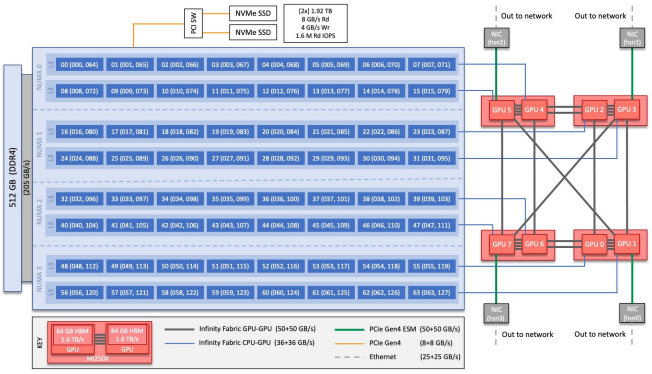


Fig. 3: Crusher compute node topology [3].

version 8.1.23 with Cray-PMI v6.1.8. Open MPI was based on v5.0.0rc11 with one additional patch for setting nearest network device [1]. The CrayMPICH runs used the default Libfabric v1.15.2.0. The Open MPI tests used the development Libfabric that includes LINKx and shared memory provider enhancements, which was from ornl-branch at SHA 38df9f668 (12 Apr 2023). All tests were taken from the OSU micro-benchmark suite version 7.0.1 [6].

In our tests we used 8 MPI processes per node to mirror the suggested usage (i.e., one for each GCD), which under Open MPI was 1 process per l3cache and process bound to core. The only exception to this mapping setting was when running inter-node point-to-point tests, which used 1 process per node (`--map-by ppr:1:node`). All tests were run using a custom testing harness that loads/unloads software modules, runs the benchmark parses results and generates the graphs. The test harness sets the `HIP_VISIBLE_DEVICES` environment variable to the best setting for Crusher during the benchmark runs.

```
mpirun \
   -x FI_USE_XPMEM -x LD_LIBRARY_PATH \
   --mca btl ^tcp,ofi,vader,openib \
   --mca pml ^ucx --mca mtl ofi \
   --mca opal_common_ofi_provider_include "shm+cxi:linkx" \
   --map-by ppr:1:l3cache --bind-to core \
   --display mapping,bindings --np 512 \
   <osu-exe> H H
    # -- or --
   <osu-exe> -d rocm D D
```

Listing 3: Example `mpirun` for Open MPI tests

```
srun \
   --cpu-bind=v,cores \
   --ntasks 512 \
   --ntasks-per-node 8 \
   -N 64 \
   -t 10000 \
   <osu-exe> H H
    --or--
   <osu-exe> -d rocm D D
```

Listing 4: Example `srun` for CrayMPICH tests

[1]We plan to include the nearest network patch in the v5.0.x branch before the Open MPI v5.0.0 release.

The command-lines for the tests were similar to those shown in Listings 3-4. The settings for LINKx indicate what Libfabric providers are to be included in the link, which in Listing 3 are the shared memory (`shm`) and SS11 (`cxi`) providers[2].

A. Point-to-point

We measured bandwidth and latency for point-to-point operations using four options: H2H (host-to-host), D2D (device-to-device), Inter-node (2 nodes), and Intra-node (1 node). These provide a baseline for the transfers within and across nodes for MPI data buffers residing in host memory and device memory. We the vendor provided CrayMPICH as a baseline measure for our performance. The graphs show three lines corresponding to CrayMPICH, Open MPI with LINKx (shm+cxi) and Open-MPI without LINKx (a single provider). The last configuration provides a functional configuration that will suffer poorer performance due to the lack of LINKx and only having access to the SS11 driver (cxi) and no sharded memory. As mentioned previously, when using the MTL framework in Open MPI, only one provider can be selected and in the non-LINKx configuration that will either be the SS11 or SHM provider, but not both.

The bi-directional bandwidth (Figure 4) shows Open MPI tracking CrayMPICH. The graphs also indicate that the LINKx layer is not degrading bandwidth when comparing the two "ompi" lines. The device-to-device performance for intra-node is very good for Open MPI as messages get larger in size, which shows the benefit of using the LINKx provider for shared memory. This also highlights the recent improvements to optimize shared memory handle caching for the Libfabric SHM provider. We found unusual results in the CrayMPICH bandwidth data for inter-node H2H, which were far lower than expected (Figure 7(a)) and needs further investigation.

The Open MPI latency (Figure 5) numbers are reasonable and generally track CrayMPICH. Open MPI with LINKx has roughly the same latency as message size gets larger, with CrayMPICH doing much better at smaller message. The CrayMPICH intra-node H2H test show much higher latency at the larger message sizes that needs further investigation.

The last point-to-point measurements show that work is needed for small message bandwidth and message rate (Figure 6). CrayMPICH has much better message rates at small messages on H2H & D2D intra-node tests. We see some difference in the Open MPI curves (with and without LINKx) for inter-node H2H & D2D that need further attention.

B. Collectives

We measured the performance of several collective operations for both H2H (host-to-host) and D2D (device-to-device) transfers. All tests were run using 8 MPI processes per node, which were mapped by l3cache and bound to core. The graphs show results from runs on Crusher using 64 nodes ($nprocs = 512$). The collective tests exercise the LINKx layer, allowing efficient on-node exchanges via SHM and efficient

[2]Note: Many of the command-line options for Open MPI are typically set via an environment file but are listed here for accuracy.

remote node communication using the CXI provider. The tests used default settings for all the collectives.

Overall we find that CrayMPICH has much better small message support than the current Open MPI with LINKx. However, the overall trends are promising when compared to CrayMPICH in Figures 7-11.

A few observations in the figures include very similar Allgather H2H data for CrayMPICH and Open MPI (Figure 9(a)). We see that Open MPI does much much better in Allgather D2D when using LINKx vs. just using CXI alone, but that at the largest messages the latency is much higher than CrayMPICH. There is an unusual spike in Alltoall D2D (Figure7(b)) with CrayMPICH that needs review. There is also a spike at the largest message size in Gather (Figure 10) for Open MPI.

V. Future Work

Support for Intel's Ponte Vecchio (PV) accelerator is planned for the near future. This will be done by adding a PV component to Open MPI's accelerator framework for use by the BTL path in Open MPI. For the MTL path, which uses the LINKx provider, the PV support in the Libfabric SHM provider will need to be evaluated.

We are currently looking into the the small message performance with Open MPI and LINKx. The LINKx capabilities are currently being reviewed, with portions being upstreamed to the public Libfabric repository. The current tests were the first to move from the Open MPI `main` branch to the `v5.0.x` branch. These v5.0.x tests have identified areas where performance drops occurred from previous measurements, and further review is needed to identify the causes for regression. We are starting to perform scale-up tests on Frontier and will work on performance improvements to collectives when using Open MPI and LINKx. The MPI one-sided support when using LINKx has a few open-issues that will be resolved in the coming months.

Lastly, we are interested to explore the use of LINKx for multi-rail support, which may be more beneficial on SS11 systems like Aurora that have more uniform on-node bandwidth between compute and network devices.

VI. Conclusion

The use of Open MPI on Slingshot 11 systems like Frontier requires care be taken to leverage shared memory and off-node communication. A new LINKx provider has been described that enables Open MPI to use both the shared memory (SHM) and Slingshot 11 (CXI) providers via the MTL framework. The LINKx provider supports joining of two or more Libfabric providers that are presented to the application as a single "linked" provider. This linking layer manages shared receive/completion queues for the linked items. The LINKx implementation has tagged message support, non-tagged and atomic Libfabric capabilities are in-progress. The shared memory provider in Librabric has also been improved, including: ROCm support, asynchronous IPC, and locking improvements.

We have outlined the key challenges address thus far to bring Open MPI up on Frontier with good performance. While there are still improvements to be made, the work is functional and performance is showing good trends compared to that of the vendor's well tuned CrayMPICH. The experiments highlight the importance of process binding to ensure good affinity to GPU and network interfaces from MPI processes, which is especially important on the Frontier system.

Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the 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. Howard Pritchard acknowledges support by the National Nuclear Security Administration. Los Alamos National Laboratory is operated by Triad National Security, LLC for the U.S. Department of Energy under contract 89233218CNA000001. This research was partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] PMIx Reference Runtime Environment. https://docs.prrte.org/en/latest/, 2023.

[2] Ralph H. Castain, David Solt, Joshua Hursey, and Aurelien Bouteiller. Pmix: Process management for exascale environments. In *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Crusher User Guide. https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html.

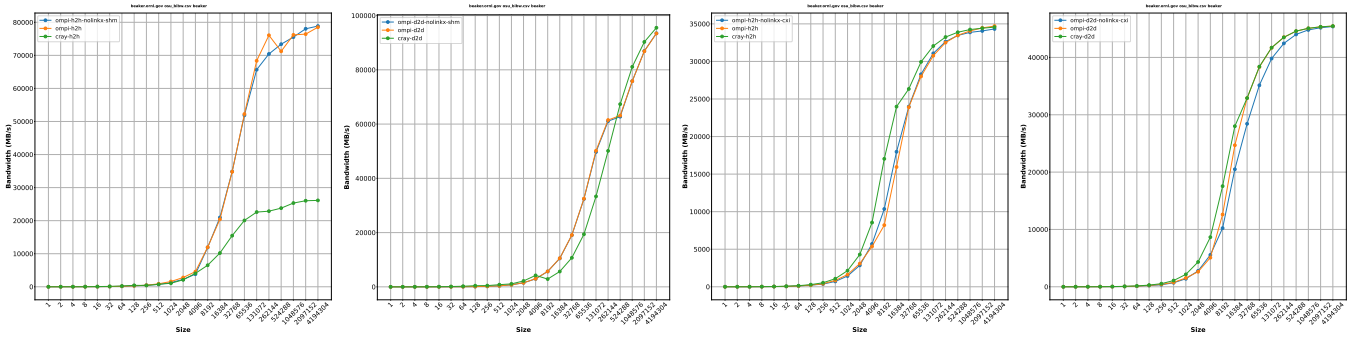
[4] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[5] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert Russell, Howard Pritchard, and Jeffrey Squyres. A Brief Introduction to the OpenFabrics Interfaces–A New Network API for Maximizing High Performance Application Efficiency. In *Proceedings of the 23rd Annual Symposium on High-Performance Interconnects*, August 2015.

[6] Ohio State University Microbenchmarks. https://mvapich.cse.ohio-state.edu/benchmarks/.

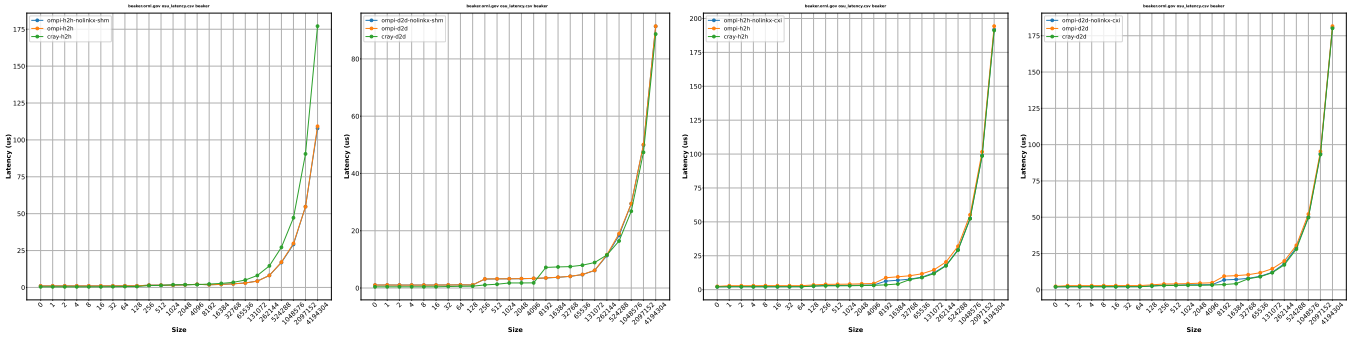
[7] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. Ucx: An open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43, 2015.

[8] Jeffrey M Squyres and Andrew 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*, pages 167–185. Springer, 2004.



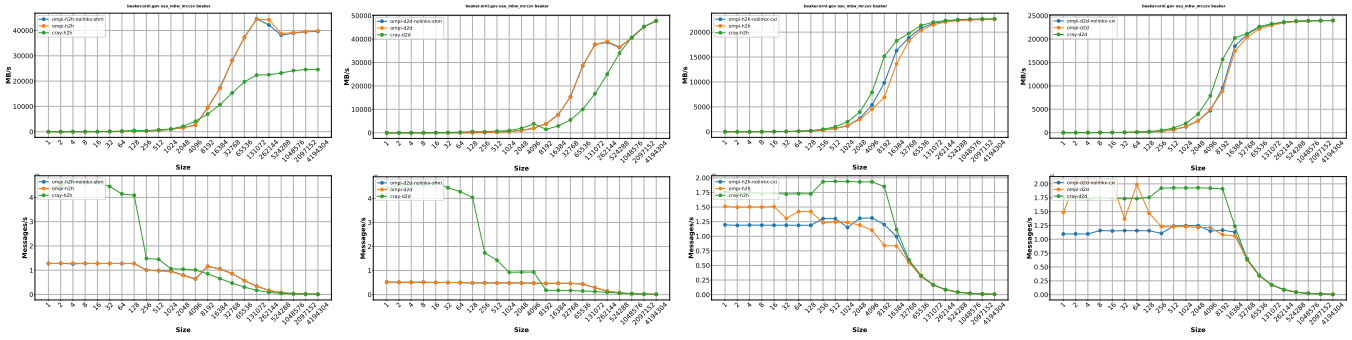
(a) H2H intra-node (b) D2D intra-node (c) H2H inter-node (d) D2D inter-node

Fig. 4: Point-to-Point Bidirectional Bandwidth `osu_bibw`



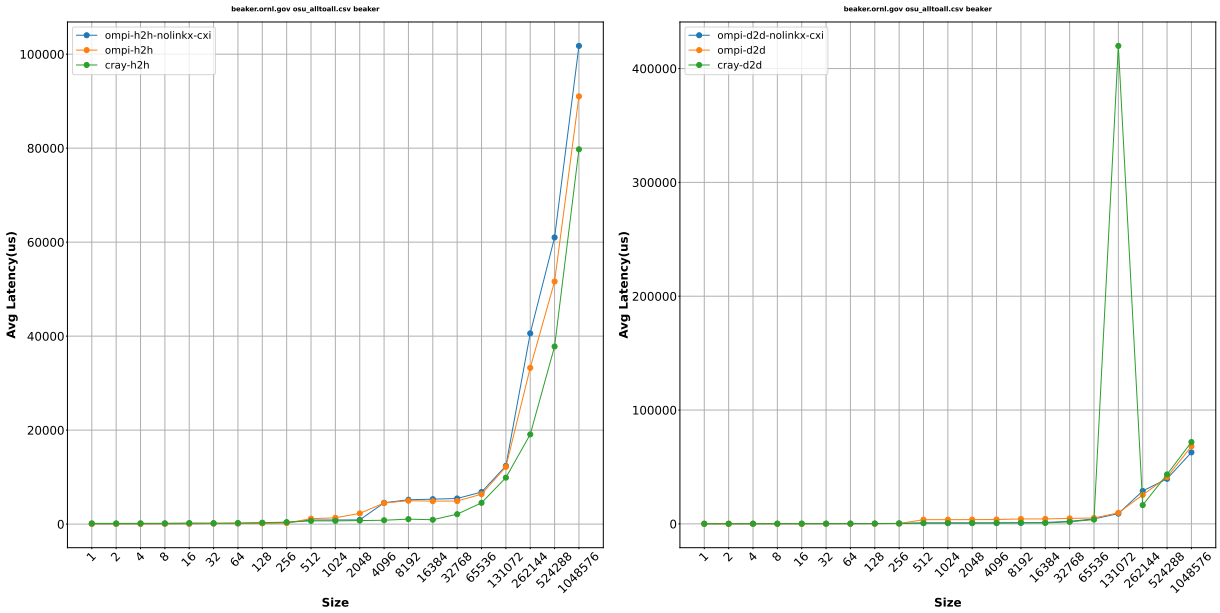
(a) H2H intra-node (b) D2D intra-node (c) H2H inter-node (d) D2D inter-node

Fig. 5: Point-to-Point Latency `osu_latency`



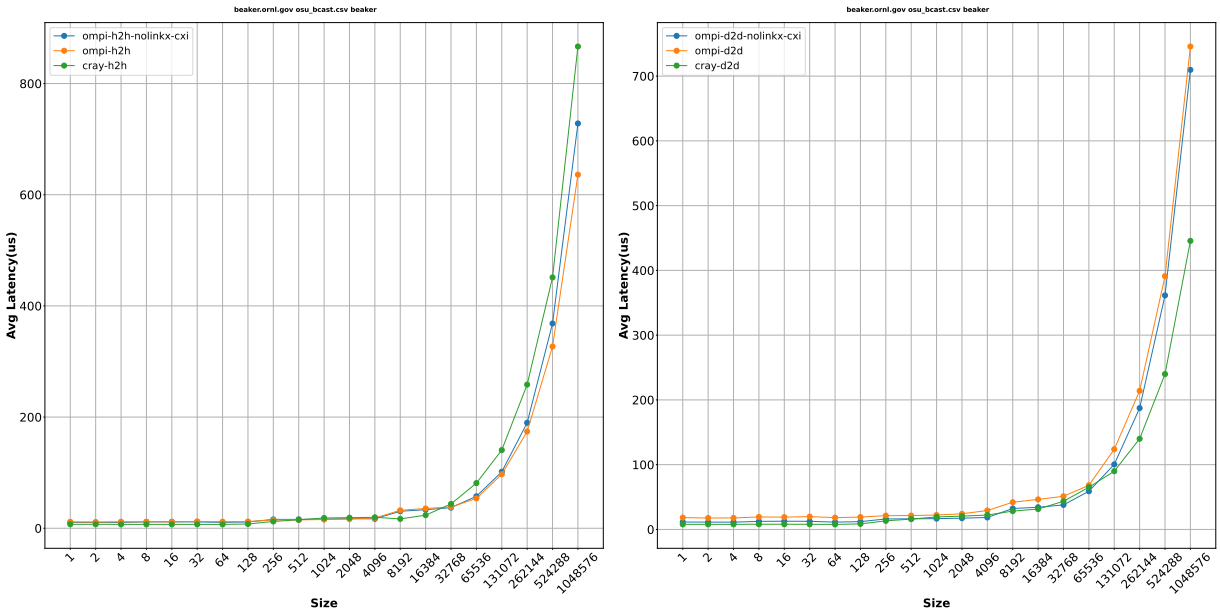
(a) H2H intra-node (b) D2D intra-node (c) H2H inter-node (d) D2D inter-node

Fig. 6: Point-to-Point Message Rate & Bandwidth `osu_mbw_mr`



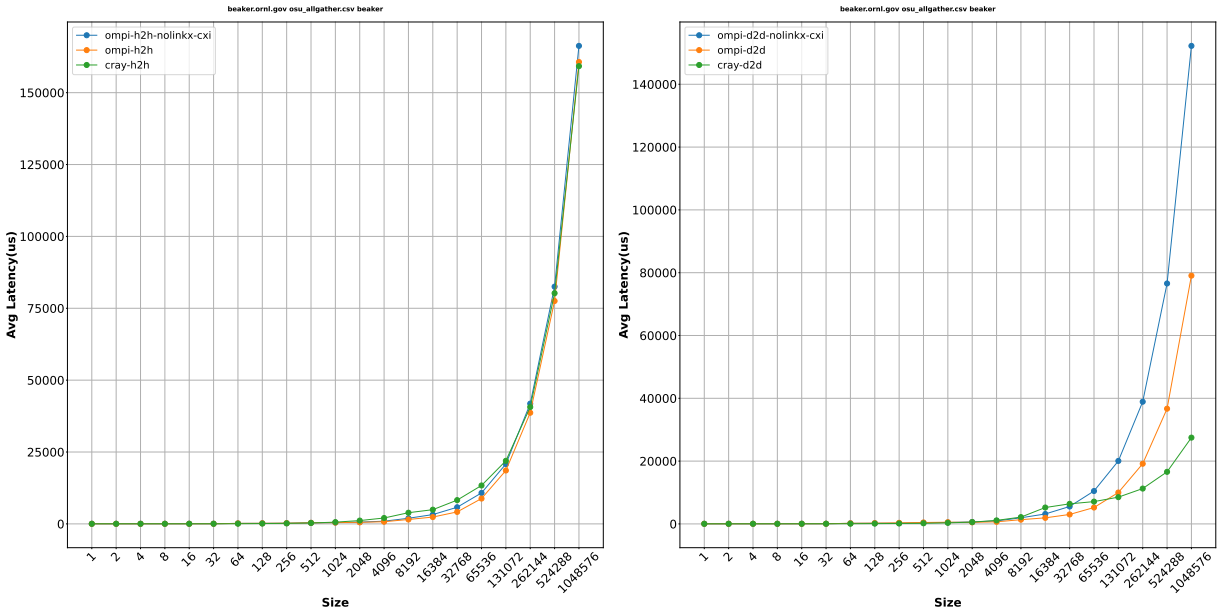
(a) H2H (b) D2D

Fig. 7: Collective Alltoall `osu_alltoall`



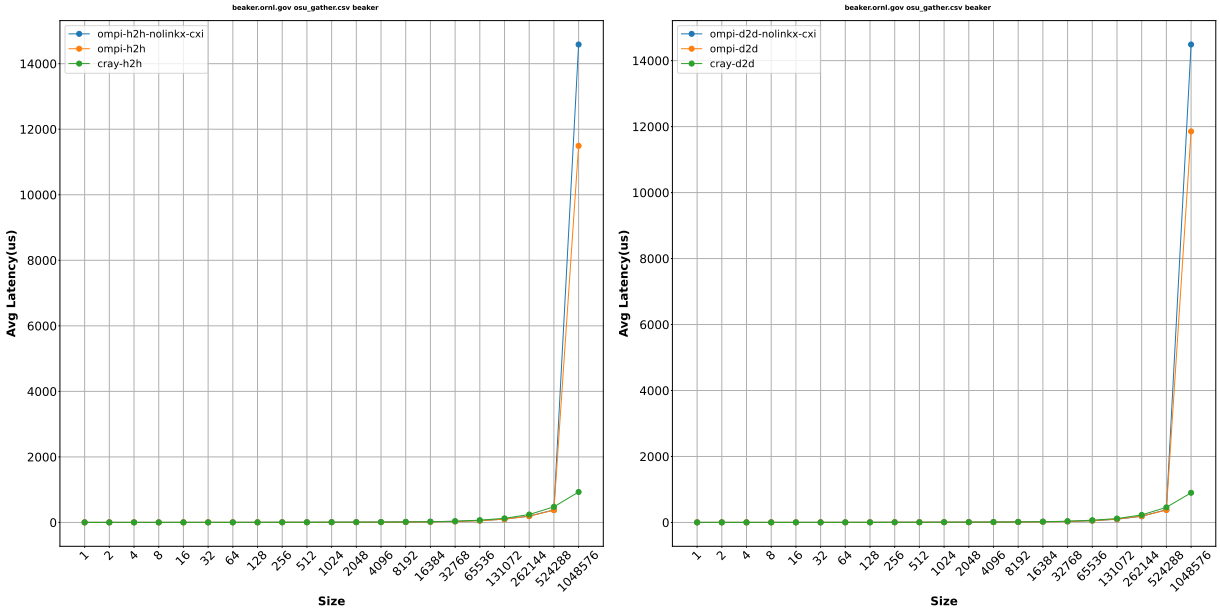
(a) H2H (b) D2D

Fig. 8: Collective Broadcast `osu_bcast`



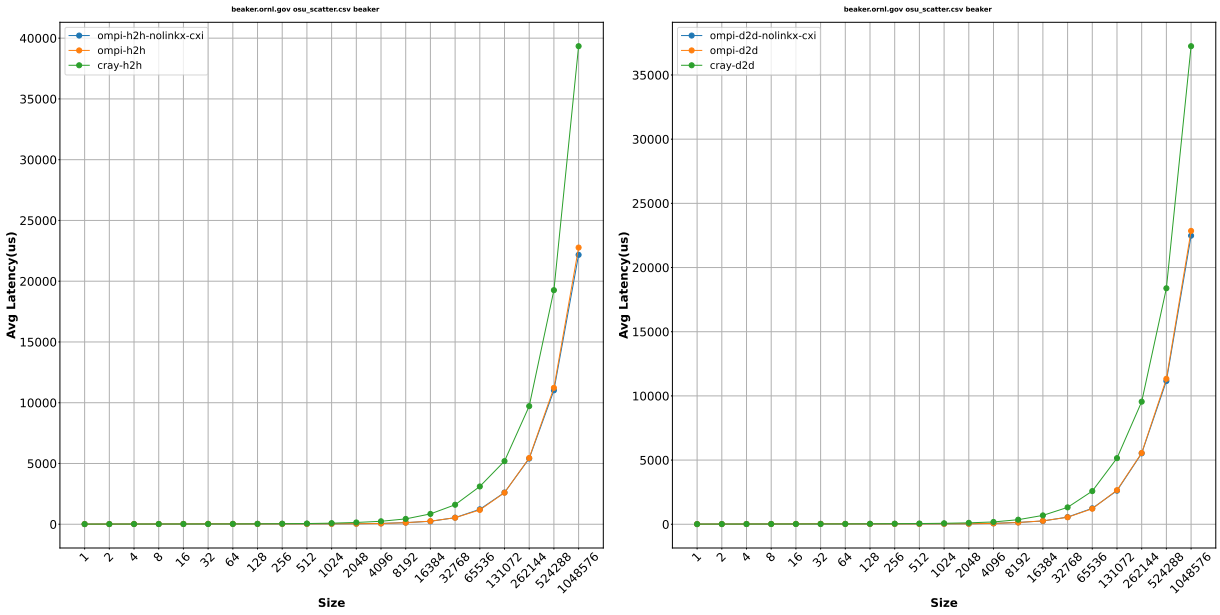
(a) H2H (b) D2D

Fig. 9: Collective Allgather `osu_allgather`



(a) H2H (b) D2D

Fig. 10: Collective Gather `osu_gather`



(a) H2H (b) D2D

Fig. 11: Collective Scatter `osu_scatter`