Optimizing Point-to-Point Communication between AMPI Endpoints in Shared Memory

Sam White & Laxmikant Kale
UIUC

ExaMPI 2017

Motivation

- Exascale trends:
    - HW: increased node parallelism, decreased memory per thread
    - SW: applications themselves becoming more dynamic
- How should applications and runtimes respond?
    - MPI+X (X=OpenMP, Kokkos, MPI, etc)?
    - New language? Legion, Charm++, HPX, etc?

Motivation

- MPI+X performance:
  - Either serialize around communication …
  - Or incur synchronization costs inside MPI
    - Semantic restrictions can help
- What if we hoist threading into MPI?
  - Performant MPI+X often requires similar hoisting
  - Threaded MPIs: MPC-MPI, FG-MPI, TMPI, AMPI
  - Similar to the MPI Endpoints proposal

Motivation

- Questions:
  - Why is AMPI's existing implementation not as fast as we might expect (vs process-based MPIs)?
    - What can be done to improve it?
  - More generally, what are the costs of process boundaries & kernel-assisted IPC?
    - What can MPI Endpoints offer in terms of pt2pt latency & bandwidth in shared memory?

Overview

- Introduction to AMPI
  - Execution model
  - Existing shared memory performance
- Shared memory optimizations
  - Locality: intra-core vs intra-process
  - Size: small vs large messages
- Conclusions & future work

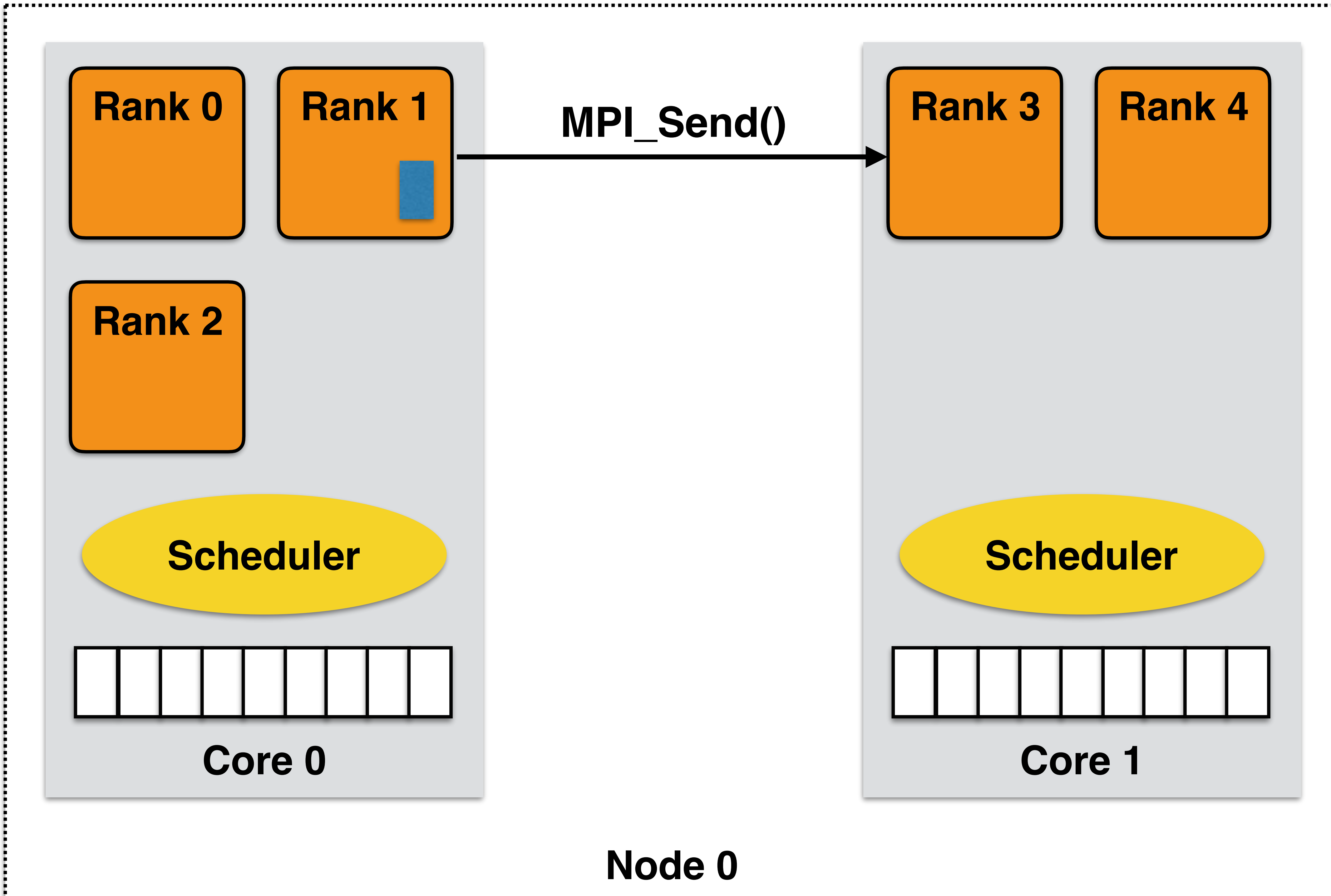
Adaptive MPI

- AMPI is an MPI implementation on top of Charm++
- AMPI offers Charm++'s application-independent features to MPI programmers:
  - Overdecomposition
  - Communication/computation overlap
  - Dynamic load balancing
  - Online fault tolerance

Execution Model

- AMPI ranks are User-Level Threads (ULTs)
  - Can have multiple per core
  - Fast to context switch
  - Scheduled based on message delivery
  - Migratable across cores and nodes at runtime
    - For load balancing & online checkpoint/restart

Execution Model

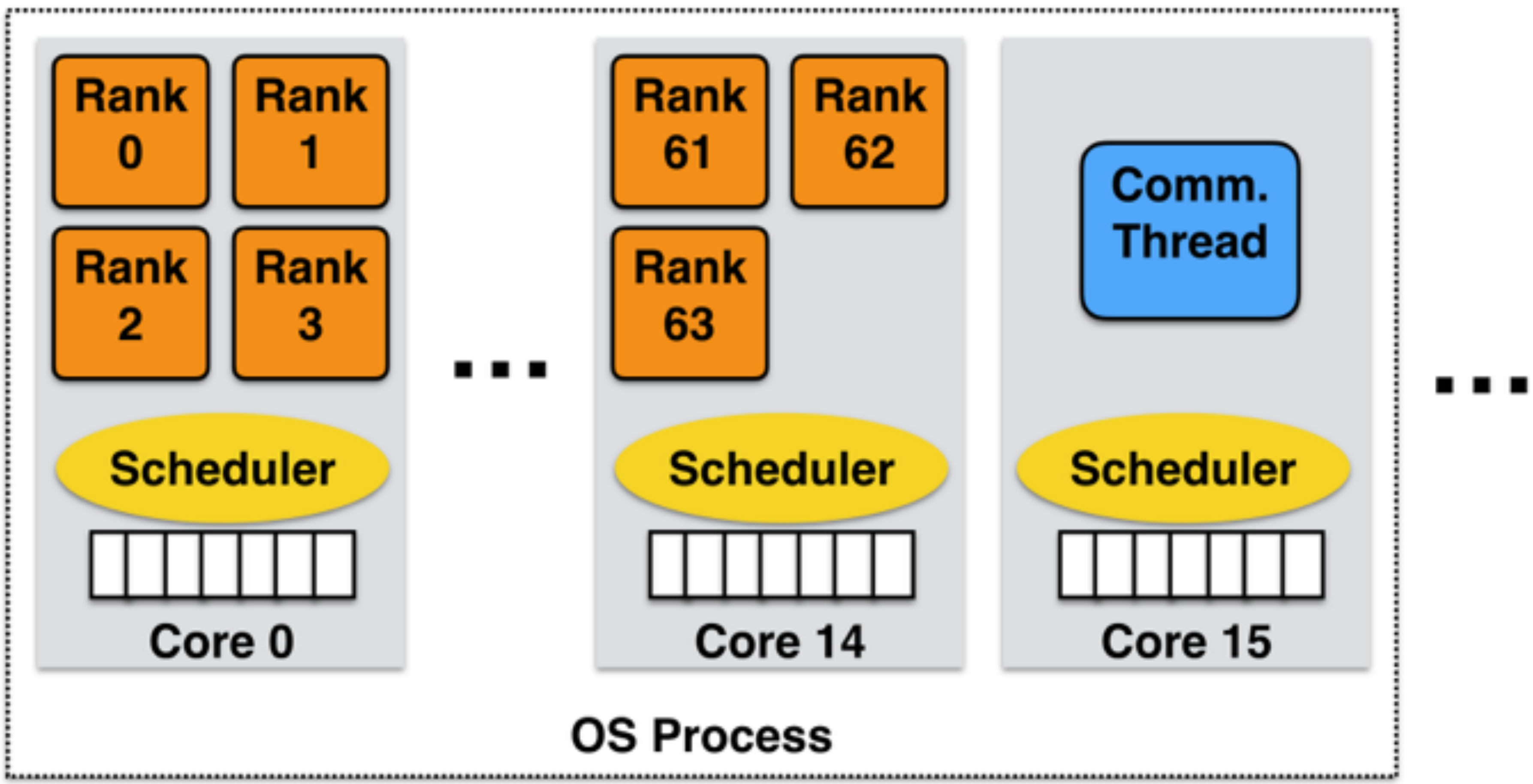


Shared Memory

- AMPI can be built in two different modes
  - Non-SMP: 1 process per core/hyperthread
  - SMP: 1 process per node/socket/NUMA domain
    - N worker threads, 1 dedicated communication thread per process

AMPI Shared Memory

- Many AMPI ranks can share the same OS process

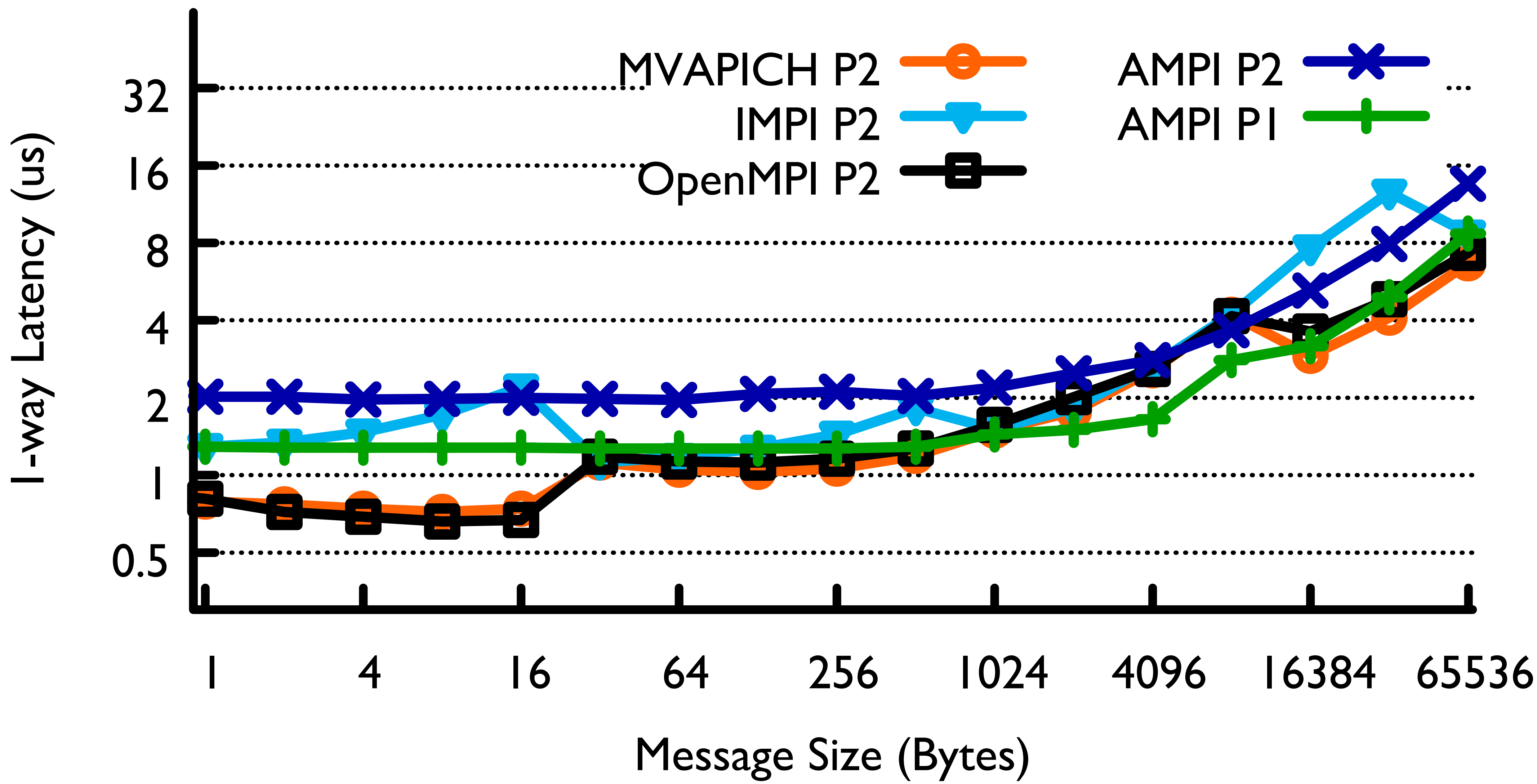


Performance Analysis

- OSU Microbenchmarks v5.3: osu_latency, osu_bibw
- Quartz (LLNL): Intel Xeon (Ivybridge)
  - MVAPICH2 2.2, Intel MPI 2018, OpenMPI 2.0.0
- Cori (NERSC): Intel Xeon (Haswell)
  - Cray MPI 6.7.0
- AMPI (6.8.0) vs AMPI-shm (6.9.0-beta)
  - P1: two ranks co-located on the same core
  - P2: two ranks on different cores, in the same OS process

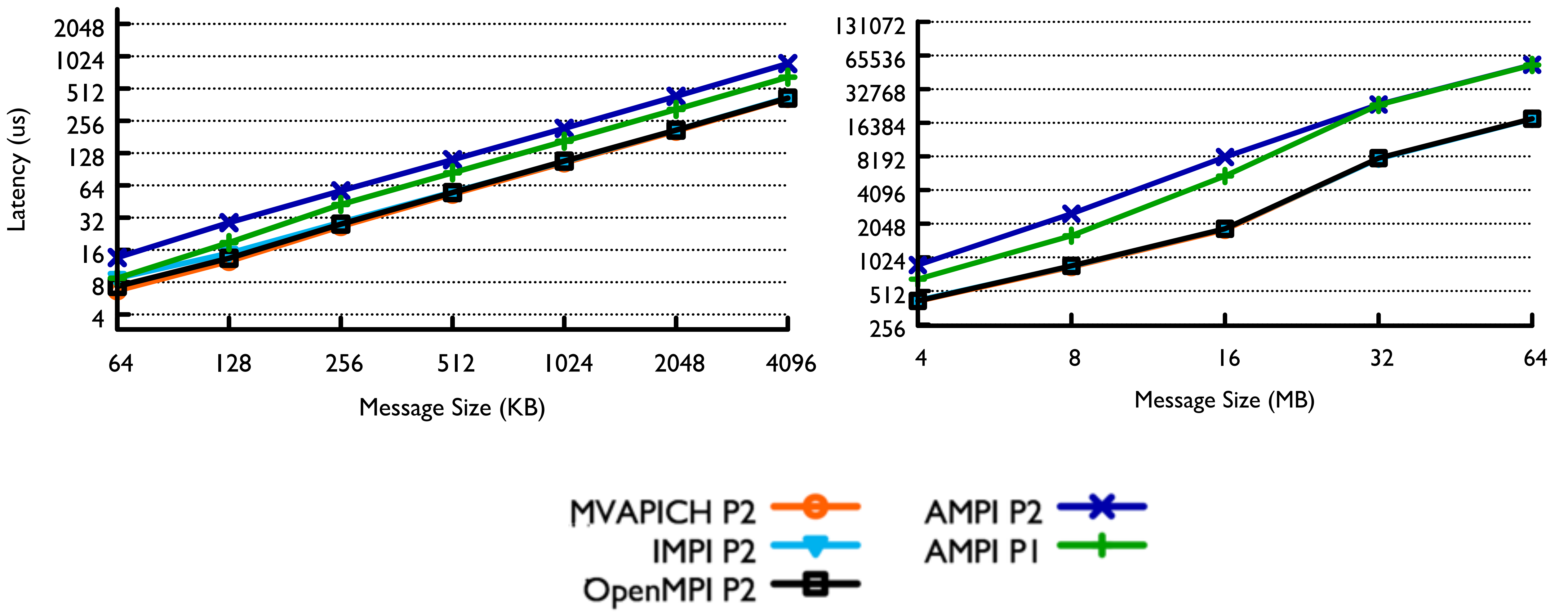
Existing Performance

- Small message latency on Quartz



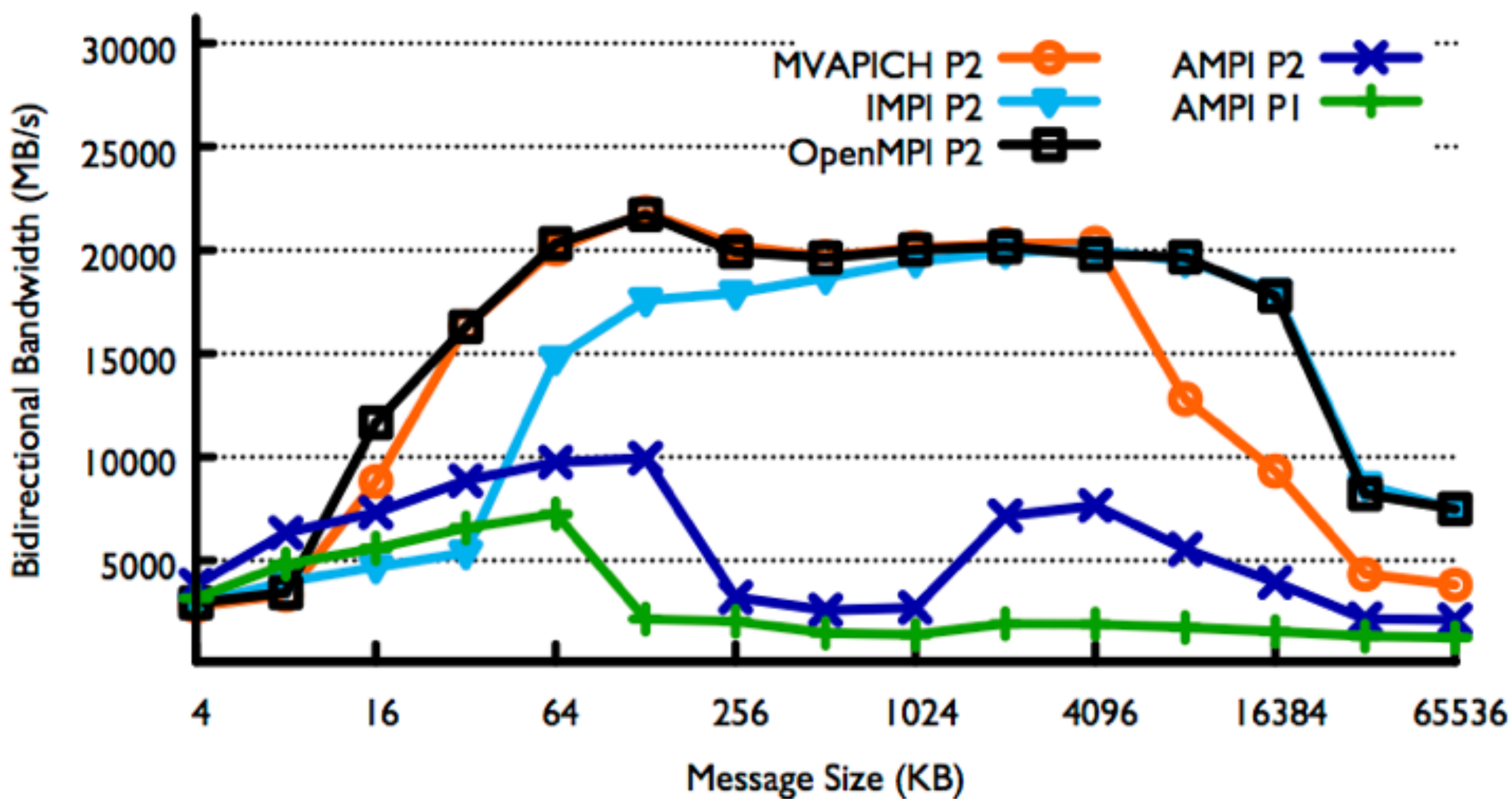
Existing Performance

- Large message latency on Quartz



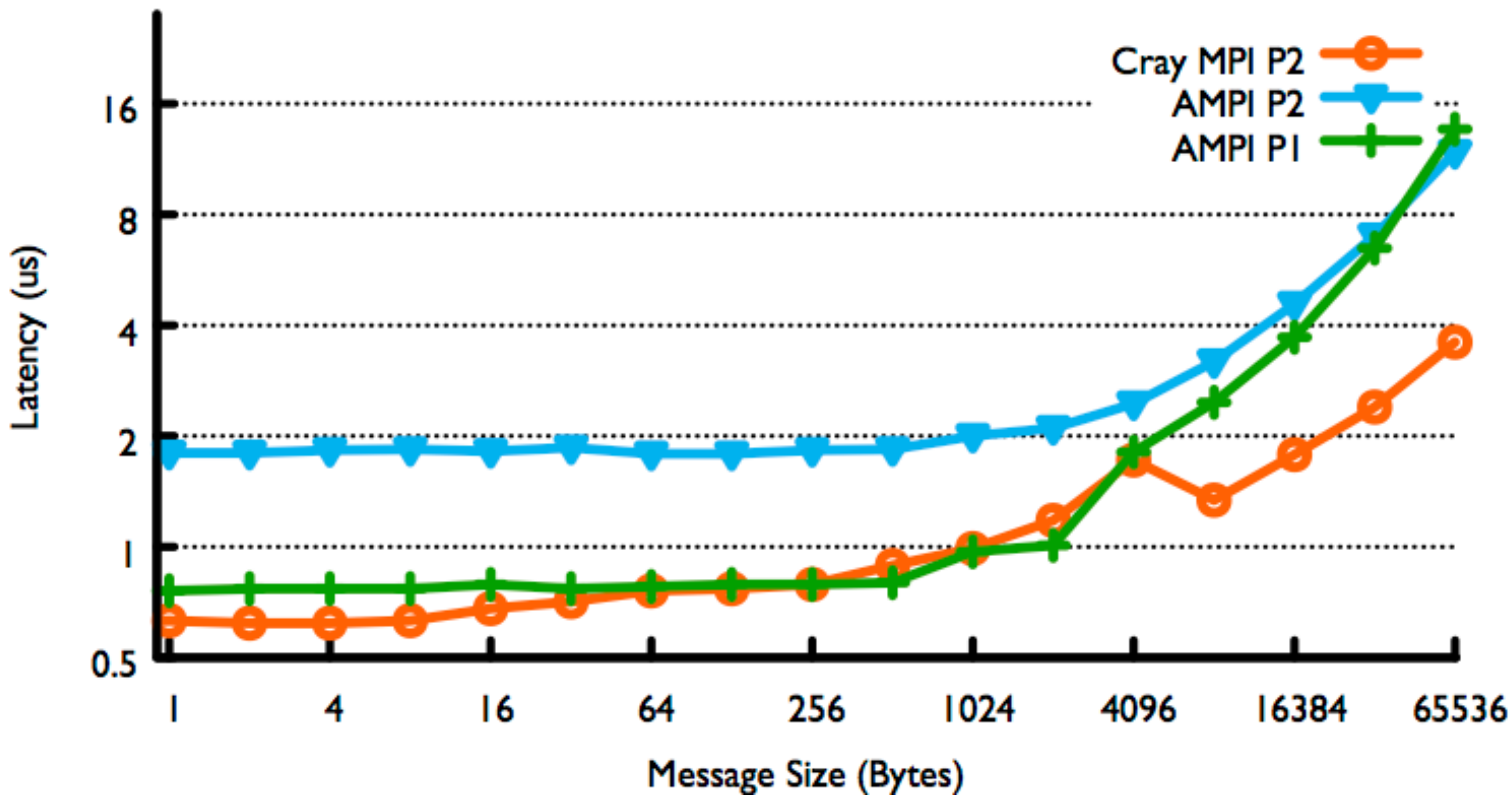
Existing Performance

- Bidirectional Bandwidth on Quartz



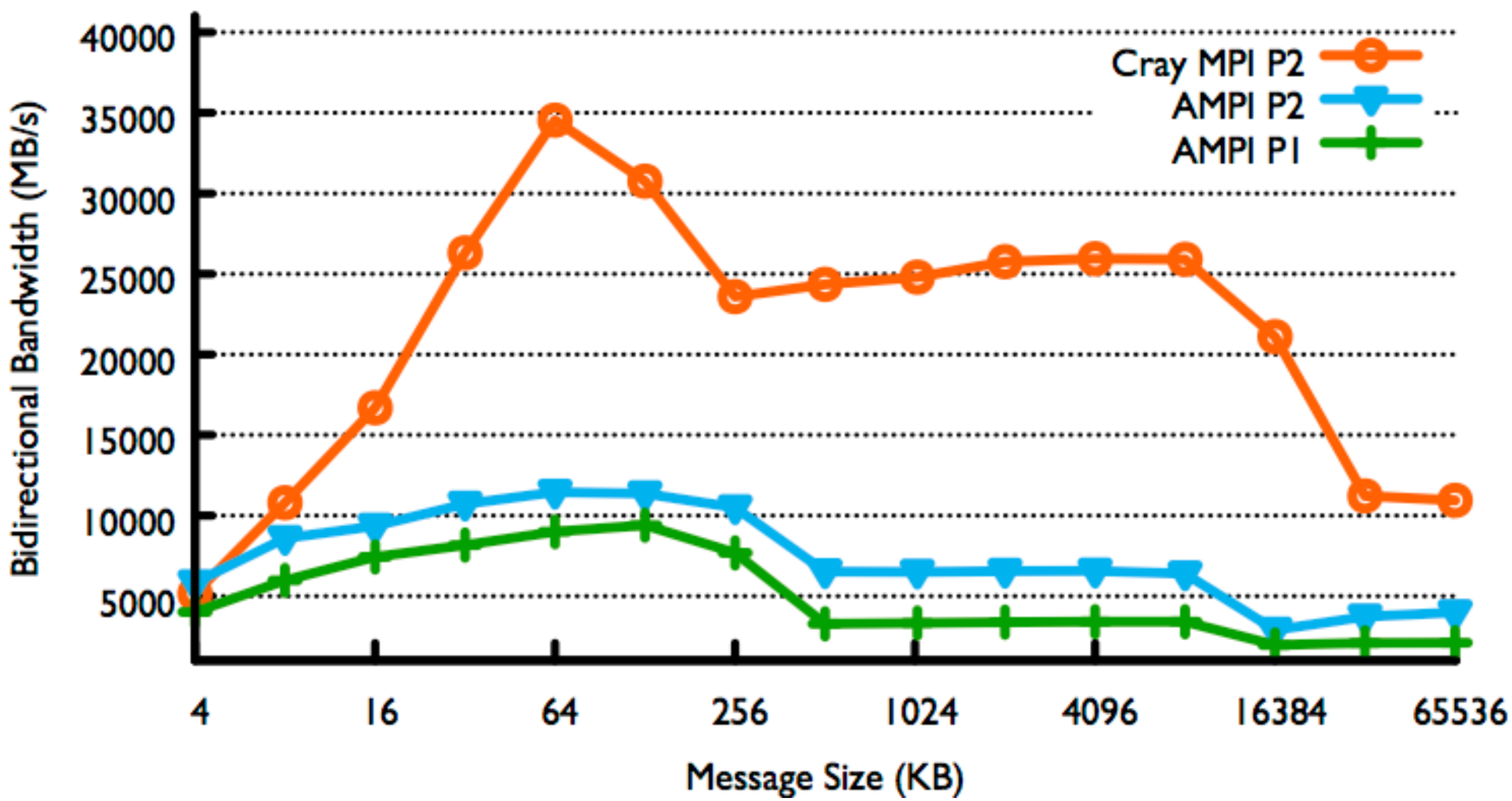
Existing Performance

- Small message latency on Cori-Haswell



Existing Performance

- Bidirectional Bandwidth on Cori-Haswell



Performance Analysis

- Breakdown of P1 time (us) per message on Quartz
  - Scheduling: Charm++ scheduler & ULT ctx
  - Memory copy: message payload movement
  - Other: AMPI message creation & matching

Overhead per message	0-B message	1-MB message
Scheduling	1.02	1.04
Memory copy	0.00	162.86
Other	0.25	1.31

Scheduling Overhead

1. Even for P1, all AMPI messages traveled thru Charm++'s scheduler
   - Use Charm++ *inline* tasks

2. ULT context switching overhead
   - Faster ULT ctx: Boost or QuickThreads ULTs

3. Avoid resuming threads without real progress
   - MPI_Waitall: keep track of "blocked on" reqs

P1 0-B latency: 1.27 us -> 0.66 us

Memory Copy Overhead

- Q: Even with *inline* tasks, AMPI P1 performs poorly for large messages. Why?
    - A: Charm++ messaging semantics do not match MPI's
        - In Charm++, messages are first class objects
        - Users pass ownership of messages to the runtime when sending and assume it when receiving
        - Only app's that can reuse message objects in their data structures can perform "zero copy" transfers

Memory Copy Overhead

- To overcome Charm++ messaging semantics in shared memory, use a rendezvous protocol:
  - Recv'er performs direct (userspace) memcpy from sendbuf to recvbuf
    - Benefit: avoid intermediate copy
    - Cost: sender must suspend & be resumed upon copy completion

P1 1-MB latency: 165 us -> 82 us

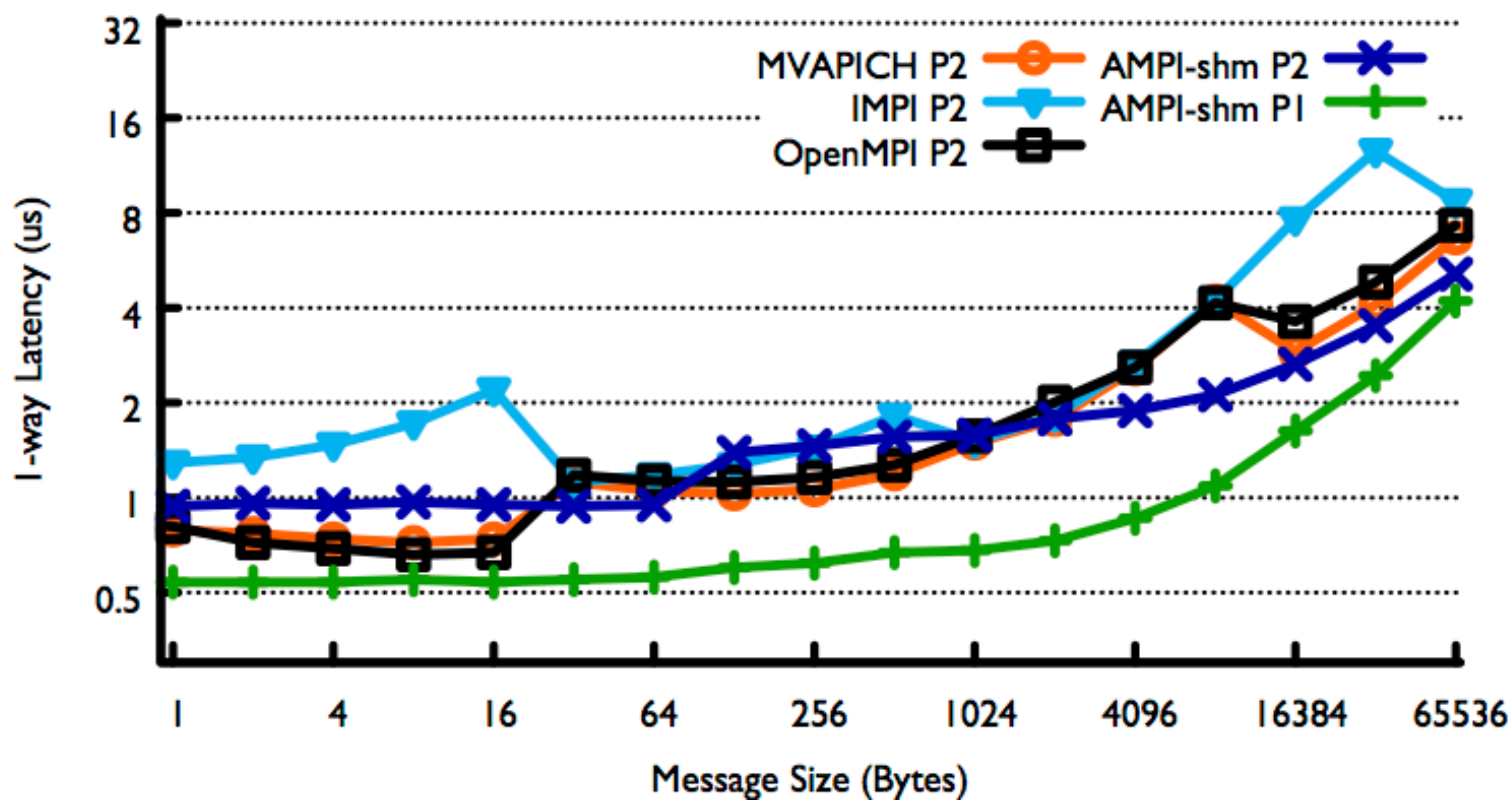
Other Overheads

- Sender-side:
  - Create a Charm++ message object & a request
- Receiver-side:
  - Create a request, create matching queue entry, enqueue in unexpected_msgs or posted_requests
- Solution: use memory pools for fixed-size, frequently-used objects

P1 0-B latency: 0.66 us -> 0.54 us

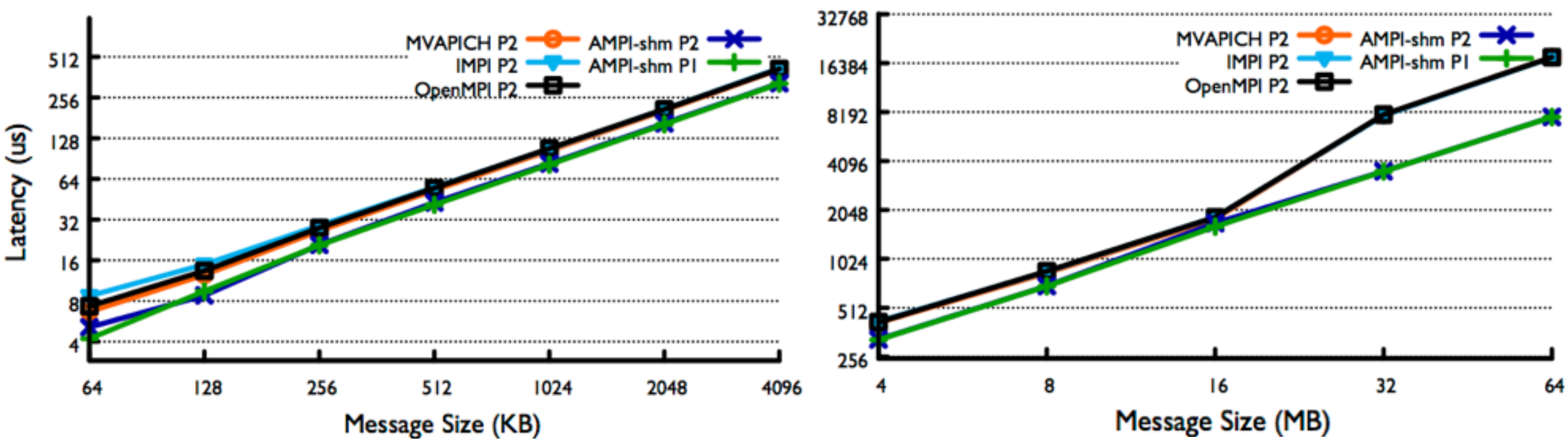
AMPI-shm Performance

- Small message latency on Quartz
  - AMPI-shm P2 faster than other impl's for 2+ KB



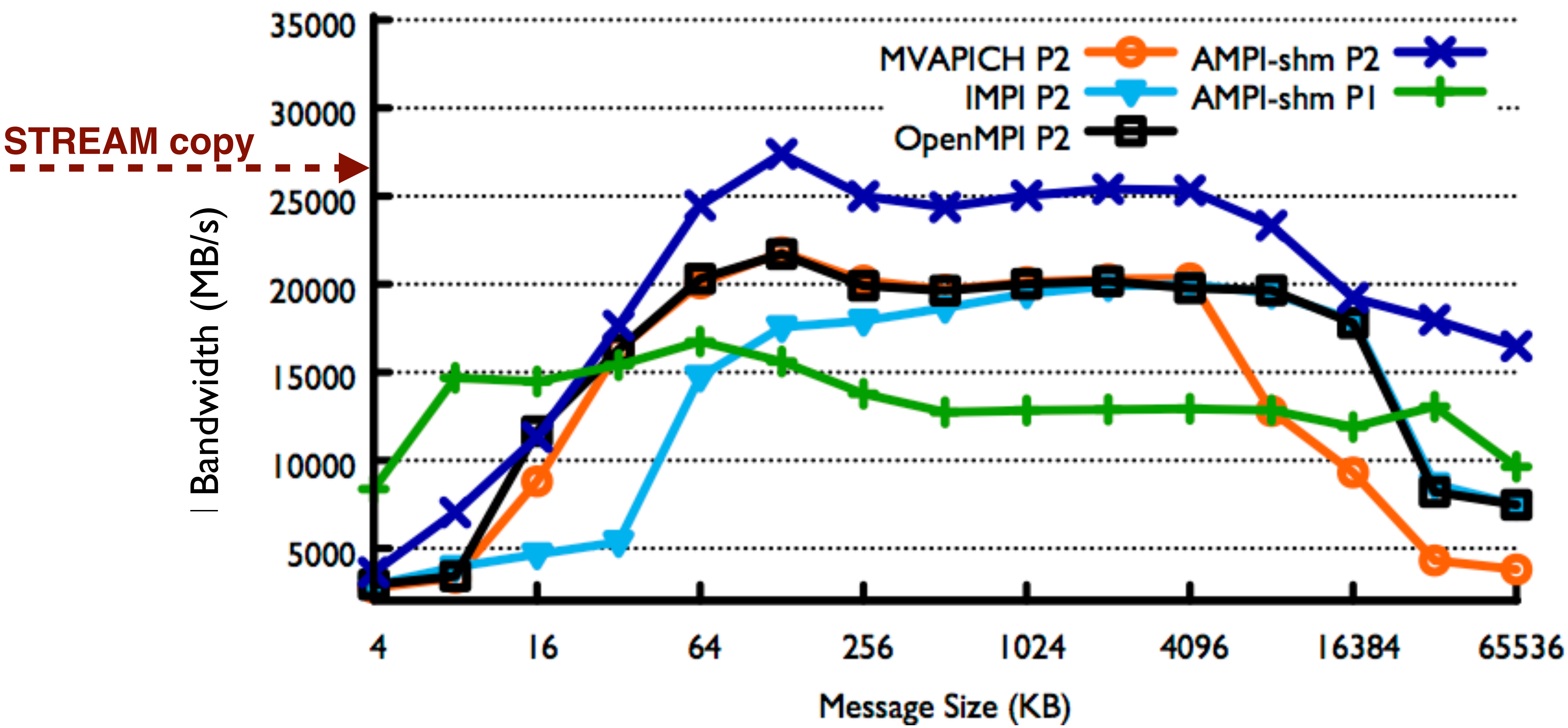
AMPI-shm Performance

- Large message latency on Quartz
  - AMPI-shm P2 fastest for all large messages, up to 2.33x faster than process-based MPIs for 32+ MB



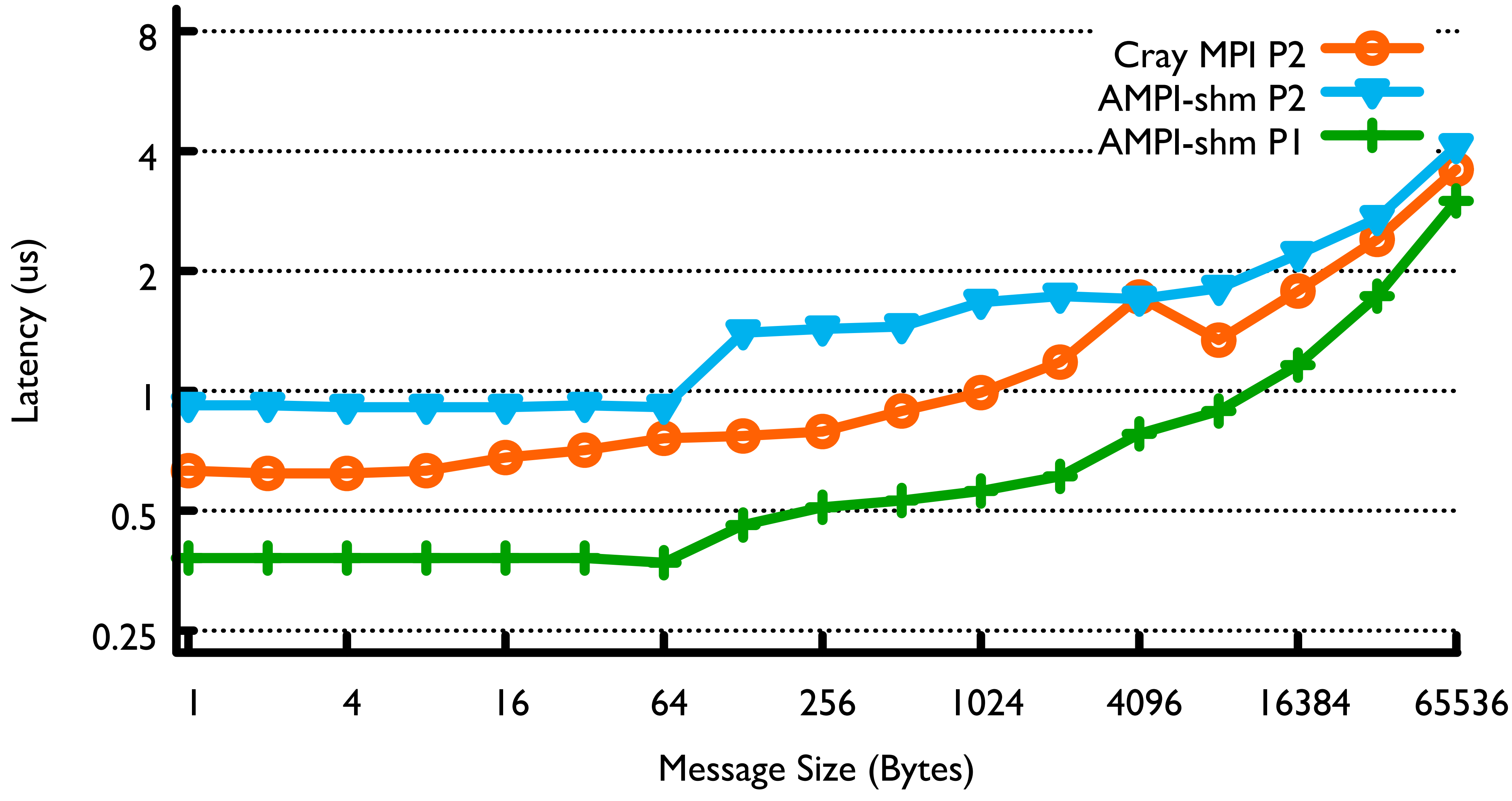
AMPI-shm Performance

- Bidirectional bandwidth on Quartz
  - AMPI-shm can utilize full memory bandwidth
  - 26% higher peak, 2x bandwidth for 32+ MB than others



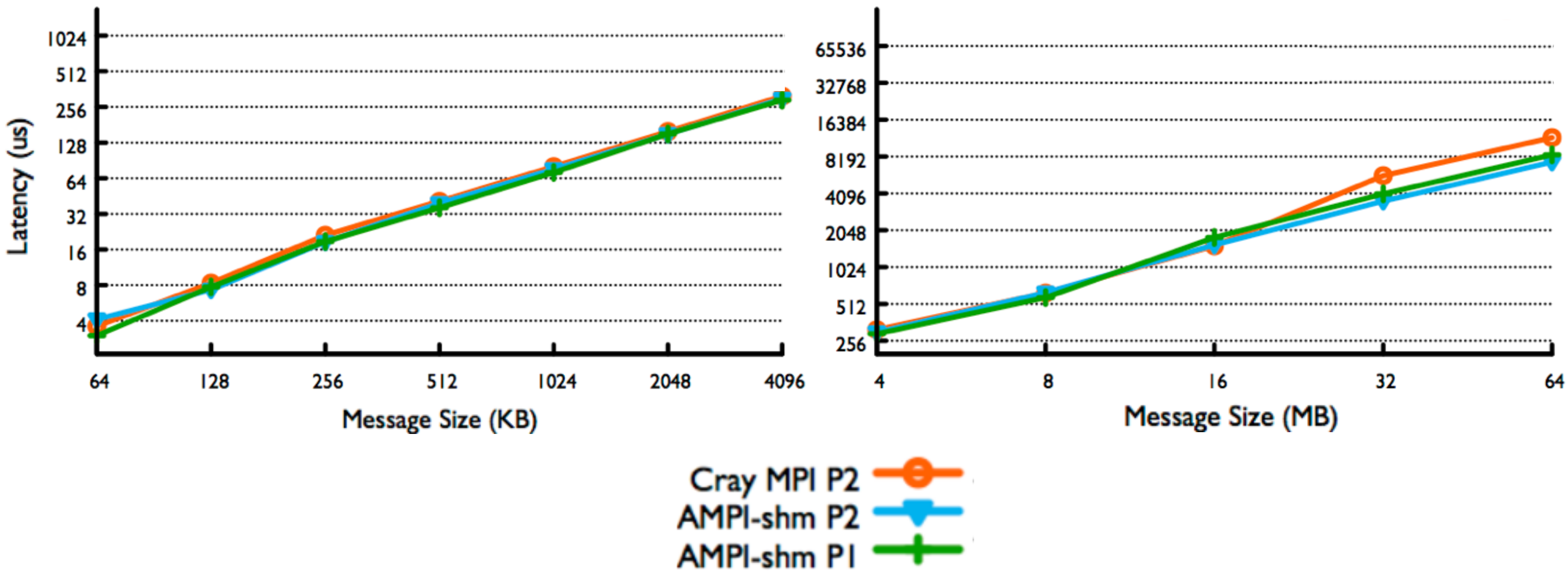
AMPI-shm Performance

- Small message latency on Cori-Haswell



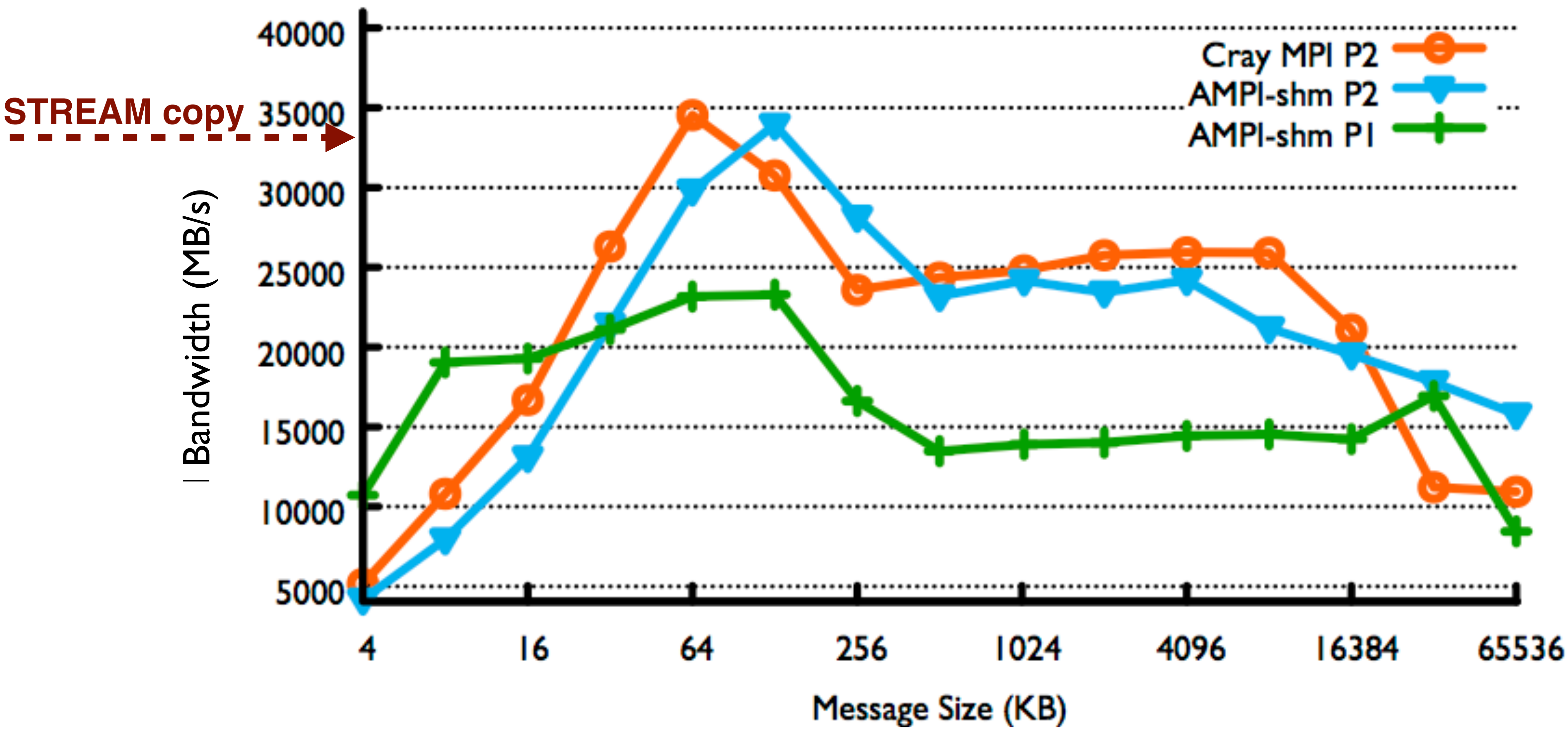
AMPI-shm Performance

- Large message latency on Cori-Haswell
  - AMPI-shm P2 is 47% faster than Cray MPI at 32+ MB



AMPI-shm Performance

- Bidirectional bandwidth on Cori-Haswell
  - Cray MPI on XPMEM performs similarly to AMPI-shm up to 16 MB



Future Work

- User-space shared memory optimizations for: Collectives, Derived Datatypes, RMA, and SHM
- Testing with applications
- Interprocess “zero copy” communication
  - Requires new Charm++ messaging semantics
    - Sender & recver both need completion callbacks
    - New messaging API under development: implementations for OFI, Verbs, uGNI, PAMI, MPI

Summary

- User-space communication offers portable intranode messaging performance
  - Lower latency: 1.5x-2.3x for large msgs
  - Higher bandwidth: 1.3x-2x for large msgs
  - Intermediate buffering unnecessary for medium/large msgs
- Shared-memory aware endpoints can provide lower latency & higher bandwidth for messaging within node

This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.

Questions?

Thank you