# The GNI Provider Layer for OFI libfabric

Howard Pritchard, Evan Harvey
*Los Alamos National Laboratory*
*Los Alamos, NM*
*Email: {howardp,eharvey}@lanl.gov*

Sung-Eun Choi, James Swaro, Zachary Tiffany
*Cray Inc.*
*St. Paul, MN*
*Email: {sungeun,jswaro,ztiffany}@cray.com*

***Abstract*—The Open Fabrics Interfaces (OFI) libfabric, a community-designed networking API, has gained increasing attention over the past two years as an API which promises both high performance and portability across a wide variety of network technologies. The code itself is being developed as Open Source Software with contributions from across government labs, industry and academia. In this paper, we present a libfabric *provider* implementation for Cray XC™ series systems using the *Generic Network Interface* (GNI) library. The provider is especially targeted for highly multi-threaded applications requiring concurrent access to the *Aries High Speed Network* with minimal contention between threads.**

***Keywords*-libfabric, networking, communication libraries, interconnects, MPI**

## I. Introduction and Motivation

Open Fabrics Interfaces (OFI) libfabric is a hardware-agnostic, user-level API for network programming. It was designed by a broad community of experts from across industry, academia, and government with the goal of providing a standardized interface for higher-level clients such as MPI and PGAS, while being portable and enabling excellent performance [1]. In fact, the design process included direct input and participation from the client community. The code itself is being developed as Open Source Software, with contributions from across a wide range of developers. In the current release of libfabric, there are five hardware-specific implementations, called *providers*, including the *Generic Network Interface* (GNI) [2] provider for the *Aries High Speed Network* [3], and two portability providers (TCP and UDP).

OFI libfabric has gained significant momentum over the past two years and is anticipated to be one of the standard APIs for future HPC systems. This gives clients a unique opportunity to port their codes to the API of future systems before they become available. While the portability providers can be used on nearly any system, the GNI provider enables clients to utilize some of the largest systems in existence today and experiment with scalability issues. Such machines include DOE's Trinity system and the NERSC Cori supercomputer. Moreover, the Aries interconnect implements a *dragonfly* topology [4], often cited as a candidate for Exascale-era systems [5] [6], making studies on these systems potentially more relevant for future systems.

In this paper, we present the GNI provider implementation of libfabric for Cray XC series systems with the Aries interconnect. Expanding on previous work [7], we describe the supported libfabric API implementations, including data transfer operations, progression model, memory registration and the tag matching engine. We also present performance results for the GNI provider using Open MPI [8], Argonne National Laboratory (ANL) MPICH and Cray MPICH.

The remainder of the paper is organized as follows. Section II, provides an overview of OFI libfabric and Section III describes the GNI provider layer. In Section IV, we present performance results for the GNI provider using libfabric directly, as well as for a variety of MPI implementations. Finally, in Section V, we conclude and describe future work.

## II. OFI libfabric

OFI libfabric is an open interface for user-level networking. It was designed by a working group of the OpenFabrics Alliance (OFA) comprised of representatives from across the industry including vendors, government laboratories and academic institutions. Members of the working group also designed a pluggable software architecture, implemented as open source software and available on GitHub [9].

The library itself is comprised of two parts: the client API and one or more fabric *providers* as well as implementations of the API for specific hardware or services. Clients such as MPI or PGAS implementations write code to the portable API and then select a specific provider at runtime. Note that providers are not required to implement the entire API or various *capabilities* of the API. The function `fi_getinfo` can be used to query the available providers and the interfaces and capabilities they implement. Some providers implement functionality that is not directly supported by the network; other providers only implement what is available natively in hardware. The GNI provider aims to support nearly all the libfabric APIs, native or not, since one of our goals is to enable clients to prepare their code for future systems.

The libfabric API is organized into four service types: control, communication, data transfer and completion. Control services aid in discovery of services available to the client on a system. Communication services are used to handle connection management and addressing. Data transfer

services export asynchronous interfaces for various communication types. Completion services are used to determine the result of the data transfer operations.

Most libfabric resources have one or more *attributes* associated with them. These attributes are used to define behavior per instantiated (in C++ parlance) resource. For example, attributes are used to define specific policies (such as message ordering requirements or progress model), set limits (such as the maximum single message size or internal buffer sizes), as well as associate other libfabric resources with the given resource (such as the wait object used for a completion queue).

The following resource types are defined in libfabric:

*fabric (`fi_fabric`):* A fabric represents a collection of hardware and software resources that access a single physical or virtual network.

*domain (`fi_domain`):* A domain typically represents a physical or virtual NIC or a hardware port, although it can be used to represent multiple such resources for fail-over or data striping. Domains are associated with a specified fabric.

*address vector (`fi_av`):* Address vectors (AVs) are an abstraction used to provide a mapping of higher level addresses, such as *rank* or *processing element* (PE), to fabric specific addresses. Address vectors are associated with a specified domain.

*endpoint (`fi_endpoint`):* Endpoints (EPs) are communication portals that can support connected and connectionless communication, both reliable and unreliable. There are three endpoint types: `FI_EP_MSG` (reliable, connection-oriented), `FI_EP_RDM` (reliable, connectionless) and `FI_EP_DGRAM` (unreliable, connectionless). Endpoints are associated with a specified domain.

*event queue (`fi_eq`):* Event queues (EQs) are used to deliver events associated with *control* operations, such as those on address vectors. Event queues are associated with a specified fabric.

*completion queue (`fi_cq`):* Completion queues (CQs) are used to deliver events that are generated by both successful and unsuccessful data transfer operations. Completion queues are associated with a specified domain.

*completion counter (`fi_cntr`):* Completion counters are a lighter-weight alternative to completion queues that simply count data transfer completion events. Counters are associated with a specified domain.

Figure 1 illustrates the relationship between the libfabric resources via a typical use case. In this example, a single fabric is being used. An event queue is associated with the fabric. One or more domains can be associated with a fabric (two are depicted in the example). Each domain can be set up differently. In this example, the first domain has two completion queues and completion counters associated with it, and the second only has a single completion queue. Finally, any number of endpoints can be associated with a domain.

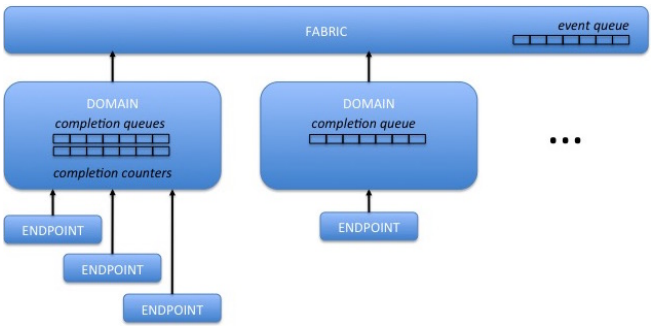


Figure 1. Example use of libfabric to illustrate the relationship between libfabric resources.

All data transfer operations are asynchronous and initiated with a specified local endpoint and remote address (obtained via an *out-of-band* method). The following data transfer operations are defined in libfabric:

*atomic operations:* Atomic transfers are used to read and update remote memory locations in an atomic manner.

*one-sided operations:* One-sided operations, or *remote memory access* (RMA) operations, are use to transfer data directly to a remote memory location, without requiring synchronization with the remote side.

*two-sided operations:* Two-sided operations, or *message* operations, implement explicit send and receive communication.

*tagged two-sided operations:* Tagged operations carry a tag with the message that is used at the receiving endpoint to match the incoming message with a corresponding receive operation.

*trigger operations:* Triggered operations defer any of the above data transfer operation until a specified condition is satisfied.

Since atomic and RMA operations can read or write directly from remote memory, all such memory must be *registered* with libfabric. Memory registration is handled via the `fi_mr` interface.

## III. The GNI provider

In this section, we describe the supported interfaces and capabilities of the GNI provider. We refer readers to previous work [7] for details on the internal mapping of libfabric resources to GNI resources.

Recall that one of the goals of the GNI provider is to give clients early access to an API available on future HPC systems. For this reason, our first release supported interfaces used primarily by existing MPI and PGAS implementations. Since then, we've continued to expand our coverage and include support for almost all the data transfer operations, supported endpoint types, and their associated capabilities.

### A. Domain Attributes

The GNI provider supports the following functional domain attributes:

*Progression models:* Manual progress (FI_PROGRESS_MANUAL) and automatic progress (FI_PROGRESS_AUTO) are supported for data transfers as well as control operations. In both cases, an additional thread is spawned to implement FI_PROGRESS_AUTO. The data progression mechanism will be discussed later in this section.

*Multi-threading support:* Full thread safety (FI_THREAD_SAFE) and *completion resource* thread safety (FI_THREAD_COMPLETION), where clients ensure thread safety on shared completion resources when using FI_PROGRESS_MANUAL, are supported.

*Resource management:* Resource management of libfabric resources to protect against overrunning of local and remote resources (FI_RM_ENABLED) is supported.

*Address Vector type:* Mapped addresses (FI_AV_MAP), where addresses are represented by a GNI device address and an identifier for a *communication domain* (a GNI abstraction for the hardware protection mechanism), and indexed addresses (FI_AV_TABLE), where addresses are represented by a simple index, are supported.

*Memory registration model:* Basic memory registration (FI_MR_BASIC), where clients register specific allocated buffers to be used in data transfer, is supported. In addition, clients are not required to register local memory regions (*i.e.*, FI_LOCAL_MR is *not* required). The memory registration cache will be discussed in detail later in this section.

### B. Endpoint Types

Currently, the GNI provider supports two of the three libfabric endpoint types: datagram (FI_EP_DGRAM) and reliable datagram (FI_EP_RDM). Not surprisingly, these two endpoint types are quite similar. Implementation-wise, the data transfer operations are exactly the same except that the internal retry mechanism is disabled for the FI_EP_DGRAM endpoint type. The data transfer operations will be discussed in more detail in the next subsection.

Most of the endpoint attributes specify various internal limits, except for the *memory tag format* attribute which enables clients to customize the use of the tag field for tagged data transfers. The tag matching engine will be discussed in detail later in this section.

Associated with each endpoint is an internal structure called a gnix_nic. This structure contains a gni_nic_handle_t handle, which in turn is associated with an Aries Fast Memory Access *(FMA) descriptor* (the Aries hardware unit used to initiate remote memory access across the network) and a handle to the associated communication domain.

### C. Data Transfer Operations

The GNI provider supports all the data transfer operations presented in the previous section, including triggered operations, for both endpoint types. The data transfer operations also support the FI_FENCE flag, which requires all previous operations targeting the same endpoint to be completed before the requested operation is initiated. All data transfer operations are initiated on a *virtual connection* (VC). VCs are obtained dynamically and maintained at the libfabric domain level. They include a handle to the GNI representation of an endpoint, gni_ep_handle_t, thus there is one VC per each remote peer that a local endpoint communicates with. VCs are also used to implement the progression model, to be discussed later in this section.

Currently, only atomic operations that are supported natively by the Aries hardware are implemented. This includes 32- and 64-bit versions of *min*, *max*, *sum*, bitwise *OR*, bitwise *AND*, bitwise *XOR*, read, write, *compare-and-swap* and masked *compare-and-swap*.

Tagged and untagged two-sided operations less than 16KB in size are sent using Aries FMA functionality as a control message payload. Larger transfers are sent via the Aries *Bulk Transfer Engine* (BTE) using a rendezvous protocol. In the rendezvous protocol, the sender first sends a short control message containing information about the source buffer. When ready, the receiver pulls the data from the sender using the Aries BTE. Once source data is moved, the receiver sends a message to the sender indicating the transfer has completed. Similarly, RMA operations shorter than 8KB are sent using Aries FMA functionality, and larger messages are transferred using the Aries BTE.

Whenever possible, data is transferred directly between the user supplied buffers. Exceptions to this include handling remote reads and atomic operations that are not four byte aligned.

### D. Progression Model

There are three types of progress managed in the GNI provider: receive (RX), transmit (TX) and deferred work (DW). A VC maintains one queue for each type of progress. The queues are independent to prevent a stall in TX processing from delaying RX processing and vice versa. Processing on any given queue stops when either the queue is empty or if an operation needs to be retried, in order to avoid livelock. Both control and data transfer operations use these VC queues.

RX progress involves servicing GNI messages and progressing the state of associated requests. If receipt of a message during RX progress requires significant additional processing, such as more network operations or acquiring certain locks, this new work is deferred and put on the DW queue. Examples of deferred work include the start of rendezvous data transfer or freeing an automatic memory registration after an RX completion.

The DW queue is processed after the RX processing, where most deferred work will be originated, and before TX

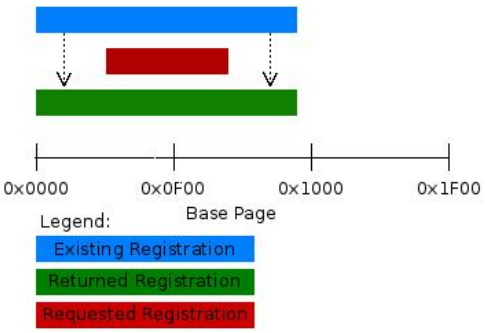


Figure 2. Aggressive registration matching: A newly requested registration that is fully within the memory region covered by an existing registration can use that existing memory registration.

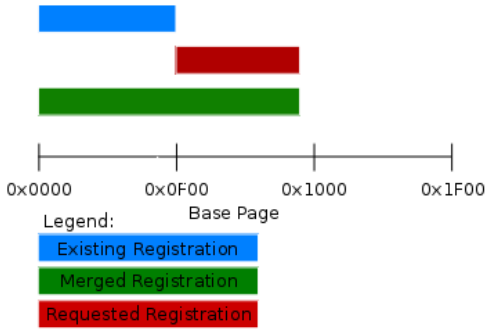


Figure 3. Registration coalescing: A new request that is adjacent to an existing memory registration is coalesced with that registration to form a new larger registration. The existing registration is then retired.

processing, giving network resource priority to TX requests which have already been initiated.

TX operations are injected into the network in the order in which they queued. The GNI provider supports the `FI_FENCE` flag by stalling the TX queue until all initiated requests have completed.

### E. Memory Registration

Memory registration for the GNI provider utilizes the *uGNI* memory registration API and an internal memory registration cache. Though the *uGNI* API provides all the functionality necessary for memory registration at a primitive level, the memory registration operation is expensive. The memory registration cache is designed to reduce redundant registrations as well as the total number of registration and deregistration calls using the *uGNI* memory registration API. It does so by combining registrations for overlapping and adjacent memory regions and employing a *lazy deregistration* scheme.

When a client requests a new memory registration, if an existing registration exists that matches the requested memory region, then there is no need to issue another registration request since an existing registration can satisfy the request. Failing that, a number of other optimizations have been implemented to reduce the number of *uGNI* memory registrations.

*Aggressive registration matching* uses the cache to search for existing registrations that fully encompass the newly requested memory region. Figure 2 illustrates an example of aggressive matching where the requested memory region is subsumed by an existing registration. In the example, the requested registration is not abutting either boundary of the existing registration and is fully satisfied by the existing registration.

If no existing memory registration satisfies the new request, we look for opportunities to combine memory registrations. *Registration coalescing* is employed when the newly requested registration is adjacent to (or would overlap with) an existing memory registration. A new, larger memory region is registered with *uGNI* and the existing (smaller) memory registration is removed from the cache and marked as *retired*. Figure 3 illustrates an example of registration coalescing. The user-requested registration is extended to the boundaries of the adjacent registration and registered with *uGNI*. The newly retired entry exists so long as the client retains a reference to the entry, *i.e.*, until `fi_close` is called with the memory registration object.

When `fi_close` is used to release a memory registration, it can be deregistered with *uGNI*. *Lazy deregistration* is a passive optimization that delays registrations from being de-allocated. Since the registrations are not deregistered immediately, these *stale* registrations may be re-used in the event that a memory registration request is received and the stale registration would satisfy the registration request. As such, lazy deregistration prevents unnecessary calls to the device to register and deregister memory.

Ultimately, application behavior determines the best policies to employ, and thus the memory registration cache is configurable. The memory registration cache supports limits on the total number of registrations and the total number of stale registrations. In addition, lazy deregistration can be disabled at runtime.

### F. Address and Tag Matching

The GNI provider uses the same framework for address and tag matching; untagged messages simply ignore the tag matching aspect. Address matching is performed for wildcard or exact matches of the addresses of inbound messages. The libfabric tags are 64-bit values with a format that is dictated by the provider and client application. Tag matching is specific to whether the message is expected (*i.e.*, posted receive) or unexpected.

Internally, tags are stored in either a simple linked list or a hash list (configurable at runtime). The linked list allows for a simple structure that is optimized for small numbers of tags. The hash list implementation uses the tag ID to hash to one of multiple bucket lists, with tags stored in order of arrival on a per-bucket basis. The best case scenario for

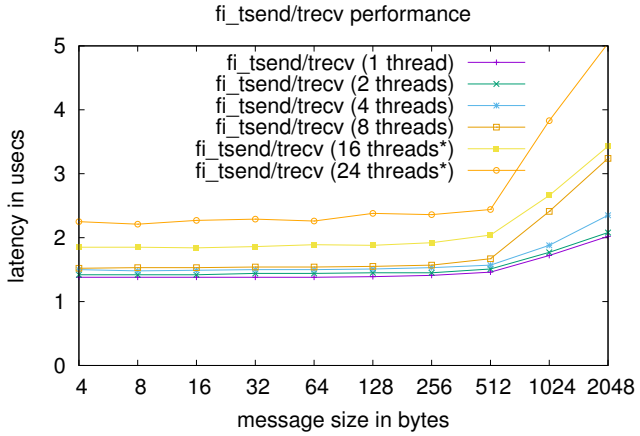


Figure 4. fi_tsend/fi_trecv ping/pong performance using multiple threads

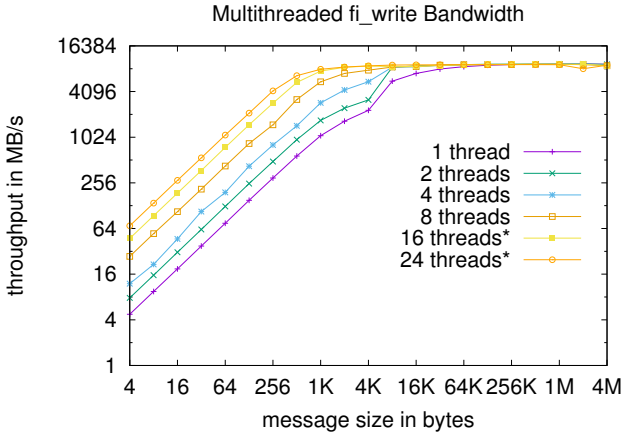


Figure 5. Multithreaded `fi_write` (onesided) bandwidth for one to 24 threads for 64 write window size (loglog scale). The 16 and 24 thread cases access the NIC from the far socket.

the hash list is tags that hash evenly across the buckets and have a single, exact match. If all tags in the tag storage are identical or simply unevenly distributed amongst the various buckets, search time with exact tags may be similar to that of the linked-list implementation. Furthermore, if a wildcard tag is used, the hash list implementation will search multiple buckets in order to find the oldest matching tag. In the worst case, all buckets must be searched, again making this equivalent to the linked list worst-case.

## IV. Experimental Results

In this section, we present results using the libfabric GNI provider, including results from low level performance tests written directly to the libfabric API, as well as results using libfabric within Open MPI and ANL MPICH.

For the experiments we used a Cray XC30 system with compute nodes comprised of two Intel Xeon IvyBridge processors (E52697 v2) with 24 cores total (12 cores/socket) without hyperthreading and 128 GB of memory running Cray Linux Environment (CLE) version 5.2up04. The job scheduler for the system was Slurm version 15.0.8.10. Release 1.3 of libfabric was used for the experiments. The library was built using gcc version 5.1.0 with optimization level O2. No special configuration options were used.

### A. libfabric Microbenchmarks

We rewrote versions of the OSU MicroBenchmarks (OMB) [10] to make direct libfabric calls to measure performance of the GNI provider without the overhead of an MPI or SHMEM implementation. We further modified the tests to optionally use multiple threads per process to exercise the GNI provider's support for multithreaded processes. All libfabric benchmarks are available at `https://github.com/ofi-cray/fabtests-cray`.

Figure 4 shows the performance of `fi_tsend` and `fi_trecv` (tagged, twosided) latency achieved by one to 24 threads. For this test, we rewrote a version of the OSU Latency Test (`osu_latency`) to use libfabric `fi_tsend` and `fi_trecv`, with each thread using its own endpoint. As we increase the number of threads, we see small increases in latency when using the socket directly attached to the Aries NIC. There is a more pronounced jump when both sockets are being fully utilized.

Figure 5 shows the onesided `fi_write` throughput achieved by one to 24 threads. For this test, we rewrote a version of the OSU Bandwidth Test (osu_bw) to use the libfabric onesided operation `fi_write` with completion events; the completion queue is checked every 64 writes (`window_size` in the original benchmark) for transfer sizes of 8K bytes or smaller. For onesided operations, there is no software involvement at the target node. As with the previous benchmark, we also create additional pthreads, each of which uses its own endpoint to communicate with a thread on the other node. Similarly, we see good scaling for small message sizes up until the 16 thread case. Beyond this, the effects of accessing the NIC from the *far* processor socket generally limits improvements to the throughput rate. Also note that beyond a message size of 8K, the underlying transfer mechanism switches to using the BTE engine, which although it enables asynchronous data transfer, does introduce a serialization point as it has a limited number of virtual channels to service the requests from the threads. Consequently, the lack of increased throughput at these larger message sizes is expected. For 16 and 24 threads, all available virtual channels of the BTE engine are saturated, and thus there is no dip in performance between 8K and 16K message sizes.

Note also that for a single thread, the FMA performance levels off around a 16K message size, which can be seen in the graph by the slight dip at the 8K message size.

### B. OSU Microbenchmarks

Basic performance characteristics of the GNI provider were investigated using the OSU MicroBenchmarks (version 5.3) using Open MPI and ANL MPICH. Both Open MPI and ANL MPICH can use libfabric as a lower level network API; *Message Transfer Layer* (MTL) in Open MPI, *Netmod* in ANL MPICH. Open MPI also has a *Byte Transfer Layer* (BTL) that uses GNI directly, allowing for comparison of implementations using the Aries network. We also compared these results to Cray MPICH.

For these experiments, we used Open MPI *master@3597a083* and ANL MPICH *master@81c1290e* utilizing the CH3 variant of the ANL MPICH libfabric (OFI) Netmod [11]. Note that ANL MPICH must be patched to work on Cray XC systems [12], but the patch is purely to enable ANL MPICH to work with Cray's Process Management Interface (PMI) implementation and has no impact on the performance of the MPICH OFI Netmod. Both `srun` and `aprun` can be used to launch the ANL MPICH with the patch applied. Cray MPICH (version 7.3.3) was used to compare with the open source MPI implementations. Cray MPICH uses a proprietary Netmod coded directly to GNI [13].

Both the Open MPI and ANL MPICH implementations make use of the `FI_EP_RDM` endpoint type and the tag matching interfaces. In the case of the Open MPI OFI MTL, the MTL is a thin layer on top of the underlying libfabric tag matching interfaces. The OFI Netmod in ANL MPICH makes use of the `MPIDI_Comm_ops_t` approach for defining `send`, `cancel`, and `probe` methods as well as the `recv_posted` method for registering an application's posted receives with the libfabric provider's tag matching component. It does not make use of the Netmod Large Message Transfer (LMT) mechanism. Both MPI implementations make use of the `fi_tinject` libfabric interface for short messages.

The GNI provider targets applications requiring thread safety, thus the OSU benchmarks were modified to request `MPI_THREAD_MULTIPLE` even though the applications were singlethreaded. Both Open MPI and ANL MPICH were compiled to be able to support this level of thread safety. In the case of Cray MPICH, the `MPICH_MAX_THREAD_SAFETY` environment variable was used to specify this level of thread safety support.

Figure 6 compares the MPI pingpong latency for Open MPI when using the GNI BTL versus going through the OFI MTL, as well as the latency achieved using the OFI Netmod in ANL MPICH and Cray MPICH. Not surprisingly, the Cray MPICH shows the best results. The OFI Netmod yields similar results, with a nearly constant overhead of about 400 nsecs over the native implementation. This can be largely accounted for by the additional overhead within the GNI provider of allocating a providerinternal request,

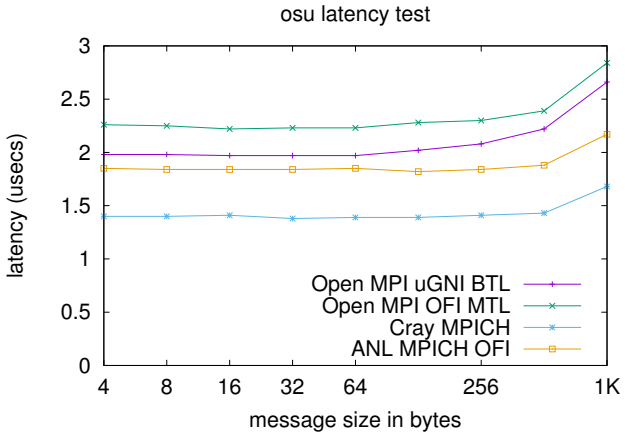


Figure 6. OSU short message latency benchmark using Open MPI, ANL MPICH and Cray MPICH.

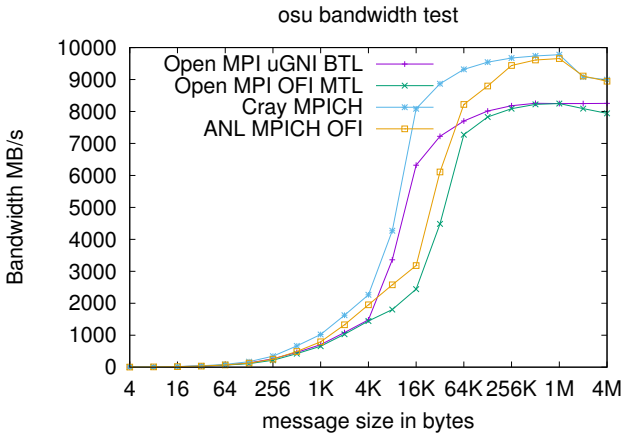


Figure 7. OSU bandwidth benchmark using Open MPI, ANL MPICH and Cray MPICH.

as well as the overhead of generating libfabric CQ entries. The finegrain locking in the GNI provider also accounts for some of the overhead. Neither of these operations is required in the Cray GNI Netmod. The overhead when using the Open MPI OFI MTL is less, around 200 nsecs for messages smaller than 1024 bytes. More of the software within Open MPI is used (tag matching, etc.) when using the native GNI path through Open MPI, whereas when using the OFI MTL, more of the software operations are occurring within the libfabric provider. Figures 7 and 8 compare, respectively, the MPI unidirectional and bidirectional bandwidths achieved for Open MPI with the GNI BTL and when going through the OFI MTL, along with a similar comparison between ANL MPICH using the OFI Netmod and Cray MPICH. The GNI provider compares well for both small and large messages for the corresponding MPI implementations, but for intermediate size messages between 8K and 64K bytes, the GNI provider under performs the native implementations

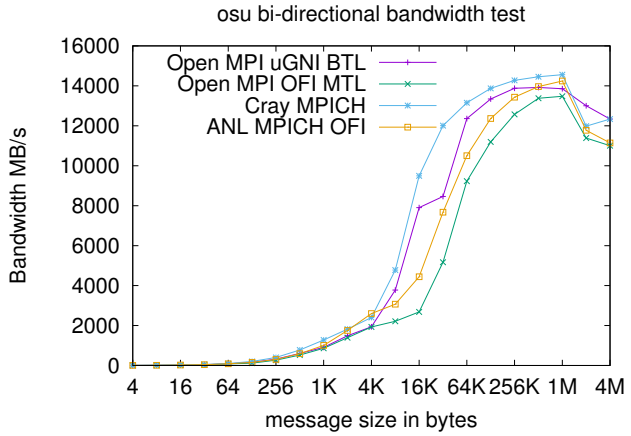


Figure 8. OSU bidirectional bandwidth benchmark using Open MPI, ANL MPICH and Cray MPICH.

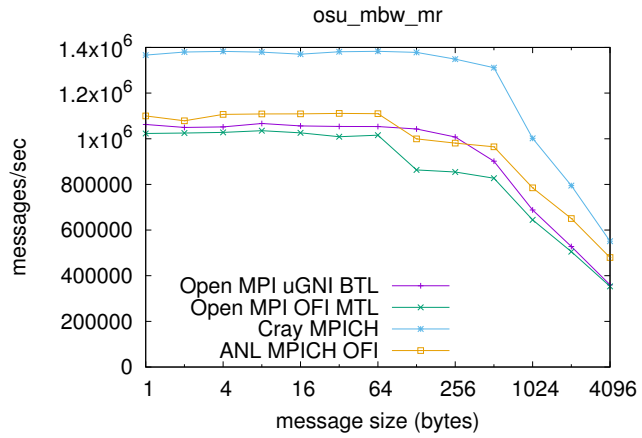


Figure 9. OSU multi-bandwidth message rate benchmark using Open MPI, ANL MPICH and Cray MPICH.

substantially. In the case of the two Open MPI results, the difference in the crossover to the rendezvous protocol (below 8K bytes when using GNI directly versus under 16K bytes for the OFI MTL) accounts for the difference at these transfer sizes for Open MPI. Analysis of the results for 32K and 64K bytes revealed that the performance difference was due to interference of the GNI provider data progress thread with the application thread. A strategy to avoid this interference is planned for the 1.4 release of libfabric. In the case of Cray MPICH, a pipelined *GET eager* protocol is used up to 128K bytes. This outperforms the *GET-based* rendezvous protocol used by the GNI provider between 16K and 128K bytes.

Figure 9 compares the short message rates when using Open MPI with the GNI BTL versus using the OFI MTL, and likewise for ANL MPICH using the OFI Netmod verses the GNI Netmod. In this experiment, two MPI processes were used. For Open MPI, the OFI BTL implementation slighly outperforms the OFI MTL implementation for messages 64 bytes and shorter. The OFI MTL uses the `fi_tinject` method for messages 64 bytes and smaller. This cuts down considerably on permessage processing on the send side as compared to the operation count when using the GNI BTL. Above 64 bytes, `fi_tsend` is used in the OFI MTL. This requires processing of CQ events on the send side, and thus more overhead. In the case of ANL MPICH, the additional overhead of allocating a GNI libfabric internal request leads to a lower message rate compared to that obtained using Cray MPICH. The ANL MPICH OFI Netmod also makes use of the `fi_tinject` so the difference in performance for 64 byte messages and smaller is not as large. Above 64 bytes, the additional overhead of the libfabric internal request allocation, as well as the need to generate libfabric CQ events both at the sender and receiver leads to a significantly lower message rate when compared to Cray MPICH. Above 1024 bytes, the time spent moving the message data begins to dominate, so the relative difference in message rate shrinks.

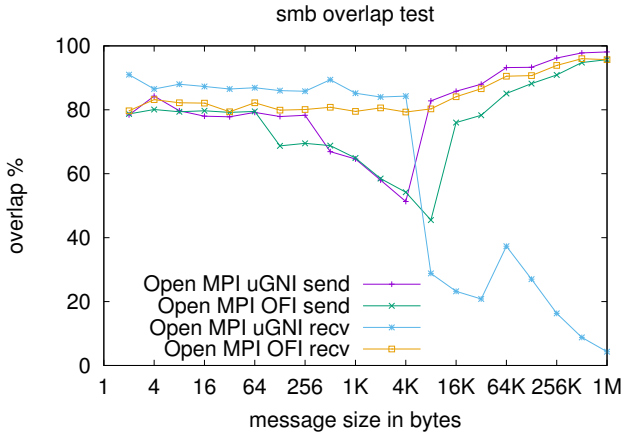


Figure 10. Sandia MPI overlap benchmark using Open MPI.

### C. Sandia Message Overhead Benchmark

The Sandia MPI message overhead benchmark [14] is a simple, but effective, benchmark for giving a first order indication of how well an MPI implementation achieves overlap of communication with computation. Because the GNI provider supports independent progression(`FI_PROGRESS_AUTO`), it is expected that for larger messages, the test should indicate good overlap of communication with computation. Figure 10 bears this out. Results for Open MPI using the GNI BTL verses the OFI MTL show that for the sender side, both methods show good overlap of communication with computation. This is expected since in both cases a receiver side GET approach is used for pulling the data from the sender's buffer into the matching receive buffer. For the receiver side however, the results show that

the GNI libfabric provider's independent progress support allows for good overlap of computation with communication for 16K bytes and longer messages.

## V. Conclusions and Future Work

In this paper, we presented the GNI provider for OFI libfabric, a new network programming API designed to be portable and deliver excellent performance. The GNI provider gives middleware clients such as MPI and PGAS libraries early exposure to the API of future systems using systems of scale, such as DOE's Trinity and Cori systems.

In addition to general performance improvements, we plan to continue implementing interfaces requested by our user community. This includes nonnative atomic operations and *scalable endpoints*, an endpoint type that multiplexes transmit and receive contexts and onnode acceleration based on `XPMEM`. We also plan to investigate additional configurability of the memory registration cache to enable better tuning for end users and a kdtree tag list implementation for clients that use highlysegmented tag formats, heavy wildcard use and deep tag queues.

## VI. Acknowledgments

The authors would like to thank Chuck Fossen, James Shimek, Tony Zinger, Ben Turrubiates and Nathan Graham for their contributions to the GNI provider. We would also like to thank the OFI working group for the design and implementation of the libfabric architecture.

**LA-UR-16-23187**

## References

[1] P. Grun, S. Hefty, S. Sur, D. Goodell, R. Russell, H. Pritchard, and J. 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.

[2] *Using the GNI and DMAPP APIs*, Cray Inc., 2011.

[3] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray Cascade: A scalable HPC system based on a Dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, November 2012.

[4] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *Proceedings of the 35th International Symposium on Computer Architecture*, June 2008.

[5] P. Kogge (Editor and Study Lead), "Exascale Computing Study: Technology Challenges in Achiveing Exascale," 2008.

[6] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR '10)*, 2010.

[7] S.-E. Choi, H. Pritchard, J. Shimek, J. Swaro, Z. Tiffany, and B. Turrubiates, "An Implementation of OFI libfabric in Support of Multithreaded PGAS Solutions," in *Proceedings of the 9th International Conference on Parititioned Global Address Space Programming Models*, September 2015.

[8] 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 of the 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[9] OpenFabrics Interfaces Working Group, "OFIWG libfabric repository," https://github.com/ofiwg/libfabric, accessed: 2016.

[10] "OSU Micro-Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks, accessed: 2016.

[11] D. Buntinas, G. Mercier, and W. Gropp, "Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem," in *CCGRID'06*, 2006, pp. 521–530.

[12] "Building and Running MPICH," https://github.com/ofi-cray/libfabric-cray/wiki/Building-and-Running-MPICH.

[13] H. Pritchard, I. Gorodetsky, and D. Buntinas, *Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ch. A uGNI-Based MPICH2 Nemesis Network Module for the Cray XE, pp. 110–119. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24449-0_14

[14] "Sandia Micro-Benchmark (SMB) Suite," www.cs.sandia.gov/smb/, accessed: 2016.