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

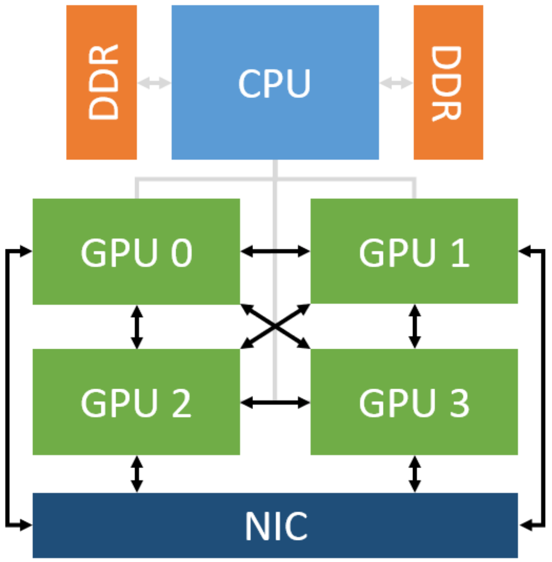
Jaemin Choi[1], Zane Fink[1], Sam White[1], Nitin Bhat[2], David F. Richards[3], Laxmikant V. Kale[1,2]

[1] University of Illinois at Urbana-Champaign
[2] Charmworks, Inc.
[3] Lawrence Livermore National Laboratory

May 17, 2021
AsHES Workshop at IPDPS

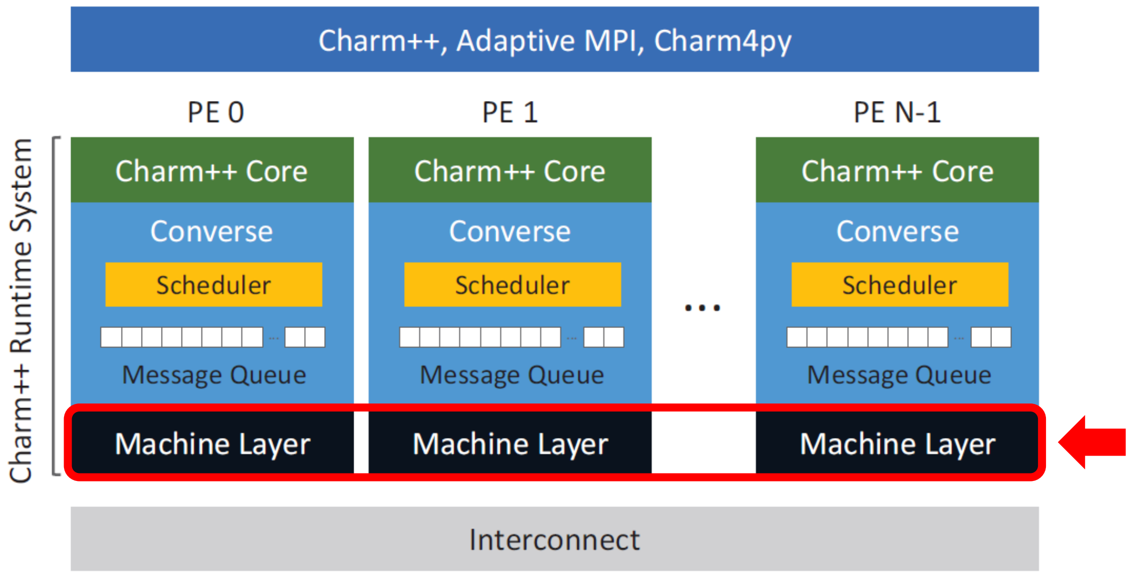
Motivation

- Increasing adoption of multi-GPU nodes and GPU acceleration in today's HPC systems
- Need for GPU-aware communication in **Charm++**
  - Multiple front-end models/languages: Charm++, Adaptive MPI, Charm4py (Python)
  - Cater to asynchronous message-driven execution
  - Implemented support using CUDA P2P memcpy and IPC
    → only for Charm++, insufficient performance
- How can we support ***efficient*** GPU-aware communication in a ***portable*** way?
  - Use **Unified Communication X (UCX)**!
  - Support NVIDIA and AMD GPUs



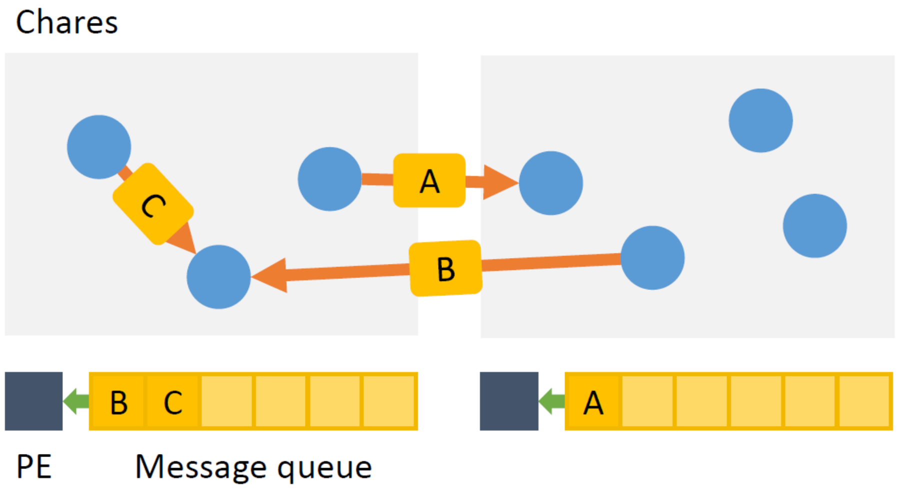
< Node Diagram of OLCF Frontier >

Approach



- Machine layer interacts with the interconnect to perform communication
- Extend the **UCX machine layer** in Charm++ RTS to support GPUs

Background



- **Charm++**
  - Object-oriented parallel programming system based on C++
  - Parallel objects (i.e., **chares**) mapped to Processing Elements (PEs, e.g., CPU cores)
  - Message-driven execution
    - Messages exchanged between chare objects drive computation
    - Computation encapsulated in **entry methods** (methods that can be invoked remotely)
    - Necessitates metadata (e.g., target chare & entry method) to be contained in messages

Background

- **Adaptive MPI (AMPI)**
  - MPI library implementation on top of Charm++
  - Virtualization: enables multiple MPI ranks per OS process
  - Can co-schedule and migrate ranks between PEs
- **Charm4py**
  - Python framework on top of Charm++
  - Cython layer connects Charm++ RTS in C++ and Charm4py RTS in Python
  - Communication through *channels* established between chares

Design Overview

- Utilize GPU-awareness in UCX tagged send/recv API
- Extend UCX machine layer in Charm++ RTS
- Multiple buffers can be sent with single entry method invocation (*no explicit receive*)

```
void Sender::foo() {
  receiver.bar(my_int, my_buffer);
}

void Receiver::bar(int dir, double* buf) {
  ...
}
```

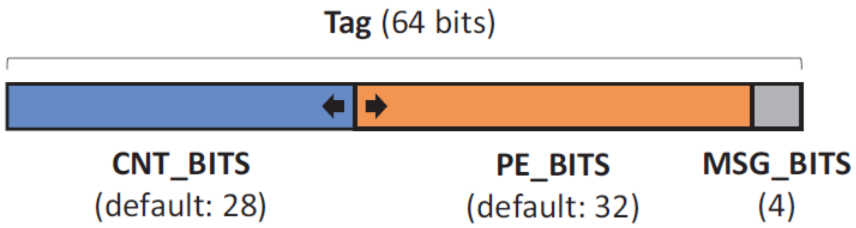
- Send **metadata and buffers on host memory** through existing mechanism
- Send **GPU buffers** separately through the UCX machine layer extension

Design Overview

- When host-side message arrives on the receiver
  - Scheduler picks up message from message queue
  - Unpacks host buffers
  - Posts receives for incoming GPU buffers using metadata
- Once all data arrives, scheduler executes target entry method
- Limitation: delay in posting receive for GPU data

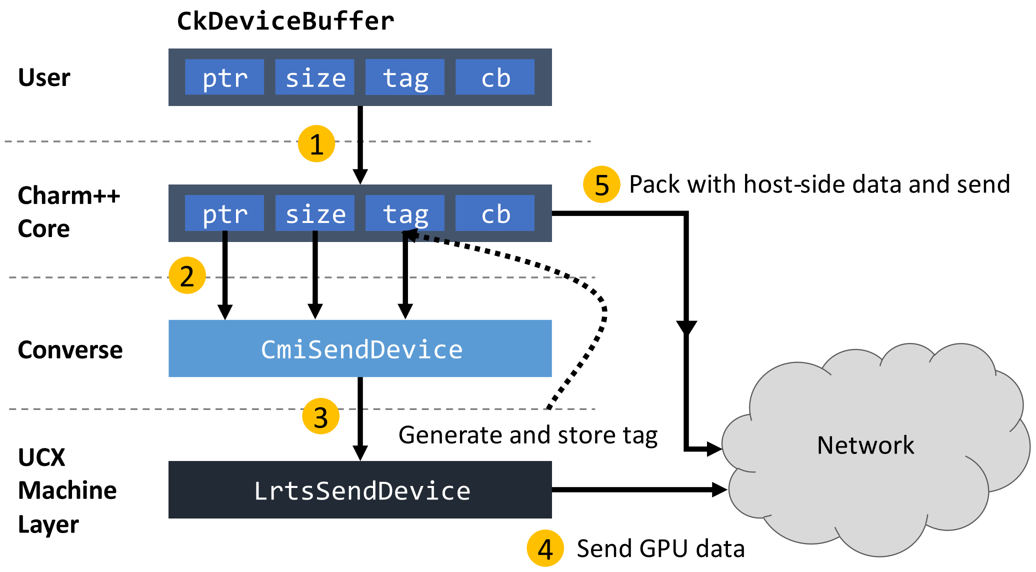
Implementation Details

- **UCX machine layer**
  - Originally contributed by Mellanox for host-side communication
  - Portable abstraction for different networking hardware supported by UCX
  - Uses UCP Tagged API: `ucp_tag_send_nb, ucp_tag_recv_nb`
- UCX tags (64 bits) for host-side messages are only used to determine the *type* of message
  - User buffers and Charm++ layer-specific data are packed inside the payload
  - How to handle GPU-aware communication (separate UCX send/receive)?
- Need some correlation to post receive for GPU buffer
  - Each PE keeps a counter
  - Incremented on every GPU-aware message send
  - Contained in the host-side metadata message for receiver



Implementation Details

- **Charm++**
  - GPU buffers can be passed to entry method invocations
  - **`nocopydevice`**: denote source GPU buffer
  - **`CkDeviceBuffer`**: store metadata for RTS
  - Receiver provides address of destination buffer




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

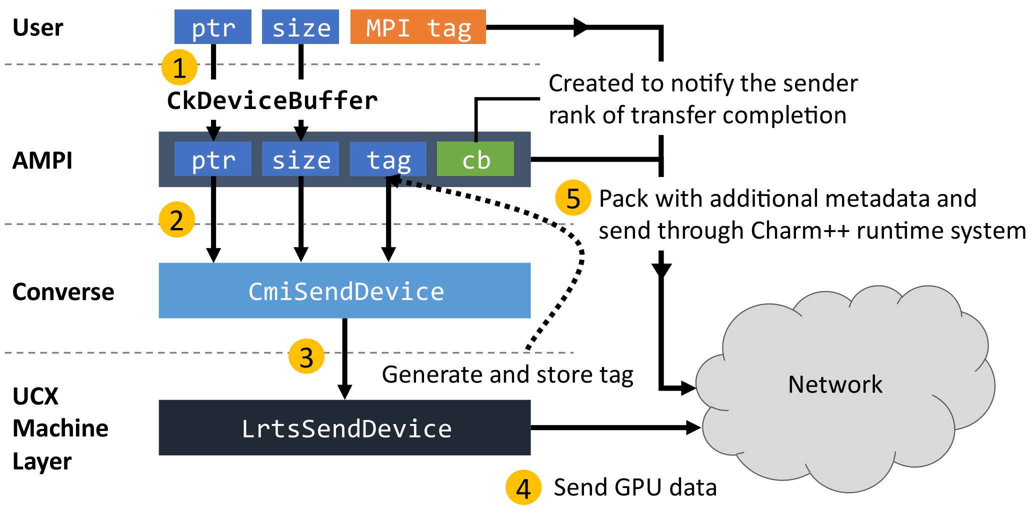
```
// C++ source file
// (1) Sender chare
void MyChare::send() {
  peer.recv(CkDeviceBuffer(send_gpu_data), size);
}

// (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) {
  // Received GPU buffer is available
  ...
}
```

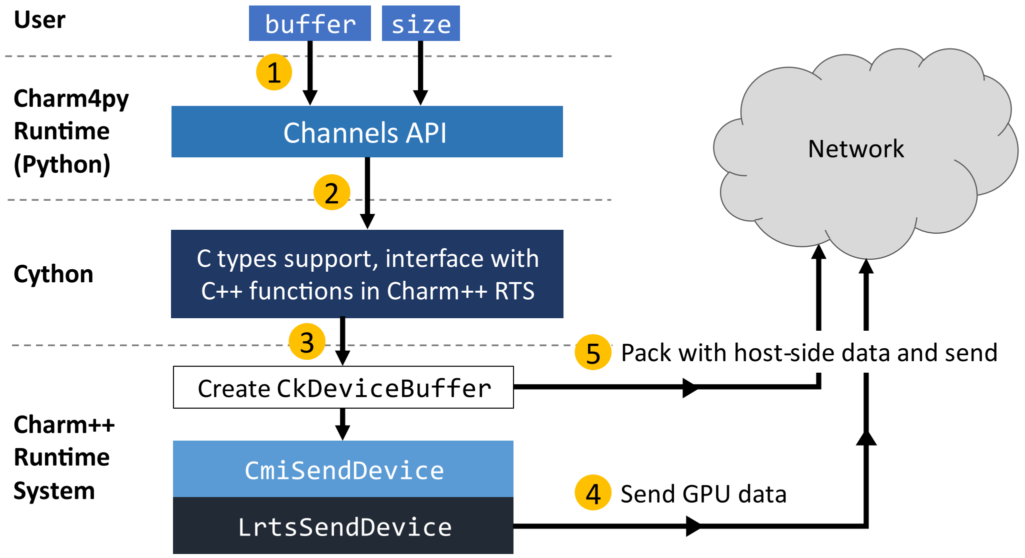
Implementation Details

- **Adaptive MPI (AMPI)**
  - Each AMPI rank is implemented as a chare object
  - Communication occurs through message exchanges between chares
- User's MPI tag is **not** provided to UCX (unlike other MPI implementations)
- Software cache for checking if source buffer is on GPU memory
- GPU buffers are sent separately, for which receives are posted only after the host-side message arrives → potential performance issue
- Charm++ callback is used to notify the receiver MPI rank when all GPU buffers have arrived



Implementation Details

- **Charm4py**
  - Utilizes channels established between chares
  - Provides functionality to explicitly post receives like MPI
  - Charm++ callback used to wake up suspended coroutine when communication completes




```python
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)
```

Experimental Setup

- OLCF Summit
  - Up to 256 nodes (1,536 NVIDIA Tesla V100 GPUs)
  - 6 processes per node, 1 process (1 CPU core) per GPU
  - NVLink: 50 GB/s, Infiniband: 12.5 GB/s
- No overdecomposition/virtualization (1 chare object per PE/CPU core)
- Benchmarks
  - OSU latency & bandwidth micro-benchmarks
  - Proxy application for 3D Jacobi iterative method (Jacobi3D)
- Compare AMPI vs. OpenMPI
  - Both uses UCX to transfer GPU data
  - Evaluate overheads caused by message-driven execution & intermediate Charm++ RTS layers

Performance Evaluation: Latency

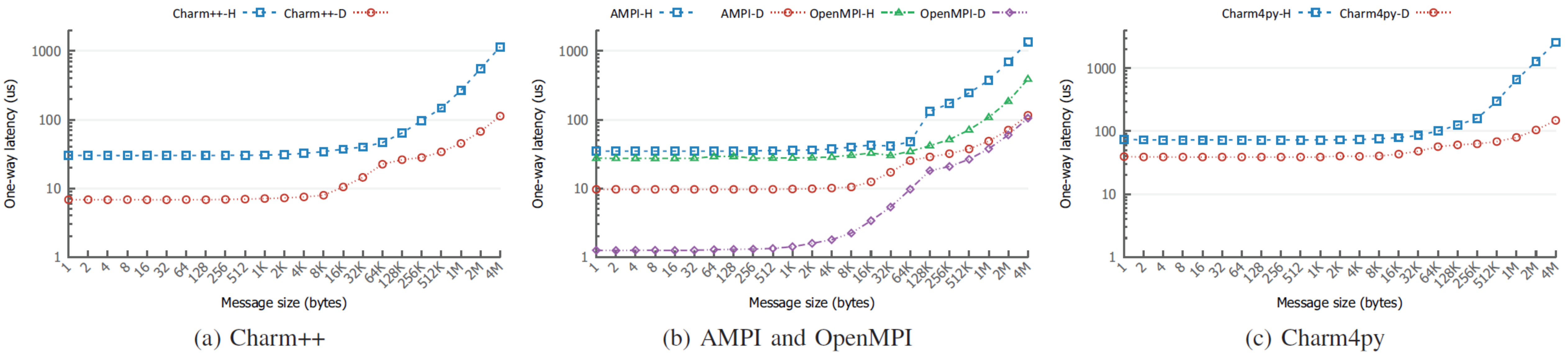
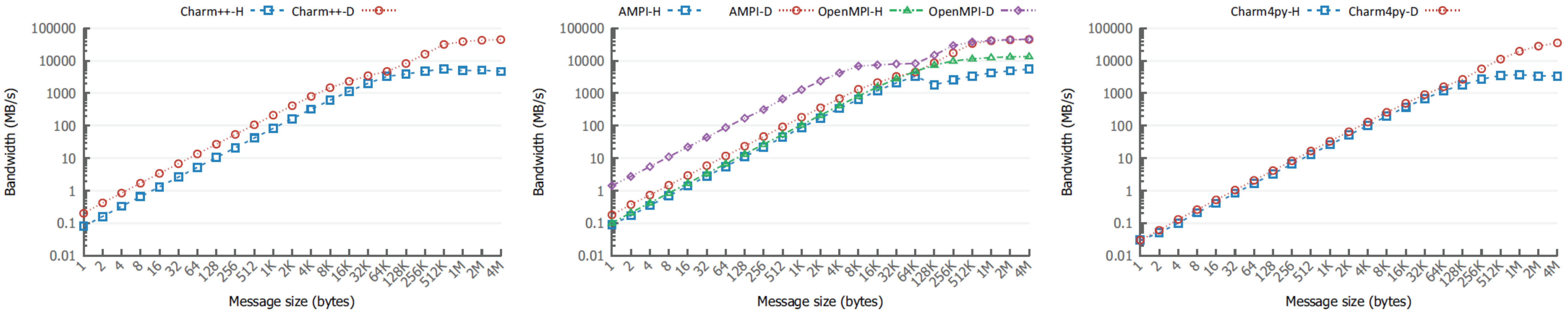


Fig. 10. Comparison of intra-node latency between host-staging and direct GPU-GPU mechanisms.

- **H**: Host-staged, **D**: Direct GPU-GPU
- Charm++: ~10.2x, AMPI: ~11.7x, Charm4py: ~17.4x
- AMPI overheads vs. OpenMPI
  - Message packing/unpacking, additional host-side metadata message & delay in posting receive, Charm++ callback invocations

Performance Evaluation: Bandwidth



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

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

- Charm++: ~9.6x, AMPI: ~10x, Charm4py: ~10.5x

Performance Evaluation: Jacobi3D

Charm++ only,
AMPI and Charm4py
in paper

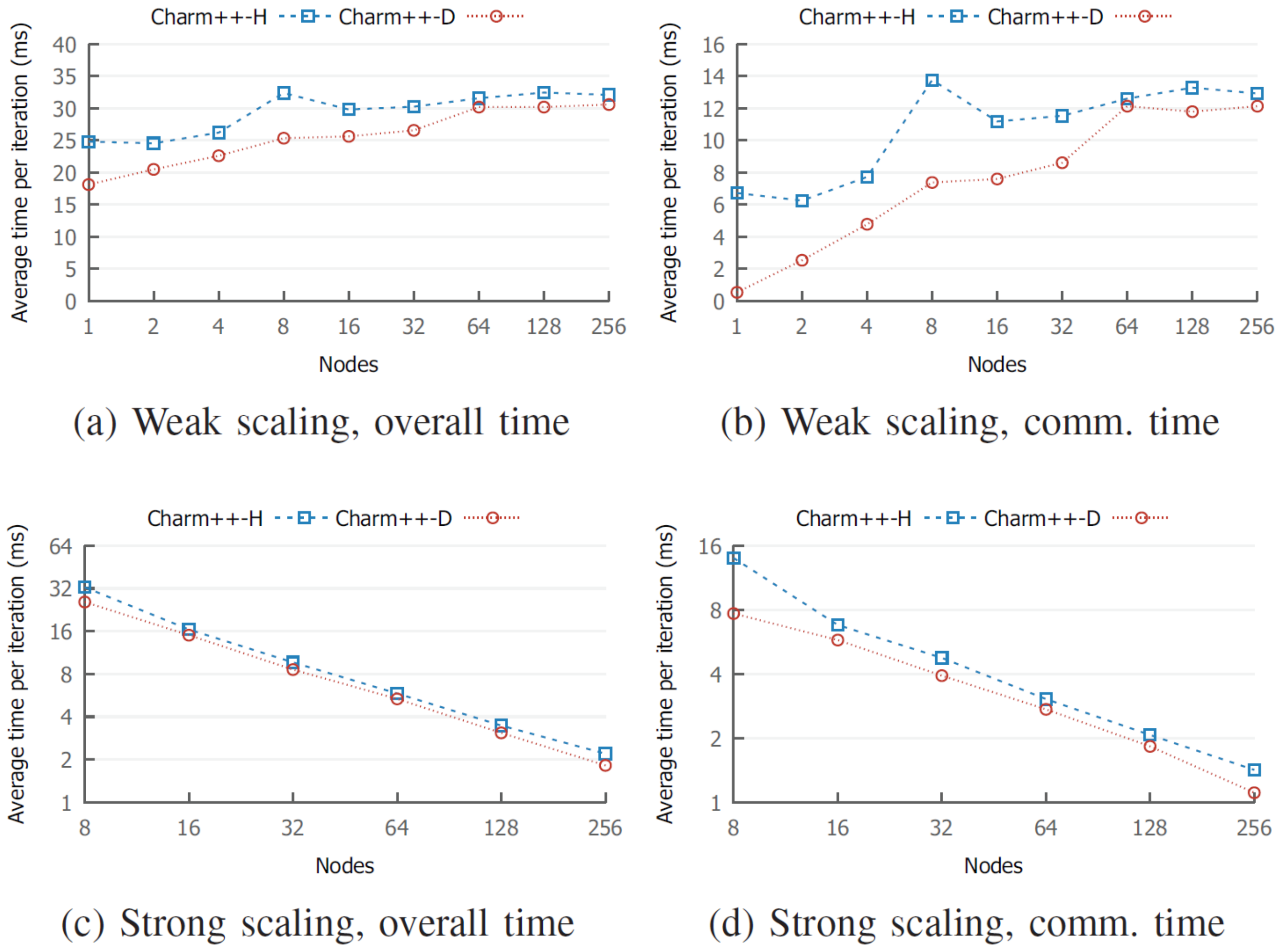
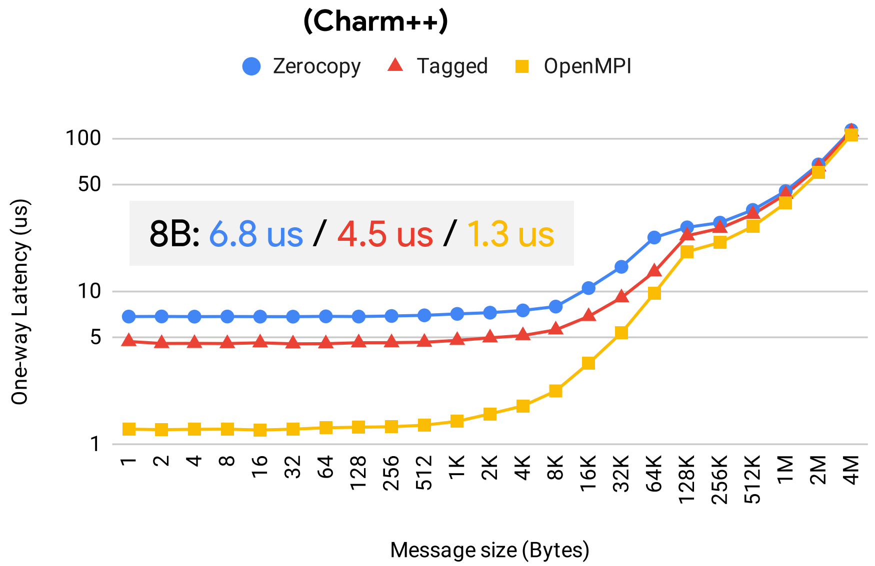
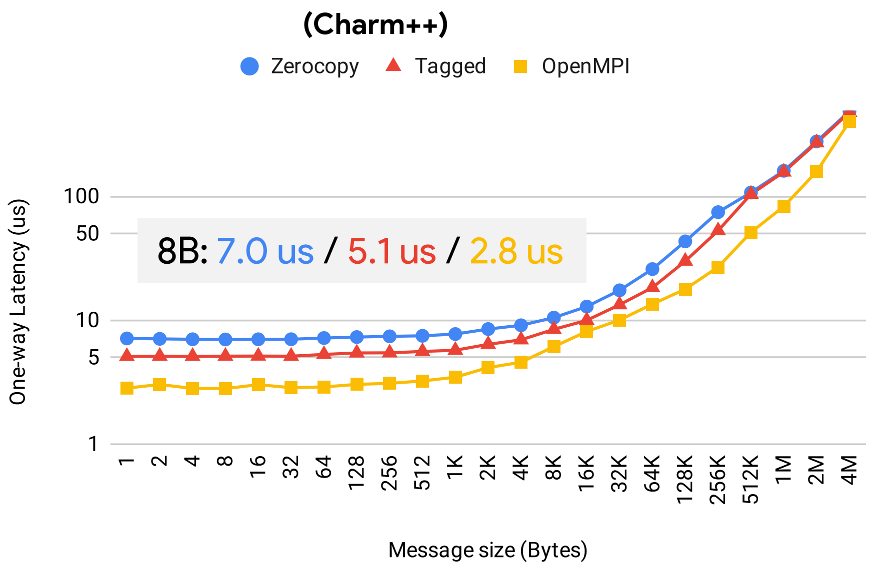


Fig. 14. Comparison of Charm++ Jacobi3D performance between host-staging and direct GPU-GPU mechanisms.

Ongoing/Future Work



< Intra-node >



< Inter-node >

- How to close the gap between Charm++/AMPI vs. OpenMPI?
- Post receive without waiting for sender's metadata
  - Charm++: Need a different API ('**Tagged API**') to push responsibility of tag generation to the user (like MPI)
  - AMPI & Charm4py: Directly utilize explicit receive, like OpenMPI
- GPU support for Active Messages API in UCX?

Conclusion

- Design and implementation to support GPU-aware communication using UCX in multiple parallel programming models: Charm++, AMPI, and Charm4py
- Design considerations to support message-driven execution and task-based runtime systems
- Evaluated performance improvements using a set of micro-benchmarks and proxy application

Thank you! Questions?