Cluster Comput (2014) 17:1139–1155
DOI 10.1007/s10586-014-0361-4

Improved MPI collectives for MPI processes in shared address spaces

Shigang Li · Torsten Hoefler · Chungjin Hu · Marc Snir

Received: 29 September 2013 / Revised: 17 February 2014 / Accepted: 18 February 2014 / Published online: 19 March 2014

Abstract As the number of cores per node keeps increasing, it becomes increasingly important for MPI to leverage shared memory for intranode communication. This paper investigates the design and optimization of MPI collectives for clusters of NUMA nodes. We develop performance models for collective communication using shared memory and we demonstrate several algorithms for various collectives. Experiments are conducted on both Xeon X5650 and Opteron 6100 InfiniBand clusters. The measurements agree with the model and indicate that different algorithms dominate for short vectors and long vectors. We compare our shared-memory allreduce with several MPI implementations—Open MPI, MPICH2, and MVAPICH2—that utilize system shared memory to facilitate interprocess communication. On a 16-node Xeon cluster and 8-node Opteron cluster, our implementation achieves on geometric average 2.3X and 2.1X speedup over the best MPI implementation, respectively. Our techniques enable an efficient implementation of collective operations on future multi- and manycore systems.

Keywords MPI · Multithreading · MPI_Allreduce · Collective communication · NUMA

Shigang Li is currently a visiting graduate student at Department of Computer Science, University of Illinois at Urbana-Champaign.

S. Li (✉) · C. Hu
School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing, China
e-mail: shigangli.cs@gmail.com

C. Hu
e-mail: huchj.cs@gmail.com

T. Hoefler
Department of Computer Science, ETH Zurich, Zurich, Switzerland
e-mail: htor@inf.ethz.ch

M. Snir
Department of Computer Science, University of Illinois at Urbana-Champaign and Argonne National Laboratory, Champaign, IL, USA
e-mail: snir@illinois.edu

1 Introduction

Applications using the message passing interface (MPI) [18] often run multiple MPI processes on each node. When evolving from petascale to exascale, the number of cores per node will keep growing to the hundreds or thousands, while memory may not be coherent. This is likely to increase the use of MPI for intranode communication.Therefore, the performance of MPI collectives becomes more and more dependent on the performance of the intranode communication component of such collectives. Furthermore, energy consumption is a major roadblock for exascale, and most energy is consumed by different forms of communication. Therefore, efficient use of shared memory by MPI is of increasing importance.

Currently, MPI processes are implemented as OS processes; MPI takes advantage of shared memory by using a shared memory segment that is attached to all the MPI processes, as shown in Fig. 1. In one approach that is used by MPICH2 [32] collectives are built atop point-to-point message passing, which uses shared memory as a transport layer inside a node. In another approach that is used by Open MPI [8] and partly by MVAPICH2 [15] collectives are implemented directly atop shared memory [8,15,28]: Data is copied from user space to shared system space so that all the processes in the communicator can share and collectively work on the data. The second approach reduces the number of memory transfers [8,28], but still requires extra data movement. Also,

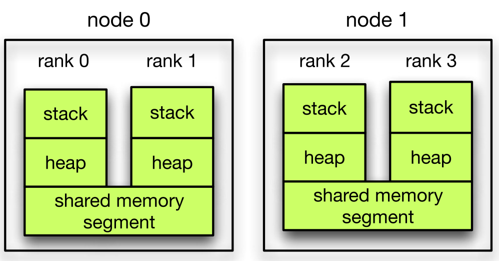


Fig. 1 Traditional use of shared memory by MPI

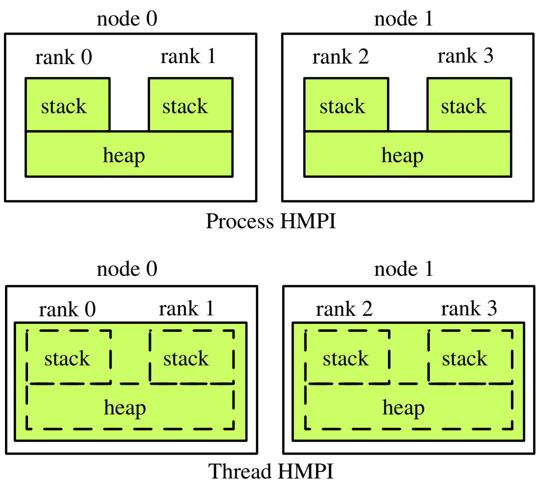


Fig. 2 HMPI use of shared memory

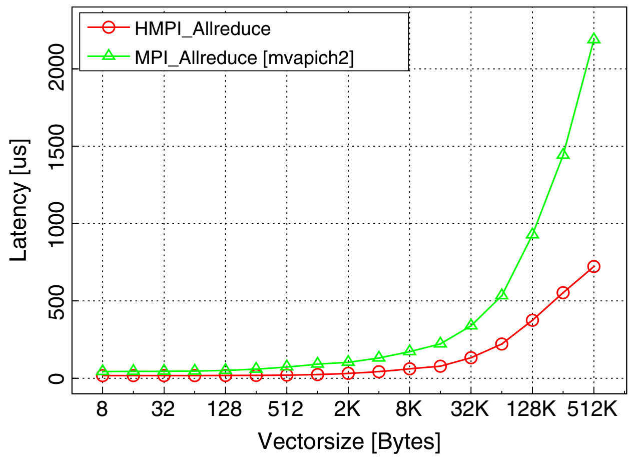


Fig. 3 Latency of HMPI_Allreduce versus traditional MPI_Allreduce on 16 Xeon X5650 nodes

memory shared across processes is usually a limited system resource.

These overheads can be avoided by ensuring that communication buffers are in memory accessible by all MPI processes at the same node. This can be done by using a shared heap, where most communication buffers are likely to reside. as shown in Fig. 2, top [5] ; or by implementing MPI processes as OS threads running in a shared address space, as shown in Fig. 2, bottom [12,24].

Sharing communication buffers directly can yield significant performance gains for two reasons: (1) communication between MPI ranks within a node requires only one copy, the minimum possible, and (2) synchronization can be accelerated by using shared synchronization variables. Applications that run in the process-based model will work with the shared-heap model with no or few modifications. They need to be linked to a different memory allocator that allocates from shared memory. In this model, the conventional communication mechanisms have to be used for stack or static variables that are not in the shared heap. A few modifications are needed for the thread-based model: Static variables need to become thread-private. Automatic privatization of global variables [19,21] can minimize the developer effort. In this model, all communication buffers are in shared address space. The hybrid MPI library (HMPI) [7] supports both models; it takes advantage of shared memory within a node and utilizes the existing MPI infrastructure for internode communication.

Prior work [6,7] has explored point-to-point communication using thread-based and process-based HMPI. We demonstrate in this paper the performance advantage of HMPI's shared-memory approach, in the context of MPI collectives, in particular, MPI_Allreduce. Figure 3 shows the motivation for our work, namely, that HMPI_Allreduce is significantly faster than traditional MPI_Allreduce. (The MPI_Allreduce functions takes as input form each MPI process a vector of values and returns as output at each MPI process the element-wise reduction of these vectors. The reduction operation is always assumed to be associative and often assumed to be commutative. Our algorithms do not use commutativity).

The paper discusses a set of algorithmic motifs, such as mixing different tree structures and multidimensional dissemination algorithms; and a set of optimization, such as utilizing shared caches and locality. These enable us to achieve highest performance for collective operations on NUMA machines. We study three shared-memory algorithms for MPI allreduce in detail: *reduce-broadcast*, *dissemination*, and *tiled-reduce-broadcast*. We establish detailed performance models for all algorithms to enable model-based algorithm selection. Experiments are conducted on a 12-core Xeon X5650 cluster and a 32-core Opteron 6100 cluster. On 16 nodes of the Xeon cluster and 8 nodes of the Opteron cluster, HMPI_Allreduce gets on average 2.3X and 2.1X speedup over MVAPICH2 1.6, gets on average 5.2X and 3.5X speedup over MPICH2 1.4, and gets on average 2.4X and 1.9X speedup over Open MPI 1.6 (all averages presented in this paper are geometric averages). Our results were obtained for the thread-based HMPI, but we expect identical results for the process-based HMPI [6]—where the communication buffers are in the shared heap.

The key contributions of this paper are as follows:

1. We design NUMA-aware algorithms for HMPI_Allreduce on clusters of NUMA nodes.
2. We show a set of motifs and techniques to optimize collective operations on multicore architectures.
3. We establish performance models based on memory access latency and bandwidth to select the best algorithm for different vector sizes.
4. We perform a detailed benchmarking study to assess the benefits of using our algorithms over state-of-the-art approaches.

This article improves the paper presented in HPDC'13 [14] by extending the collective algorithm and enriching the experimental results on different architectures. In the next section, we discuss the models to estimate the cost of intranode data movement, and introduce the implementations and performance models of our NUMA-aware allreduce algorithms, including reduce-broadcast, dissemination, and parallel-reduce followed by a broadcast. Experimental results and analyses are presented in Sect. 3. Section 3.4 discusses parameter estimation for our model and algorithm selection. A comparison with OpenMP reduction is presented in Sect. 3.5. Sections 3.6 and 3.7 present the performance of the remaining thread-based MPI collectives using microbenchmarks and applications on CMP clusters. Section 4 discusses related work, and Sect. 5 summarizes and concludes.

2 Allreduce algorithms in shared memory

Several factors, such as thread affinity, memory contention, and cache coherency, must be considered when designing multithreading algorithms. We assume that threads are bound to cores and use the following techniques to address these factors:

1. All the algorithms discussed in the following subsections are NUMA-aware to reduce intersocket memory traffic.
2. To reduce capacity cache misses for large-vector reductions, we use strip mining for large vectors in all algorithms. With this technique, large vectors are divided into chunks so that each chunk can fit into cache.
3. A high-performance, tree-based barrier is used to synchronize all threads in a communicator. Flag variables used in the synchronization operations are padded to prevent false sharing.

2.1 Performance model

We use performance models to guide the selection of the best collective algorithms. Since internode communication is not affected by our design, we focus on a performance model for intranode communication. The key operation is a read or write of contiguous data. The average latency for each cache line is less than the latency for the first few cache lines, because consecutive memory accesses benefit from hardware prefetching [1,11]. Therefore, we approximate the time required to access m consecutive cache lines as $a + bm$.

The values of a and b depend on whether the data is on the local or the remote socket, the level of memory hierarchy, the cache line state (e.g., modified cache lines need to be written back before they are evicted), and whether one reads or writes the data (e.g., a write miss may cause a load triggered by write-allocate). The dominant factor, and the only one we consider in this paper, is the distinction between intersocket and intrasocket communication.

The layout of threads is critical when designing the algorithms. For instance, a random mapping of threads in a node may lead to more intersocket communication, which can be several times slower than intrasocket communication [17]. In order to minimize communication overhead, all the following algorithms are designed and implemented hierarchically according to the hierarchy detected by the HWLOC library [4], namely intramodule (for cores sharing L2 cache), intrasocket (for cores sharing L3 cache), intersocket, and internode. To simplify the problem, when introducing the algorithms implementation, we assume that each core has a private L2 cache and each socket contains a shared L3 cache, such as for the Xeon X5650 and Opteron 6100. The intrasocket memory access time is expressed as $a_\alpha + b_\alpha m$, and the intersocket memory access time is expressed as $a_\beta + b_\beta m$, where $a_\alpha + b_\alpha < a_\beta + b_\beta$ and $1/b_\alpha > 1/b_\beta$. These formulas ignore congestion [17]. To model congestion, we use the formula $a + Btm$ to represent the delay for t simultaneous accesses where $1/B$ is the total bandwidth of the shared link and $1/(Bt) \leq 1/b$. We use $1/B_\alpha$ to represent intrasocket total bandwidth and $1/B_\beta$ to represent intersocket total bandwidth, and $B_\alpha < B_\beta$.

We use s for the number of sockets and q for the number of threads running in each socket, for a total of $p = qs$ threads per node.

2.2 Reduce-broadcast

Our reduce-broadcast algorithm uses a tree reduction, followed by a tree broadcast. For the reduction phase, we may use a regular n-ary tree. Each thread performs a reduction after it has synchronized with $n - 1$ children and finished its previous step. If it is the first child of its parent, then it acts as

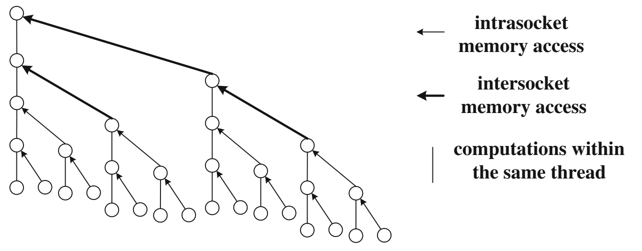

intrasocket memory access
intersocket memory access
computations within the same thread

Fig. 4 Binary reduction tree, for 1 node with 4 sockets; each socket includes 4 cores

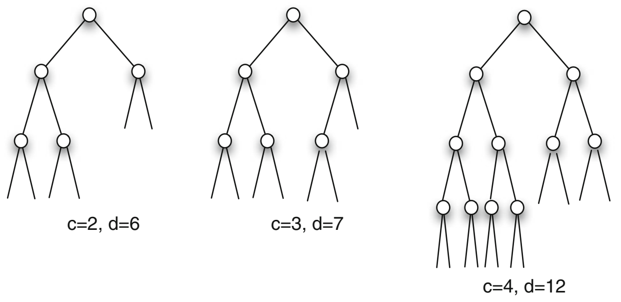


Fig. 5 Largest number d of leaves in a minimum depth binary tree with at most c concurrent operations

parent in the broadcast part. Otherwise it synchronizes with the parent.

A binary reduction tree is illustrated in Fig. 4, vertical lines connect computations within the same thread and arrows show communication across threads. The n-ary tree is mapped onto the node topology so that intersocket communication is used only at the top levels.The height of an intrasocket reduction tree is $\lceil \log_n q \rceil$ and the height of an intersocket n-ary tree is $\lceil \log_n s \rceil$. Assuming no congestion, the time for the intrasocket d-ary reduction tree is $(n-1)(a_\alpha + b_\alpha m)\lceil \log_n q \rceil$. This function is monotonically increasing in n, so a binary tree is the optimal choice.

Intrasocket reduction may cause congestion in the lower level of the tree, since simultaneous memory accesses share the last-level cache. Let $c = \lfloor b/B \rfloor$ be the largest number of threads that can concurrently access the last level cache without congestion. A minimal depth binary reduction tree with d leaves never has more than c simultaneously active threads if $d \leq 2^{1+\lfloor \log_2 c \rfloor} + c$. We provide a "proof by example" in Fig. 5.

The reduction will use the entire bandwidth until the number of active threads falls below this bound. Thus, if

$$\begin{aligned} d &= \min\left(q, \left\lfloor 2^{1+\lfloor \log_2 c \rfloor} + c \right\rfloor\right) \\ &= \min\left(q, 2^{1+\lfloor \log_2 (b/B) \rfloor} + \lfloor b/B \rfloor\right) \qquad (1) \end{aligned}$$

Then an optimal intrasocket reduction algorithm consists of one phase where the number of active threads is reduced from q to d, in time $a_\alpha + (q-d)B_\alpha m$, followed by a second phase where the remaining reduction completes in time $(a_\alpha + b_\alpha m)\lceil \log_2 d \rceil$. The total time is

$$T_{intra-red} = a_\alpha + (q-d)B_\alpha m + (a_\alpha + b_\alpha m)\lceil \log_2 d \rceil,$$

where d is defined by Eq. (1). The first phase can be executed by having $q-d$ simultaneous reductions. In practice, it may be more efficient to split this phase into subphases, each with at most c active reductions.

For intersocket reduction, no memory accesses share the same link (in modern NUMA processors, sockets are linked with each other by point-to-point links, e.g., Intel QPI and AMD HT). The time for the intersocket binary reduction tree is

$$T_{inter-red} = (a_\beta + b_\beta m)\lceil \log_2 s \rceil \qquad (2)$$

The total time for reduction is

$$T_{red} = T_{intra-red} + T_{inter-red} \qquad (3)$$

When broadcast is done by using shared memory with a n-ary tree, one thread writes the vector, and n threads read the vector simultaneously. Communication will go through the cache, and data is not written back to memory. Caches have a high bandwidth and hence can support a large fan-out. Experiments described in Sect. 3.2 show that, for the two target systems, only two configurations need to be considered: (1) a one-stage broadcast, where all the threads read the vector simultaneously, and (2) a two-stage broadcast, where a "socket master" at each socket reads the vector in the first step, and then all threads within a socket, except the socket master, read the local copy simultaneously, as illustrated in Fig. 6. The first approach always gets the best performance on a dual-socket Westmere CMP, and the second approach performs better for the 4-socket Magny-Cours CMP.

In a two-stage broadcast, the time for the intersocket broadcast (no shared intersocket link in this stage) is $(a_\beta + b_\beta m)$, and the time for the intrasocket broadcast is $(a_\alpha + B_\alpha(q-1)m)$, so that the total time overhead is $a_\alpha + B_\alpha(q-1)m + a_\beta + b_\beta m$. In a one-stage broadcast the time is dominated by the intersocket memory accesses and each intersocket link is shared by q memory accesses. Simultaneous remote accesses to the same data benefit from synergistic prefetching [30]: After one thread reads one chunk of data

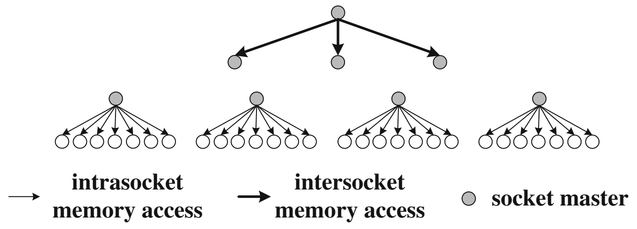


Fig. 6 Two-stage broadcast, for 1 node with 4 sockets; each socket includes 4 cores

from the remote socket, other threads can read this chunk of data directly from the socket-local shared cache. However, we found that, in practice, it does suffer from congestion to some extent, so that we approximate the time cost of one-stage broadcast as $a_\beta + B_\beta qm$. In general, the broadcast takes time

$$T_{bcst} = \min((a_\alpha + B_\alpha(q-1)m + a_\beta + b_\beta m), (a_\beta + B_\beta qm)) \tag{4}$$

Considering both reduction and broadcast phases for tree-based allreduce, the total time taken by a n-ary reduction tree combined with a one- or two-stage broadcast is

$$T_{red-bcst} = T_{red} + T_{bcst} \tag{5}$$

This algorithm requires an associative reduction operator, but does not require commutativity.

2.3 Dissemination

The next algorithm we introduce is dissemination algorithm [9,10], which achieves complete dissemination of information among $p = 2^k$ threads in k synchronized steps. During step i $(i = 0, 1, ..., N-1)$, thread j $(j = 0, 1, ..., p-1)$ combines the data from thread $(j - 2^i)\mathrm{mod}p$ with its own data. After step i, thread j has the reduction of data from threads $j, (j-1)\mathrm{mod}p, (j-2^i)\mathrm{mod}p$. This algorithm has fewer steps but more total communication than the reduce-broadcast algorithm. It also requires commutativity.

If p is a power of 2, then so are q and s. The dissemination algorithm can be laid out so that intersocket communication happens only in the last $\log_2 s$ steps, as presented in Fig. 7. In the last $\log_2 s$ steps, all threads within a socket need the same chunk of data from the other sockets. While this approach benefits from synergistic prefetching [30], it does suffer from congestion to some extent in practice.

In each step of dissemination all the threads communicate simultaneously, so we use the total bandwidth $1/B$ to approximate the time overhead. Although p memory accesses share the intersocket link (in the case of 2 sockets), intersocket links always have equal bidirectional bandwidth in modern

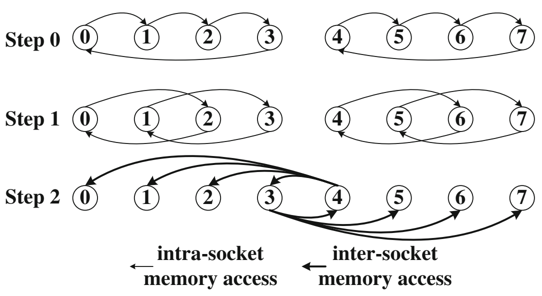


Fig. 7 Dissemination, for 1 node with 2 sockets; each socket includes 4 cores

NUMA processors, so that we can always use q as the number of memory accesses sharing the link. If p is a power of 2 the total time taken by dissemination is

$$T_{dis-pwr2} = (a_\alpha + B_\alpha qm)\log_2 q + (a_\beta + B_\beta qm)\log_2 s \tag{6}$$

If p is not a power of 2, then the dissemination algorithm can be completed in $\lceil \log_2 p \rceil + 1$ steps. Let $\hat{p} = 2^{\lfloor \log_2 p \rfloor}$ be the largest power of two that is smaller or equal to p. In the first step, we execute $p - \hat{p}$ pairwise reductions in parallel, reducing the problem size to $\hat{p}$. We then use the dissemination algorithm over $\hat{p}$ threads; and complete the computation with one more step of $p - \hat{p}$ parallel point-to-point communications. The $\hat{p}$ threads can be chosen so that each socket has at least $\lfloor \hat{p}/s \rfloor$ of them. But $\hat{p}/s > p/(2s) = q/2$. Thus, the first $\lfloor \log_2(q/2) \rfloor$ rounds of the dissemination algorithm, as well as the first and last step are local to each socket. It follows that the running time can be bound by

$$\begin{aligned} T_{dis-non-pwr2} = {} & (a_\alpha + B_\alpha qm)(\lfloor \log_2 q \rfloor + 2) + \\ & (a_\beta + B_\beta qm)(\lceil \log_2 p \rceil - \lfloor \log_2 q \rfloor - 1) \end{aligned} \tag{7}$$

The bound can be tightened for specific values of q and s. A dissemination algorithm for $p = 12$ is given in Fig. 8.

2.4 Tiled-reduce-broadcast

In this section we present the tiled-reduce-broadcast algorithm. Since all threads can access all the vectors, a straightforward algorithm is for each thread to compute sequentially one tile of the final result and then broadcast it to all threads. This algorithm works for long vectors and can be expected to have better performance than other algorithms for large vector sizes, because it can keep all the threads busy and make better use of bandwidth.

We use the tiled approach only within sockets; the limited intersocket bandwidth means that a tree reductions performs better at that stage. Each send buffer of a thread is partitioned into q chunks as evenly as possible, and then each thread simultaneously reduces its corresponding portion into a temporary buffer. In order to prevent false sharing, a scenario where individual processors treat distinct data sets in a cache line as if they are shared, the temporary buffer is padded with dummy data to the cache line boundary and partitioned at cache line granularity. In Fig. 9, $Thread_0$ in $Socket_1$ reduces all the B_0 blocks onto a temporary buffer. Next, we reduce in parallel these q chunks across sockets, using a tree reduction, in $\lceil \log_2 s \rceil$ steps.

Simultaneous memory accesses happen within each socket, so that we use the total bandwidth $1/B$ to estimate the reduce time. The time spent in the intrasocket reduction is:

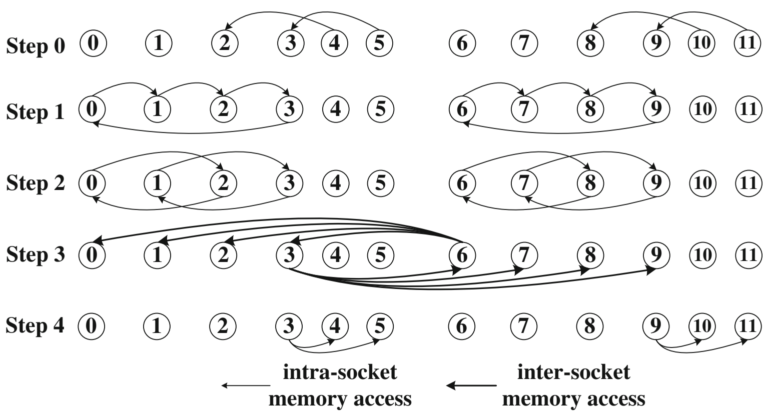


Fig. 8 Dissemination, for 1 node with 2 sockets; each socket includes 6 cores

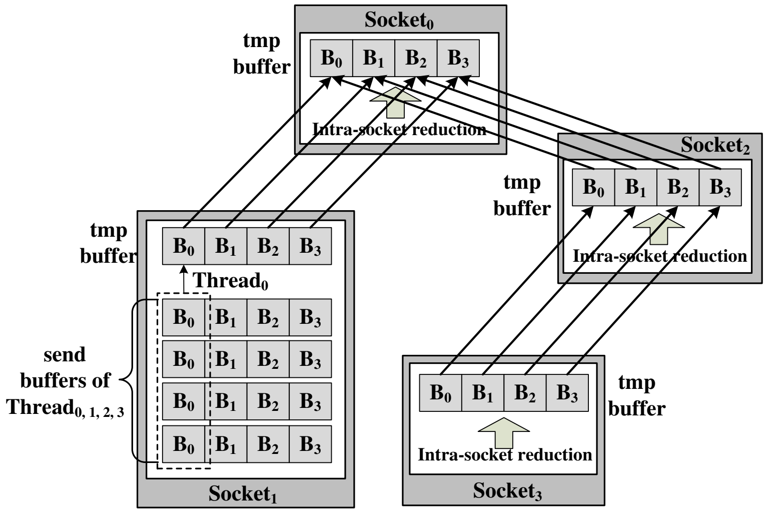


Fig. 9 Hierarchical parallel-reduce, for 1 node with 4 sockets; each socket includes 4 cores

$$T_{tiled-intra-red} = (a_\alpha + B_\alpha q(m/q))q = a_\alpha q + B_\alpha qm \tag{8}$$

The time spent in intersocket reduction is

$$\begin{aligned} T_{tiled-inter-red} &= (a_\beta + B_\beta (m/q)q)\lceil \log_2 s \rceil \\ &= (a_\beta + B_\beta m)\lceil \log_2 s \rceil \end{aligned} \tag{9}$$

The total reduction time is

$$T_{tiled-red} = a_\alpha q + B_\alpha mq + (a_\beta + B_\beta m)\lceil \log_2 s \rceil \tag{10}$$

For the broadcast phase, we use one- or two-stage broadcast, as presented in Sect. 2.2. The total time for the tiled-reduce-broadcast is

$$\begin{aligned} T_{tiled-red-bcst} &= a_\alpha q + B_\alpha mq + (a_\beta + B_\beta m)\lceil \log_2 s \rceil + \\ &\quad min(a_\alpha + B_\alpha (q-1)m + a_\beta + b_\beta m, a_\beta + B_\beta qm) \end{aligned} \tag{11}$$

2.5 Internode communication

Allreduce can be executed on multiple network nodes by performing a reduce computation within each node, an allreduce across nodes, and a broadcast within each node. The intranode communication is the same as for a reduce-broadcast or a tiled-reduce-broadcast. Therefore, to first approximate, an optimized allreduce is obtained by composing the best intranode allreduce with the best internode allreduce.

Different from reduce-broadcast and tiled-reduce-broadcast, internode dissemination is used after intranode dissemination for the dissemination algorithm across nodes. The internode dissemination is similar to the shared-memory algorithm described in Sect. 2.3. Assume there are P nodes (P is power of 2). It needs $N = \lceil \log_2 P \rceil$ steps to accomplish internode dissemination. In each step, to reduce communication overhead, only one thread in a node communicates with another node using point-to-point communication. After receiving the data, each thread within the node combines the received data with its own data simultaneously. Again, dissemination may has fewer steps but may have more communication than the other two algorithms. Experiments described in Sect. 3.6 compare the performance of these three algorithms on distributed memory.

2.6 Other collective operations

Our allreduce algorithms can be extended to other collective operations in NUMA shared memory systems. A reduce can be implemented as the reduction phase in allreduce. Different from allreduce, one needs to allocate an extra temporary buffer for each parent thread to store the intermediate results, since only the root thread has an output buffer. Broadcast can be implemented as the broadcast phase of allreduce.

An intranode barrier is implemented as a reduce-broadcast allreduce with zero workload. In the reduction phase, each thread sets a flag variable to indicate its arrival by an n-ary reduction tree. The root thread then informs other threads to continue by a one- or two-stage broadcast tree. For the internode barrier, an existing MPI_Barrier is called by the root thread between the reduction and broadcast phases.

For scatter, a temporary buffer is allocated within each node. The "global" Scatter is called by the "node master" to scatter the send buffer evenly among all the nodes, and the temporary buffer on each node is used as the receive buffer. The temporary buffer on each node then is evenly scattered among all the threads within a node. Similar to the broadcast phase of allreduce, the intranode scatter phase can be implemented as one- or two-stage scatter. Reduce-scatter is implemented by a reduce phase (tree-based algorithm or tiled reduce) followed by a scatter phase (one- or two-stage scatter).

3 Evaluation

Experiments were conducted on both Intel Xeon X5650 (Westmere) and AMD Opteron 6100 (Magny-Cours) clusters. One Xeon X5650 node has two 2.67 GHz Westmere processor sockets. Each socket has 6 cores and a 12 MB inclusive shared L3 cache. The architecture of the Xeon X5650 is illustrated in Fig. 10. One Opteron 6100 node has 2.4 GHz Magny-Cours sockets with 8 cores in each of the sockets. Each socket has a shared 10 MB noninclusive L3 cache; 2 MB out of the 10 MB of L3 cache are used to record the data in L1 and L2 caches. The architecture of the Opteron 6100 is illustrated in Fig. 11.

In a Xeon X5650 node, intersocket data transfer goes through the Quick Path Interconnect (QPI), while in an Opteron 6100 node, intersocket data transfer goes through hypertransport (HT 3) point-to-point links. Both the Xeon X5650 nodes and Opteron 6100 nodes are connected with Voltaire QDR InfiniBand in the cluster. The operating

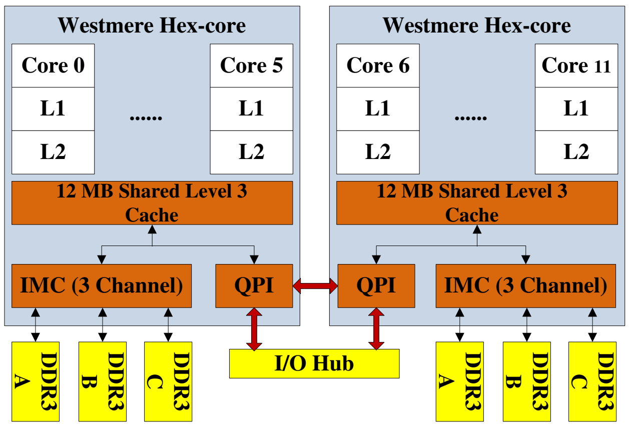


Fig. 10 Westmere, 2 sockets, total 12 cores

system on the Xeon X5650 cluster is Scientific Linux 6.1; the operating system on Opteron 6100 cluster is CentOS 5.5.

We compare the performance of HMPI's Allreduce with several currently popular MPI implementations, including MPICH2 1.4.1.pl, MVAPICH2 1.6, and Open MPI 1.6 in both shared-memory and distributed-memory environments. All the experiments are run 256 times, and we present the average values in the following figures.

We define the *speedup* S as $S = \frac{T_{ref}}{T}$. This means that an optimized operation that runs in 50 % of the latency (time) of the reference operation is said to have a speedup of 2 (also denoted as 2X).

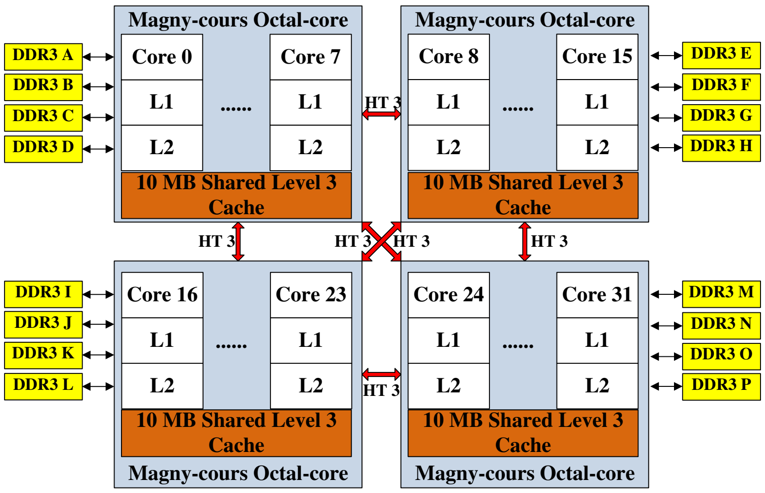
3.1 Reduce-broadcast

In this section, we compare the performance of different broadcast and reduction tree structures, in order to select the reduce-broadcast algorithm. Figures 12 and 13 present the performance comparison of different broadcast trees on Westmere and Magny-Cours CMPs respectively. On the 12-core Westmere, a one-stage broadcast where 11 threads read data from the root thread simultaneously always gets the best performance for all vector sizes. The reason is that the inclusive L3 cache of Westmere exhibits affordable contention when all the threads accessing it simultaneously.

Different from Westmere, on the 32-core Magny-Cours, a two-stage broadcast always gets the best performance for all vector sizes. The reason is probably that Magny-Cours has more sockets and cores than does Westmere, so that the benefit of finishing broadcast in one step cannot compensate for the high contention overhead.

In both Figs. 12 and 13, flatter broadcast trees become much more advantageous when the vector size is larger than 768 KB. The reason is that the total data set size is larger than the L3 cache and threads need to load the data from main memory (DRAM). The bandwidth to main memory is much lower than the L3 cache, so that reducing the number of passes becomes more important.

Fig. 11 Magny-Cours, 4 sockets, total 32 cores



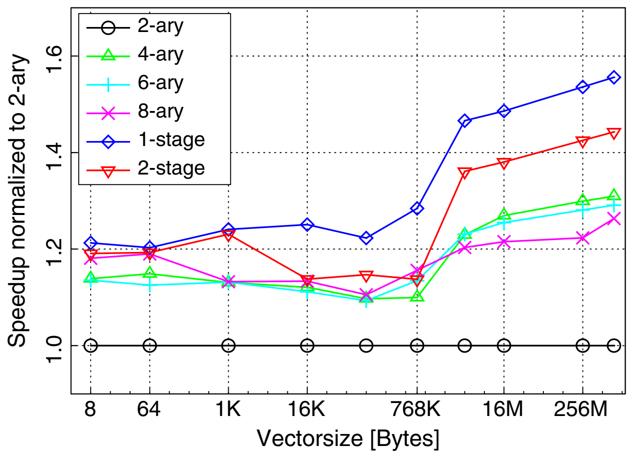


Fig. 12 Performance comparison of different n-ary broadcast trees and 1-stage/2-stage broadcast trees on 12-core Westmere

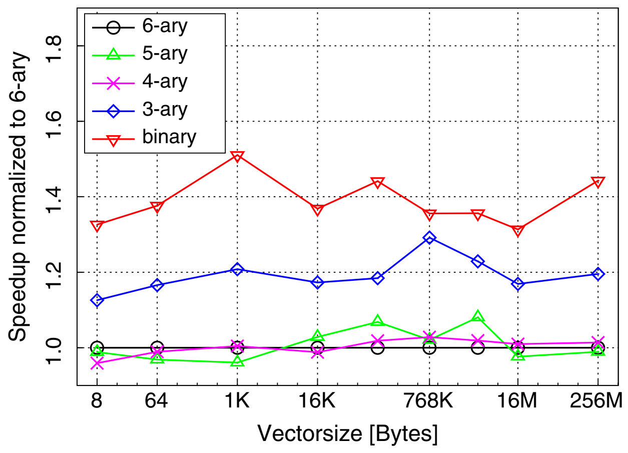


Fig. 14 Performance comparison of different n-ary reduction trees on 12-core Westmere

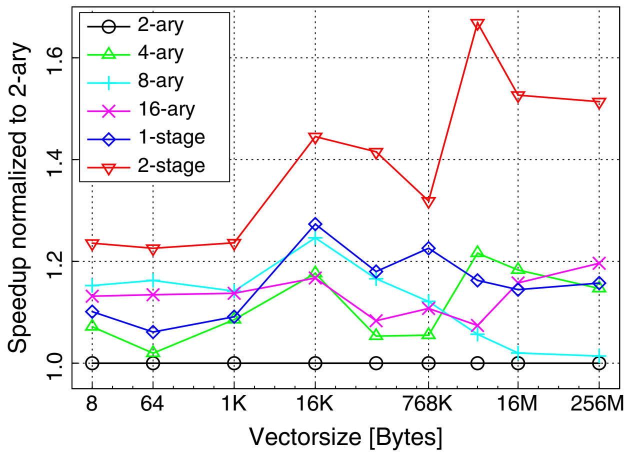


Fig. 13 Performance comparison of different n-ary broadcast trees and 1-stage/2-stage broadcast trees on 32-core Magny-Cours

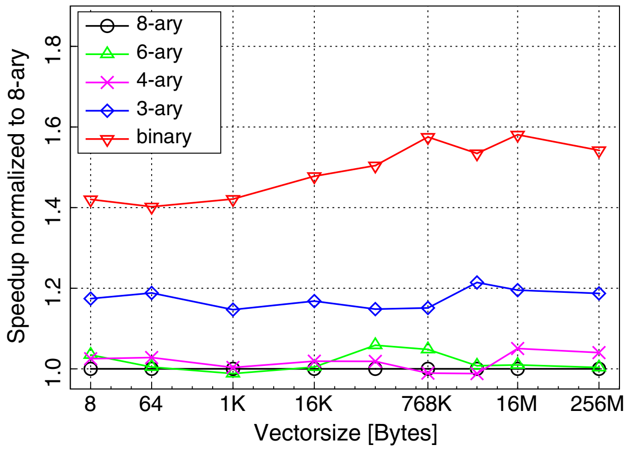


Fig. 15 Performance comparison of different n-ary reduction trees on 32-core Magny-Cours

Figures 14 and 15 presents the performance comparison of different n-ary reduction trees on Westmere and Magny-Cours respectively. As expected, a binary reduction tree dominates all other n-ary reduction trees. In summary, the best reduce-broadcast algorithm is a binary reduction tree followed by a one-stage or two-stage broadcast tree, which is abbreviated as "tree" algorithm in the remainder of the paper.

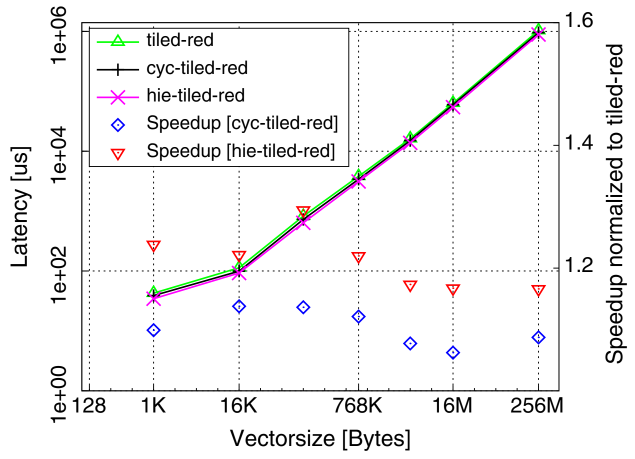


Fig. 16 Tiled-reduce on 32-core Magny-Cours

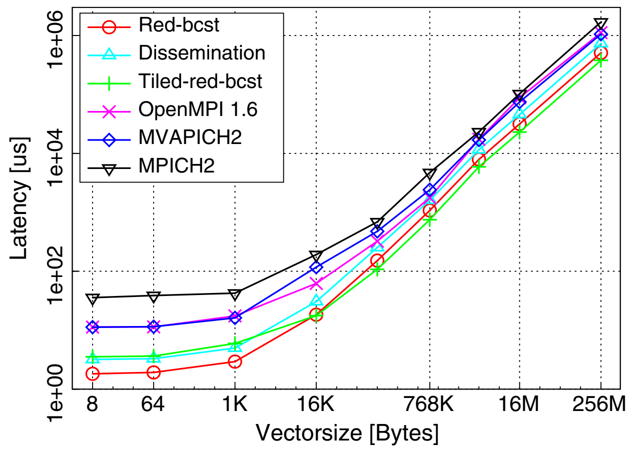


Fig. 17 Performance comparison between HMPI's Allreduce algorithms and several MPI implementations on a 12-core Westmere CMP

3.2 Performance of tiled-reduce

We evaluate the performance of the hierarchical tiled-reduce algorithm presented in Sect. 2.4 with other similar implementations, namely, a naive tiled-reduce and cyclic tiled-reduce [15]. Tiled-reduce does the reduction in parallel but without consideration of the NUMA hierarchy. This leads to high contention for intersocket memory accesses. Mamidala et al. [15] proposed a cyclic tiled-reduce algorithm where the order of send buffer (input buffer) accesses are interleaved leading to lower contention than tiled-reduce. Figure 16 show the results on Magny-Cours.Cyclic tiled-reduce performs slightly better than the original tiled-reduce algorithm. The hierarchical algorithm that uses tiled-reduce inside sockets and tree reduction across sockets has significantly better performance. Similar results have been obtained from Westmere.

3.3 HMPI's allreduce versus traditional MPIs

First we compare the best HMPI Allreduce algorithms, including tree, dissemination, and tiled-reduce followed by a broadcast, with the current MPI implementations, including MPICH2, Open MPI 1.6, and MVAPICH2, on shared memory. The number of launched threads in HMPI Allreduce and the number of launched processes in the traditional Allreduce are equal to the number of cores on each architecture. Figure 17 shows the result on Westmere.

HMPI's Allreduce always outperforms traditional approaches used in other MPI implementations on Westmere. This performance is due partially to direct memory access and low overhead of synchronization and partially to the aggressive NUMA optimizations in HMPI. Among all the HMPI Allreduce algorithms, dissemination almost always exhibits the worst performance on both architectures (only better than tiled-reduce-broadcast for small vectors). This probably results from the redundant computation and contention caused by combining reduction and broadcast together, and the extra overhead for non-power-of-2 thread counts, as reflected in Eq. (7).

Among all the MPI implementations, MPICH2 always gets the worst performance. Recall that in MPICH2, collectives are built on the point-to-point message passing using shared memory merely as a transport layer; in Open MPI 1.6 and MVAPICH2, collectives are implemented and optimized independently by eliminating point-to-point message passing as the underlying communication protocol. Implementing collectives on top of point-to-point message passing has the most buffer copies. Our best implementation achieves on average 3.2X lower latency than Open MPI 1.6, 3.9X lower latency than MVAPICH2, and 5.9X lower latency than MPICH2.

Overall, on Westmere, the tree-based algorithm gets the best performance when the vector size is less than 16 KB, while tiled-reduce followed by a broadcast gets the best performance when vector size grows larger than 16 KB. When comparing Eq. (5) with Eq. (11), the latency of tiled-reduce followed by a broadcast is higher than tree but the bandwidth term is more favorable. When the vectors are small, latency is the limiting factor in the time overhead, so that tree performs better than parallel-reduce followed by a broadcast. When the vector size grows larger and bandwidth becomes the bottleneck, tiled-reduce followed by a broadcast performs better than tree.

Figure 18 shows the result on Magny-Cours. The same as that on Westmere, tree-based algorithm dominates the performance for small vector size while tiled-reduce-broadcast dominates the performance for large vector size, but the crosspoint is different. In summary, the best algorithm is different on different machines as well as different vector size, and this is the motive of designing performance model for algorithm selection.

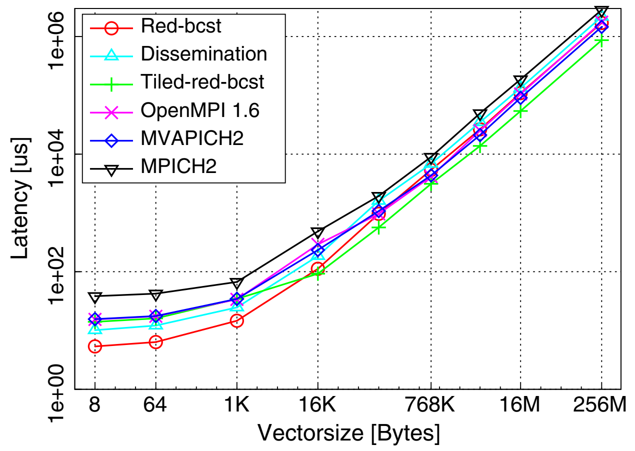


Fig. 18 Performance comparison between HMPI's Allreduce algorithms and several MPI implementations on a 32-core Magny-Cours CMP

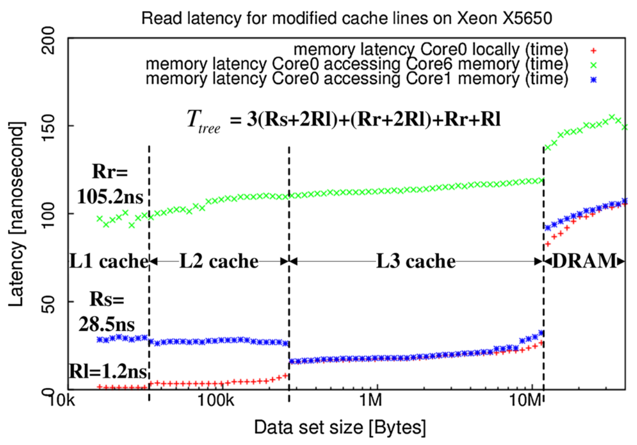


Fig. 19 Modified cache line read latency on Westmere. Rl denotes latency of read local L1 cache, Rs denotes latency of read other L1 cache but within the same socket, and Rr denotes latency of read other L1 cache from remote socket

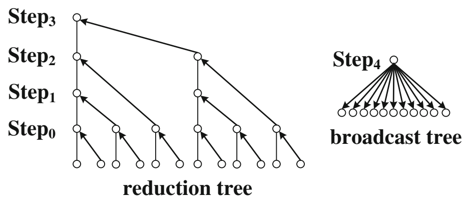


Fig. 20 Steps in the tree-based algorithm on Westmere

3.4 Algorithm selection

In this section we use memory access latency and bandwidth to verify the performance models and then select the best Allreduce algorithms for different vector sizes. Several factors, such as cache coherence protocol, hardware and software prefetch, and page size (TLB), affect the cache line transfer latency and bandwidth. The configurations of these parameters for both Westmere and Magny-Cours are presented below. Page size is set to 4 KB, hardware prefetch and adjacent line prefetcher are turned on, and no software prefetch is used in the original code or in the compiler options. The various cache line states also affect performance. In order to build the performance model accurately, all the data in send buffer and receive buffer are set to the *modified state*, which models the common scenario of a local write followed by a global communication of the written buffer.

As mentioned, the tree-based algorithm gets the best performance for small vector sizes. The time overhead of a tree-based algorithm is shown in Eq. (5), where $n = 2$. We set the vector size to one cache line ($m = 1$) and use experimentally measured cache line transfer latency to verify the model. To determine the latencies, we use BenchIT [17], which provides memory latency benchmarks for multicore and multiprocessor x86-based systems.

Figure 19 shows the results on Xeon X5650. The three curves show the latency of local access, intrasocket access (Core 0 accessing Core 1), and intersocket access (Core 0 accessing Core 6) respectively. By varying the data set size, the performance of the full memory hierarchy is exposed. The latency of reading local L1 cache (Rl) is 1.2 ns, the latency of reading other L1 cache but in the same socket (Rs) is 28.5 ns, and the latency of reading L1 cache on the other socket (Rr) is 105.2 ns.

On the 12-core Westmere, Eq. (5) is unfolded as $T_{red-bcst} = (a_\alpha + 2B_\alpha m) + 2(a_\alpha + b_\alpha m) + (a_\beta + b_\beta m) + (a_\beta + B_\beta q m)$, where $m = 1$, $q = 6$. Figure 20 shows the steps in the tree-based algorithm on Westmere.

(1) In $Step_0$, $Step_1$, and $Step_2$, a thread reads a cache line from its local L1 cache and a cache line from a remote L1 cache but within the same socket and then writes to its local L1 cache. Because writing to the local L1 cache is a write hit, we assume its latency is equal to the read latency. In $Step_0$, there are 3 simultaneous memory accesses within each socket; however, because the data size is very small and the shared L3 cache serves as central unit for intercore communication, the congestion is ignored. Thus, the time overhead of the first three steps is $3(Rs + 2Rl)$, corresponding to $(a_\alpha + 2B_\alpha m) + 2(a_\alpha + b_\alpha m)$ in the model.
(2) In $Step_3$, a thread performs the same operations as in the first three steps except that there is an intersocket cache line read. The time overhead of $Step_3$ is $Rr + 2Rl$, corresponding to $(a_\beta + b_\beta m)$ in the model.
(3) In $Step_4$, all other threads read data from the root thread and write to their own receive buffer. Because of the same reason mentioned in $Step_0$, the congestion is ignored. Hence, $Step_4$ can be simplified to one thread reading a cache line from remote socket and writes to local L1

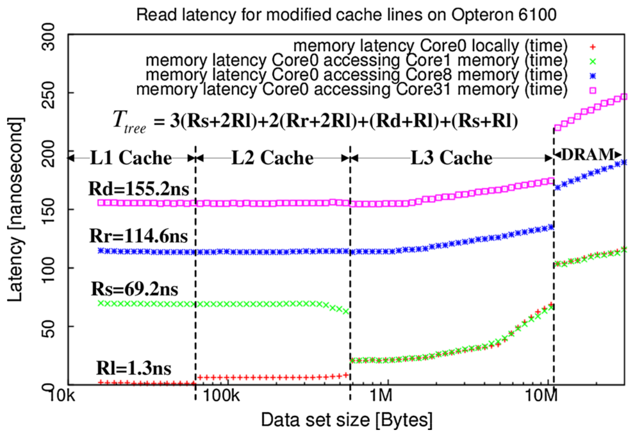


Fig. 21 Modified cache line read latency on Magny-Cours. Rr denotes the L1 read latency from a horizontal or vertical remote socket, and Rd denotes the L1 read latency from a diagonal remote socket

cache. The write is a write hit and we assume it equals to the read latency. So the time overhead in $Step_4$ is $Rr+Rl$, corresponding to $(a_\beta + B_\beta qm)$ in the model.

To sum up, the overall time overhead of tree-based algorithm on Xeon X5650 is $3(Rs + 2Rl) + (Rr + 2Rl) + (Rr + Rl)$= 306.7 ns. We use an indirect method to measure the practical runtime, namely, the runtime of one cache line workload minus the runtime of zero workload. The practical runtime is 339.8 ns, which is a little higher than that the model predicted. The deviation is due to complex interactions in the microarchitecture (e.g., pipelining and superscalar units) that have only low-order influence on the runtime and that we thus excluded from the model.

Similar results have been obtained on Magny-Cours. We note that on the four-socket Opteron 6100, the latency of reading an L1 cache line from a diagonal remote socket is higher than that from horizontal or vertical remote socket, as illustrated in Fig. 21. The time overhead is $3(Rs + 2Rl) + 2(Rr + 2Rl) + (Rd + Rl) + (Rs + Rl) = 608.4$ ns, in which Rd denotes latency of read other L1 cache from diagonal remote socket. The practical runtime is 647.4 ns, which is also a little higher than that model prediction.

The tiled-reduce followed by a broadcast gets the best performance for large vector sizes. In this section, we utilize the performance models to select the best algorithm for different vector sizes. We measure all the values (latency and bandwidth) in Eq. (5) and Eq. (11) on Westmere, and we present the predicted runtime obtained from the performance models and the real runtime of tree-based algorithm and tiled-reduce-broadcast in Figs. 22 and 23 respectively. We see that the models can predict latencies accurately and the relative errors are all below 5 %.

Figure 24 shows that, on Westmere, the tree-based algorithm gets the best performance for small vector size, while

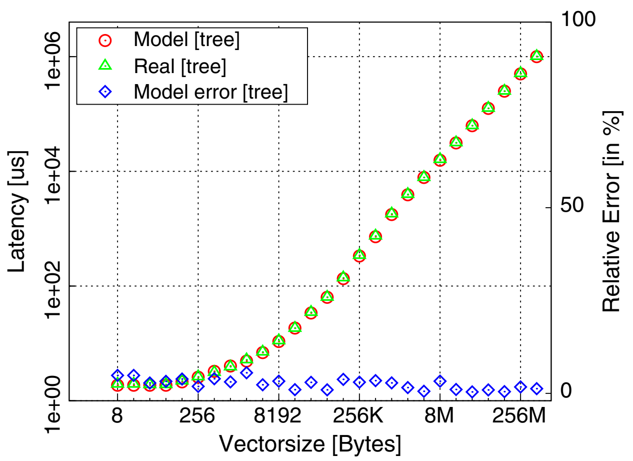


Fig. 22 Performance model for tree-based algorithm on Westmere

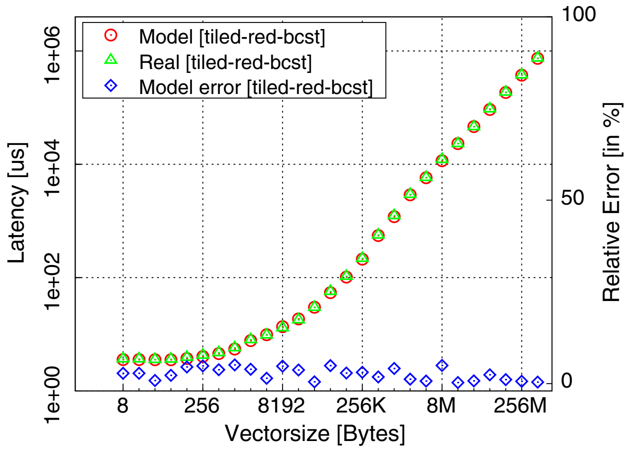


Fig. 23 Performance model for tiled-reduce-broadcast on Westmere

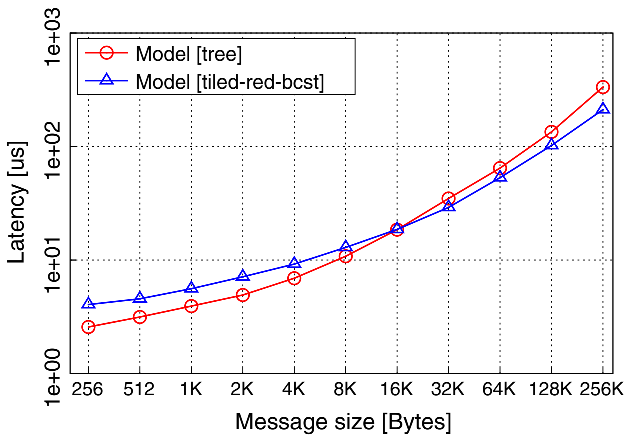


Fig. 24 Selecting the best algorithm with the performance model on Westmere

the crosspoint at 16 KB indicates that the best algorithm switches to tiled-reduce followed by a broadcast. The crosspoint on Magny-Cours is 14 KB, as illustrated in Fig. 25.

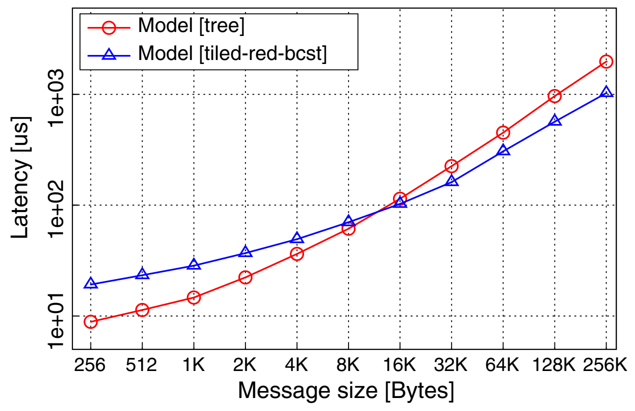


Fig. 25 Selecting the best algorithm with the performance model on Magny-Cours

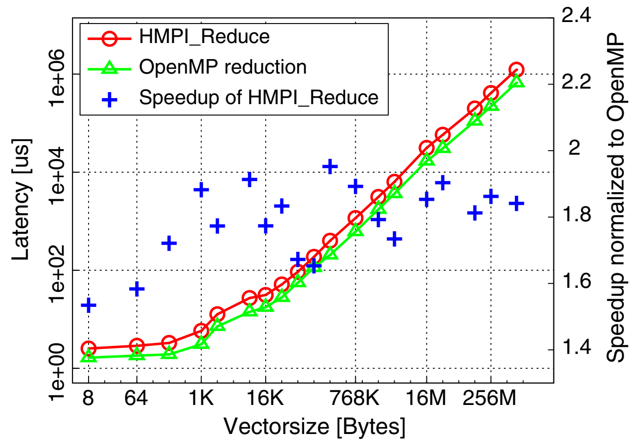


Fig. 27 Performance comparison between the best HMPI Reduce algorithm and OpenMP reduction on Magny-Cours

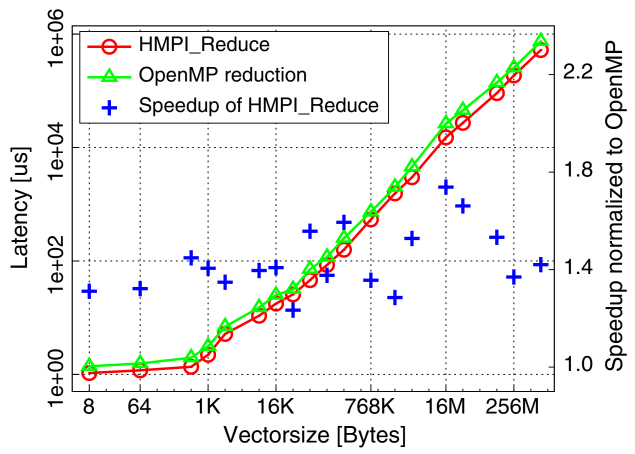


Fig. 26 Performance comparison between the best HMPI Reduce algorithm and OpenMP reduction on Westmere

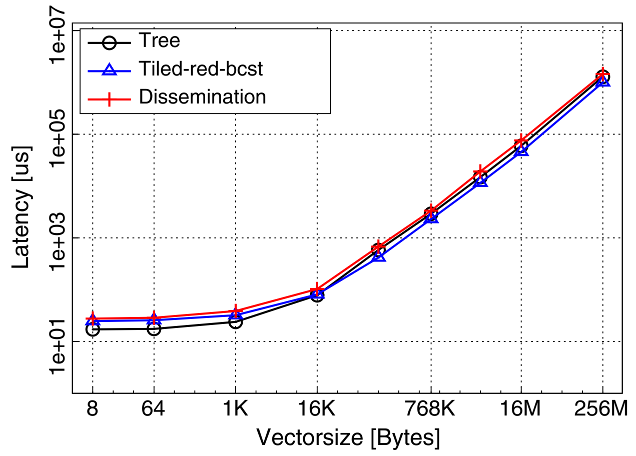


Fig. 28 Performance comparison between HMPI_Allreduce algorithms on 16-node Xeon X5650 running on 192 cores

3.5 Comparison with OpenMP

We compare our performance with another native shared-memory programming environment, OpenMP. We previously discussed how our techniques can be used in the context of MPI. However, we could not easily quantify the source of the benefits because current MPI implementations do not exploit direct shared-memory communication. Thus, in this section, we implemented HMPI Reduce with the techniques described above and compare it with OpenMP reductions that have been optimized for direct shared-memory accesses.

Figures 26 and 27 compare our best Reduce algorithms with an OpenMP REDUCTION clause [20] on Westmere and Magny-Cours respectively. C based OpenMP does not support reduction on vectors, so we use Fortran based OpenMP reduction for our comparison. We use 12 threads for both HMPI Reduce and OpenMP on Westmere. We see that HMPI Reduce achieves on average 1.5X and 1.7X speedup over OpenMP for all the vector sizes on Westmere and Magny-Cours respectively, due to the hierarchy-aware HMPI Reduce implementation on NUMA machines.

3.6 Performance on distributed memory

We now compare the algorithms of thread-based Allreduce on 16-node Xeon cluster combining inter- and intranode communications. As on shared memory, on-node dissemination exhibits the worst performance among our algorithms, as illustrated in Fig. 28. For the internode dissemination, each node has to communicate with another node by point-to-point communication, probably causing more communication overhead than the current process-based MPI allreduce implementation [22,26,27], which is used for internode allreduce in tree and tiled-reduce-broadcast.

We compare thread-based MPI allreduce, broadcast, and reduce with MPICH2 1.4, MVAPICH2 1.6 and Open MPI 1.6. For both MPI and HMPI, the total number of launched ranks is equal to the total number of cores in the cluster.

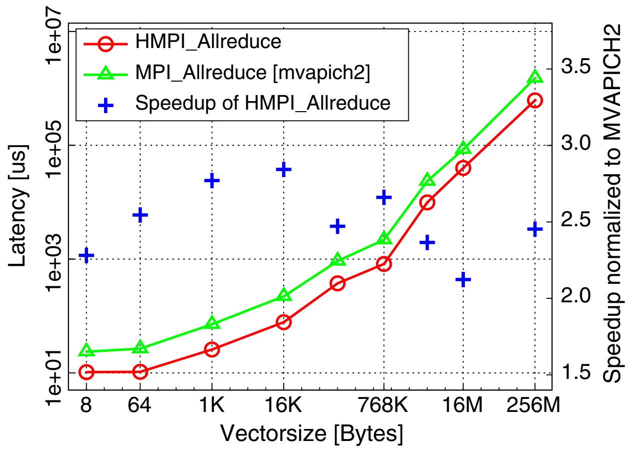


Fig. 29 Performance comparison between HMPI's Allreduce and MVAPICH2 on 16-node Xeon X5650 running on 192 cores

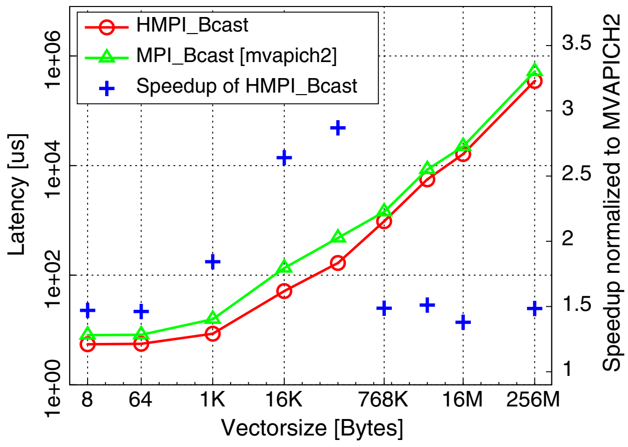


Fig. 31 Performance comparison between HMPI's Bcast and MVAPICH2 on 16-node Xeon X5650 running on 192 cores

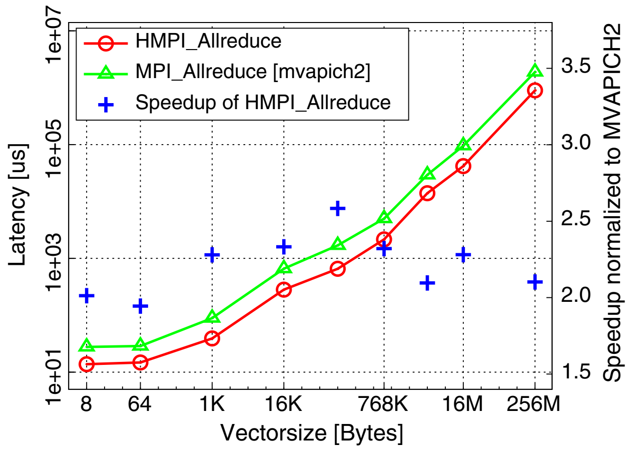


Fig. 30 Performance comparison between HMPI's Allreduce and MVAPICH2 on 8-node Opteron 6100 running on 256 cores

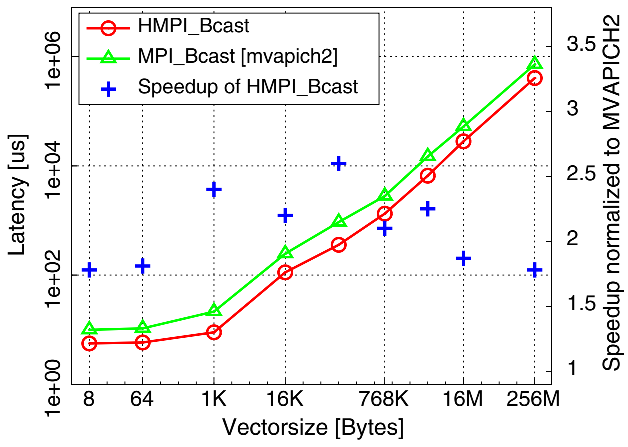


Fig. 32 Performance comparison between HMPI's Bcast and MVAPICH2 on 8-node Opteron 6100 running on 256 cores

On 16-node Xeon X5650 and 8-node Opteron 6100 clusters, HMPI_Allreduce gets on average 2.3X and 2.1X speedup over MVAPICH2, gets on average 5.2X and 3.5X speedup over MPICH2, and gets on average 2.4X and 1.9X speedup over Open MPI. The performance comparison between HMPI_Allreduce and MVAPICH2 on 16-node Xeon X5650 cluster and 8-node Opteron 6100 cluster are shown in Figs. 29 and 30 respectively. Figures 31 and 32 show that HMPI_Bcast gets on average 1.7X and 2.0X speedup over MVAPICH2 on the 16-node Xeon X5650 cluster and 8-node Opteron 6100 cluster respectively. Figures 33 and 34 show that HMPI_Reduce on average 1.3X and 1.5X speedup over MVAPICH2 on the 16-node Xeon X5650 cluster and 8-node Opteron 6100 cluster respectively. Figures 35 and 36 show that HMPI_Barrier scales better than MVAPICH2. Figures about the comparison with MPICH2 and Open MPI aren't presented here to save space. The results indicate that the thread-based MPI collectives design, which is a true zero-copy approach with NUMA-aware topology optimization, has significant advantage over traditional process-based MPI collectives.

3.7 Application comparison

Two applications, dense matrix vector multiplication and tree-building in Barnes-Hut, are used to evaluate the performance of HMPI mainly stressing our collective optimizations. We compare HMPI implementation with MVAPICH2, which is the best performing MPI implementation in our earlier experiments. The dense matrix vector multiplication is computed for 128 iterations and the matrix size is 49,152 $\times$ 49,152. The matrix is partitioned column-wise and scattered to all the processes using MPI_Scatter. The vector is broadcast to all the processes using MPI_Bcast. Within each iteration, each process uses an MPI_Reduce to sum the corresponding part of the intermediate results. As illustrated in Figs. 37 and 38, HMPI has better scalability and gets on average 1.2X and 1.3X speedup over MVAPICH2 on the

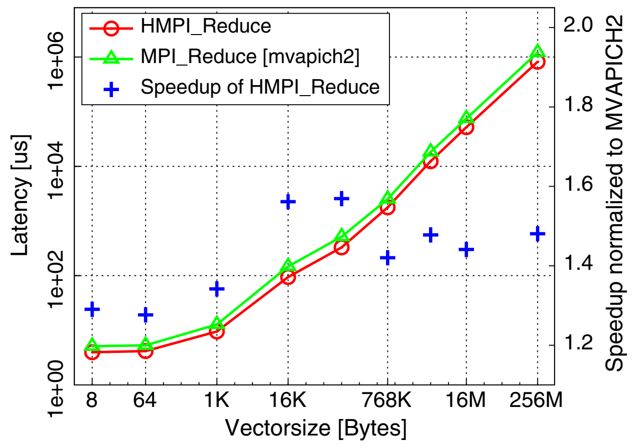


Fig. 33 Performance comparison between HMPI_Reduce and MVAPICH2 on 16-node Xeon X5650 running on 192 cores

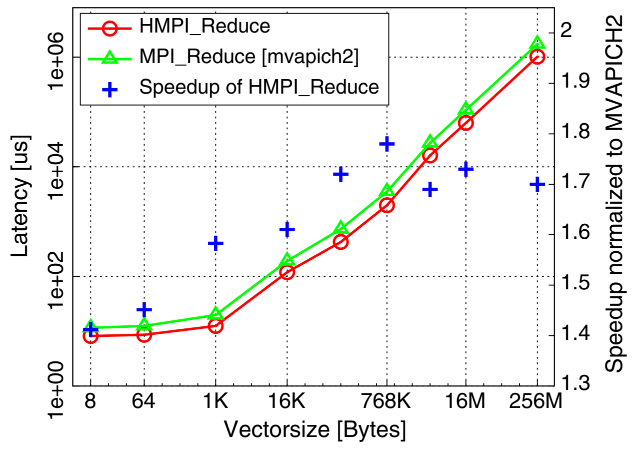


Fig. 34 Performance comparison between HMPI_Reduce and MVAPICH2 on 8-node Opteron 6100 running on 256 cores

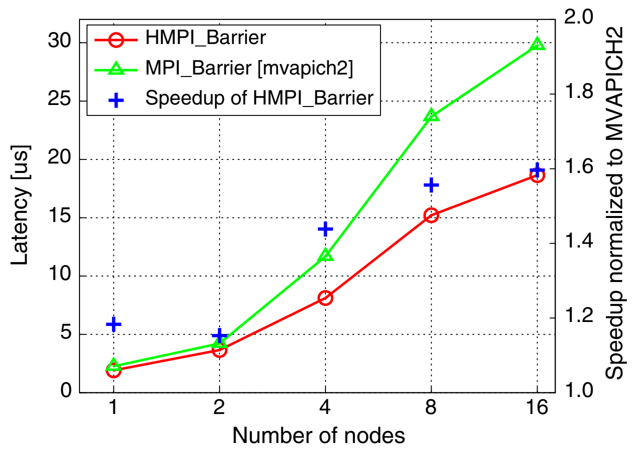


Fig. 35 Performance comparison between HMPI_Barrier and MVAPICH2 on Xeon X5650 cluster (12, 24, 48, 96, and 192 cores)

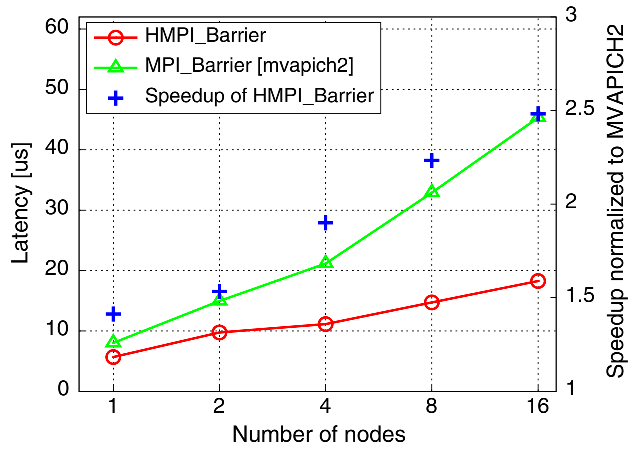


Fig. 36 Performance comparison between HMPI_Barrier and MVAPICH2 on Opteron cluster (32, 64, 128, 256, and 512 cores)

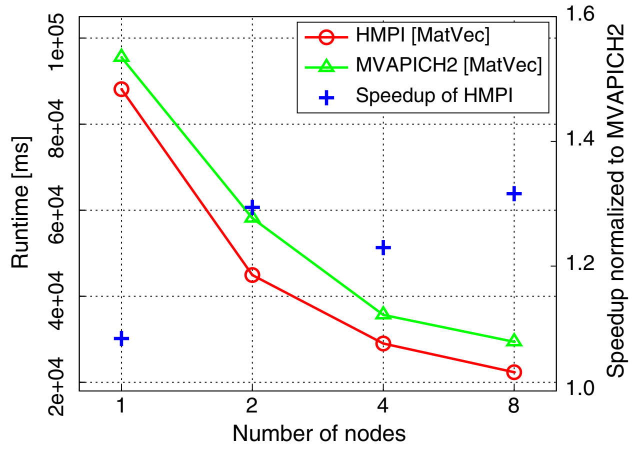


Fig. 37 Dense matrix vector multiplication on Xeon X5650 cluster (12, 24, 48, and 96 cores)

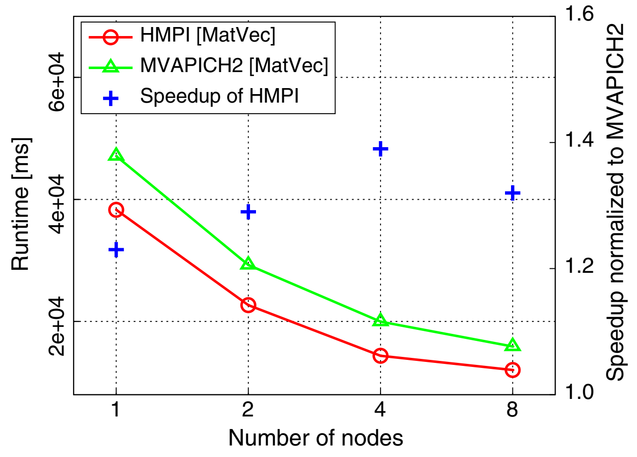


Fig. 38 Dense matrix vector multiplication on Opteron 6100 cluster (32, 64, 128, and 256 cores)

16-node Xeon X5650 cluster and 8-node Opteron 6100 cluster respectively.

Allreduce is a key operation in the tree-building algorithm of the Barnes-Hut n-body simulation [31]. Tree building relies on allreduce to achieve high level alignment of space partitions. Processes compute local costs of subspaces and then use an allreduce to sum local costs of subspaces to get

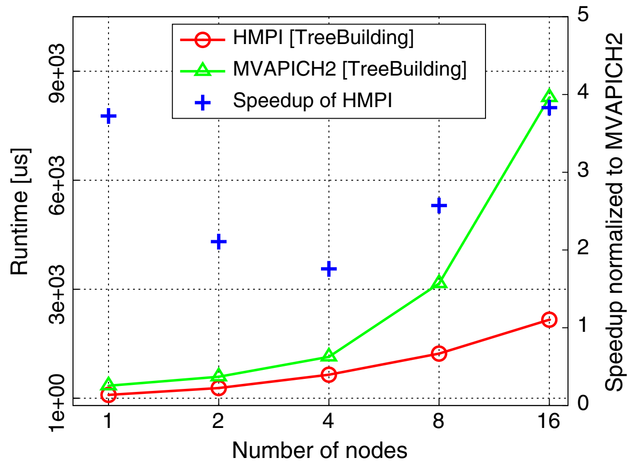


Fig. 39 Tree-building in Barnes-hut on Xeon X5650 cluster (12, 24, 48, 96, and 192 cores)

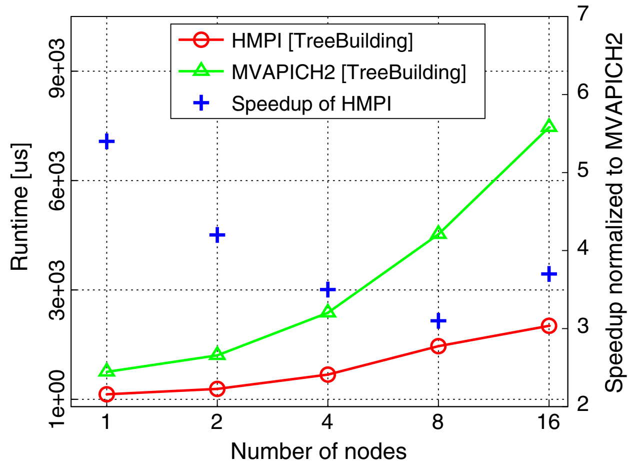


Fig. 40 Tree-building in Barnes-hut on Opteron 6100 cluster (32, 64, 128, 256, and 512 cores)

the global cost of these subspaces. Because processes split the simulation space recursively, multiple allreduce operations on different vector length are used until there is no fat subspace. We set the number of bodies to 4 million and test the allreduction phase of the tree building algorithm on 1, 2, 4, 8, and 16 Xeon X5650 nodes and Opteron 6100 nodes, and HMPI shows on average 2.5X and 3.8X speedup over MVAPICH2 respectively. Moreover, as the number of nodes increases, the improvement becomes more apparent, as illustrated in Figures 39 and 40. Overall, results on real applications indeed validate the advantage of the thread-based MPI collectives design.

4 Related work

Several MPI implementations have optimizations for shared memory based on a process per MPI rank model. In MVAPICH2, shared-memory-based collectives have been enabled for MPI applications running over OFA-IB-CH3, OFA-iWARP-CH3, and uDAPL-CH3 stack. Currently, this support is available for the following collective operations: MPI_Allreduce, MPI_Reduce, MPI_Barrier, and MPI_Bcast [15]. Open MPI provides sm BTL (shared-memory Byte Transfer Layer) as a low-latency, high-bandwidth mechanism for transferring data between two processes via shared memory. According to the hardware architecture, Open MPI will choose the best BTL available for each communication. Other MPI implementations, such as LA-MPI [2] and Sun MPI [23], also have support for shared memory.

Graham and Shipman [8] have examined the benefits of creating shared-memory optimized multiprocess collectives for on-node operations. They indicated the importance of taking advantage of shared caches and reducing intersocket memory traffic. Kielmann et al. [13] developed MagPIe, a hierarchy-aware library of collective communication operations for wide area systems. We utilize this hierarchical design method to implement NUMA-aware collectives of HMPI. Tang and Yang [25] presented thread-based MPI system for SMP clusters and showed that multi-threading, which provides a shared-memory model within a process, can yield performance gain for MPI communication because of speeding the synchronization and reducing the buffering and orchestration overhead. Their experimental results indicated that even in a cluster environment for which internode network latency is relatively high, exploiting thread-based MPI execution on each node can deliver substantial performance gains. A hybrid of multiprocess and multithreading runtime system for Partitioned Global Address Space languages is presented in [3].

Tree-based barriers, such as the combining tree barrier [29] and the MCS barrier [16], are designed to distribute hot-spot accesses over a software tree. This rationale is also used in HMPI when designing synchronization operations, such as HMPI_Barrier, and also collective operations, such as tree-based HMPI_Allreduce. Dissemination-based barriers [9, 16] achieve complete dissemination of information among p processes in $log_2 p$ synchronized steps. We utilize this algorithm to implement dissemination-based allreduce which further evolves to 3D dissemination (intrasocket, intersocket and then internode) to reduce communication overhead. Zhang et al. [30] have exploited program-level transformations to lift the parallel programs to be cache-sharing-aware, which motivated us when optimizing the collective algorithms to take advantage of shared cache.

5 Conclusions and discussion

In the era of multicore or manycore, parallel programming languages or libraries need to provide high performance and low power consumption for scientific computing applications on both shared and distributed memory. In this paper, we improve MPI performance, the most popular library interface for high-performance computing, using multithreading for collective communications. Multithreading has several advantages over multiprocessing on shared memory for collectives: direct memory access can reduce buffer copying and system resource overhead; and multithreading features fast synchronization between threads.

For multithreading-based HMPI_Allreduce, we design hierarchy-aware algorithms to reduce intersocket data transfer, utilize shared last-level cache in modern CMPs to reduce data transfer latency, and adopt strip mining to improve the cache efficiency when the dataset size exceeds the capacity of the last-level cache.

We find that tree-based HMPI_Allreduce is best for small vector sizes while tiled-reduce followed by a broadcast is best for large vector sizes. We compare the best allreduce algorithms of HMPI with other MPI implementations. Experimental results show that multithreading yields significant performance improvement for MPI collective communication. On 16-node Xeon cluster and 8-node Opteron cluster, HMPI_Allreduce gets on average 2.3X and 2.1X speedup over MVAPICH2 1.6, gets on average 5.2X and 3.5X speedup over MPICH2 1.4, and gets on average 2.4X and 1.9X speedup over Open MPI 1.6. We also establish performance models for all the algorithms of HMPI_Allreduce. The consistency of predicted and measured running time shows the correctness of the performance models, which can be used for algorithm selection on new platforms.

Architecture trends indicate that the number of cores will grow continuously and that deep memory hierarchies will be necessary to reduce power consumption and contention on buses. Thus, we expect that NUMA effects will be even more important on future systems. Our developed techniques, algorithms, and model form a basis for implementing parallel communication algorithms on such future architectures.

Acknowledgments The work is supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under Award number DE-FC02-10ER26011 and DOE Office of Science, Advanced Scientific Computing Research, under Award number DE-AC02-06CH11357. Li is supported in part by National Key Basic Research and Development Program of China under No.2013CB329605 and No.2013CB329606, and Key Project of the National 25th Year Research Program of China under No.2011BAK08B04. This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under Award number DE-FC02-10ER26011, program manager Lucy Nowell.

References

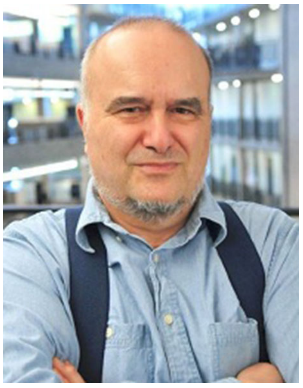
1. AMD: Software optimization guide for AMD family 15h processors (2012).
2. Aulwes, R., Daniel, D., Desai, N., Graham, R., Risinger, L., Taylor, M., Woodall, T., Sukalski, M.: Architecture of LA-MPI, a network-fault-tolerant MPI. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium, p. 15 (2004).
3. Blagojević, F., Hargrove, P., Iancu, C., Yelick, K.: Hybrid PGAS runtime support for multicore nodes. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model PGAS '10, pp. 3:1–3:10. ACM (2010).
4. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: A generic framework for managing hardware affinities in HPC applications. In: Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP '10, pp. 180–186. IEEE Computer Society (2010).
5. Feind, K., McMahon, K.: An ultrahigh performance MPI implementation on SGI ccNUMA Altix systems. Comput. Methods Sci. Technol., 67–70 (2006).
6. Friedley, A., Bronevetsky, G., Lumsdaine, A., Hoefler, T.: Hybrid MPI: efficient message passing for multi-core systems. In: Proceedings of the SC13 IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (2013).
7. Friedley, A., Hoefler, T., Bronevetsky, G., Lumsdaine, A., Ma, C.C.: Ownership passing: efficient distributed memory programming on multi-core systems. In: Proceedings of the 18th ACM symposium on Principles and Practice of Parallel Programming PPoPP'13 (Accepted) (2013).
8. Graham, R.L., Shipman, G.: MPI support for multi-core architectures: optimized shared memory collectives. In: Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 130–140. Springer, Berlin (2008).
9. Hensgen, D., Finkel, R., Manber, U.: Two algorithms for barrier synchronization. Int. J. Parallel Program. **17**(1), 1–17 (1988)
10. Hoefler, T., Mehlan, T., Mietke, F., Rehm, W.: Fast barrier synchronization for InfiniBand. In: Proceedings of the 20th International Parallel and Distributed Processing Symposium IPDPS (2006).
11. Intel: Intel 64 and IA-32 Architectures optimization reference manual (2012).
12. Kamal, H., Wagner, A.: Fg-mpi: fine-grain mpi for multicore and clusters. In: Proceedings of the IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), pp. 1–8 (2010).
13. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MagPIe: MPI's collective communication operations for clustered wide area systems. In: Proceedings of the seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP '99, pp. 131–140. ACM, New York (1999)
14. Li, S., Hoefler, T., Snir, M.: Numa-aware shared-memory collective communication for mpi. In: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing HPDC '13, pp. 85–96. ACM, New York (2013)
15. Mamidala, A., Kumar, R., De, D., Panda, D.: MPI collectives on modern multicore clusters: performance optimizations and communication characteristics. In: Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid CCGRID '08, pp. 130–137 (2008).
16. Mellor-Crummey, J.M., Scott, M.L.: Synchronization without contention. SIGPLAN Notice **26**(4), 269–278 (1991)
17. Molka, D., Hackenberg, D., Schone, R., Muller, M.S.: Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques PACT '09, pp. 261–270. IEEE Computer Society, Washington (2009)
18. MPI Forum: MPI: A Message-passing interface standard. version 2.2 (2009).
19. Negara, S., Zheng, G., Pan, K.C., Negara, N., Johnson, R.E., Kalé, L.V., Ricker, P.M.: Automatic MPI to AMPI program transformation using photran. In: Proceedings of the Conference on Parallel Processing Euro-Par, pp. 531–539. Springer, Berlin (2011)

20. Board, OpenMP Architecture Review: Application program interface version **3**, 1 (2011)
21. Pérache, M., Carribault, P., Jourdren, H.: MPC-MPI: an MPI implementation reducing the overall memory consumption. In: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 94–103. Springer, Berlin (2009).
22. Rabenseifner, R.: Optimization of collective reduction operations. Computational Science-ICCS pp. 1–9. Springer, Berlin (2004).
23. Sistare, S., Vaart, R., Loh, E.: Optimization of MPI collectives on clusters of large-scale SMP's. In: Proceedings of the ACM/IEEE 1999 Conference on Supercomputing (1999).
24. Tang, H., Shen, K., Yang, T.: Program transformation and runtime support for threaded MPI execution on shared-memory machines. ACM Trans. Program. Lang. Syst. (TOPLAS) **22**(4), 673–700 (2000)
25. Tang, H., Yang, T.: Optimizing threaded MPI execution on SMP clusters. In: Proceedings of the 15th International Conference on Supercomputing ICS '01, pp. 381–392. ACM (2001).
26. Thakur, R., Gropp, W.: Improving the performance of collective operations in MPICH. In: Proceedings of the 10th European PVM/MPI User's Group Meeting in Recent Advances in Parallel Virtual Machine and Message Passing Interface. Lecture Notes in Computer Science, vol. 2840, pp. 257–267. Springer, Berlin (2003).
27. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. Int. J. High Perform. Comput. Appl. **19**, 49–66 (2005)
28. Tipparaju, V., Nieplocha, J., Panda, D.: Fast collective operations using shared and remote memory access protocols on clusters. In: Proceedings of the International IEEE on the Parallel and Distributed Processing Symposium, (2003).
29. Yew, P.C., Tzeng, N.F., Lawrie, D.: Distributing hot-spot addressing in large-scale multiprocessors. IEEE Trans. Comput. **36**(4), 388–395 (1987)
30. Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP '10, pp. 203–212. ACM (2010).
31. Zhang, J., Behzad, B., Snir, M.: Optimizing the BarnesspsHut algorithm in UPC. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC '11, pp. 75:1–75:11. ACM (2011).
32. Zhu, H., Goodell, D., Gropp, W., Thakur, R.: Hierarchical collectives in MPICH2. In: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 325–326. Springer, Berlin (2009).

Shigang Li is a Ph.D candidate in the School of Computer and Communication Engineering, University of Science and Technology Beijing. His major is Computer architecture. He is very interested in parallel programming model, like MPI and OpenCL, and large-scale parallel applications.

Torsten Hoefler is an Assistant Professor for Computer Science at ETH Zürich where he leads the Scalable Parallel Computing Laboratory. He is the co-chair of the collective operations working group in the MPI Forum and he is interested in Collective Communications, Process Topologies, One Sided Operations, and Hybrid Programming in MPI.

Chungjin Hu is a Professor in School of Computer and Communication Engineering, University of Science and Technology Beijing. His research area includes parallel computing, parallel compiler technology, parallel software engineering, grid computing and network storage architecture.



Marc Snir is Michael Faiman and Saburo Muroga Professor in the Department of Computer Science, University of Illinois at Urbana-Champaign. He has pursued theoretical and practical research in parallel computing for more than 30 years, published numerous papers, and involved in several major projects in parallel computing, including the design of the MPI library and the design if IBM's high-performance-computing platforms.