Improving the Communication Performance of Hybrid Programming Models

Wenbin Lu[1]; Tony Curtis[1]; Barbara Chapman, PhD[1,2]
[1]Stony Brook University, [2]Brookhaven National Laboratory

Abstract

Modern HPC clusters have nodes that incorporate large amounts of shared memory cores and powerful network adaptors. The efficiency of hybrid programming models are critical for scientific applications to reach desired levels of scalability on those machines.

In this work, we combine API extensions in distributed programming models that enable fine grained control of communication in multi-threaded codes, and an implementation of those APIs that utilizes a low-latency network library which is designed to drive the network to its full potential using multiple CPU cores. The resulting hybrid microbenchmarks achieved up to 64 times higher communication throughput when compared to the traditional implementation, for small to medium message sizes and atomic operations.

Motivation

The next generation of supercomputers and HPC clusters will have fewer but more powerful nodes. For example, with 44 IBM POWER9 CPU cores and six NVIDIA Volta V100 GPUs per node, the Summit supercomputer at ORNL achieved more than eight times the peak FLOPS of the 18,688-node supercomputer TITAN, using only 4,608 nodes. Following this trend, different programming models need to be able to work together efficiently to avoid introducing more overheads that impedes scalability.

Unfortunately, this is usually not the case. As an example, the universally supported `MPI_THREAD_MULTIPLE` mode, while allowing all threads to call MPI routines simultaneously, incurs high overhead from locking and provides little benefit in return. The `FUNNELED` mode is equally flawed as it requires careful tuning for each hardware/software combination to achieve good overlap, while maintaining balanced loads[1].

Past works [4], [5] have shown that replacing MPI processes with threads decreases the communication throughput, especially for small message sizes.

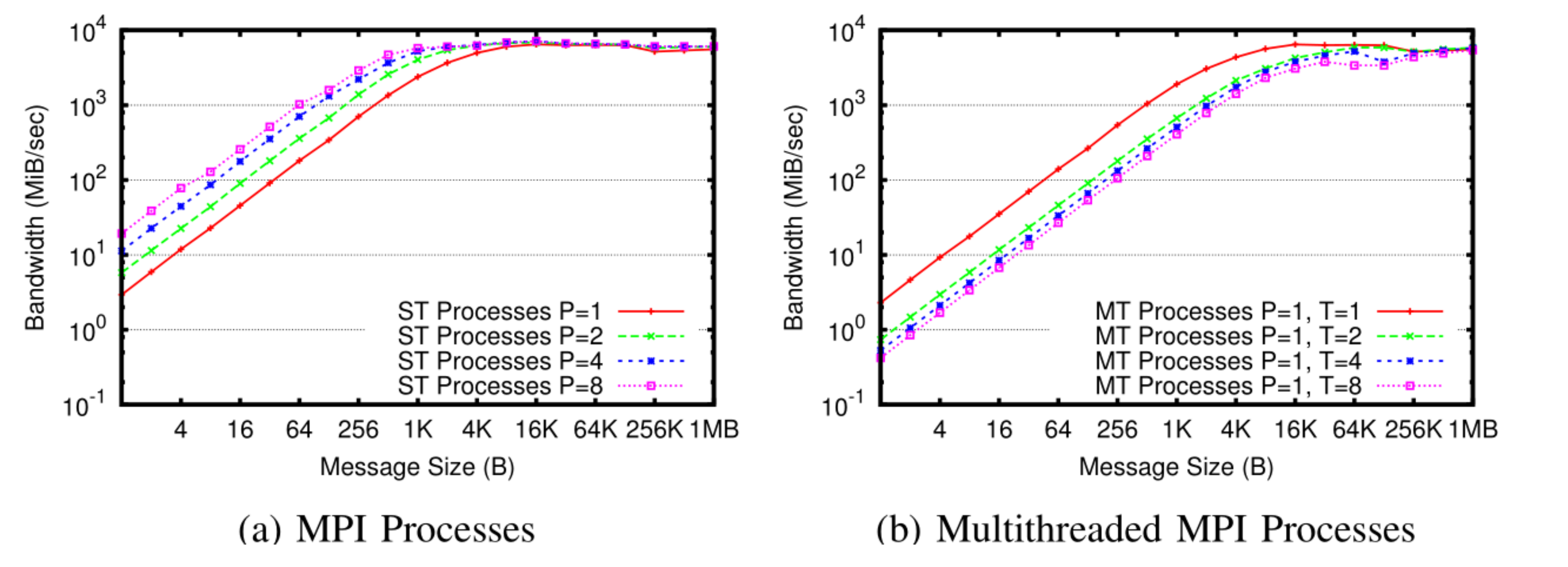

Figure 1: Bandwidth versus message size, comparing MPI processes and multithreaded MPI using the OSU microbenchmark suite. Test setup: Intel Xeon E5-2697, Mellanox FDR InfiniBand network adaptors, Intel MPI 4.1.3, Intel C Compiler 14.0.2. Source: [4]

The OpenUCX (Unified Communication X) Library

OpenUCX is an HPC communication library that abstracts vendor-specific network interfaces, and provides a set of unified communication primitives which can be used to implement various programming models.

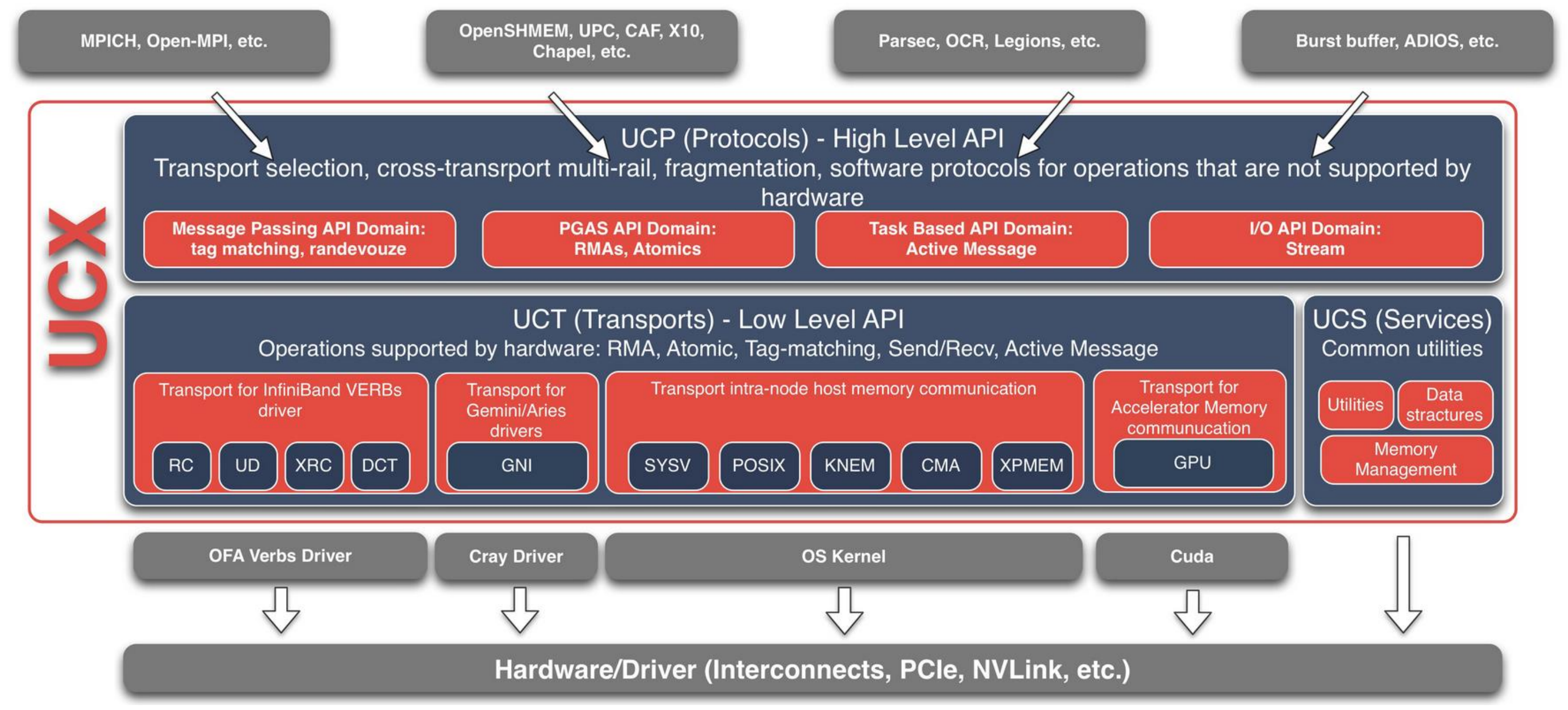
MPICH, Open-MPI, etc.
OpenSHMEM, UPC, CAF, X10, Chapel, etc.
Parsec, OCR, Legions, etc.
Burst buffer, ADIOS, etc.
UCX
UCP (Protocols) - High Level API
Transport selection, cross-transrport multi-rail, fragmentation, software protocols for operations that are not supported by hardware
Message Passing API Domain: tag matching, randevouze
PGAS API Domain: RMAs, Atomics
Task Based API Domain: Active Message
I/O API Domain: Stream
UCT (Transports) - Low Level API
Operations supported by hardware: RMA, Atomic, Tag-matching, Send/Recv, Active Message
Transport for InfiniBand VERBs driver
RC
UD
XRC
DCT
Transport for Gemini/Aries drivers
GNI
Transport intra-node host memory communication
SYSV
POSIX
KNEM
CMA
XPMEM
Transport for Accelerator Memory communucation
GPU
UCS (Services)
Common utilities
Utilities
Data structures
Memory Management
OFA Verbs Driver
Cray Driver
OS Kernel
Cuda
Hardware/Driver (Interconnects, PCIe, NVLink, etc.)

Figure 2: OpenUCX architecture. Source: [6]

OpenUCX provides workers objects that act as network injection points and communication managers. Workers' tag matching spaces are independent of each other, as well as their Send/Recv/RDMA progression and completion environments. Being able to select the optimized backend for each type of interconnect, the workers can effectively utilize hardware resources and thus have performance comparable to individual MPI ranks, even if they are created inside a single process.

The New Approach and Results

Instead of letting all threads to share a single worker like in the mainstream implementations, each thread will now use a private worker. To evaluate this new approach, we have created several multi-threaded microbenchmarks that perform different communication operations, and recorded the average latencies. Up to 64x speedup is achieved for 24 threads.

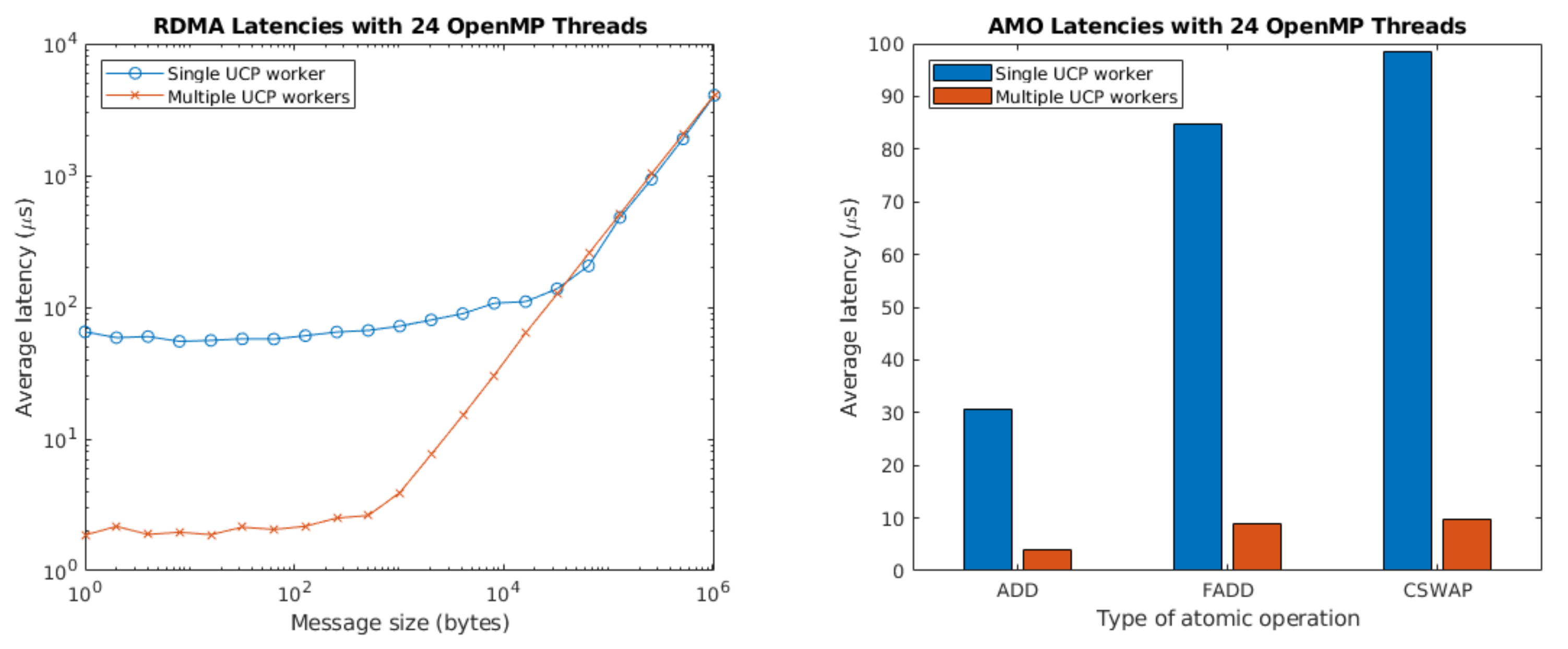

Figure 3: Message latency versus message size, comparing single UCP worker and multiple UCP workers. Test setup: Intel Xeon E5-2690, Mellanox InfiniBand ConnectX-5 network adaptors, OpenUCX 1.5.1, GCC/GOMP 8.3.0.

Programming Model Integration

Our work will be integrated into implementations of distributed programming models in the following ways:

- OpenSHMEM Contexts: Containers that define independent ordering and completion environments that can be used to manage the communication operations performed by separate threads within a multi-threaded PE.
- MPI Endpoints: Ongoing MPI proposal that adds the ability to create sub-ranks within ranks, allowing threads to be addressed directly.

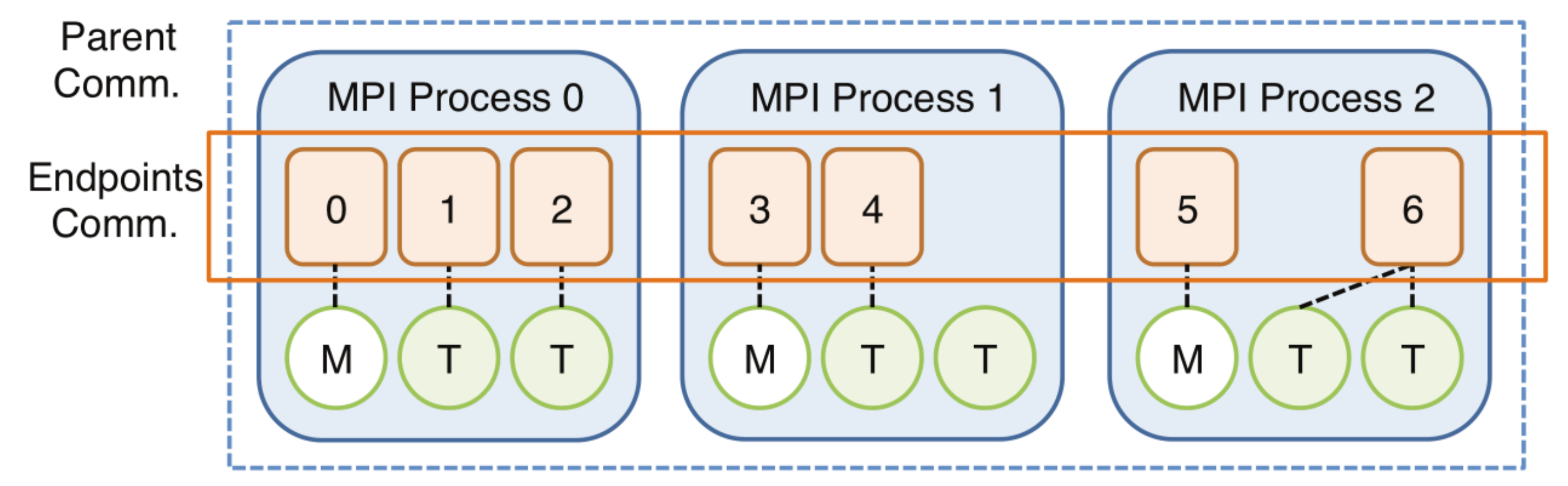

Figure 4: MPI endpoints proposal. Source: [4]

Related Works

- Intel Omni-Path multi-EP [2][3]: Starting from Intel MPI release 2019, `MPI_THREAD_SPLIT` is added to allow threads to use the multiple network injection points in the Omni-Path architecture. This approach is very restrictive in which sets of threads can communicate with each other.
- EP-lib by Srinivas Sridharan, James Dinan, et al [4]. An experimental library for MPI endpoints support. Thread offload MPI calls to proxy MPI processes using POSIX shared memory and library overloading.
- Single MPI process offloading by Karthikeyan Vaidyanathan, Dhiraj Kalamkar, et al [5]. This work reduced MPI library call overhead by offloading all MPI calls to a single process, so the threads can go back to computation immediately. It does not eliminate library internal overhead.

Future Work

- Translate this performance gain to MPI/OpenSHMEM implementations. Mutexes that were used to protect the internal data structures need to be removed, and the levels of indirections can be reduced.
- Improve the GPU support of distributed programming models by exploring the GPU-allocated memory support in OpenUCX. Coordinate the communication between threads, CUDA/ROCm kernels and remote nodes more efficiently.

References

1. OpenMP Tasking and MPI in a Lattice QCD Benchmark, Larry Meadows and Ken-ichi Ishikawa, IWOMP 2017
2. Multiple Endpoints for Improved MPI Performance on a Lattice QCD Code, Larry Meadows, Kenichi Ishikawa, Taisuke Boku, Masashi Horikoshi, HPC ASIA 18
3. Accelerating HPC codes on Intel Omni-Path Architecture networks: From particle physics to Machine Learning, Peter Boyle, Michael Chuvelev, Guido Cossu, Christopher Kelly, Christoph Lehner, Lawrence Meadows, arXiv:1711.04883
4. Enabling Efficient Multithreaded MPI Communication Through a Library-Based Implementation of MPI Endpoints, Srinivas Sridharan, James Dinan, and Dhiraj D. Kalamkar, Super Computing 2014
5. Improving concurrency and asynchrony in multithreaded MPI applications using software offloading, Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, Kiran Pamnany, Jeff R. Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, Thomas Jefferson, Super Computing 2015
6. OpenUCX. https://www.openucx.org/