NUMA-Aware Shared-Memory Collective Communication for MPI

Shigang Li

Torsten Hoefler

Marc Snir

Presented By: Shafayat Rahman

Motivation

- Number of cores per node keeps increasing
- So it becomes important for MPI to leverage shared memory for **intranode** communication.

Introduction

- The paper investigates:
  - The design and optimizations of MPI collectives
  - For cluster of NUMA nodes

- **Keywords**: MPI, Multithreading, MPI_Allreduce, Collective Communication, NUMA

Some basic definitions

- **MPI**: a standardized and portable message passing system
- **Collective Functions**: functions involving communication among all processes within a process group. Example: MPI_Bcast call
- **NUMA**: Non Uniform Memory Access. A computer memory design used in multiprocessing where the memory access time depends on the memory location relative to the processor.

Contributions

- Developed performance model for collective communication using shared memory
- Developed several algorithm for various collectives.
- Conducted experiments on Xeon X5650 and Operton 6100 Infiniband clusters
- Compared their shared-memory allreduce algorithm with several traditional MPI implementations: Open MPI, MPICH2, and MVAPICH2

Findings

- Different algorithms dominate for short vectors and long vectors
- On a 16-node Xeon cluster and 8-node Operton cluster, their implementation achieves on average 2.5X and 2.3X sppedup over MPICH2.

Background

- Applications using MPI often run multiple MPI processes in each node.
- With increasing number or cores per node and non-uniform memory, performance of MPI collectives depend more and more on **intranode** communications.

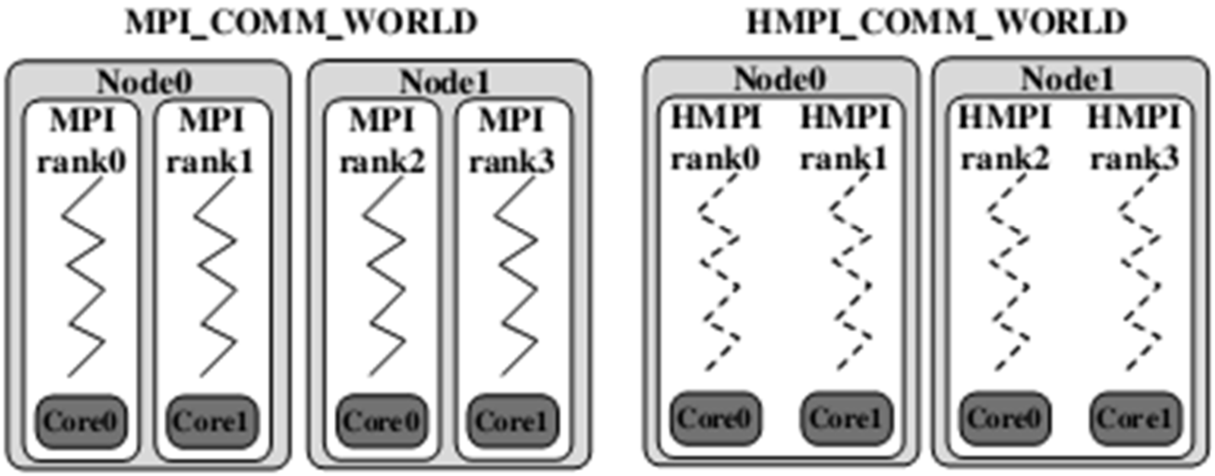
Background

- Current implementations of MPI collectives take advantage of shared memory in two ways:
  - Collectives are built using point-to-point message passing, which uses shared memory as a transport layer inside a node. (MPICH2)
  - Collectives are implemented directly on top of shared memory. Data is copied from user space to shared memory space. (Open MPI and MVAPICH2).
- The second approach reduces the number of memory transfers but still requires extra data movement. Plus, shared memory is a limited system resource.

HMPI: Hybrid MPI

- Replacing the process-based rank design with a thread-based rank.
- All MPI ranks (threads) on a node are running within the same address space.
- Utilizes existing MPI infrastructure for **internode** communication
- A similar approach is to use a globally shared heap for all MPI processes on a node (XPMEM)

HMPI: Hybrid MPI

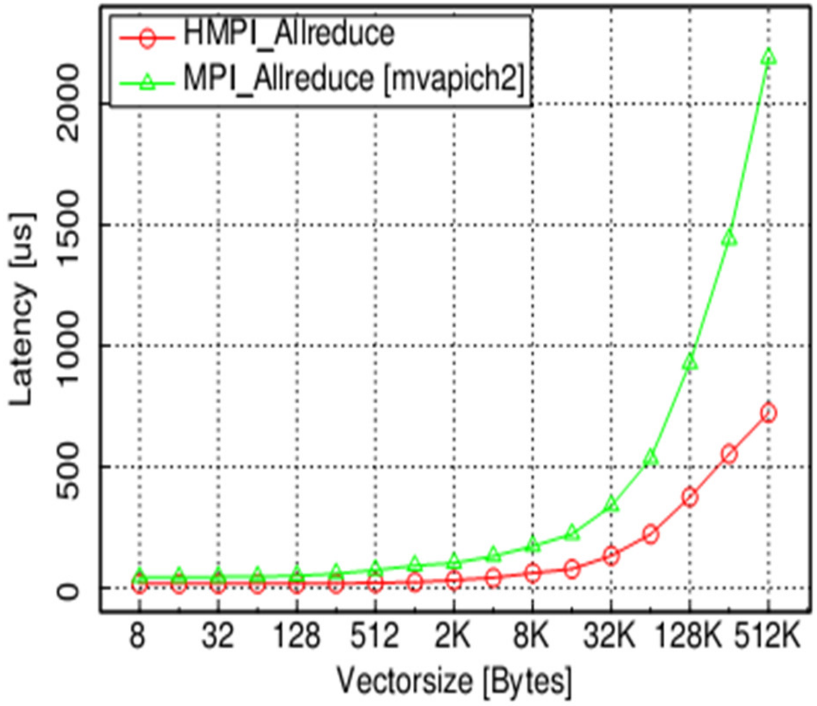


Contributions

- Designed NUMA-aware algorithms for thread-based HMPI_Allreduce on cluster of NUMA nodes
- Showed a set of motifs and techniques to optimize collective operations on multicore architecture
- Established performance models based on memory access latency and bandwidth to select the best algorithm for different vector sizes
- Performed a detailed benchmarking study to assess the benefits of using their algorithms over existing approaches

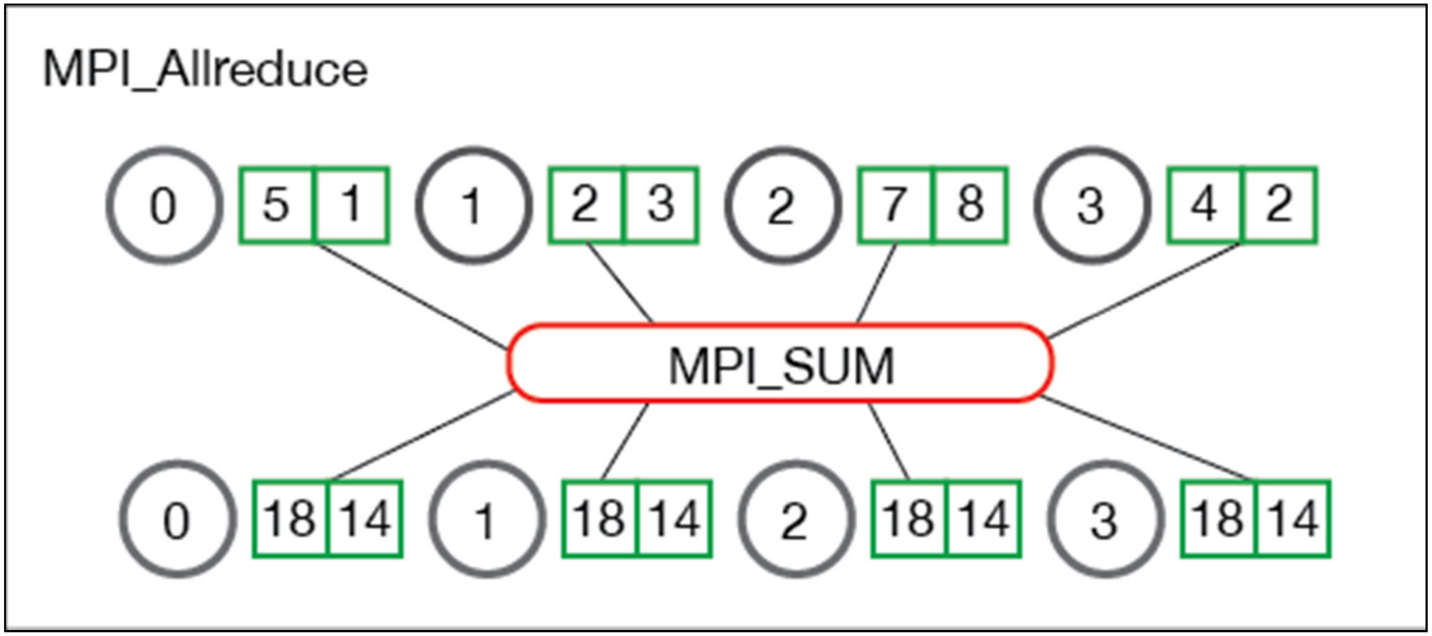
Contributions

- They specifically selected MPI_Allreduce to show the benefits of HMPI.



MPI_Allreduce

- A combination of MPI_Reduce and MPI_broadcast

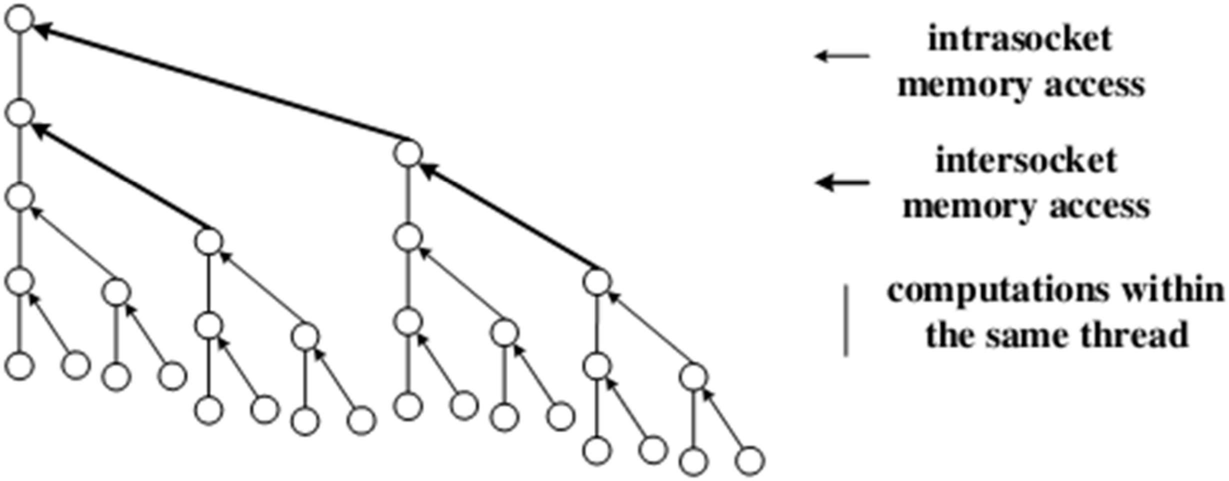


Performance Model

- Ignored internode communication and focused on intranode communication.
- Each algorithm is designed and implemented hierarchically between **intramodule** (for cores sharing L2 cache), **intrasocket** (for cores sharing L3 cache), **intersocket**, and **internode**.
- Assumptions:
  - Each core has a private L2 cache
  - Each socket contains a shared L3 cache
- If there are s sockets and q threads running in each socket, then total thread per node is $p = q * s$

Algorithm 1: Reduce_Broadcast

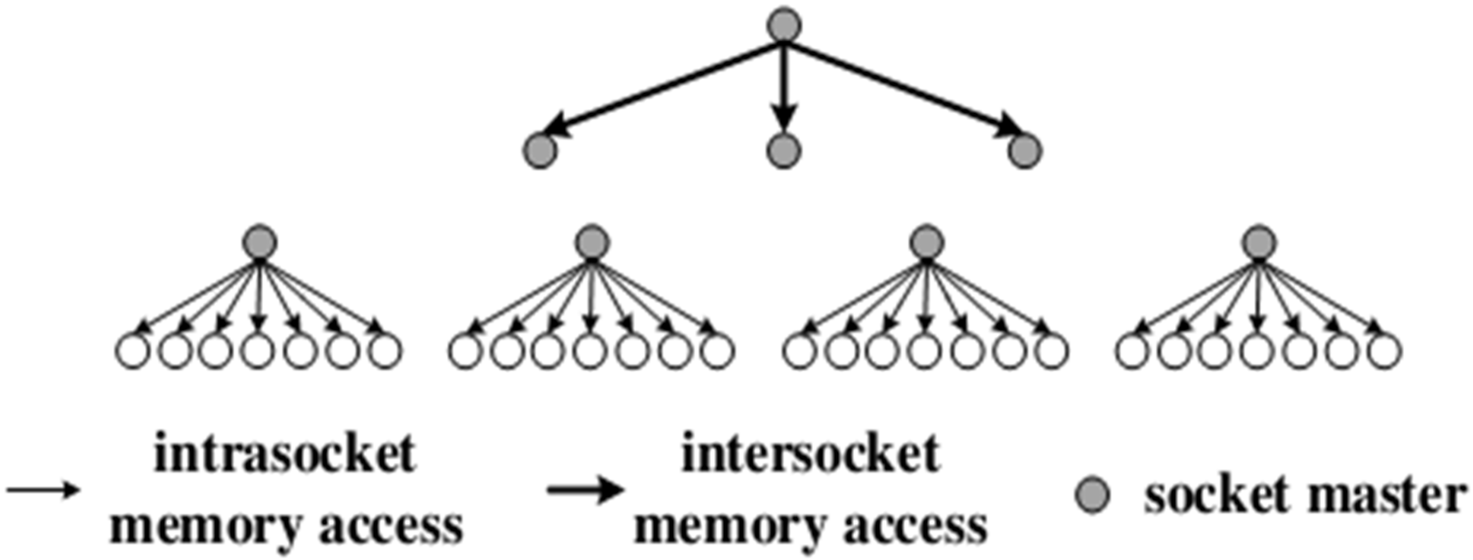
- Uses a *tree reduction*, followed by a *tree broadcast*
- In reduction, each thread performs a reduction after it has synchronized with n-1 children and finished its previous step. The first child acts as a parent in next step.


intrasocket memory access
intersocket memory access
computations within the same thread

Algorithm 1: Reduce_Broadcast

- In broadcast, one thread writes the vector, and n threads read the vector simultaneously.
- Communication involves cache, and no memory write back.
- Broadcast can be done in two ways:
  - **One stage,** where all threads read the vector simultaneously
  - **Two-stage**, where a "socket-master" at each socket reads the vector at first step, and then all threads within a socket, excluding the master, reads it simultaneously.

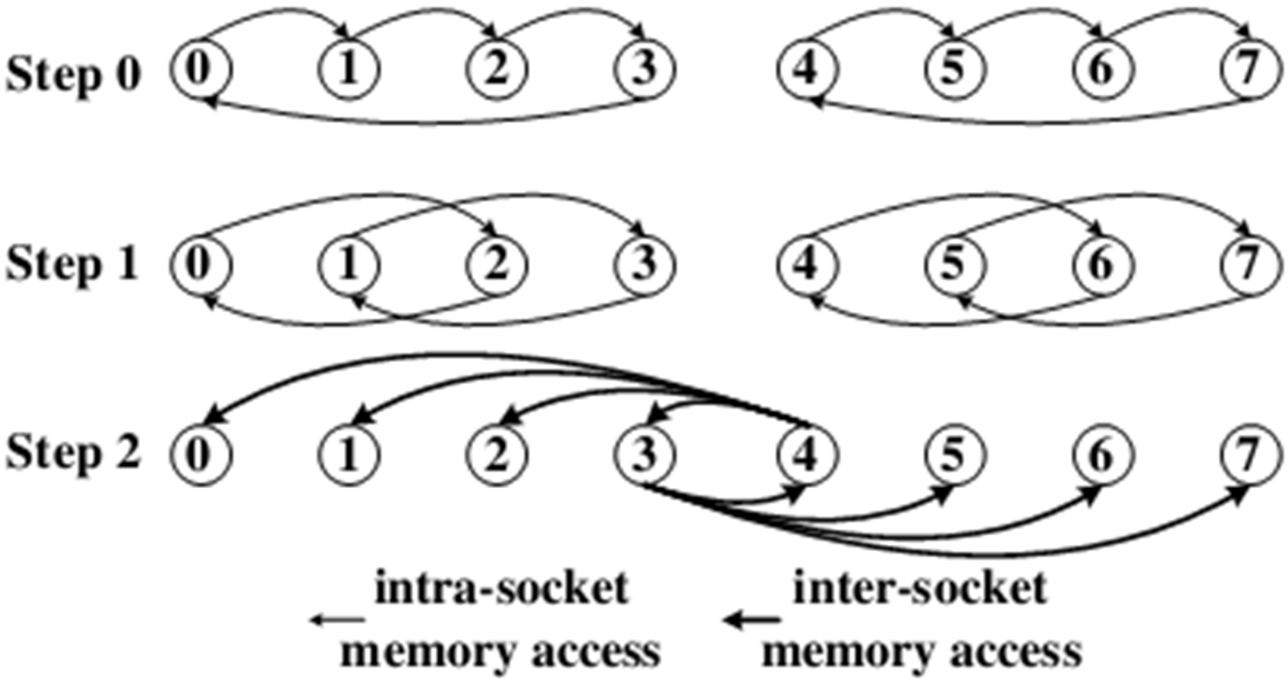
Algorithm 1: Reduce_Broadcast



Algorithm 2: Dissemination

- The dissemination algorithm achieves complete dissemination of information among p threads in $\lceil \log_2 p \rceil$ synchronized steps.
- The algorithm has fewer steps but more total communication than reduce-broadcast algorithm.

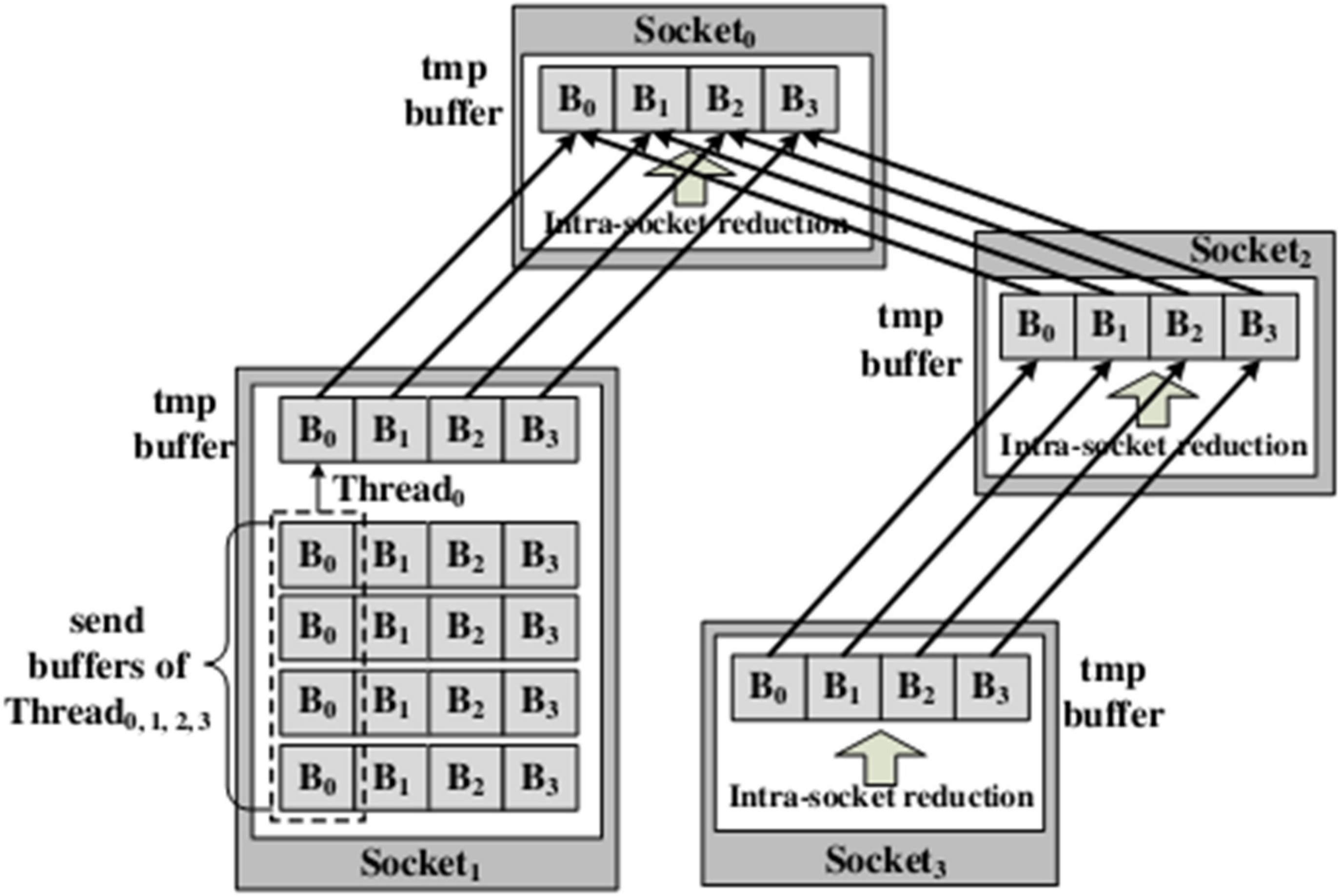
Algorithm 2: Dissemination



Algorithm 3: Tiled-Reduce Broadcast

- Since all threads can access all vectors, each thread compute a tile of final result and then broadcast it to all other threads.
- Works for long vectors and can be expected to have better performance than other algorithms for large vectors, because it can keep all the threads busy and make better use of bandwidth.
- Used only within sockets, tree reduction is better for intersocket because of limited intersocket bandwidth.
- For broadcasting, one or two stage broadcasting is used.

Algorithm 3: Tiled-Reduce Broadcast

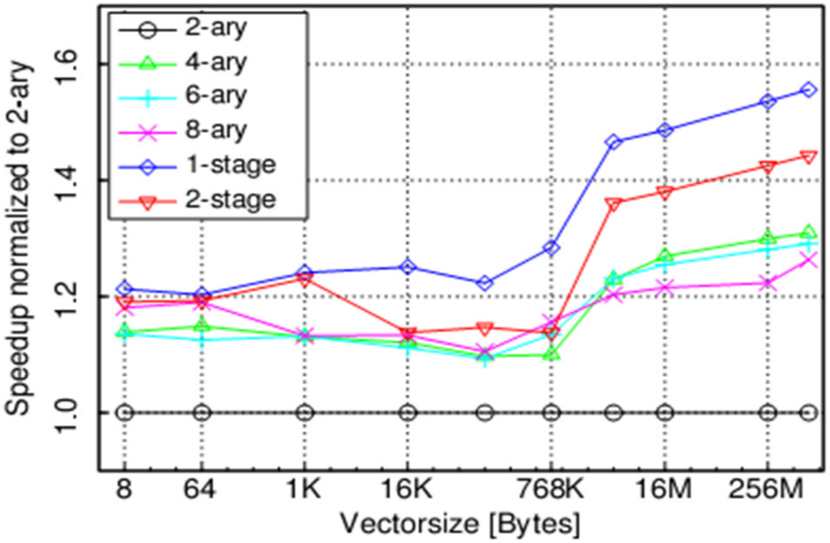


Evaluation

- Intel Xeon X5650 (Westmere), 2 sockets, 12 cores
- AMD Operton 6100 (Magny-Cours), 4 sockets, 32 cores

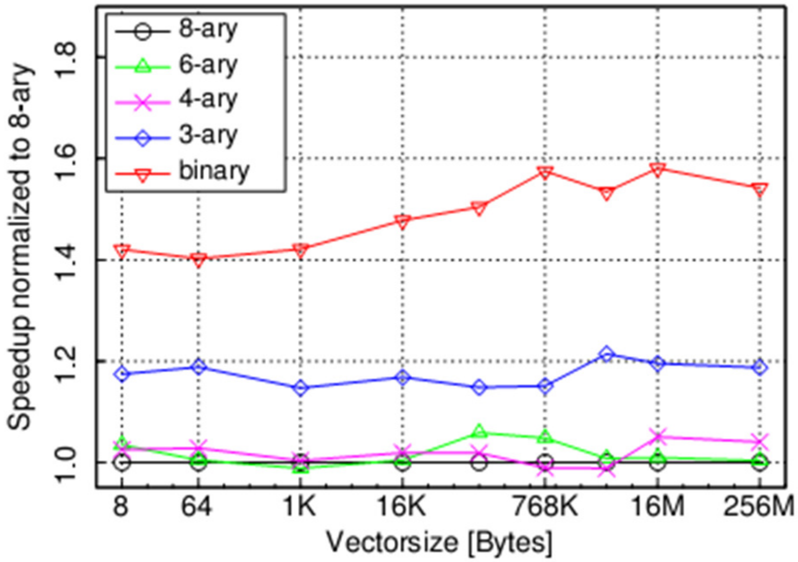
- Performance of HMPI's allreduce against MPICH2, MVAPICH2, Open MPI

Performance Analysis: Reduce Broadcast



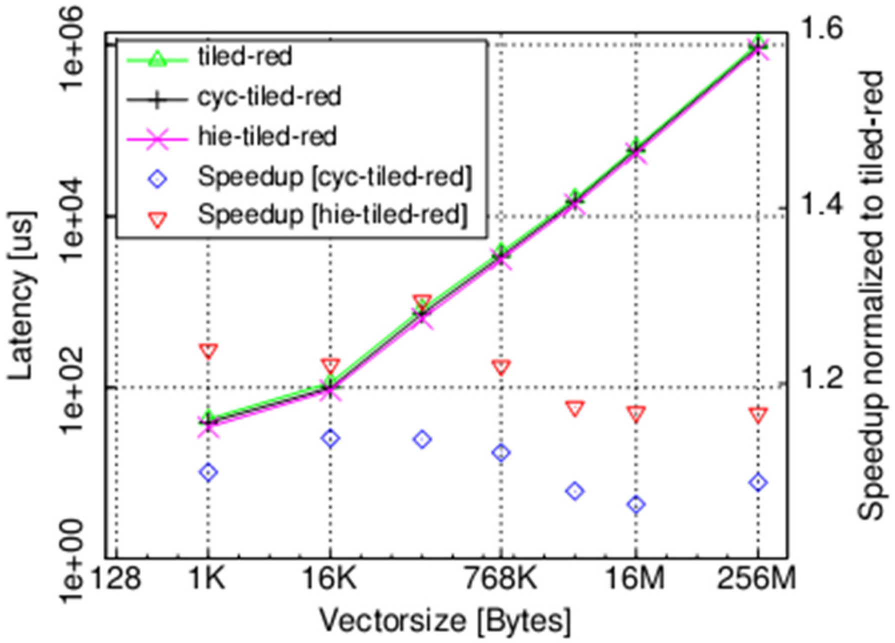
- One stage broadcast (where all threads read from the root thread) works better for 12-core Westmere.
- The reason is inclusive L3 cache exhibits affordable contention with all the threads accessing it simultaneously.

Performance Analysis: Reduce Broadcast



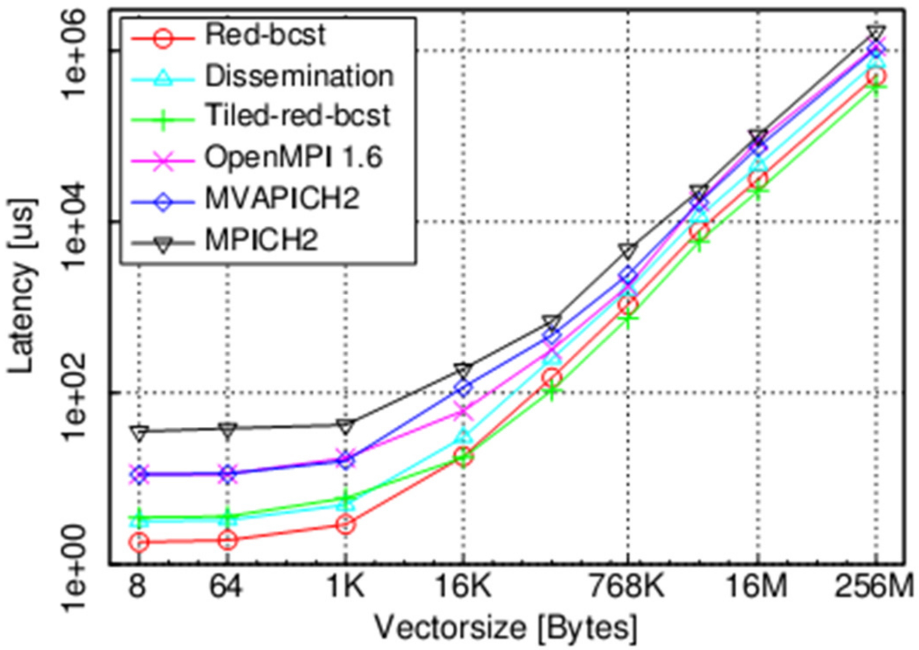
- Two stage broadcast works better for 32-core Magny-Cours. The reason is: more sockets and cores, so high contention overhead for one-step broadcast.
- For large vector sizes, flatter broadcast trees becomes advantageous. Reason? Data size is bigger than L3 cache, so more memory access needed, so reducing the numberof passes is important.

Performance Analysis: Tiled Reduce



- Naive tiled-reduce does the reduction in parallel, but without considering NUMA, so high contention.
- Cyclic tiled-reduce perform slightly better.
- Hierarchical tiled-reduce, that uses tile reduce inside nodes and tree reduction across sockets performs best

Performance Analysis: HMPI Allreduce vs. Traditional MPI

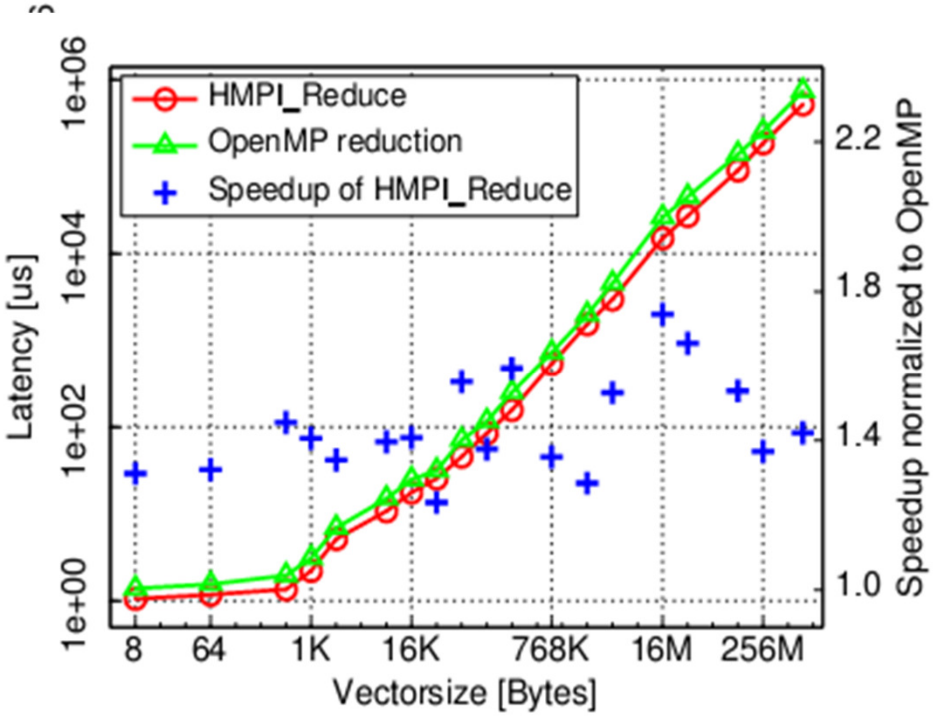


- HMPI outperforms traditional MPIs
- Reasons:
  - Direct memory access
  - Low synchronization overhead
  - Aggreesive NUMA

Performance Analysis: HMPI Allreduce vs. Traditional MPI

- Dissimination is the worst algorithm in all platform, because of redundant computation.
- MPICH2 shows the worst performance, bacause it uses shared memory only as a transport layer.
- Proposed implementations gets 3.6X lower latency than Open MPI, 4.3X than MVAPICH2.
- Tree based algorithms gets the best performance if vector size < 16 KB
- Tiled-reduce followed by broadcast has best performance if vector size > 16 KB

Comparison with OpenMP

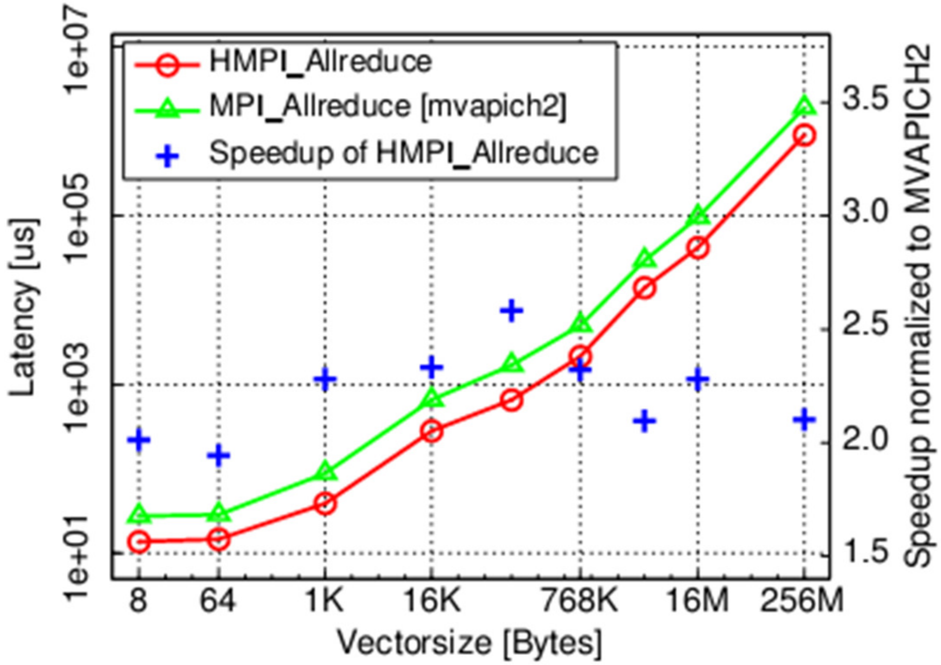


- OpenMP is another shared-memory programming environment. Its reductions have been optimized for direct shared-memory access.

  HMPI reduce achieves on average 1.5X speedup over OpenMP for all vector sizes.

  Probably due to hierarchy-aware HMPI reduce implementations on NUMA machines.

Performance on Distributed Memory



- On a 16-node Xeon cluster
- Dissemination works worst because of high internode communication overhead
- HMPI_Bcast and HMPI_Reduce get on avergae 1.8X and 1.4X speedup over MVAPICH2

Conclusion

- Multithreading has several advantages over multiprocessing on shared memory for collectives
- Using this principle, they improved MPI performance
- Proposed new algorithms for MPI_Allreduce
- Performed experiments and analyzed results