GPU-aware Communication with UCX in Parallel Programming Models: Charm++, MPI, and Python

Jaemin Choi*, Zane Fink*, Sam White*, Nitin Bhat†, David F. Richards‡, Laxmikant V. Kale*†
*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
†Charmworks, Inc., Urbana, Illinois, USA
‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California, USA
Email: {jchoi157,zanef2,white67,kale}@illinois.edu, nitin@hpccharm.com, richards12@llnl.gov

***Abstract*—As an increasing number of leadership-class systems embrace GPU accelerators in the race towards exascale, efficient communication of GPU data is becoming one of the most critical components of high-performance computing. For developers of parallel programming models, implementing support for GPU-aware communication using native APIs for GPUs such as CUDA can be a daunting task as it requires considerable effort with little guarantee of performance. In this work, we demonstrate the capability of the Unified Communication X (UCX) framework to compose a GPU-aware communication layer that serves multiple parallel programming models of the Charm++ ecosystem: Charm++, Adaptive MPI (AMPI), and Charm4py. We demonstrate the performance impact of our designs with microbenchmarks adapted from the OSU benchmark suite, obtaining improvements in latency of up to 10.2x, 11.7x, and 17.4x in Charm++, AMPI, and Charm4py, respectively. We also observe increases in bandwidth of up to 9.6x in Charm++, 10x in AMPI, and 10.5x in Charm4py. We show the potential impact of our designs on real-world applications by evaluating a proxy application for the Jacobi iterative method, improving the communication performance by up to 12.4x in Charm++, 12.8x in AMPI, and 19.7x in Charm4py.**

***Index Terms*—GPU communication, UCX, Charm++, AMPI, CUDA-aware MPI, Python, Charm4py**

I. Introduction

The parallel processing power of GPUs have become central to the performance of today's High Performance Computing (HPC) systems, with seven of the top ten supercomputers in the world equipped with GPUs [1]. GPU-accelerated applications often store the bulk of their data in device memory, increasing the importance of efficient inter-GPU data transfers on modern systems.

Although vendors provide GPU programming models such as CUDA for executing kernels and transferring data, their limited functionality makes it challenging to implement a general communication backend for parallel programming models on distributed-memory machines. Direct GPU-GPU communication crossing the process boundary can be implemented using CUDA Inter-process Communication (IPC), but requires extensive optimization such as IPC handle cache and pre-allocated device buffers [2]. Direct inter-node transfers of GPU data cannot be implemented solely with CUDA and requires additional hardware and software support [3]. Adding support for GPUs from other vendors such as AMD or Intel requires another round of development and optimization efforts that could have been spent elsewhere.

There have been a number of software frameworks aimed at providing a unified communication layer over the various types of networking hardware, such as GASNet [4], libfabric [5], and UCX [6]. While they have been successfully adopted in many parallel programming models including MPI and PGAS for communication involving host memory, UCX is arguably the first framework to support production-grade, high-performance inter-GPU communication on a wide range of modern GPUs and interconnects. In this work, we utilize the capability of UCX to perform direct GPU-GPU transfers to support GPU-aware communication in multiple parallel programming models from the Charm++ ecosystem including MPI and Python: Charm++, Adaptive MPI (AMPI), and Charm4py. We extend the UCX machine layer in the Charm++ runtime system to enable the transfer of GPU buffers and expose this functionality to the parallel programming models, with model-specific implementations to support their user applications. Our tests on a leadership-class system show that this approach substantially improves the performance of GPU communication for all models.

The major contributions of this work are the following:

- We present designs and implementation details to enable GPU-aware communication using UCX as a common abstraction layer in multiple parallel programming models: Charm++, AMPI, and Charm4py.
- We discuss design considerations to support message-driven execution and task-based runtime systems by performing a metadata exchange between communication endpoints.
- We demonstrate the performance impact of our mechanisms using a set of microbenchmarks and a proxy application representative of a scientific workload.

II. Background

A. GPU-aware Communication

GPU-aware communication has developed out of the need to rectify productivity and performance issues with data transfers involving GPU buffers. Without GPU-awareness, additional code is required to explicitly move data between host and device memory, which also substantially increases latency and reduces attainable bandwidth.

The GPUDirect [7] family of technologies have been leading the effort to resolve such issues on NVIDIA GPUs. Ver-

978-1-6654-3577-2/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPSW52791.2021.00079

sion 1.0 allows Network Interface Controllers (NICs) to have shared access to pinned system memory with the GPU and avoid unnecessary memory copies, and version 2.0 (GPUDirect P2P) enables direct memory access and data transfers between GPU devices on the same PCIe bus. GPUDirect RDMA [8] utilizes Remote Direct Memory Access (RDMA) technology to allow the NIC to directly access memory on the GPU. Based on GPUDirect RDMA, the GDRCopy library [9] provides an efficient low-latency transport for small messages. The Inter-Process Communication (IPC) feature introduced in CUDA 4.1 enables direct transfers between GPU data mapped to different processes, improving the performance of communication crossing the process boundary [2].

MPI is one of the first parallel programming models and communication standards to adopt these technologies and support GPUs in the form of CUDA-aware MPI, which is available in most MPI implementations. Other parallel programming models have either built direct GPU-GPU communication mechanisms natively using GPUDirect and CUDA IPC, or adopted a GPU-aware communication framework.

B. UCX

Unified Communication X (UCX) [6] is an open-source, high-performance communication framework that provides abstractions over various networking hardware and drivers, including TCP, OpenFabrics Alliance (OFA) verbs, Intel Omni-Path, and Cray uGNI. It is currently being developed at a fast pace with contributions from multiple hardware vendors as well as the open-source community.

With support for tag-matched send/receive, stream-oriented send/receive, Remote Memory Access (RMA), and remote atomic operations, UCX provides a high-level API for parallel programming models to implement a performance-portable communication layer. Projects using UCX include Dask, OpenMPI, MPICH, and Charm++. GPU-aware communication is supported on NVIDIA and AMD GPUs through its tagged and stream APIs. When provided with pointers to GPU memory, these APIs utilize the respective CUDA or ROCm libraries to perform efficient GPU-GPU transfers.

C. Charm++

Charm++ [10] is a parallel programming system based on the C++ language, developed around the concept of migratable objects. A Charm++ program is decomposed into objects called *chares* that execute in parallel on the Processing Elements (PEs, typically CPU cores), which are scheduled by the runtime system. This object-centric approach enables *overdecomposition*, where the problem domain is decomposed into a larger number of chares than the number of available PEs. Overdecomposition empowers the runtime system to control the mapping and scheduling of chares onto PEs, facilitating computation-communication overlap and dynamic load balancing.

The execution of a Charm++ program is driven by messages exchanged between chare objects. Each message encapsulates information about the work to be performed on the receiver chare (i.e., *entry method* in Charm++) and relevant data. Incoming messages are stored in a message queue associated with each PE, which are picked up by the scheduler. Communication in Charm++ is asynchronous as the sender does not wait for any reply or acknowledgement from the receiver, and messages are asynchronously received in the message queue. Communication operations initiated by chare objects pass through various layers in the Charm++ runtime system until they eventually reach the *machine layer*. Charm++ supports various low-level transports with different machine layer implementations, including TCP/IP, Mellanox Infiniband, Cray uGNI, IBM PAMI, and UCX.

Charm++ has support for GPU-GPU transfers implemented using CUDA memory copies and IPC, but it is limited to a single node and has inadequate performance. This work enables GPU-aware communication seamlessly within and across nodes using UCX, improving the performance of GPU-accelerated applications developed with any of the parallel programming models in the Charm+ ecosystem.

D. Adaptive MPI

Adaptive MPI (AMPI) [11] is an MPI library implementation developed on top of the Charm++ runtime system. AMPI virtualizes the concept of an MPI rank: whereas a traditional MPI library equates ranks with operating system processes, AMPI supports execution with multiple ranks per process. This empowers AMPI to co-schedule ranks that are located on the same PE based on the delivery of messages. Users can tune the number of ranks they run with based on performance. AMPI ranks are also migratable at runtime for the purposes of dynamic load balancing or checkpoint/restart-based fault tolerance.

Communication in AMPI is handled through Charm++ and its optimized networking layers. Each AMPI rank is associated with a chare object. AMPI optimizes communication based on locality of the recipient rank as well as the size and datatype of the message buffer. Small buffers are packed inside a regular Charm++ message in an eager fashion, and the Zero Copy API [12] is used to implement a rendezvous protocol for larger buffers. The underlying runtime optimizes message transmission based on locality over user-space shared memory, Cross Memory Attach (CMA) for within-node, or RDMA across nodes. This work extends such optimizations to the context of multi-GPU nodes connected by a high performance network programmable with the UCX API.

E. Charm4Py

Charm4Py [13] is a parallel programming framework based on the Python language, developed on top of the Charm++ runtime system. It seeks to provide an easily-accessible parallel programming environment with improved programmer productivity through Python, while maintaining high scalability and performance of the adaptive C++-based runtime. Being based on Python, Charm4py can readily take advantage of many widely-used software libraries such as NumPy, SciPy, and pandas.

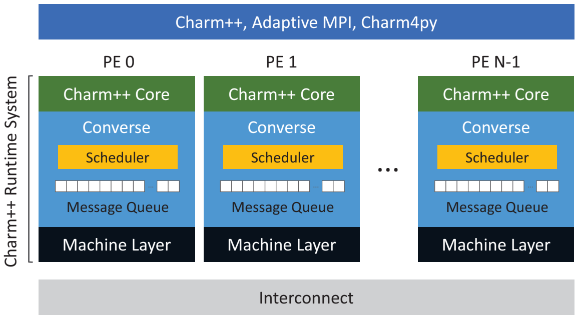


Fig. 1. Software stack of the Charm++ family of parallel programming models.

```
// Sender object's method
void Sender::foo() {
  // Send a message to the receiver object
  // to execute the 'bar' entry method
  receiver.bar(my_val1, my_val2);
}

// Receiver object's entry method,
// executed once the sender's message
// is picked up by the scheduler
void Receiver::bar(int val1, double val2) {
  // val1 and val2 are available
  ...
}
```

Fig. 2. Message-driven execution in Charm++.

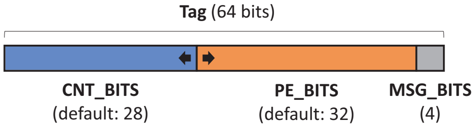


Fig. 3. Tag generation for GPU communication in UCX machine layer.

Chare objects in Charm4py communicate with each other by asynchronously invoking entry methods as in Charm++. The parameters are serialized and packed into a message that is handled by the underlying Charm++ runtime system. This allows our extension of the UCX machine layer to also support GPU-aware communication in Charm4py.

Aside from the Charm++-like communication through entry method invocations, Charm4py also provides a functionality to establish streamed connections between chares, called *channels* [14]. Channels provide explicit send/receive semantics to exchange messages, but retains asynchrony by suspending the caller object until the respective communication is complete. We extend the channels feature to support GPU-aware communication in Charm4py, which is discussed in detail in Section III-D.

III. Design and Implementation

To accelerate communication of GPU-resident data, we utilize the capability of UCX to directly send and receive GPU data through its tagged APIs. UCX is supported as a machine layer in Charm++, positioned at the lowest level of the software stack directly interfacing the interconnect, as illustrated in Figure 1. As AMPI and Charm4py are built on top of the Charm++ runtime system, all host-side communication travels through the Charm++ core and Converse layers where layer-specific headers are added or extracted, with actual communication primitives executed by the machine layer.

The main idea of enabling GPU-aware communication in the Charm++ family of parallel programming models is to retain this route to send metadata and host-side data, while separately supplying GPU data to the UCX machine layer. The metadata is necessitated by the message-driven execution model in Charm++, as shown in Figure 2. The sender object provides the data it wants to send to the entry method invocation, but the receiver does not post an explicit receive function. Instead, the sender's message arrives in the message queue of the PE that currently owns the receiver object. When the message is picked up by the scheduler, the receiver object and target entry method are resolved using the metadata contained in the message. Any host-resident data destined for the receiving chare is unpacked from the message and delivered to the receiver's entry method.

With our GPU-aware communication scheme, the sender object's GPU buffers are not included as part of the message. Only metadata containing information about the GPU data transfer initiated by the sender and sender's data on host memory are contained in the message. Source GPU buffers are directly provided to the UCX machine layer to be sent, and a receive for the incoming GPU data is posted once the host-side message arrives on the receiver. A noticeable limitation of this approach is the delay in posting the receive caused by the need to wait for the host-side message containing the metadata. We are currently working on an improved mechanism where explicit receives can be posted in advance. Note that while the UCX machine layer provides the fundamental capability to transfer buffers directly between GPUs, additional implementations to each of the parallel programming models are required as described in the following sections.

A. UCX Machine Layer

Originally contributed by Mellanox, the UCX machine layer in Charm++ is designed to handle low-level communication using the UCP tagged API, providing a portable implementation over all the networking hardware supported by UCX. To support GPU-aware communication, we extend the UCX machine layer to provide an interface for sending and receiving GPU data with the UCP tagged API. We adopt a tag generation scheme specific to GPU-GPU transfers to separate this path from the existing host-side messaging, as shown in Figure 3. The first four bits (MSG_BITS) of the 64-bit tag are used to differentiate the message type, where the new UCX_MSG_TAG_DEVICE type is added for inter-GPU communication. The remainder of the tag is split into the source PE index (PE_BITS, 32 by default) and the value of

```
// Charm++ Interface (CI) file
// Exposes chare objects and entry methods
chare MyChare {
  entry MyChare();
  entry void recv(nocopydevice char data[size],
                  size_t size);
};
```

```
// C++ source file
// (1) Sender chare
void MyChare::send() {
  peer.recv(CkDeviceBuffer(send_gpu_data), size);
}

// (2) Receiver's post entry method
void MyChare::recv(char*& data, size_t& size) {
  // Set the destination GPU buffer
  // Receive size is optional
  data = recv_gpu_data;
}

// (3) Receiver's regular entry method
void MyChare::recv(char* data, size_t size) {
  // Receive complete, GPU data is available
  ...
}
```

Fig. 4. GPU-aware communication interface in Charm++.

```
// Converse layer metadata
struct CmiDeviceBuffer {
  const void* ptr; // Source GPU buffer address
  size_t size;
  uint64_t tag; // Set in the UCX machine layer
  ...
};

// Charm++ core layer metadata
struct CkDeviceBuffer : CmiDeviceBuffer {
  CkCallback cb; // Support Charm++ callbacks
  ...
};
```

Fig. 5. Metadata object used for GPU communication in Charm++.

a counter maintained by the source PE (CNT_BITS, 28 by default). This division can be modified by the user to allocate more bits to one side or the other to accommodate different scaling configurations.

The core functionalities of GPU-aware communication in the UCX machine layer are exposed as the following functions:

```
void LrtsSendDevice(int dest_pe, const void*& ptr,
                    size_t size, uint64_t& tag);
void LrtsRecvDevice(DeviceRdmaOp* op,
                    DeviceRecvType type);
```

LrtsSendDevice provides the functionality to send GPU data using the information provided by the calling layer including the destination PE, address of the source GPU buffer, size of the data, and a reference to the 64-bit tag to be set. The tag is generated within this function by incrementing the tag counter of the source PE, and included as metadata by the caller to be sent along with any host-side data. Once the destination UCP endpoint is determined, the source GPU buffer is sent separately with ucp_tag_send_nb using the generated tag.

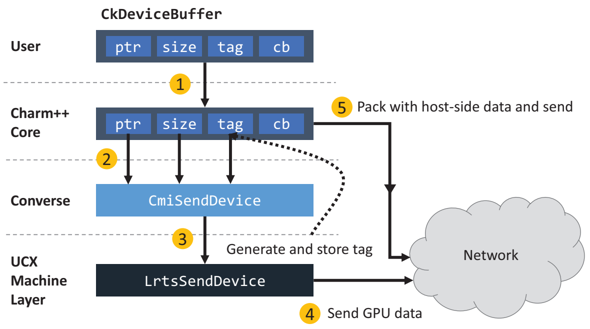


Fig. 6. Sender-side logic of GPU-aware communication in Charm++.

Once the metadata arrives on the destination PE, the corresponding receive for the incoming GPU data is posted with LrtsRecvDevice. The DeviceRdmaOp struct passed by the calling layer contains metadata necessary to post the receive with ucp_tag_recv_nb, such as the address of the destination GPU buffer, size of the data, and the tag set by the sender. DeviceRecvType denotes which parallel programming model has posted the receive, so that the appropriate handler function can be invoked once the GPU data has been received. The following sections describe in detail how the different parallel programming models build on the UCX machine layer to perform GPU-aware communication.

B. *Charm++*

Communication in Charm++ occurs between chare objects that may be scheduled on different PEs. It should be noted that multiple parameters can be passed to a single entry method invocation, as in Figure 2. We provide an additional attribute in the Charm++ Interface (CI) file, nocopydevice, to annotate parameters on GPU memory. Figure 4 illustrates this extension as well as the usage of a CkDeviceBuffer object, which wraps the address of a source GPU buffer and is used by the runtime system to store metadata regarding the GPU-GPU transfer. The structure of CkDeviceBuffer is presented in Figure 5.

1) Send: An entry method invocation such as peer.recv() in Figure 4 executes a generated code block that prepares a message containing data on host memory and sends it to the receiver object. We modify the code generation to send GPU buffers in tandem, using the CkDeviceBuffer objects provided by the user (one per buffer). These objects hold information necessary for the UCX machine layer to send the GPU buffers with LrtsSendDevice. The tags set by the machine layer are stored in the CkDeviceBuffer objects, which are packed with host-side data as well as other metadata needed by the Converse and Charm++ core layers. This packed message is sent separately, also using the UCX machine layer. Figure 6 illustrates this process.

2) Receive: To receive the incoming GPU data directly into the user's destination buffers and avoid extra copies, we provide a mechanism for the user to specify the addresses

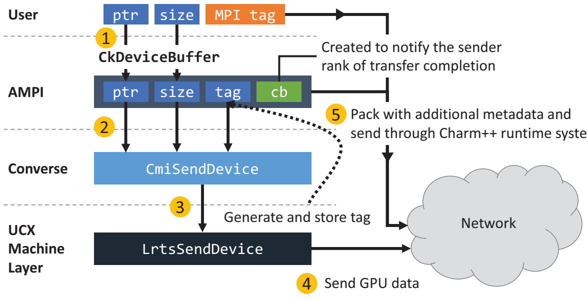


Fig. 7. Sender-side logic of GPU-aware communication in AMPI.

of the destination GPU buffers by extending the Zero Copy API [12] in Charm++. The user can provide this information to the runtime system in the *post entry method* of the receiver object, which is executed by the runtime system before the actual target entry method, i.e., *regular entry method*. As can be seen in Figure 4, the post entry method has a similar function signature as the regular entry method, with parameters passed as references so that they can be set by the user.

When the message containing host-side data and metadata (including CkDeviceBuffer objects) arrives, the post entry method of the receiver chare is first executed. Using information about destination GPU buffers provided by the user in the post entry method and source GPU buffers in the CkDeviceBuffer objects, the receiver instructs the UCX machine layer to post receives for the incoming GPU data with LrtsRecvDevice. Once all the GPU buffers have arrived, the regular entry method is invoked, completing the communication.

C. Adaptive MPI

Each AMPI rank is implemented as a chare object on top of the Charm++ runtime system, to enable virtualization and adaptive runtime features such as load balancing. Communication between AMPI ranks occurs through an exchange of AMPI messages between the respective chare objects. An AMPI message adds AMPI-specific data such as the MPI communicator and user-provided tag to a Charm++ message, and we modify how it is created to support GPU-aware communication with the CkDeviceBuffer metadata object. This change is transparent to the user, and GPU buffers can be directly provided to AMPI communication primitives such as MPI_Send and MPI_Recv like any CUDA-aware MPI implementation.

1) Send: The user application can send GPU data by invoking a MPI send call with parameters including the address of the source buffer, number of elements and their datatype, destination rank, tag, and MPI communicator. The chare object that manages the destination rank is first determined, and the source buffer's address is checked to see if it is located on GPU memory. A software cache containing addresses known to be on the GPU is maintained on each PE to optimize this process. Figure 7 illustrates the mechanism that is executed when the source buffer is found to be on the GPU, where a CkDeviceBuffer object is first created in the AMPI runtime to store the information provided by the user. A Charm++ callback object is also created and stored as metadata, which is used by AMPI to notify the sender rank when the communication is complete. The source GPU buffer is sent in an identical manner as Charm++ through the UCX machine layer with LrtsSendDevice. The tag that is needed by the receiver rank to post a receive for the incoming GPU data is also generated and stored inside the CkDeviceBuffer object. Note that this tag is separate from the MPI tag provided by the user, which is used to match the host-side send and receive.

```
if not gpu_direct:
  # Host-staging mechanism (not GPU-aware)
  # Transfer GPU buffer to host memory and send
  charm.lib.CudaDtoH(h_send_data, d_send_data, size,
                     stream)
  charm.lib.CudaStreamSynchronize(stream)
  channel.send(h_send_data)

  # Receive and transfer to GPU buffer
  h_recv_data = partner_channel.recv()
  charm.lib.CudaHtoD(d_recv_data, h_recv_data, size,
                     stream)
  charm.lib.CudaStreamSynchronize(stream)
else:
  # GPU-aware communication
  # Send and receive using GPU buffers directly
  channel.send(d_send_data, size)
  channel.recv(d_recv_data, size)
```

Fig. 8. Channel-based communication in Charm4py. CUDA functions are included in the Charm++ library as C++ functions and exposed through Charm4py's Cython layer.

2) Receive: Because there are explicit receive calls in the MPI model in contrast to Charm++, there are two possible scenarios regarding the host-side message that contains metadata: the message arrives before the receive is posted, and vice versa. If the message arrives first, it is stored in an unexpected message queue, which is searched for a match when the receive is posted later. If the receive is posted first, it is stored in a request queue to be matched when the message arrives. The receive for the incoming GPU data is posted after this match of the host-side message, with LrtsRecvDevice in the UCX machine layer. Another Charm++ callback is created for the purpose of notifying the destination rank, which is invoked by the machine layer when the GPU data arrives.

D. Charm4py

GPU-aware communication in Charm4py is built around the Channel API, which provides functionality for the user to provide the address of the destination GPU buffer. While the API itself is in Python, its core functionalities are implemented with Cython [15] and the underlying Charm++ runtime system is comprised of C++. Cython generates C extension modules to support C constructs and types to be used with Python for interoperability and performance, and is used extensively in the Charm4py runtime. The Cython layer is also used to interface with the Charm++ runtime, which performs the bulk of the

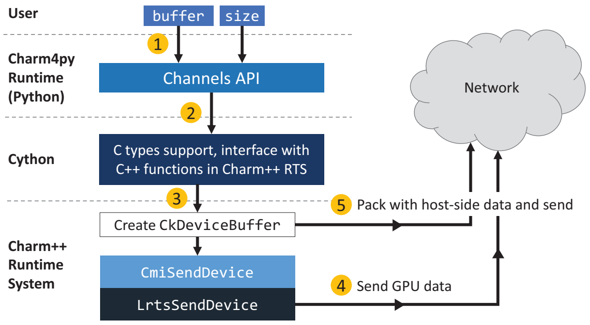


Fig. 9. Sender-side logic of GPU-aware communication in Charm4py.

work for GPU-aware communication with the UCX machine layer. Note that the Python interface for UCX, UCX-Py [16], is not used in this work as Charm4py can directly utilize the UCX functionalities in C/C++ through the Charm++ runtime system.

Figure 8 compares our GPU-aware communication support against the host-staging mechanism in a ping-pong exchange of GPU data. The two chares involved establish a channel, through where data on host or GPU memory is exchanged depending on `gpu_direct` flag. The host-staging version needs to explicitly move data between host and device memory using the CUDA API, adding complexity to the programmer and degrading performance. Note that the channel send and receive calls are asynchronous; the coroutine posting the receive is suspended until the message arrives. Such asynchronous communication is implemented with futures [17], a key component of Charm4py.

1) Send: As can be seen from Figure 8, addresses of the source and destination GPU buffers can be directly provided to Charm4py's channel API. The address and size of the buffer are propagated to the Charm++ runtime system through the Cython layer, which are used to construct the `CkDeviceBuffer` metadata object. The steps after this point are similar to Charm++ and AMPI, where the metadata is used by the UCX machine layer to send the source GPU buffer, and the metadata itself is packed together with the host-side data and Charm4py-specific information to be sent separately to the receiver object. This process is illustrated in Figure 9.

2) Receive: When the host-side message containing metadata about the GPU-GPU transfer arrives, it is used to post the receives for the incoming GPU data in the UCX machine layer. A Charm++ callback is created and tied to the `LrtsRecvDevice` function, so that it can be invoked when the GPU-GPU transfer is complete. This callback invocation fulfills the future that has suspended the channel receive call, allowing the user application (coroutine) to continue.

IV. Performance Evaluation

In this section, we describe the hardware platform and software configurations, as well as the set of micro-benchmarks and proxy application used to evaluate the performance of our GPU-aware communication designs.

A. Experimental Setup

The Summit supercomputer at Oak Ridge National Laboratory is used to evaluate the performance of GPU-aware communication mechanisms implemented in Charm++, AMPI and Charm4py. The experiments are scaled up to 256 nodes of Summit, where each IBM AC922 node contains two IBM Power9 CPUs and six NVIDIA Tesla V100 GPUs. Each CPU is connected to three GPUs, which are interconnected via NVLink with a theoretical peak bandwidth of 50 GB/s. For a GPU to communicate with another GPU connected to the other CPU, data needs to travel through the X-Bus that connects the CPUs with a bandwidth of 64 GB/s. The network interconnect is based on Mellanox Enhanced Data Rate (EDR) Infiniband, providing up to 12.5 GB/s of bandwidth.

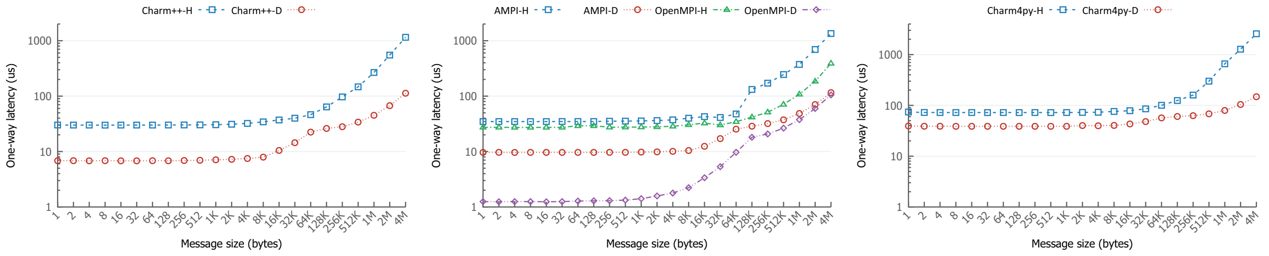
Charm++, AMPI and Charm4py are configured to use the non-SMP build, using one CPU core as the single PE for each process and one process per GPU device. On a single node of Summit, for example, six PEs (and processes) execute in parallel using all six available GPUs. To accurately evaluate the impact of GPU-awareness on communication performance by separating communication from computation, the problem domain is decomposed into the same number of chare objects as the number of PEs and GPUs (no overdecomposition in Charm++/Charm4py, no virtualization in AMPI).

For reference, the performance of OpenMPI is provided along with the AMPI results, which also maps one process to each GPU. Since both AMPI and OpenMPI utilize UCX to transfer GPU data, this comparison isolates the performance differential incurred by the layers above UCX. Note that AMPI delivers messages through the Charm++ runtime system for its adaptive runtime features, in contrast to OpenMPI which can directly utilize UCX for communication.

B. Micro-benchmarks

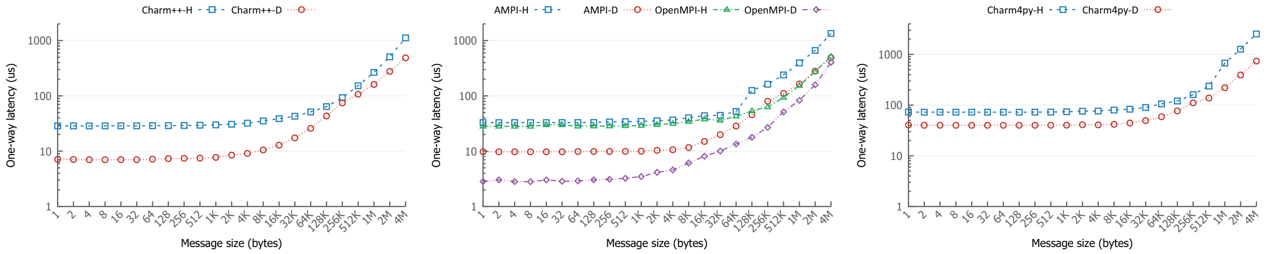
To evaluate the performance of point-to-point communication primitives involving GPU memory, we adapt the widely used OSU micro-benchmark suite [18] to Charm++ and Charm4py. We also add an option to use the host-staging mechanism, which stages the GPU buffer on host memory before performing communication, to measure the performance impact of our implementations to enable GPU-aware communication. This option is added to the original MPI versions of the benchmarks as well for AMPI and OpenMPI. Performance results are presented with both axes in log-scale, comparing the GPU-aware version of the benchmark (suffixed with D) against the host-staging version (suffixed with H).

1) Latency: The OSU latency benchmark repeats ping-pong iterations for different message sizes, where the sender sends a message to the receiver and waits for a reply. Once the message arrives, the receiver sends a message with the same size back to the sender, completing one iteration. GPU-aware communication allows the message buffers to be supplied directly to the communication primitives, whereas the host-staging version requires additional data transfers between host and device.



(a) Charm++ (b) AMPI and OpenMPI (c) Charm4py

Fig. 10. Comparison of intra-node latency between host-staging and direct GPU-GPU mechanisms.



(a) Charm++ (b) AMPI and OpenMPI (c) Charm4py

Fig. 11. Comparison of inter-node latency between host-staging and direct GPU-GPU mechanisms.

Figures 10 and 11 illustrate the improvements in intra-node and inter-node latency with GPU-awareness in Charm++, AMPI and Charm4py. The range of performance improvements in the latency benchmark is summarized in Table I, where the achieved speedups with small messages using the eager protocol are denoted in a separate row. The observed improvement in latency increases with message size with large messages in all three programming models, as the host-staging mechanism suffers significant slowdowns caused by host memory copies in the Charm++ runtime system.

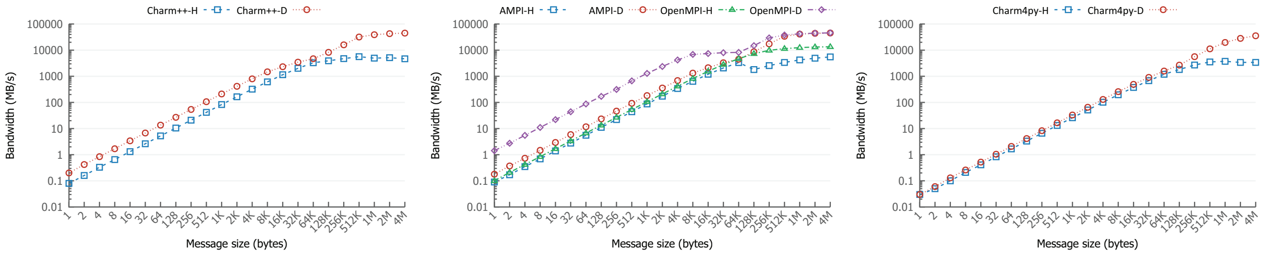
Although the performance of AMPI improves substantially with GPU-aware communication, it does not quite match the latency of CUDA-aware OpenMPI. To further investigate this issue, we isolate the time taken in UCX by taking advantage of the modular property of the UCX machine layer. We can easily disable the `CmiSend/RecvDevice` calls in the Converse layer and invoke receive handlers directly, allowing us to determine the time taken outside of UCX. This turns out to be about 8 μs, which tells us that the GPU-GPU transfer itself with UCX has a latency of less than 2 μs, similar to OpenMPI. Thus most of the overhead is AMPI-specific, which has multiple factors: message packing and unpacking, additional host-side message that contains metadata, Charm++ callback invocations, and the fact that the receiver rank cannot post a receive until the metadata message is received. There are also a couple of heap memory allocations that are used to retain metadata for the UCX machine layer. We plan to further analyze and optimize the code to get AMPI's performance as close to OpenMPI as possible.

It should be noted that the detection of the GDRCopy library by UCX is essential in order to achieve low latencies with small messages, which is not included in the default library search path on Summit. With the rendezvous protocol, UCX switches to the CUDA IPC transport for intra-node transfers, and to the pipelined host-staging mechanism that stages GPU data on host memory in chunks for inter-node communication.

2) Bandwidth: In the OSU bandwidth benchmark, the sender performs a number of back-to-back non-blocking sends designated by the window size for each message size, then waits for a reply from the receiver. The receiver performs the reverse, posting multiple non-blocking receives followed by a send. The increases in bandwidth achieved by our GPU-aware communication mechanisms are illustrated in Figures 12 and 13, with the range of improvement detailed in Table I. Charm++ and AMPI achieve close to the maximum attainable bandwidth (50 GB/s for intra-node, 12.5 GB/s for inter-node), with Charm++ demonstrating up to 44.7 GB/s and 10 GB/s, and AMPI up to 45.4 GB/s and 10 GB/s for intra-node and inter-node, respectively. It is worth noting that the host-staging version of AMPI (AMPI-H) suffers a degradation in bandwidth at 128 KB due to a sudden increase in latency, which is being investigated. Charm4py's bandwidth only reaches 35.5 GB/s for intra-node and 6.0 GB/s for inter-node in the given range of message sizes, but we observe that it keeps increasing as messages become larger than 4 MB.

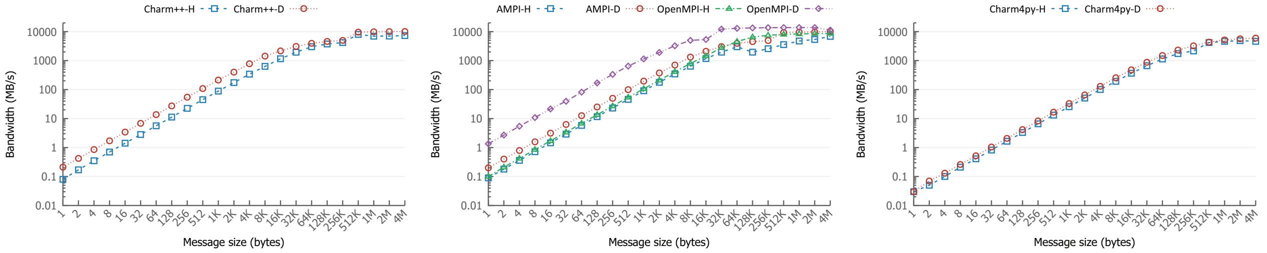
C. Proxy Application: Jacobi3D

To assess the impact of GPU-aware communication on application performance, we implement a proxy application, Jacobi3D, on all three parallel programming models: Charm++, AMPI, and Charm4py. Jacobi3D performs the Jacobi iterative method in a three-dimensional space, using CUDA kernels to



(a) Charm++ (b) AMPI and OpenMPI (c) Charm4py

Fig. 12. Comparison of intra-node bandwidth between host-staging and direct GPU-GPU mechanisms.



(a) Charm++ (b) AMPI and OpenMPI (c) Charm4py

Fig. 13. Comparison of inter-node bandwidth between host-staging and direct GPU-GPU mechanisms.

TABLE I
Improvement in latency and bandwidth with GPU-aware communication.

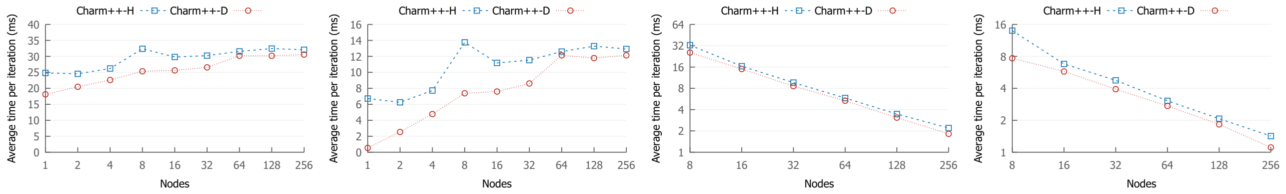
Improvement	Type	Charm++	Intra-node AMPI	Charm4py	Charm++	Inter-node AMPI	Charm4py
Latency	Range	2.1x – 10.2x	1.9x – 11.7x	1.8x – 17.4x	1.2x – 4.1x	1.8x – 3.5x	1.5x – 3.4x
	Eager	4.4x	3.6x	1.9x	4.1x	3.4x	1.8x
Bandwidth	Range	1.4x – 9.6x	1.3x – 10.0x	1.3x – 10.5x	1.2x – 2.7x	1.3x – 2.6x	1.0x – 1.5x

perform stencil computations on the GPU. The problem domain is decomposed into equal-size cuboid blocks, minimizing surface area. Each block exchanges its halo data on the GPU with up to six neighbors, which are either provided directly to the communication primitives (GPU-aware) or staged through host memory. Note that Jacobi3D is configured to run for a set number of iterations without convergence checks, to evaluate the performance of point-to-point communication.

We evaluate both weak and strong scaling performance of Jacobi3D using up to 256 nodes (1,536 GPUs) of Summit, comparing the overall time and communication time per iteration of the host-staging and GPU-aware communication mechanisms. Jacobi3D is weak scaled with a base domain size of $1,536^3$ double values and each dimension doubled in x, y, z order. Strong scaling experiments executed on eight to 256 nodes maintain the domain size of $3,072^3$ doubles.

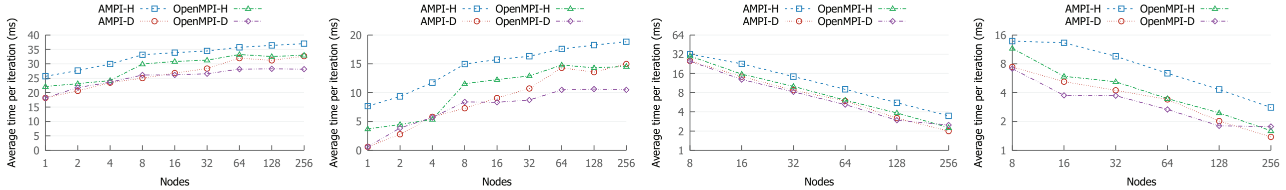
1) Charm++: Figure 14 shows the weak and strong scaling performance of the Charm++ version of Jacobi3D. With weak scaling, the GPU-aware version (Charm++-D) demonstrates a speedup between 1.1x and 12.4x in communication performance, with the largest speedup obtained on a single node. This is an expected result as the improvement in latency and bandwidth are more significant for intra-node communication. The improved communication performance translates into reductions in overall iteration time, between 5% and 37%. The relative speedup obtained with GPU-aware communication decreases as the number of nodes increases, as slower inter-node communication starts to dominate intra-node communication. With strong scaling, the improvement in communication performance ranges between 12% and 82% and overall iteration time between 9% and 27%, with the largest speedup obtained on a single node.

2) AMPI: Figure 15 illustrates the weak and strong scaling performance of the AMPI version of Jacobi3D, with the performance of OpenMPI provided as reference. With weak scaling, GPU-awareness improves the communication performance by factors between 1.3x and 12.8x, accelerating the overall performance up to 41%. The GPU-aware communication performance in AMPI is similar to that of OpenMPI up to 16 nodes, but starts to fall behind at larger scales. We suspect that this is due to the additional metadata exchange performed in AMPI whose performance impact becomes more pronounced at large node counts, but plan to look into this issue in more detail. With strong scaling, AMPI



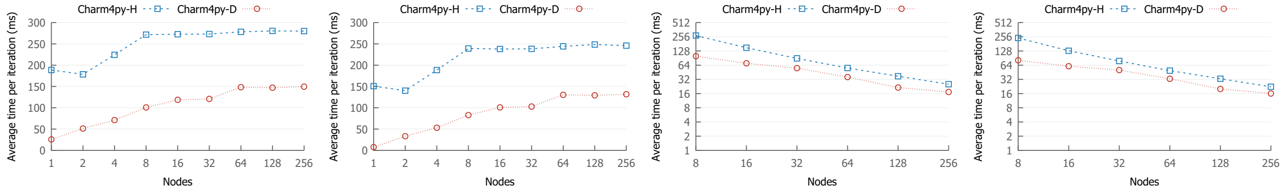
(a) Weak scaling, overall time (b) Weak scaling, comm. time (c) Strong scaling, overall time (d) Strong scaling, comm. time

Fig. 14. Comparison of Charm++ Jacobi3D performance between host-staging and direct GPU-GPU mechanisms.



(a) Weak scaling, overall time (b) Weak scaling, comm. time (c) Strong scaling, overall time (d) Strong scaling, comm. time

Fig. 15. Comparison of AMPI Jacobi3D performance between host-staging and direct GPU-GPU mechanisms.



(a) Weak scaling, overall time (b) Weak scaling, comm. time (c) Strong scaling, overall time (d) Strong scaling, comm. time

Fig. 16. Comparison of Charm4py Jacobi3D performance between host-staging and direct GPU-GPU mechanisms.

achieves a speedup between 1.9x and 2.6x in communication performance and an improvement in overall iteration time between 27% and 74%.

3) Charm4py: The weak and strong scaling performance of Charm4py are depicted in Figure 16. As the support for GPU-aware communication in Charm4py significantly improves performance especially for large messages as seen in Figures 10c and 11c, communication performance is improved by factors between between 1.9x and 19.7x with weak scaling. Because communication performance has a greater impact on the overall performance in Charm4py compared to other parallel programming models, we observe speedups in overall execution time between 1.9x and 7.3x. With strong scaling, the improvement in communication performance ranges between 1.4x and 3.0x, resulting in speedups between 1.5x and 2.7x in the overall iteration times.

V. Related Work

There have been many publications on supporting GPU-aware communication in the context of parallel programming models. Works from the MVAPICH group [2], [3], [19] utilize CUDA and GPUDirect technologies to optimize inter-GPU communication in MPI. Hanford et al. [20] highlights shortcomings of current GPU communication benchmarks and shares experiences with tuning different MPI implementations. Khorassani et al. [21] evaluates the performance of various MPI implementations on GPU-accelerated OpenPOWER systems. Chen et al. [22] proposes compiler extensions to support GPU communication in the UPC programming model. This work is distinguished from other related studies in demonstrating designs for GPU-aware communication and their performance in multiple parallel programming models built on a common abstraction layer based on UCX.

VI. Conclusion

In this work, we have discussed the importance of GPU-aware communication in today's GPU-accelerated supercomputers, and the associated technologies that are involved in supporting direct GPU-GPU transfers for several parallel programming models: Charm++, AMPI, and Charm4py. We leverage the capability of the UCX framework to seamlessly support inter-GPU communication through a set of high-performance APIs, implementing an extension to the UCX machine layer in the Charm++ runtime system to provide a performance-portable communication layer for the Charm++ family of parallel programming models. With designs to utilize the UCX machine layer for GPU-aware communication while retaining the semantics of message-driven execution, we demonstrate substantial improvements in performance using latency and bandwidth benchmarks adapted from the OSU benchmark suite, as well as a proxy application representing a widely used stencil algorithm.

With GPU-aware communication support in place for the Charm++ ecosystem, we plan to incorporate computation-communication overlap with overdecomposition [23] to minimize communication overheads on modern GPU systems. We also plan on supporting collective communication of GPU data, using this work as the basis to translate collective communication primitives to point-to-point calls.

While UCX proves to be an effective framework for universally accelerating GPU communication, there is still room for performance improvement as indicated by the differential between AMPI and OpenMPI. One of the potential areas of improvement is GPU support in the active messages API of UCX, which could better fit the message-driven execution model of Charm++. Another is supporting user-provided tags in the Charm++ runtime system, which would eliminate the need to delay the posting of the receive for GPU data until the arrival of the metadata message.

Acknowledgment

We thank the UCX developer team, including Akshay Venkatesh, Devendar Bureddy, and Yossi Itigin for their assistance with technical issues on the Summit supercomputer.

This work was performed under the auspices of the U.S. Department of Energy (DOE) by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-819099).

This research was supported by the Exascale Computing Project (17-SC- 20-SC), a collaborative effort of the U.S. DOE Office of Science and the National Nuclear Security Administration.

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. DOE under Contract No. DE-AC05-00OR22725.

References

[1] (2021) Top500 list, november 2020. [Online]. Available: https://www.top500.org/lists/top500/2020/11/

[2] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, “Optimizing mpi communication on multi-gpu systems using cuda inter-process communication,” in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, 2012, pp. 1848–1857.

[3] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, “Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus,” in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 80–89.

[4] D. Bonachea and P. H. Hargrove, “Gasnet-ex: A high-performance, portable communication library for exascale,” in *Languages and Compilers for Parallel Computing*, M. Hall and H. Sundar, Eds. Cham: Springer International Publishing, 2019, pp. 138–158.

[5] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, “A brief introduction to the openfabrics interfaces - a new network api for maximizing high performance application efficiency,” in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 34–39.

[6] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, “Ucx: An open source framework for hpc network apis and beyond,” in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43.

[7] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, “The development of mellanox/nvidia gpudirect over infiniband–a new model for gpu to gpu communications,” *Comput. Sci.*, vol. 26, no. 3–4, p. 267–273, Jun. 2011. [Online]. Available: https://doi.org/10.1007/s00450-011-0157-1

[8] (2021) Gpudirect rdma :: Cuda toolkit documentation. [Online]. Available: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html

[9] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, and D. K. D. K. Panda, “Designing efficient small message transfer mechanism for inter-node mpi communication on infiniband gpu clusters,” in *2014 21st International Conference on High Performance Computing (HiPC)*, 2014, pp. 1–10.

[10] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, “Parallel programming with migratable objects: Charm++ in practice,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’14. IEEE Press, 2014, p. 647–658. [Online]. Available: https://doi.org/10.1109/SC.2014.58

[11] C. Huang, O. Lawlor, and L. V. Kalé, “Adaptive mpi,” in *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 306–322.

[12] (2021) Charm++ zero copy messaging api. [Online]. Available: https://charm.readthedocs.io/en/v6.10.2/charm++/manual.html#zero-copy-messaging-api

[13] J. J. Galvez, K. Senthil, and L. Kale, “Charmpy: A python parallel programming model,” in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 423–433.

[14] (2021) Charm4py channels api. [Online]. Available: https://charm4py.readthedocs.io/en/latest/introduction.html#channels

[15] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, “Cython: The best of both worlds,” *Computing in Science Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[16] (2021) Ucx-py. [Online]. Available: https://github.com/rapidsai/ucx-py

[17] (2021) Charm4py futures api. [Online]. Available: https://charm4py.readthedocs.io/en/latest/introduction.html#futures

[18] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda, “Omb-gpu: A micro-benchmark suite for evaluating mpi libraries on gpu clusters,” in *Recent Advances in the Message Passing Interface*, J. L. Träff, S. Benkner, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 110–120.

[19] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, “Mvapich2-gpu: Optimized gpu to gpu communication for infiniband clusters,” *Comput. Sci.*, vol. 26, no. 3–4, p. 257–266, Jun. 2011. [Online]. Available: https://doi.org/10.1007/s00450-011-0171-3

[20] N. Hanford, R. Pankajakshan, E. A. León, and I. Karlin, “Challenges of gpu-aware communication in mpi,” in *2020 Workshop on Exascale MPI (ExaMPI)*, 2020, pp. 1–10.

[21] K. S. Khorassani, C.-H. Chu, H. Subramoni, and D. K. Panda, “Performance evaluation of mpi libraries on gpu-enabled openpower architectures: Early experiences,” in *High Performance Computing*, M. Weiland, G. Juckeland, S. Alam, and H. Jagode, Eds. Cham: Springer International Publishing, 2019, pp. 361–378.

[22] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou, “Unified parallel c for gpu clusters: Language extensions and compiler implementation,” in *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC’10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 151–165.

[23] J. Choi, D. F. Richards, and L. V. Kale, “Achieving computation-communication overlap with overdecomposition on gpu systems,” in *2020 IEEE/ACM 5th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2020, pp. 1–10.