Adaptive MPI:
Overview & Recent Developments

Sam White
UIUC

Motivation

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

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

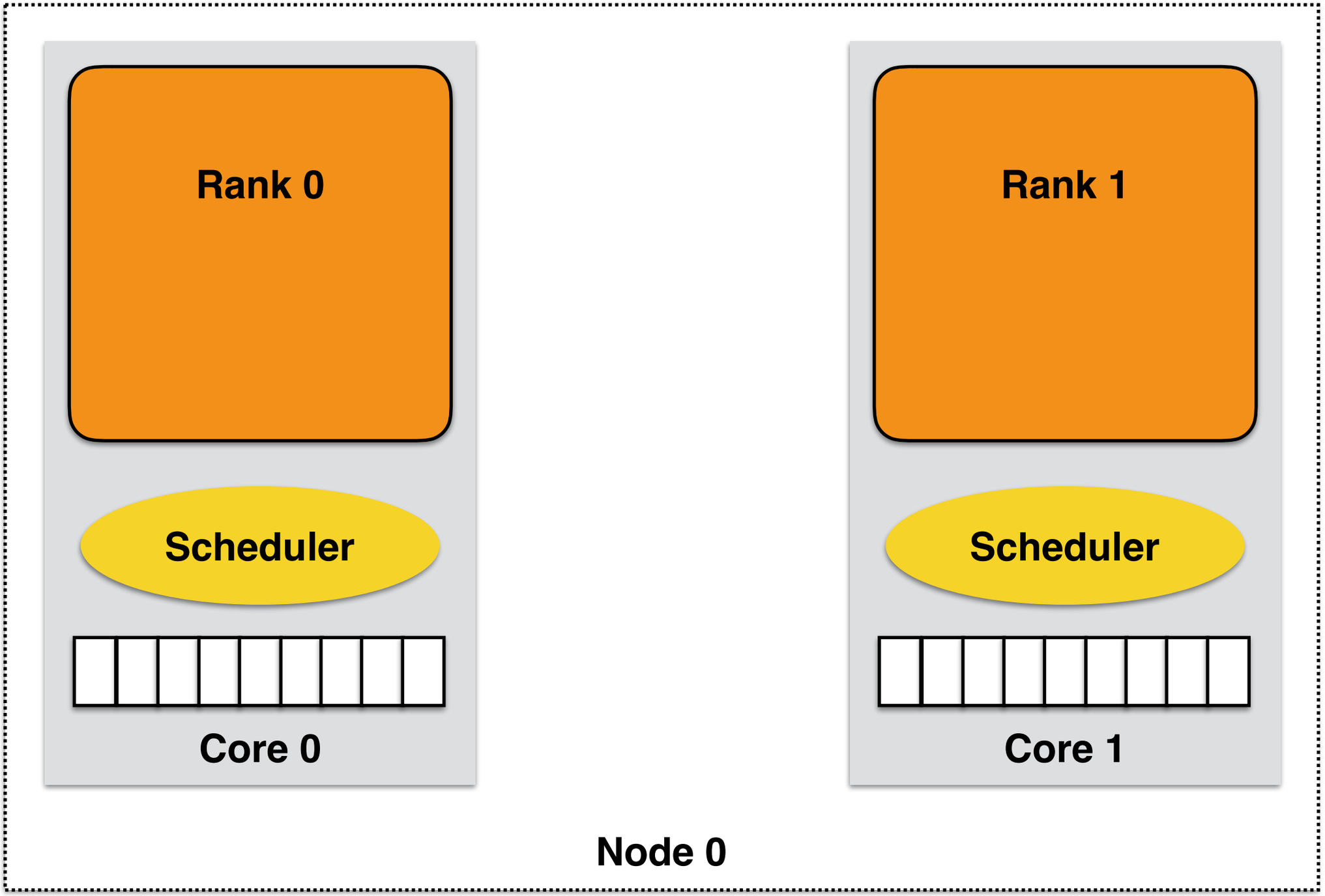
Overview

- Introduction
- Features
- Shared memory optimizations
- Conclusions

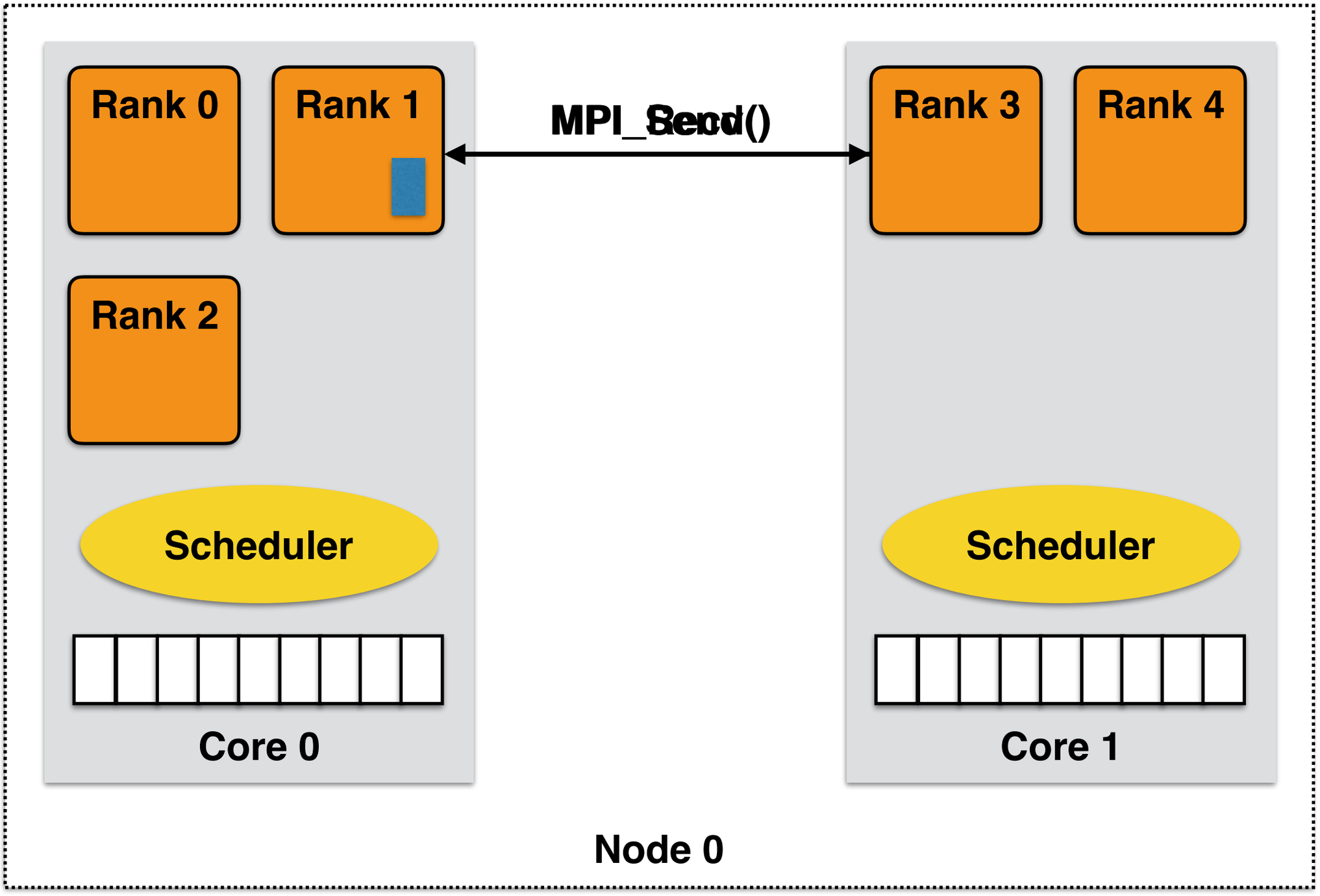
Execution Model

- 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 & fault tolerance

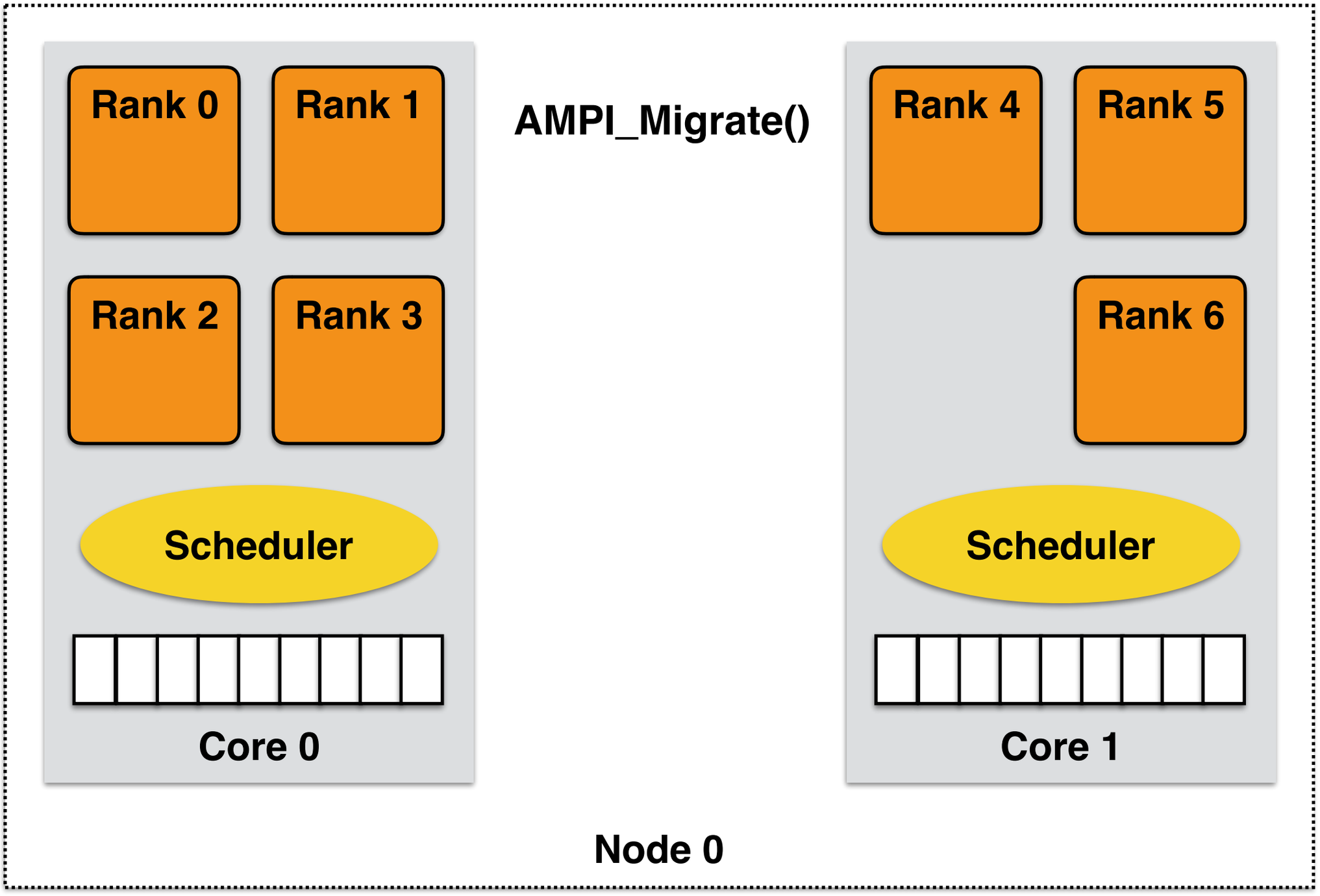
Execution Model



Execution Model



Execution Model



Thread Safety

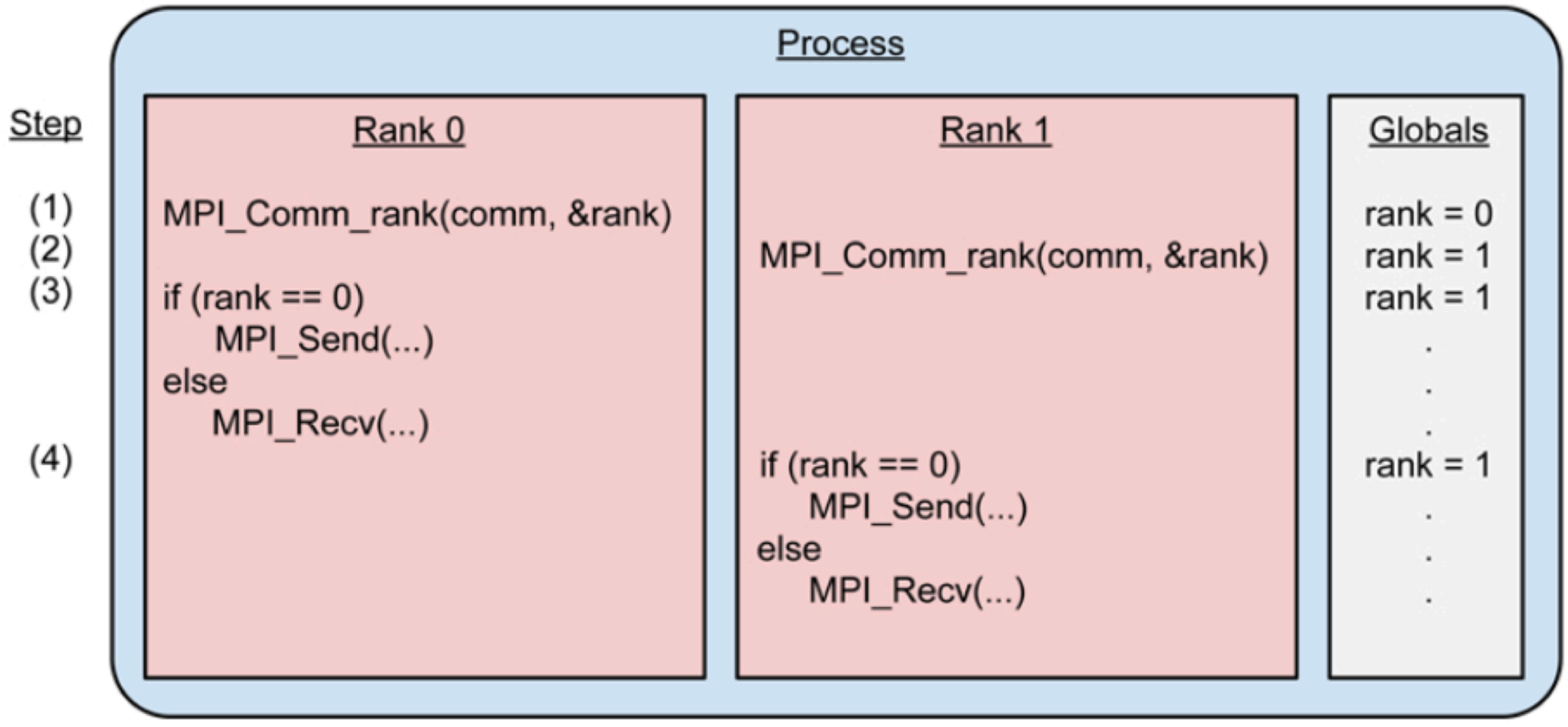
- AMPI virtualizes ranks as threads: is this safe?

```
int rank, size;
int main(int argc, char *argv[]) {

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_Barrier(MPI_COMM_WORLD);

  if (rank==0) MPI_Send(...);
  else if (rank==1) MPI_Recv(...);

  MPI_Finalize();
}
```

Charmworks

Thread Safety

- AMPI virtualizes ranks as threads: is this safe?
  No, global variables are defined per process



Thread Safety

- AMPI programs are MPI programs without mutable global variables
- Solutions:
  1. Refactor the application to not use globals/statics, instead pass them around on the stack
  2. Swap ELF Global Offset Table entries at ULT context switch
  3. Swap Thread Local Storage pointer during ctx
     - Tag unsafe vars with C/C++ ‘thread_local’ or OpenMP ‘threadprivate’, the runtime manages TLS
     - Work in progress: have the compiler privatize them for you, i.e., *icc -fmpc-privatize*

Charmworks

Conversion to AMPI

- AMPI programs are MPI programs, with 2½ caveats:

1. Without mutable global/static variables
   - Or with them properly handled
2. Possibly with calls to AMPI's extensions
   - *AMPI_Migrate()*

2½. Fortran main & command line args

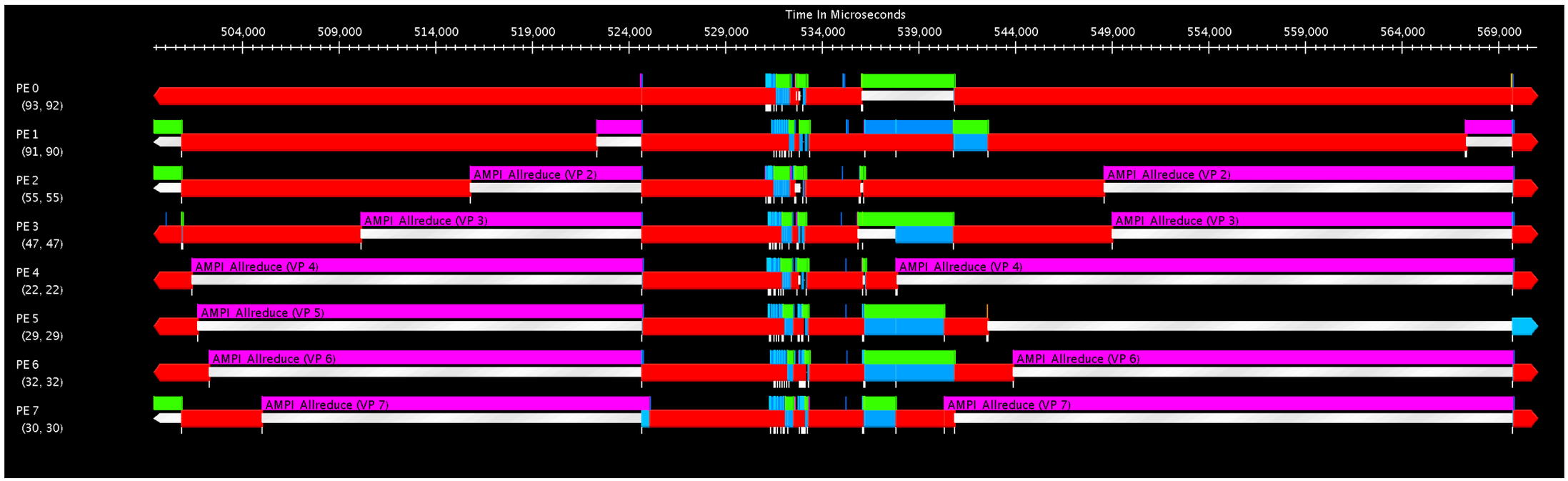
AMPI Fortran Support

- AMPI implements the F77 and F90 MPI bindings
- MPI -> AMPI Fortran conversion:
  - Rename ‘program main’ -> ‘subroutine mpi_main’
  - AMPI_ command line argument parsing routines
  - Automatic arrays: increase ULT stack size

Charmworks

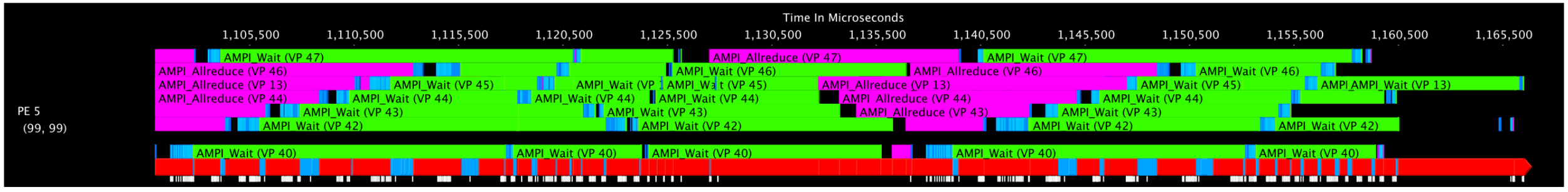
Overdecomposition

- Bulk-synchronous codes often underutilize the network with compute/communicate phases
  - LULESH v2.0:



Overdecomposition

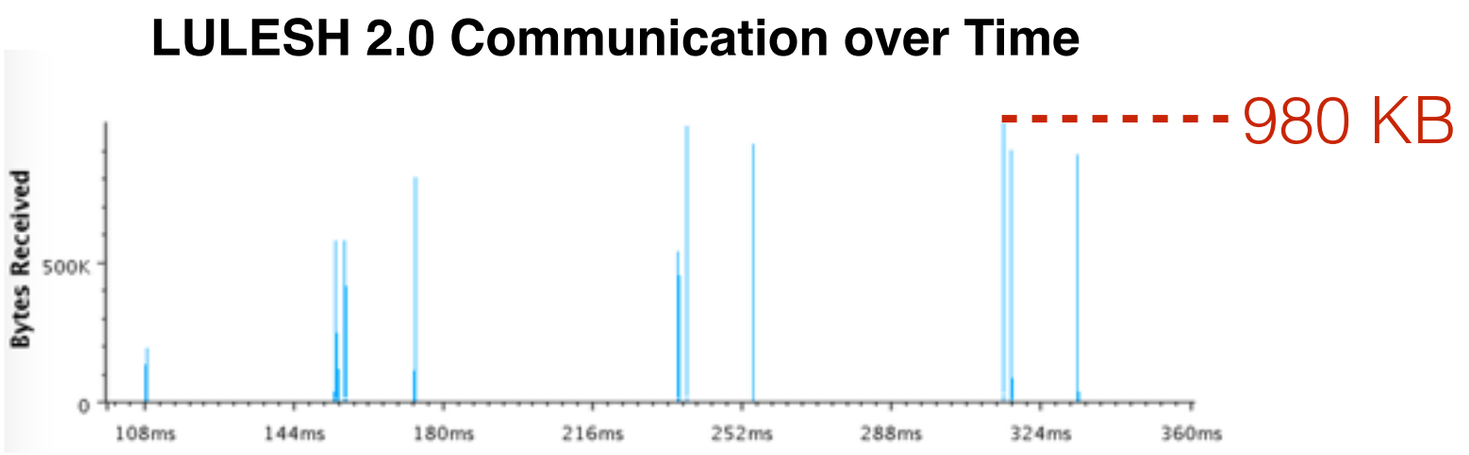
- With overdecomposition, overlap communication of one rank with computation of others on its core



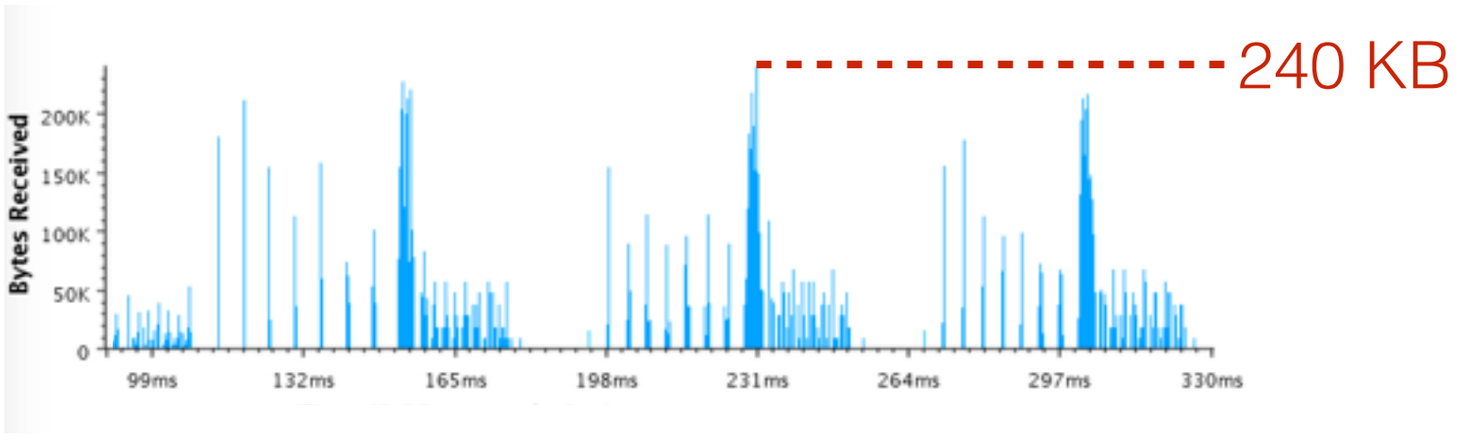
Message-driven Execution

- Overdecomposition spreads network injection over the whole timestep

1 rank/core



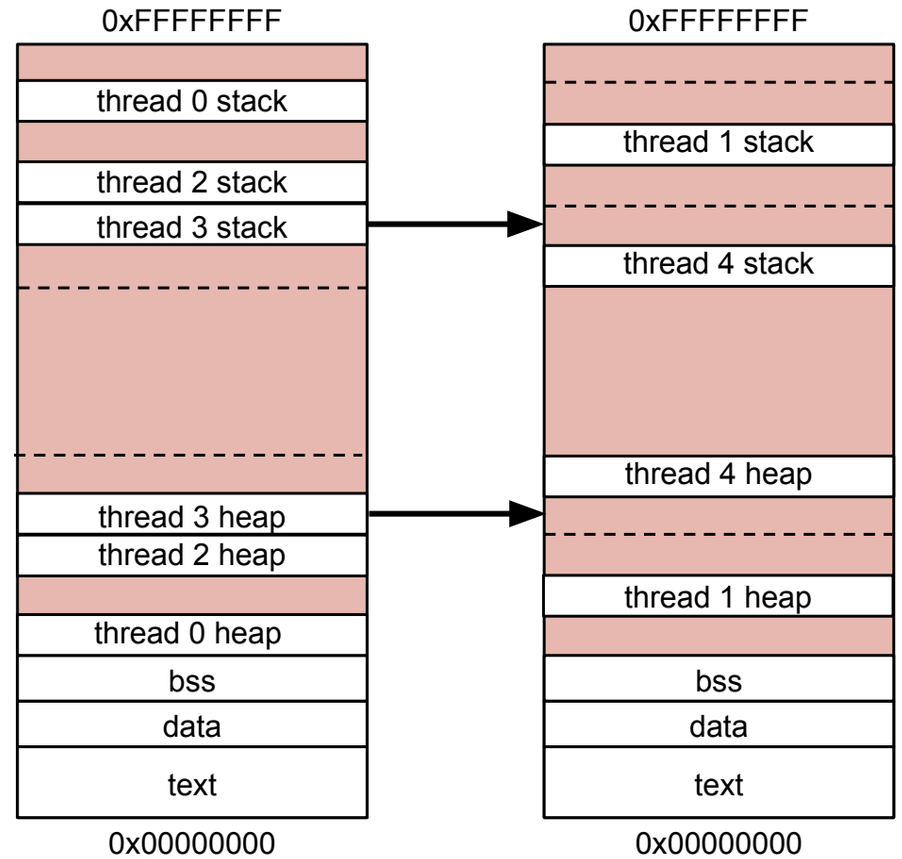
8 ranks/core



Charmworks

Migratability

- AMPI ranks are migratable at runtime between address spaces
  - User-level thread stack + heap
- Isomalloc memory allocator makes migration automatic
  - No user serialization code
  - Works everywhere but BGQ & Windows



Load Balancing

- To enable load balancing in an AMPI program:
  1. Insert a call to *AMPI_Migrate(MPI_Info)*
     - Info object is LB, Checkpoint, etc.
  2. Link with Isomalloc and a load balancer:

     ampicc -memory isomalloc -module CommonLBs

  3. Specify the number of virtual processes and a load balancing strategy at runtime:

     srun -n 100 ./pgm +vp 1000 +balancer RefineLB

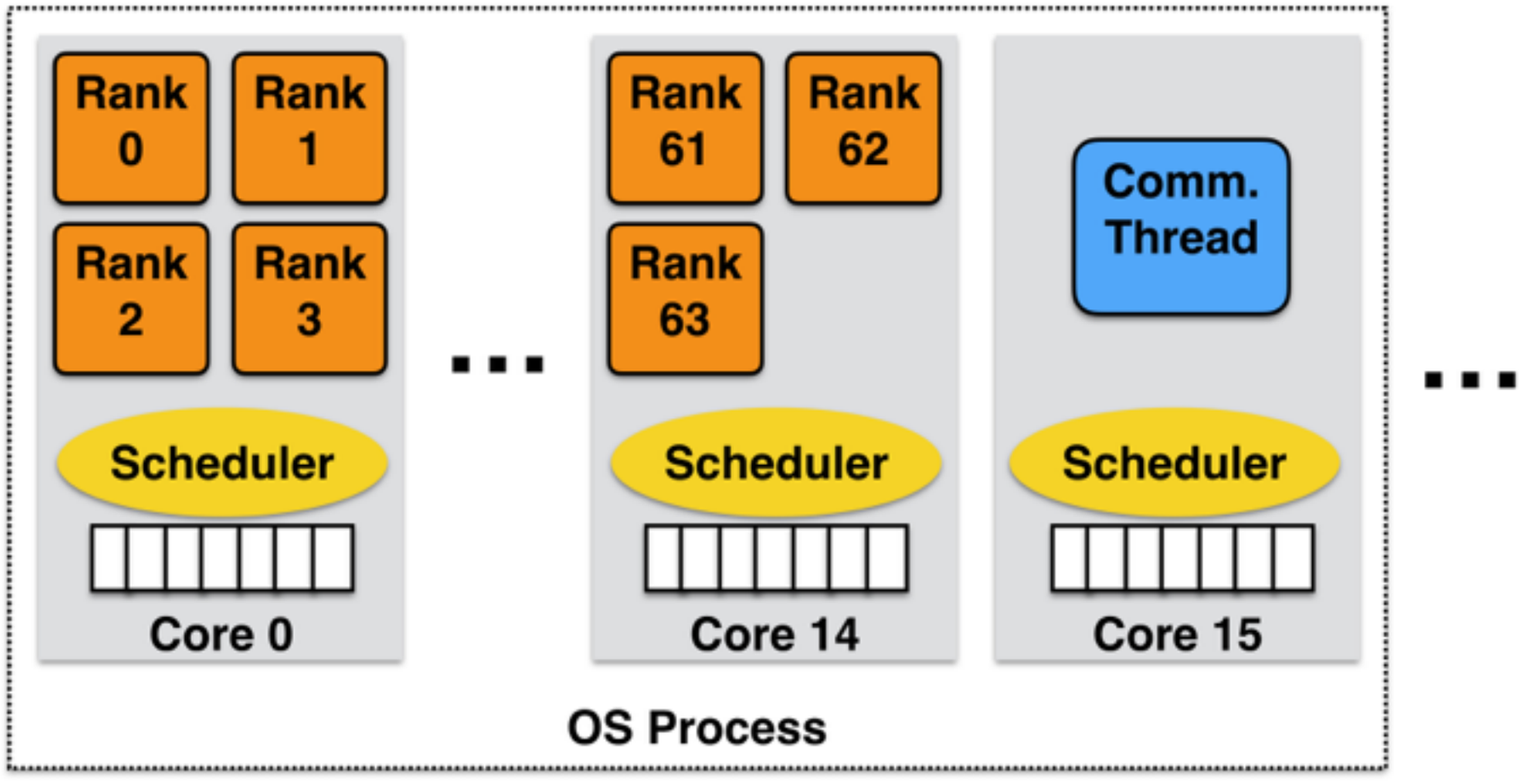
Charmworks

Recent Work

- AMPI can optimize for communication locality
  - Many ranks can reside on the same core
    - Same goes for process/socket/node
  - Load balancers can take communication graph into consideration

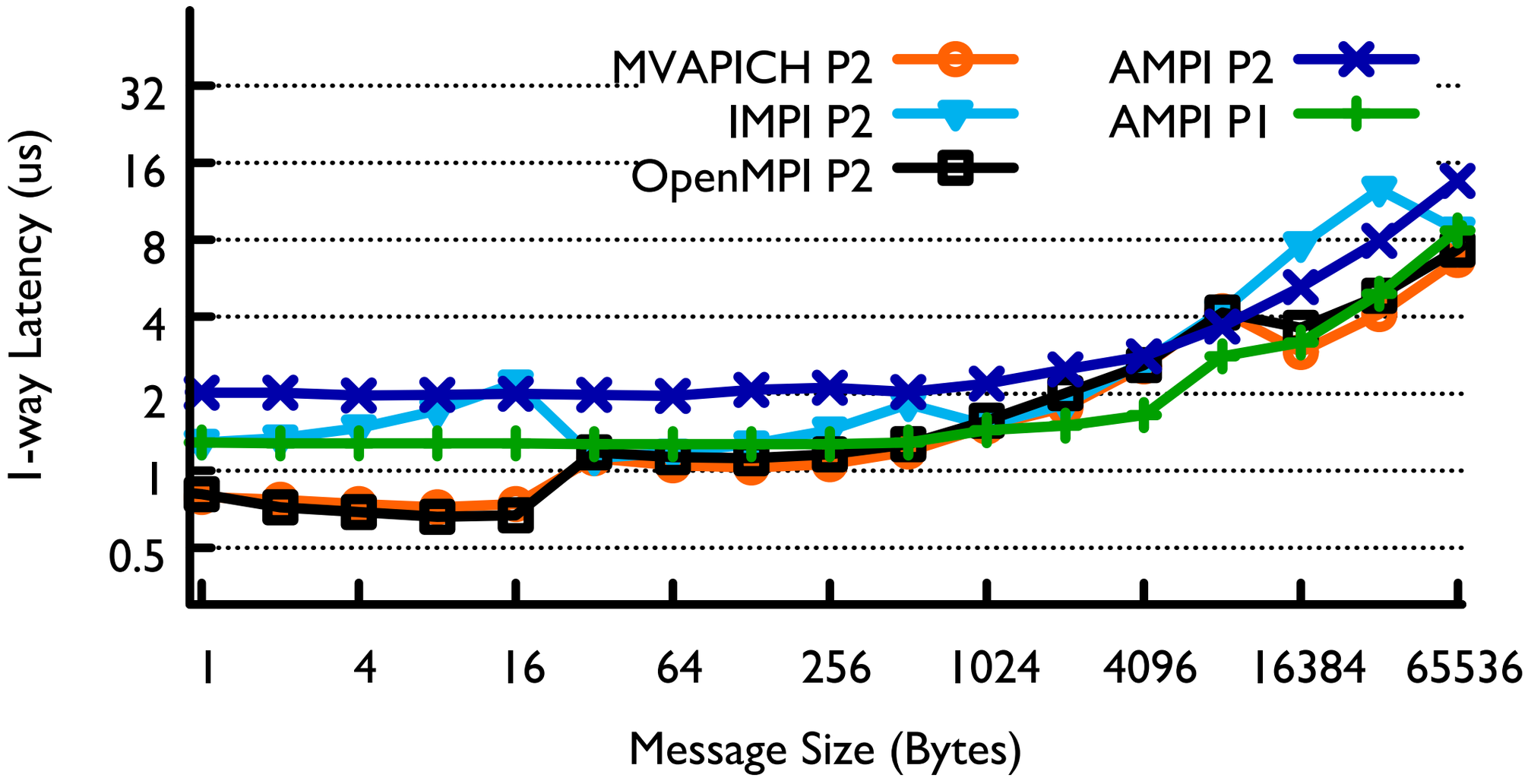
AMPI Shared Memory

- Many AMPI ranks can share the same OS process



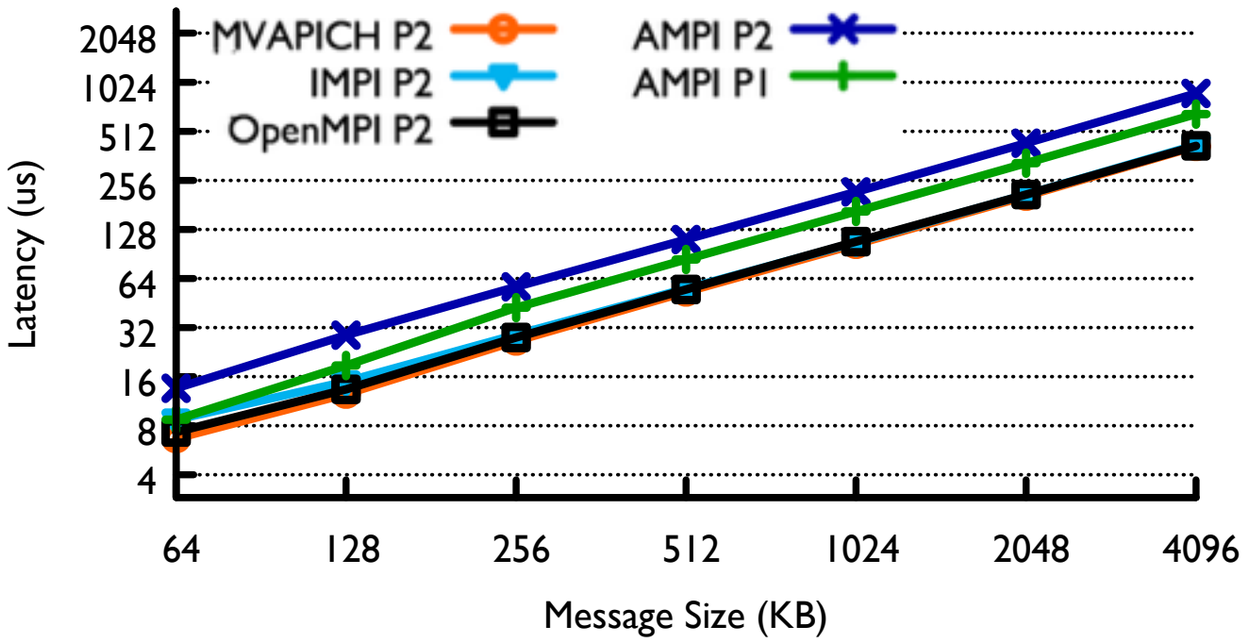
Existing Performance

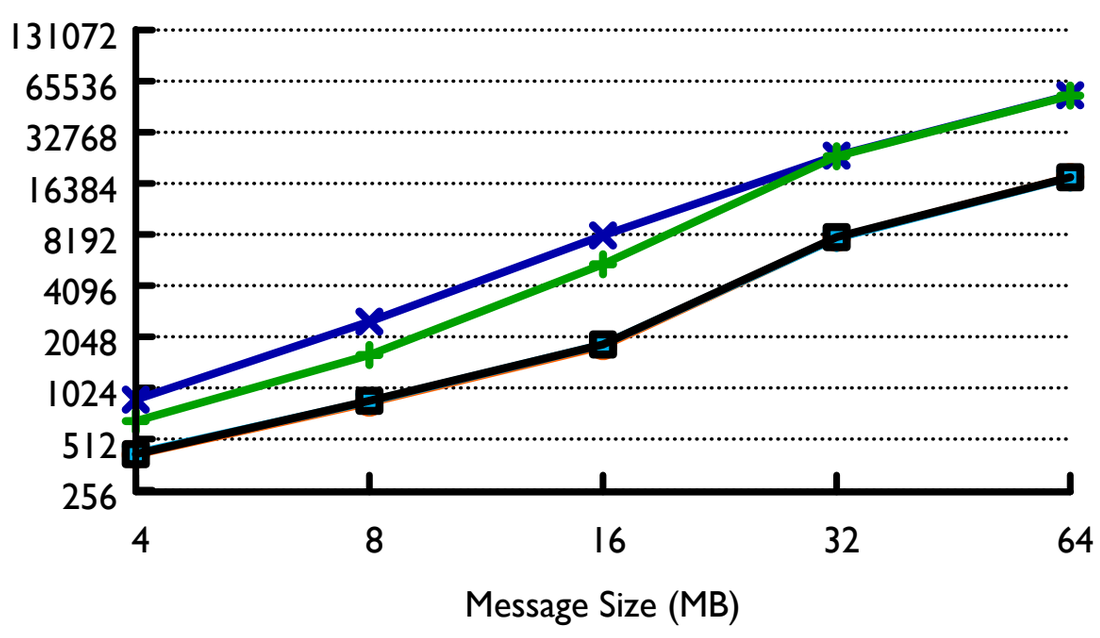
- Small message latency on Quartz (LLNL)



Existing Performance

- Large message latency on Quartz





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

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: synchronization, sender must suspend & be resumed upon copy completion

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

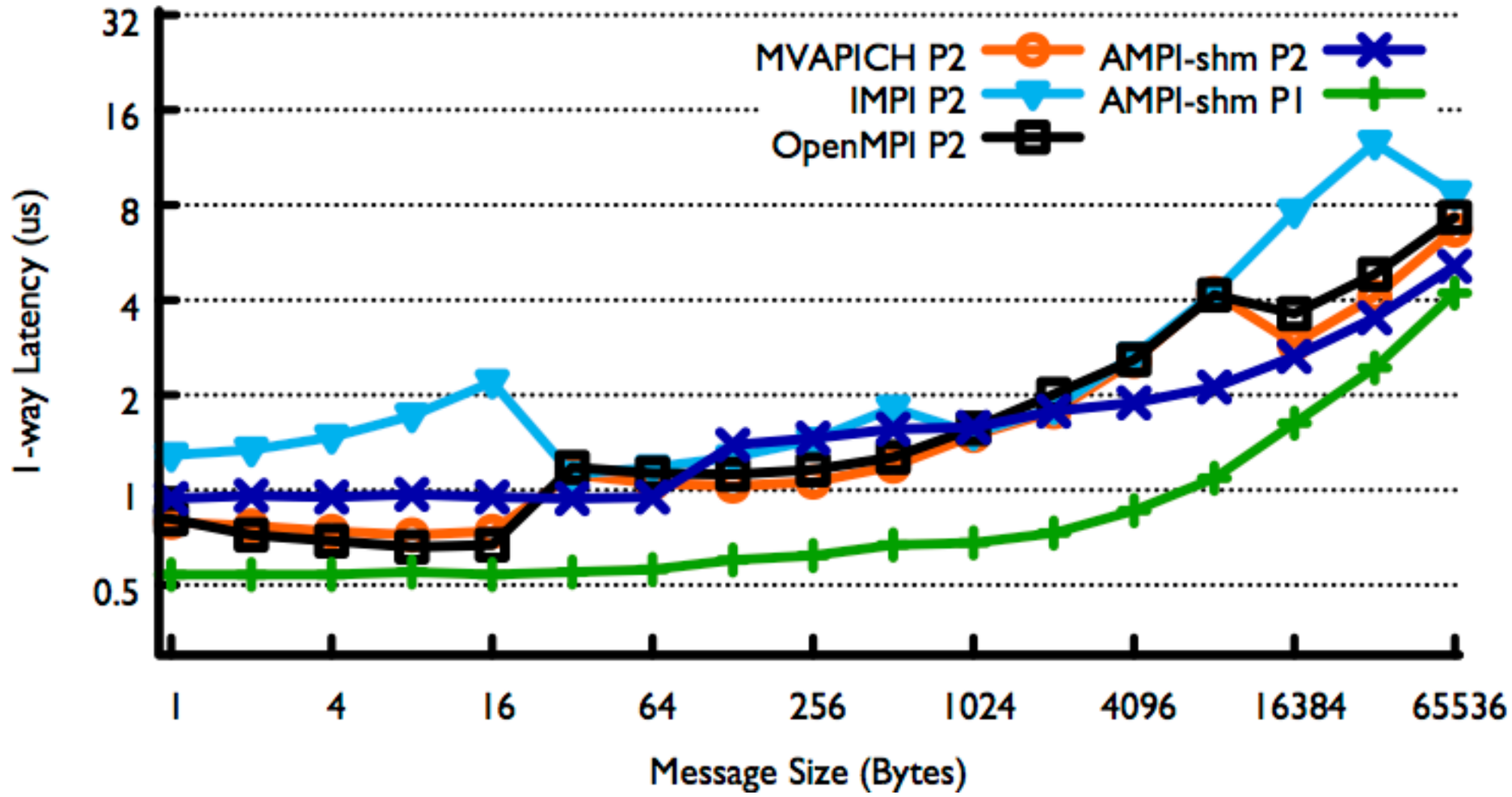
Other Overheads

- Sender-side:
  - Create a Charm++ message object & a request
- Receiver-side:
  - Create a request, create matching queue entry, dequeue from unexpectedMsgs or enqueue in postedReqs
- Solution: use memory pools for fixed-size, frequently-used objects
  - Optimize for common usage patterns, i.e. MPI_Waitall with a mix of send and recv requests

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

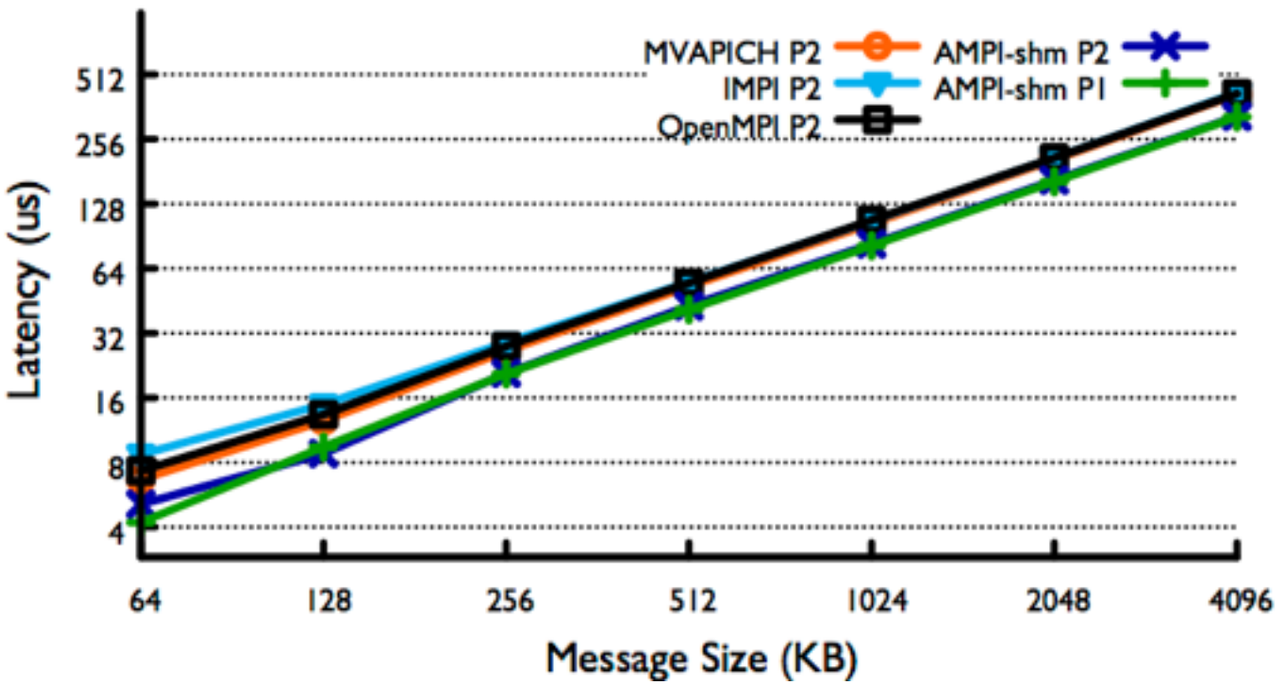
AMPI-shm Performance

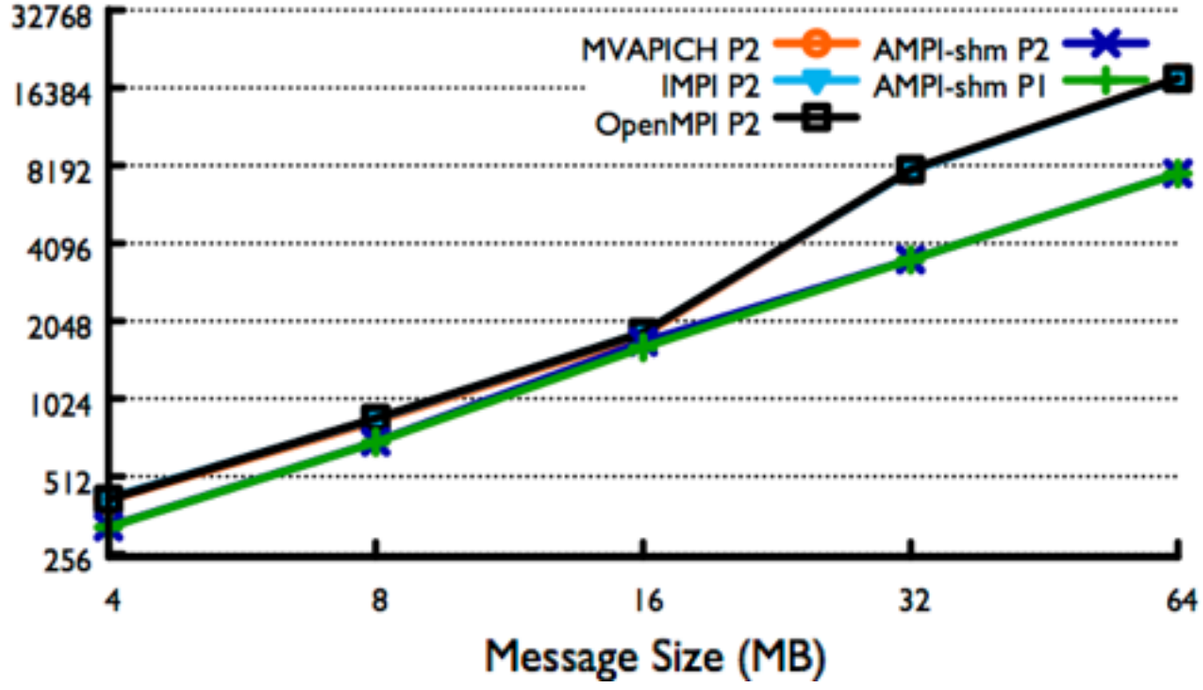
- Small message latency on Quartz
  - AMPI-shm P2 faster than other impl's for 2+ KB



AMPI-shm Performance

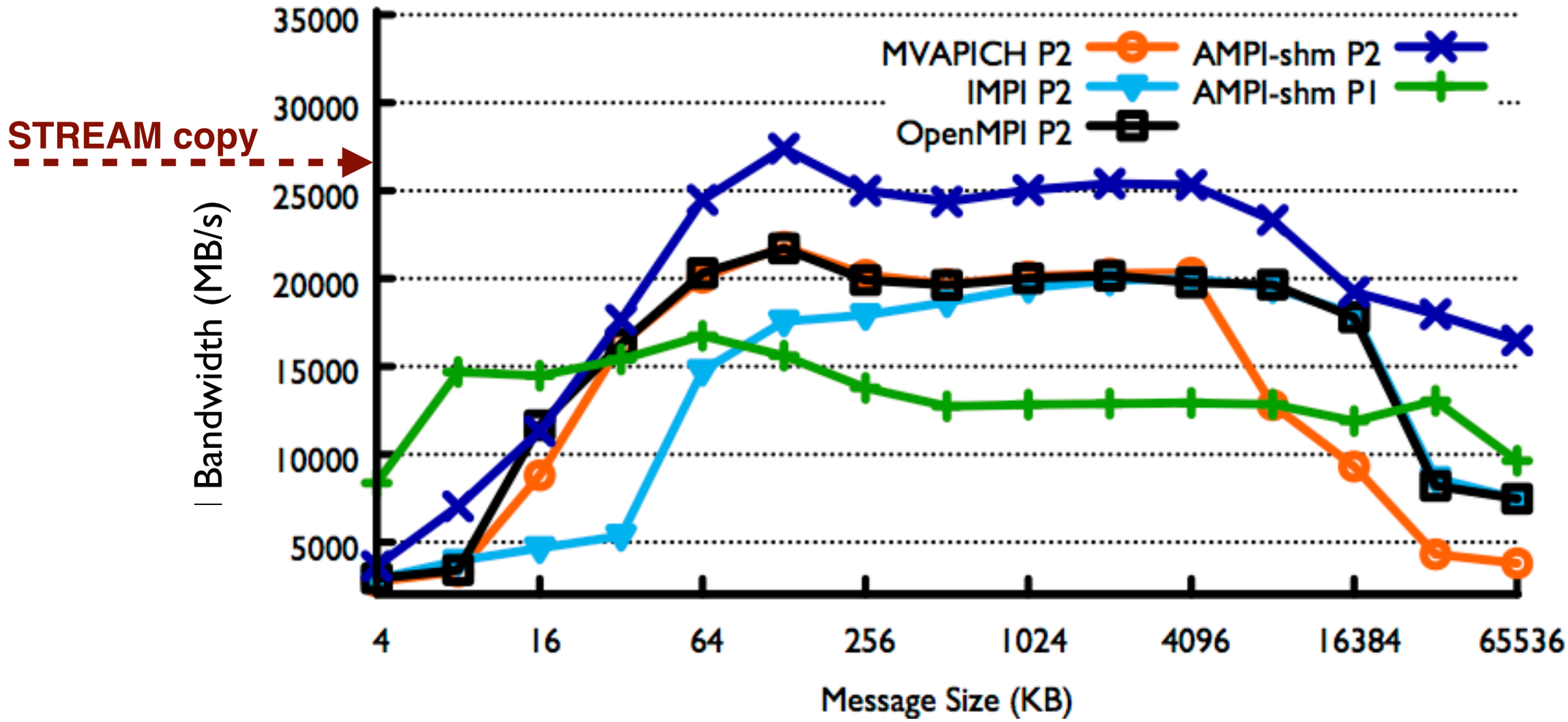
- Large message latency on Quartz
  - AMPI-shm P2 fastest for all large messages, up to 2.33x faster than process-based MPIs for 32+ MB





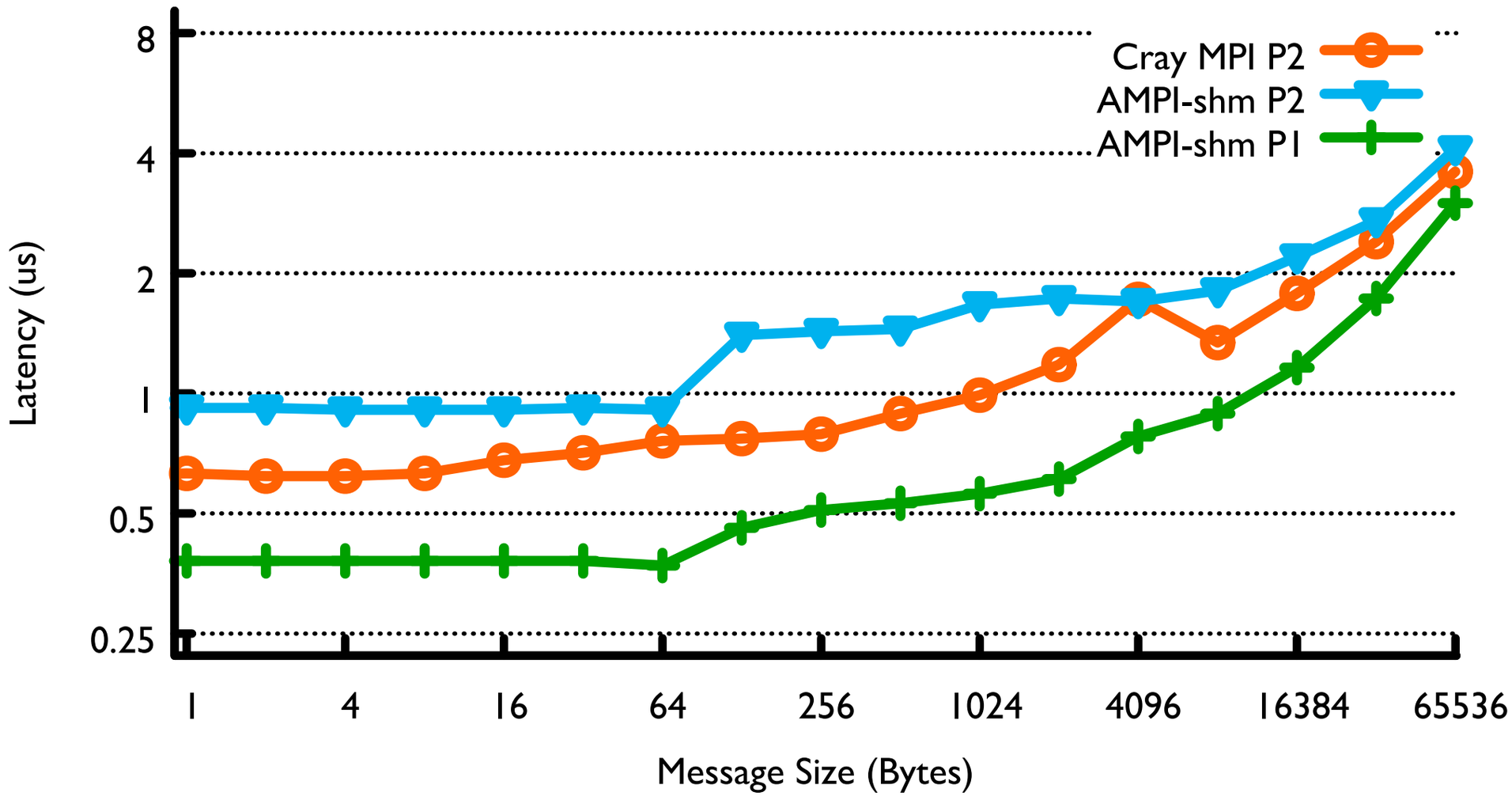
AMPI-shm Performance

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



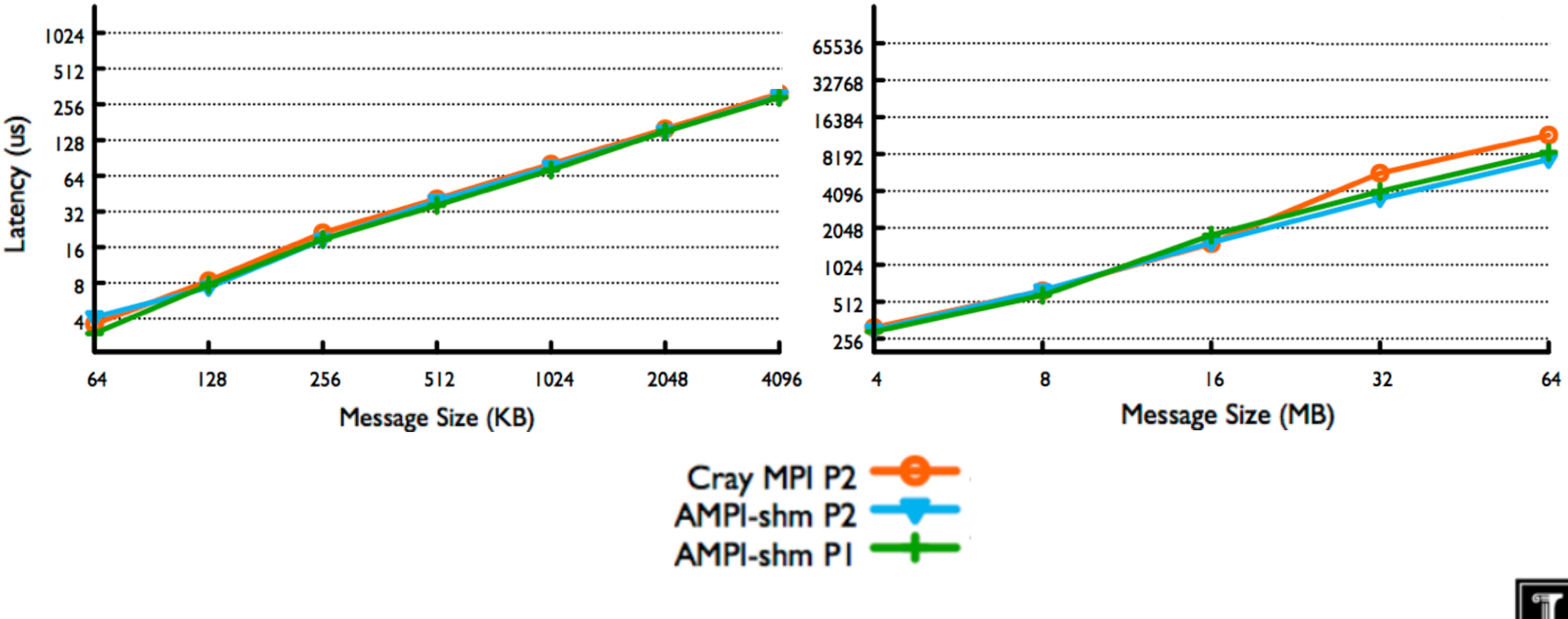
AMPI-shm Performance

- Small message latency on Cori-Haswell



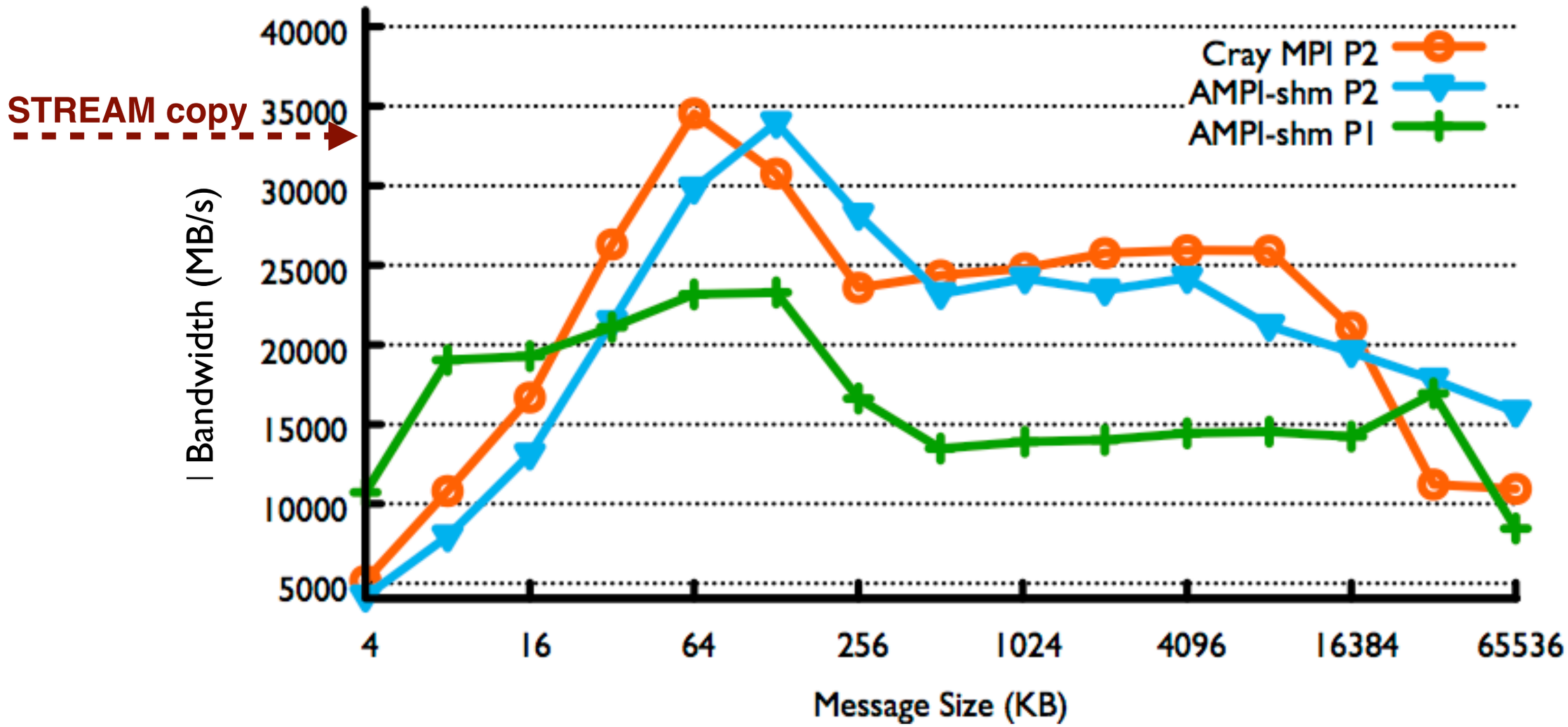
AMPI-shm Performance

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



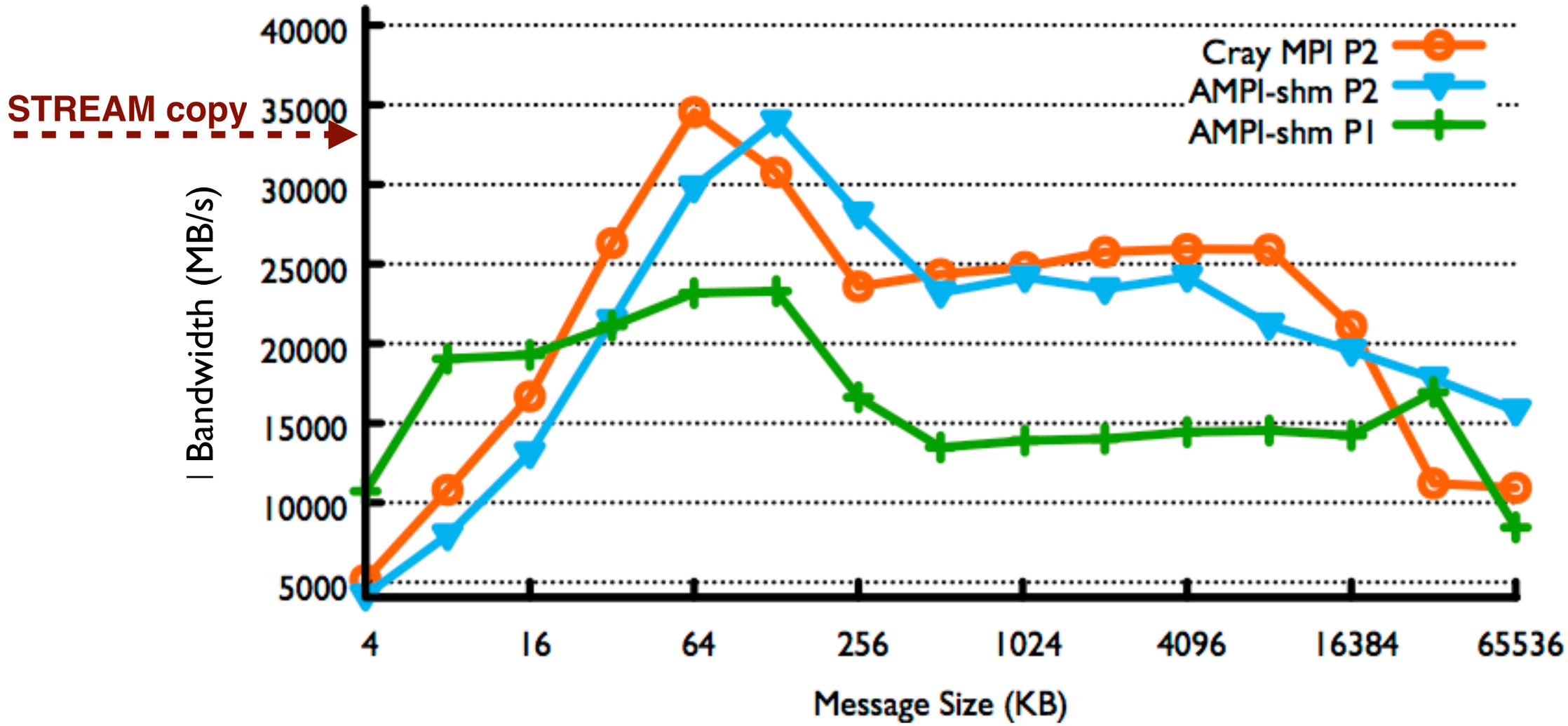
AMPI-shm Performance

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



AMPI-shm Performance

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



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

Conclusions

- AMPI provides application-independent runtime support for existing MPI applications:
  - Overdecomposition
  - Latency tolerance
  - Dynamic load balancing
  - Automatic fault detection & recovery

- See the AMPI manual for more info

Charmworks

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