NUMA-Aware Shared-Memory Collective Communication for MPI

Shigang Li*
School of Computer and Communication Engineering
University of Science and Technology Beijing
shigangli.cs@gmail.com

Torsten Hoefler
Department of Computer Science
ETH Zurich
htor@inf.ethz.ch

Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
and Argonne National Laboratory
snir@illinois.edu

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 optimizations of MPI collectives for clusters of NUMA nodes. We develop performance models for collective communication using shared memory, and we develop 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 traditional 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 average 2.5X and 2.3X speedup over MVAPICH2, respectively. Our techniques enable an efficient implementation of collective operations on future multi- and manycore systems.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

Keywords

MPI, multithreading, MPI_Allreduce, collective communication, NUMA

1. INTRODUCTION

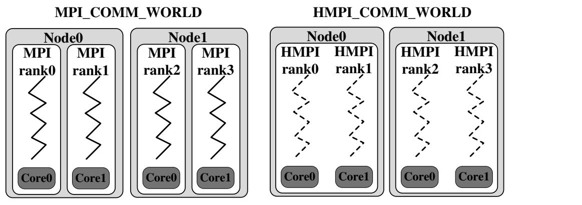

Figure 1: Runtime model of traditional MPI vs. Hybrid MPI (HMPI)

Applications using the Message Passing Interface (MPI) [14] often run multiple MPI processes at each node. As the number of cores per node keeps growing and as nodes exhibit increasingly nonuniform memory, the performance of MPI collectives becomes more and more dependent on the performance of the intranode communication component of such collectives. Currently, implementations of MPI collectives take advantage of shared memory in two ways. In the first way, collectives are built by using point-to-point message passing, which uses shared memory as a transport layer inside a node. Collectives in MPICH2 [28] are implemented in this way. In the second way, Collectives are implemented directly atop shared memory [6, 11, 23]: 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. Collectives in Open MPI [6] and some collectives in MVAPICH2 [11] are implemented in this way. The second approach reduces the number of memory transfers [6, 23], but still requires extra data movement. Also, shared memory is usually a limited system resource.

The hybrid MPI library (HMPI) [5] avoids these overheads by replacing the common process-based rank design with a thread-based rank; all MPI ranks (threads) on a node are running within the same address space, as illustrated in Figure 1. HMPI takes advantage of shared memory within a node and utilizes the existing MPI infrastructure for internode communication. A similar technique can be applied to existing MPI implementations by using techniques such as cross-memory attach (XPMEM [24]) to create a globally shared heap for all MPI processes on a node. The detailed techniques for sharing the heap are well understood [20] and thus not a topic of this paper.

Sharing heap data directly can yield significant perfor-

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

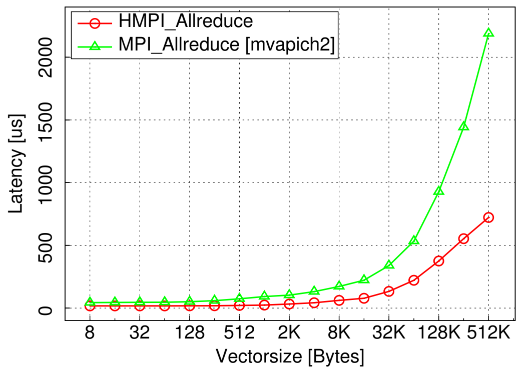

Figure 2: Latency of HMPI_Allreduce vs. traditional MPI_Allreduce on 16 Xeon X5650 nodes

mance gains for the following 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 flags. Applications that run in the process-based model will work with few modifications in the thread-based model: Static variables need to become thread-private. Automatic privatization of global variables [15, 17] can minimize the developer effort. Shared-heap techniques such as XPMEM change the memory allocator to allocate from shared memory, and no further changes are needed.

We demonstrate in this paper the performance advantage of HMPI's thread-based approach, in the context of MPI collectives, in particular, MPI_Allreduce. Figure 2 shows the motivation for our work, namely, that HMPI_Allreduce is significantly faster than traditional MPI_Allreduce. The remainder of this 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 thread-based 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 2.5X and 2.3X speedup over MVAPICH2 1.6, and gets on average 6.3X and 4.7X speedup over MPICH2 1.4.

The key contributions of this paper are as follows:

1. We design NUMA-aware algorithms for thread-based 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.

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 Section 3. Section 3.4 discusses parameter estimation for our model and algorithm selection. A comparison with OpenMP reduction is presented in Section 3.5. Section 3.6 and Section 3.7 present the performance of our thread-based MPI collectives using microbenchmarks and applications on CMP clusters. Section 4 discusses related work, and Section 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 the last-level 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. We assume that the time overhead of consecutive memory access by a thread is approximated as $a+bm$, where m is the vector length (number of cache lines), $a+b$ is the latency for the first cache line access and $1/b$ denotes the bandwidth (relative to cache lines). The average latency for each cache line is less than the latency for the first cache line, because consecutive memory accesses benefit from hardware or software prefetching [9, 1]. 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 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 [13]. 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],

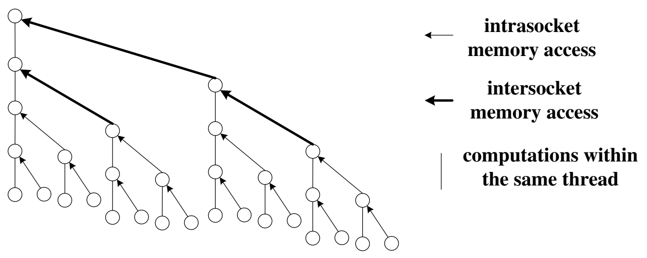
intrasocket memory access
intersocket memory access
computations within the same thread

Figure 3: Binary reduction tree, for 1 node with 4 sockets; each socket includes 4 cores.

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 [13]. To model congestion, we use the formula $a + Bdm$ to represent the delay for d simultaneous accesses where $1/B$ is the total bandwidth of the shared link and $(1/(Bd) \leq 1/b)$. We use $1/B_\alpha$ to represent intrasocked total bandwidth and $1/B_\beta$ to represent intersocket total bandwidth.

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 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 parent in the next round. Otherwise it synchronizes with the parent.

A binary reduction tree is illustrated in Figure 3; 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. If h is the height of the intrasocket reduction tree, then $q \leq (n^{h+1}-1)/(n-1)$, and $h = \lceil \log_n((n-1)q+1) \rceil - 1$. To simplify this formula, we assume q is much larger than 1, so that $h = \lceil \log_n((n-1)q+1)\rceil - 1 = \lceil \log_n(n-1) + \log_n q \rceil - 1 \approx \lceil 1 + \log_n q \rceil - 1 = \lceil \log_n q \rceil$. Similarly, the height of intersocket n-ary tree can be expressed as $\lceil \log_n s \rceil$. Assuming no congestion, the time for the intrasocket n-ary reduction tree is $(n-1)(a_\alpha + b_\alpha m)\lceil \log_n q \rceil$. However, intrasocket reduction may cause congestion in the lower level of the tree, since simultaneous memory accesses share the last-level cache. However, the number of simultaneous accesses decreases as the tree goes to higher levels. To be more accurate, we use

$$T_{intra-red} = (n-1)\sum_{i=0}^{\lceil \log_n q \rceil} max(a_\alpha + b_\alpha m, a_\alpha + B_\alpha n^i m) \tag{1}$$

to model the time for the intrasocket n-ary reduction tree. 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 n-ary reduction tree is

$$T_{inter-red} = (n-1)(a_\beta + b_\beta m)\lceil \log_n s \rceil \tag{2}$$

The total time for reduction is

$$T_{red} = T_{intra-red} + T_{inter-red} \tag{3}$$

This function is monotonically increasing in n, so that the best algorithm is always obtained for $n = 2$. This is validated by the experiments described in Section 3.2.

When broadcast is done by using shared memory with an 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 fanout. Experiments described in Section 3.2 show that, for the systems under consideration, 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 Figure 4. 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 [26]: After one thread reads one chunk of data 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 an n-ary reduction tree combined with a one- or two-stage broadcast is

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

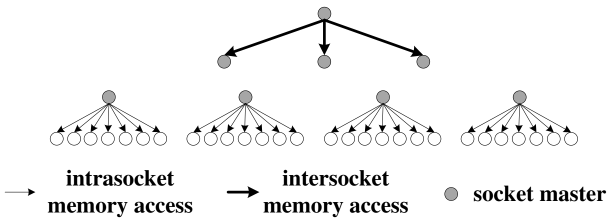

Figure 4: Two-stage broadcast, for 1 node with 4 sockets; each socket includes 4 cores.

2.3 Dissemination

The dissemination algorithm [7, 8] achieves complete dissemination of information among p threads in $\lceil \log_2 p \rceil$ synchronized steps. If the number of threads p is a power of 2, it needs $N = \lceil \log_2 p \rceil$ steps to accomplish an allreduce. During step i ($i = 0, 1, ..., N-1$), thread j ($j = 0, 1, ..., p-1$) combines the data from thread $(j - 2^i + p) \mod p$ with its own data. This algorithm has fewer steps but more total communication than does the reduce-broadcast algorithm.

The dissemination algorithm can be laid out so that intersocket communication happens only in the last $\lceil \log_2 s \rceil$ steps, as presented in Figure 5. In the last $\lceil \log_2 s \rceil$ steps, all threads within a socket need the same chunk of data from the other sockets. While this approach benefits from synergistic prefetching [26], it does suffer from congestion to some extent in practice.

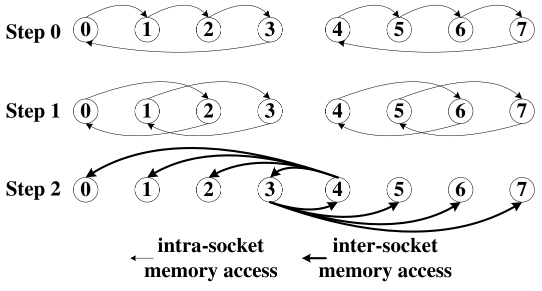

Figure 5: Dissemination, for 1 node with 2 sockets; each socket includes 4 cores.

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 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

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

If the number of threads p is not a power of 2, the final results may not be correct using the above algorithm. To solve this problem, we put the excess threads in a different set that is handled separately, so that the number of the remaining threads is the largest possible power of 2. A regular dissemination algorithm can be used for the remaining threads; the excess threads then copy the final result from a thread in the same socket. A dissemination algorithm for $p = 12$ is given in Figure 6.

Let $u = \lfloor log_2 q \rfloor$, $q' = 2^u$ and $r = q - q'$. If p is not a power of 2, then the total time taken by dissemination is

$$T_{dis-non-pwr2} = (a_\alpha + B_\alpha q' m)\lceil \log_2 q' \rceil + (a_\beta + B_\beta q' m)\lceil \log_2 s \rceil + 2(a_\alpha + B_\alpha rm) \quad (7)$$

2.4 Tiled Reduce-Broadcast

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

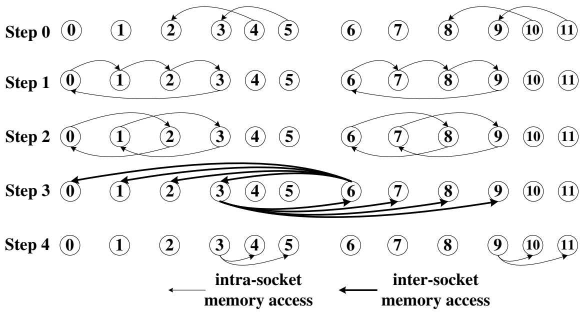

Figure 6: Dissemination, for 1 node with 2 sockets; each socket includes 6 cores.

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, the temporary buffer is padded with dummy data to the cache line boundary and partitioned at cache line granularity. In Figure 7, $Thread_0$ 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.

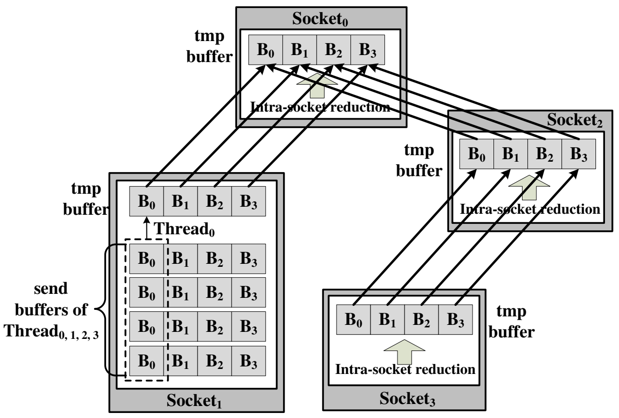

Figure 7: Hierarchical parallel-reduce, for 1 node with 4 sockets; each socket includes 4 cores.

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

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

The time spent in intersocket reduction is

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

The total reduction time is

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

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

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

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 approximation, an optimized allreduce is obtained by composing the best intranode allreduce with the best internode allreduce. Internode collectives have been extensively studied [22, 18, 21], so we focus on the intranode part.

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 Section 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 Section 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.

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 Figure 8. 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 Figure 9.

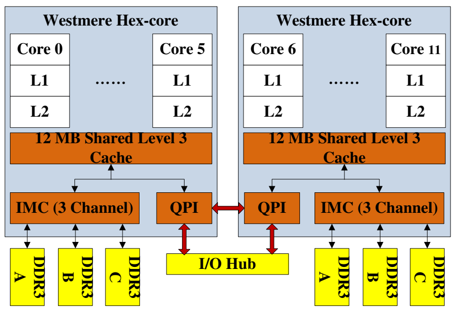

Figure 8: Westmere, 2 sockets, total 12 cores

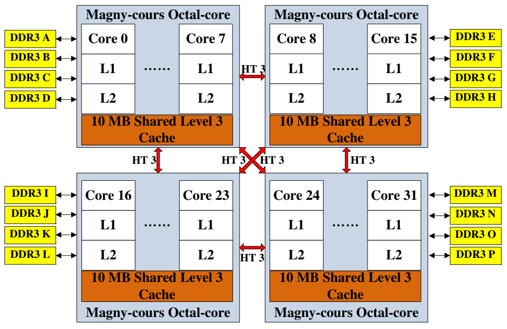

Figure 9: Magny-Cours, 4 sockets, total 32 cores

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 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. To save space, if similar results are obtained from both systems, we present the results only on one architecture.

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 10 and 11 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.

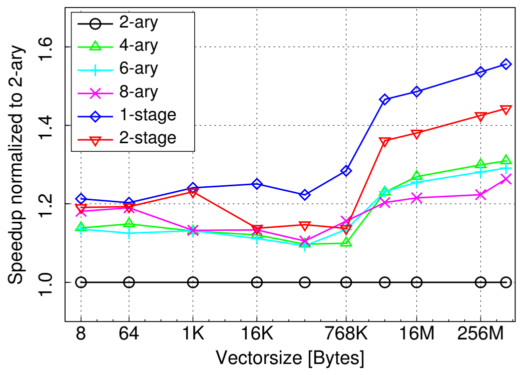

Figure 10: Performance comparison of different n-ary broadcast trees and 1-stage/2-stage broadcast trees on 12-core Westmere

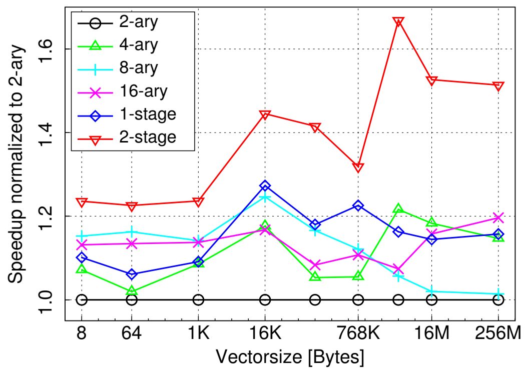

Figure 11: Performance comparison of different n-ary broadcast trees and 1-stage/2-stage broadcast trees on 32-core Magny-Cours

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 Figures 10 and 11, 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.

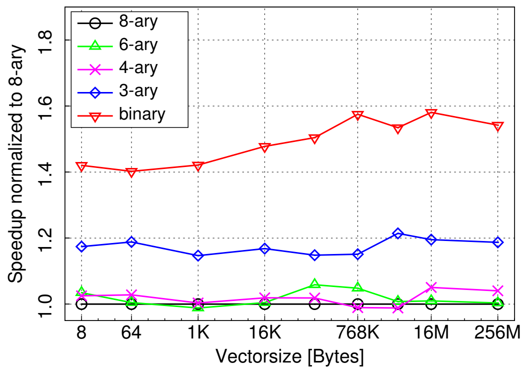

Figure 12: Performance comparison of different n-ary reduction trees on 32-core Magny-Cours

Figure 12 presents the performance comparison of different n-ary reduction trees on Magny-Cours. As expected, a binary reduction tree dominates all other n-ary reduction trees. Similar results have been obtained for Westmere. 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.

3.2 Performance of Tiled-Reduce

We evaluate the performance of the hierarchical tiled-reduce algorithm presented in Section 2.4 with other similar implementations, namely, a naive tiled-reduce and cyclic tiled-reduce [11]. Tiled-reduce does the reduction in parallel but without consideration of the NUMA hierarchy. This leads to high contention for intersocket memory accesses. Mamidalaet al. [11] 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 13 show the results on Magny-Cours. Cyclic tiled-

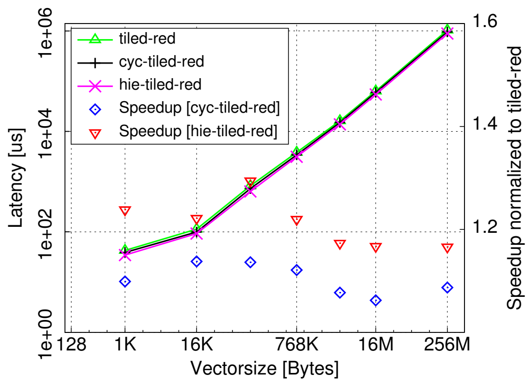

Figure 13: Tiled-reduce on 32-core Magny-Cours

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 vs. 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 14 shows the result on Westmere.

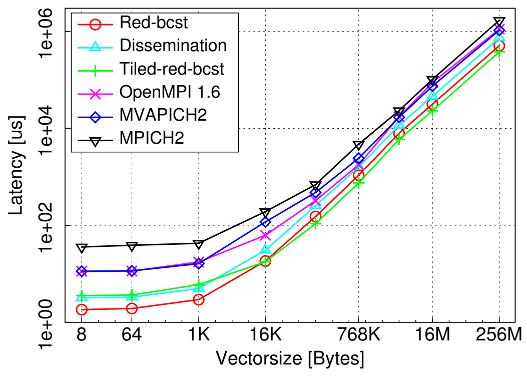

Figure 14: Performance comparison between HMPI's Allreduce algorithms and several MPI implementations on a 12-core Westmere CMP

HMPI's Allreduce always outperforms traditional approaches used in other MPI implementations. 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 nonpower-of-2 thread counts, as reflected in Equation (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 3.6X lower latency than Open MPI 1.6, 4.3X lower latency than MVAPICH2, and 8.8X lower latency than MPICH2.

Overall, 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 Equation (5) with Equation (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. Similar results have been achieved on Magny-Cours.

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 Equation (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 [13], which provides memory latency benchmarks for multicore and multiprocessor x86-based systems.

Figure 15 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.

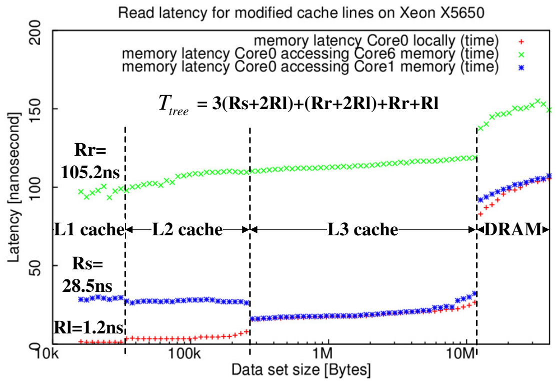

Figure 15: Modified cache line read latency on Westmere. Rl denotes latency of read local L1 cache, Rs denotes latency of read other L1 cache but within the same socket, and Rr denotes latency of read other L1 cache from remote socket.

On the 12-core Westmere, Equation (5) is unfolded as $T_{red-bcst} = (a_\alpha + 2B_\alpha m) + 2(a_\alpha + b_\alpha m) + (a_\beta + b_\beta m) + (a_\beta + B_\beta qm)$, where $m = 1, q = 6$. Figure 16 shows the steps in the tree-based algorithm on Westmere.

(1) In $Step_0$, $Step_1$, and $Step_2$, a thread reads a cache

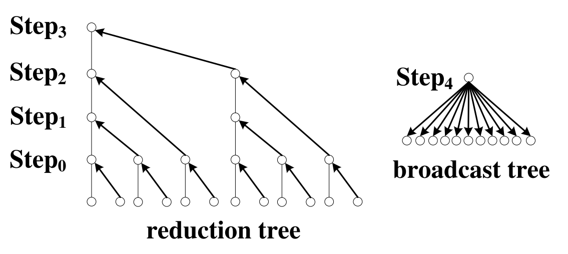

Figure 16: Steps in the tree-based algorithm on Westmere

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 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 Figure 17. 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 Equation (5) and Equation (11) on Westmere, and we present the predicted runtime obtained from the performance models and the real runtime of parallel reduce followed by a broadcast and tree-based algorithms in

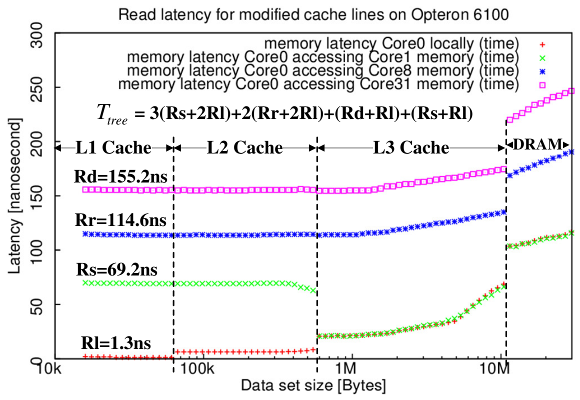

Figure 17: Modified cache line read latency on Magny-Cours. Rr denotes the L1 read latency from a horizontal or vertical remote socket, and Rd denotes the L1 read latency from a diagonal remote socket.

Figure 18. We see that the models can predict latencies accurately and the relative errors are all below 5%.

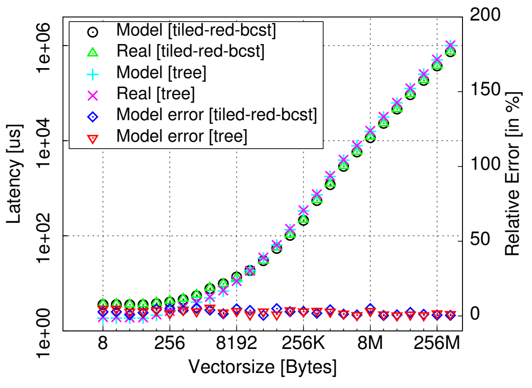

Figure 18: Performance model for tree and tiled-reduce-broadcast on Westmere

Figure 19 shows that the tree-based algorithm gets the best performance for small vector size, while the crosspoint at 16 KB indicates that the best algorithm switches to tiled-reduce followed by a broadcast. Similarly, the crosspoint on Magny-Cours is 14 KB.

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.

Figure 20 compares our best Reduce algorithms with an OpenMP REDUCTION clause [16] on Westmere. C based OpenMP does not support reduction on vectors, so we use Fortran based OpenMP reduction for our comparison. We

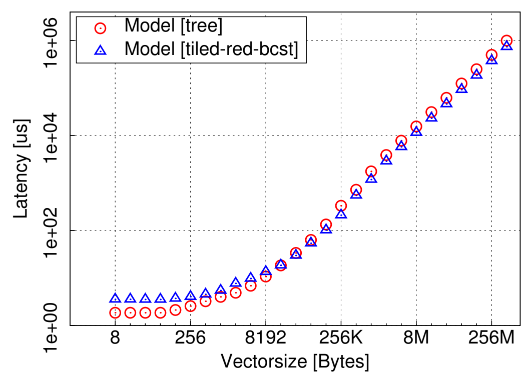

Figure 19: Selecting the best algorithm with the performance model on Westmere

use 12 threads for both HMPI Reduce and OpenMP on Westmere. We see that HMPI Reduce achieves on average 1.5X speedup over OpenMP for all the vector sizes, probably due to the hierarchy-aware HMPI Reduce implementation on NUMA machines.

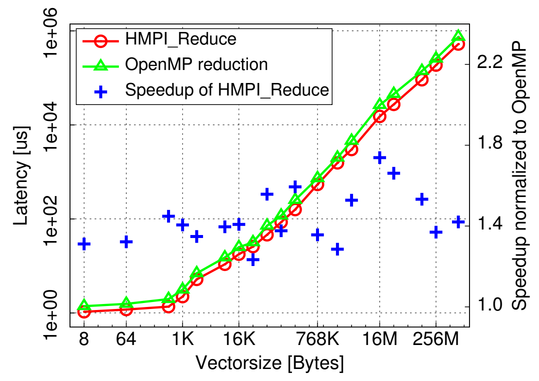

Figure 20: Performance comparison between the best HMPI Reduce algorithm and OpenMP reduction on Westmere

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 Figure 21. 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, 18, 21], 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 and MVAPICH2 1.6. For both MPI and HMPI, the total number of launched ranks is equal to the total number of cores in the cluster. On 16-node Xeon X5650 and 8-node Opteron 6100 clusters, HMPI_Allreduce gets on average 2.5X and 2.3X speedup over MVAPICH2, and gets on average 6.3X and 4.7X speedup over MPICH2. To save space, only the performance comparison between HMPI_Allreduce and MVAPICH2 on 8-node Opteron 6100

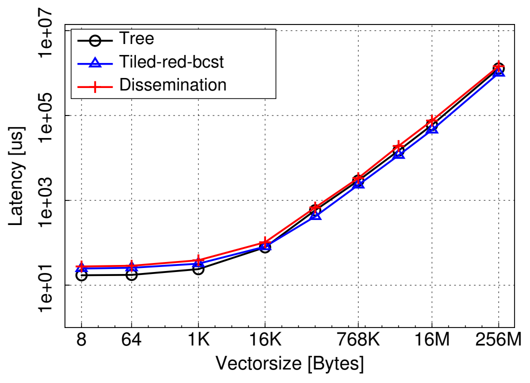

Figure 21: Performance comparison between HMPI_Allreduce algorithms on 16-node Xeon X5650 running on 192 cores

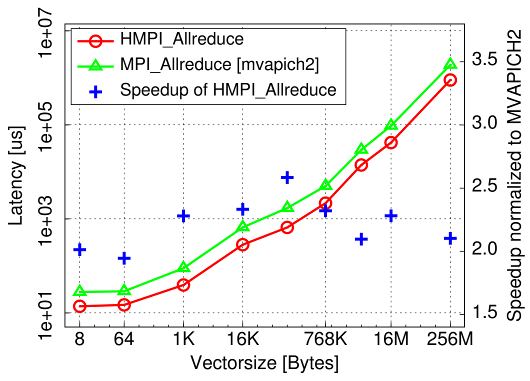

Figure 22: Performance comparison between HMPI's Allreduce and MVAPICH2 on 8-node Opteron 6100 running on 256 cores

cluster is shown in Figure 22. Figures 23 and 24 show the performance of HMPI_Bcast and HMPI_Reduce on the 16-node Xeon X5650 cluster. HMPI_Bcast and HMPI_Reduce get on average 1.8X and 1.4X speedup over MVAPICH2 for all vector sizes respectively. Figure 25 shows that HMPI_Barrier scales better than MVAPICH2. 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 columnwise 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 use an MPI_Reduce to sum the corresponding part of the intermediate results. As

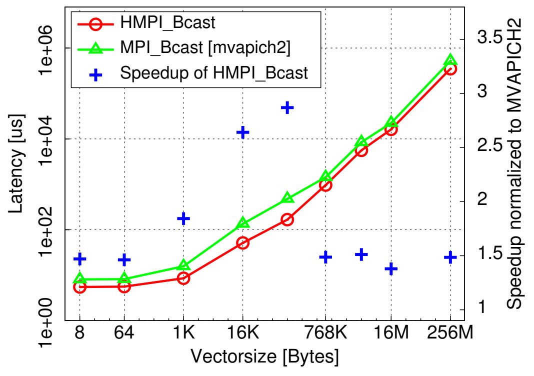

Figure 23: Performance comparison between HMPI's Bcast and MVAPICH2 on 16-node Xeon X5650 running on 192 cores

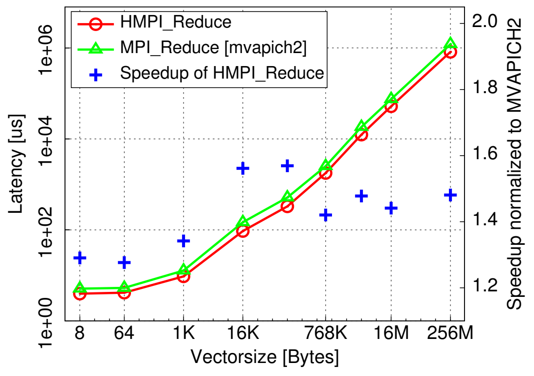

Figure 24: Performance comparison between HMPI_Reduce and MVAPICH2 on 16-node Xeon X5650 running on 192 cores

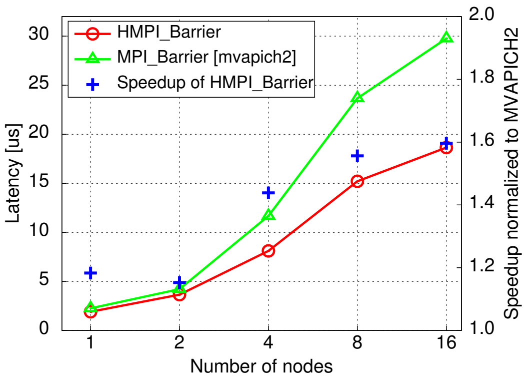

Figure 25: Performance comparison between HMPI_Barrier and MVAPICH2 on Xeon X5650 cluster (12, 24, 48, 96, and 192 cores)

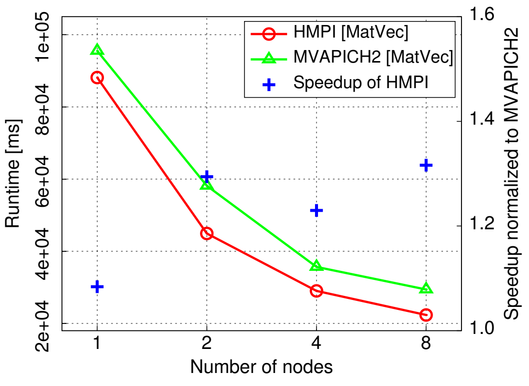

Figure 26: Dense matrix vector multiplication on Xeon X5650 cluster (12, 24, 48, and 96 cores)

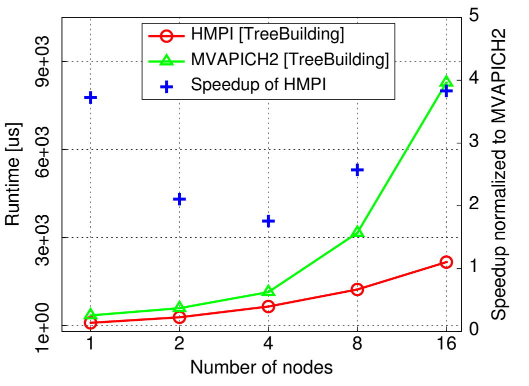

Figure 27: Tree-building in Barnes-hut on Xeon X5650 cluster (12, 24, 48, 96, and 192 cores)

illustrated in Figure 26, HMPI has better scalability and gets on average 1.2X speedup over MVAPICH2.

Allreduce is a key operation in the tree-building algorithm of the Barnes-Hut n-body simulation [27]. 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 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. HMPI shows on average 2.8X speedup over MVAPICH2. Moreover, as the number of nodes increases, the improvement becomes more apparent, as illustrated in Figure 27. 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 [11]. 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 [19], also have support for shared memory.

Graham and Shipman [6] 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. [10] 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 [20] 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 [25] and the MCS Barrier [12], are designed to distribute hotspot 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 [7, 12] 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. [26] 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 2.5X and 2.3X speedup over MVAPICH2 1.6, and gets on average 6.3X and 4.7X speedup over MPICH2 1.4. 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.

6. 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.2013CB329606 and Key Project of the National Twelfth-Five Year Research Program of China under No.2011BAK08B04.

7. REFERENCES

[1] AMD. Software Optimization Guide for AMD Family 15h Processors, January 2012.

[2] R. Aulwes, D. Daniel, N. Desai, R. Graham, L. Risinger, M. Taylor, T. Woodall, and M. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium.*, page 15, April 2004.

[3] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick. Hybrid PGAS runtime support for multicore nodes. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 3:1–3:10. ACM, 2010.

[4] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186. IEEE Computer Society, 2010.

[5] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma. Ownership Passing: efficient distributed memory programming on multi-core systems. In *PPoPP'13. Proceedings of the 18th ACM symposium on Principles and Practice of Parallel Programming*, 2013. Accepted at PPoPP'13.

[6] R. L. Graham and G. Shipman. 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*, pages 130–140, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] D. Hensgen, R. Finkel, and U. Manber. Two

algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, Feb. 1988.

[8] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. Fast barrier synchronization for InfiniBand. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[9] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, April 2012.

[10] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. 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, pages 131–140, New York, NY, USA, 1999. ACM.

[11] A. Mamidala, R. Kumar, D. De, and D. Panda. MPI collectives on modern multicore clusters: performance optimizations and communication characteristics. In *8th IEEE International Symposium onCluster Computing and the Grid. CCGRID '08.*, pages 130 –137, May 2008.

[12] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. *SIGPLAN Not.*, 26(4):269–278, April 1991.

[13] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 261–270, Washington, DC, 2009. IEEE Computer Society.

[14] MPI Forum. MPI: A Message-Passing Interface standard. version 2.2, September 2009.

[15] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kalé, and P. M. Ricker. Automatic MPI to AMPI program transformation using photran. In *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, pages 531–539, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] OpenMP Architecture Review Board. Application Program Interface Version 3.1. July 2011.

[17] M. Pérache, P. Carribault, and H. Jourdren. 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*, pages 94–103, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] R. Rabenseifner. Optimization of collective reduction operations. *Computational Science-ICCS*, pages 1–9, 2004.

[19] S. Sistare, R. Vaart, and E. Loh. Optimization of MPI collectives on clusters of large-scale SMP's. In *Supercomputing, ACM/IEEE 1999 Conference*, November 1999.

[20] H. Tang and T. Yang. Optimizing threaded MPI execution on SMP clusters. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 381–392. ACM, 2001.

[21] R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 257-267 10th European PVM/MPI User's Group Meeting*, pages 257–267. Springer Verlag, 2003.

[22] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.

[23] V. Tipparaju, J. Nieplocha, and D. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International.* IEEE, 2003.

[24] M. Woodacre, D. Robb, D. Roe, and K. Feind. The SGI AltixTM 3000 global shared-memory architecture, 2005.

[25] P.-C. Yew, N.-F. Tzeng, and D. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, april 1987.

[26] E. Z. Zhang, Y. Jiang, and X. Shen. 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, pages 203–212. ACM, 2010.

[27] J. Zhang, B. Behzad, and M. Snir. Optimizing the Barnes-Hut algorithm in UPC. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 75:1–75:11. ACM, 2011.

[28] H. Zhu, D. Goodell, W. Gropp, and R. Thakur. 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*, pages 325–326, Berlin, Heidelberg, 2009. Springer-Verlag.