Cache-oblivious MPI all-to-all communications based on Morton order

Shigang Li, *Member IEEE,* Yunquan Zhang, *Member IEEE,* and Torsten Hoefler, *Member IEEE*

Abstract—Many-core systems with a rapidly increasing number of cores pose a significant challenge to parallel applications to use their complex memory hierarchies efficiently. Many such applications rely on collective communications in performance-critical phases, which become a bottleneck if they are not optimized. We address this issue by proposing cache-oblivious algorithms for MPI_Alltoall, MPI_Allgather, and the MPI neighborhood collectives to exploit the data locality. To implement the cache-oblivious algorithms, we allocate the send and receive buffers on a shared heap and use Morton order to guide the memory copies. Our analysis shows that our algorithm for MPI_Alltoall is asymptotically optimal. We show an extension to our algorithms to minimize the communication distance on NUMA systems while maintaining optimality within each socket. We further demonstrate how the cache-oblivious algorithms can be applied to multi-node machines. Experiments are conducted on different many-core architectures. For MPI_Alltoall, our implementation achieves on average 1.40X speedup over the naive implementation based on shared heap for small and medium block sizes (less than 16 KB) on a Xeon Phi KNC, achieves on average 3.03X speedup over MVAPICH2 on a Xeon E7-8890, and achieves on average 2.23X speedup over MVAPICH2 on a 256-node Xeon E5-2680 cluster for block sizes less than 1 KB.

Index Terms—cache-oblivious algorithms, collective communication, NUMA, MPI_Alltoall, MPI_Allgather, neighborhood collectives.

1 Introduction

While both frequency and Dennard scaling have ended, Moore's law still holds and leads to a steadily growing number of cores. Many-core processors with massive intra-node parallelism and complex memory hierarchies are now commonplace. Compute nodes are composed of complex Networks-on-Chip (NoCs) arranged in cache-coherent multi-chip configurations with increasingly expensive data movement costs. The Message Passing Interface (MPI) [1] is used ubiquitously for communication in parallel applications. For many MPI applications, collective operations ("collectives") are performance critical and directly determine scalability. Thus it is imperative to achieve highest performance of collective data-transfers using algorithms that exploit the inherent data-locality as well as the memory hierarchy for intra- and inter-node parts of the communication.

Intra-node communication is implemented using cache line transfer on the NoC [2]. However, designing optimal communication algorithms in terms of cache efficiency is non-trivial. The first challenge comes from MPI itself: MPI often launches multiple processes at each node, and each process has a private virtual address space. Traditionally, data is copied in and then copied out of a shared system space [3], which leads to extra memory copies. To deal with this problem, three approaches have been developed: (1) kernel-assisted communication [4], [5], (2) thread-based ranks [6], [7], and (3) shared heaps [8], [9]. All three approaches can reduce the number of memory copies of intra-node communication to one (the minimum possible in MPI). However, traditional algorithms for MPI collectives [10] mainly focus on reducing the latency and bandwidth overhead over the network, and ignore the cache efficiency. In fact, inter-node communication faces the same problem, when considering Dynamic Random Access Memory (DRAM) as a private cache for each node on a multi-node machine.

A second challenge stems from the diversity of many-core hardware itself: Processors may have very different memory hierarchies, such as a two-level cache for Intel® Xeon Phi™ KNC [11] or a three-level cache for Intel® Xeon E7 [12]. The cache capacity of each level may also be different. Furthermore, there are various arrangements of main memory, including Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA), which has to be considered in algorithm design to enable best performance [7], [13]. This hardware diversity causes a high programming effort to tune the algorithms.

Cache-oblivious algorithms [14], [15], [16], [17] are asymptotically optimal (often within a factor of two) in terms of cache complexity without considering any hardware parameters. Thus, these algorithms enable portable performance on different architectures. To carry these benefits towards implementations of MPI collectives, we propose cache-oblivious algorithms for MPI all-to-all style operations, including MPI_Alltoall, MPI_Allgather, and their neighborhood versions (many-to-many collectives), and demonstrate their performance advantages. Figure 1 motivates our work. As expected, a naive implementation based on shared heap for MPI_Alltoall, in which each process sequentially copies data blocks into its own receive buffer, is faster than the traditional MPI because of less memory copies. However, compared with the naive

- *S. Li is with State Key Laboratory of Computer Architecture, Institute of Computing Technologies, Chinese Academy of Sciences, Beijing, China. E-mail: shigangli.cs@gmail.com (Corresponding Author)*
- *Y. Zhang is with State Key Laboratory of Computer Architecture, Institute of Computing Technologies, Chinese Academy of Sciences, Beijing, China. E-mail: yunquan.cas@gmail.com*
- *T. Hoefler is with Department of Computer Science, ETH Zurich, Switzerland. E-mail: htor@inf.ethz.ch*

implementation based on shared heap, our cache-oblivious implementation further achieves 1.40X speedup in the geometric mean when the block size is less than 16 KB (i.e., the geometric mean of the speedups for the block sizes from 8 bytes to 8 KB is 1.40X), and performs equally for larger block sizes.

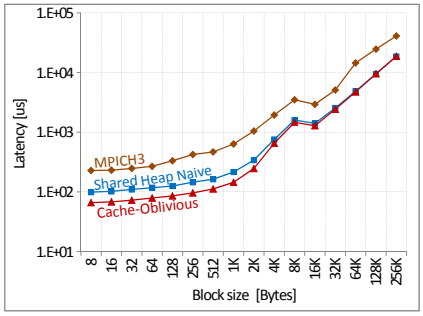


Fig. 1. Latency comparison of MPI_Alltoall between traditional MPI, naive shared heap, and cache-oblivious implementations based on shared heap on 60-core Intel Xeon Phi KNC.

The key idea is to arrange the order of transferring the send buffers to the corresponding receive buffers in Morton order [18]. The algorithms are implemented based on a shared heap, which is established by POSIX shared memory and overriding of dynamic memory allocation functions. If the send and receive buffers in a users' code are allocated on the heap, the code directly benefits from our cache-oblivious implementations without any modification. However, if the send or receive buffers are on the stack, users have to change them to be allocated on the heap to benefit from our implementations. We also demonstrate how the idea can be applied to multi-node machines. Compared with well-tuned MPI libraries, our cache-oblivious implementations achieve significant performance improvements on several different architectures. The key contributions are as follows:

1) We propose cache-oblivious algorithms for MPI all-to-all style collectives based on Morton order, and prove the optimality by cache complexity analysis.
2) We improve the proposed cache-oblivious algorithms for NUMA architectures to minimize the total distance of data transfers.
3) We extend the cache-oblivious algorithms for multi-node machines, in which DRAM is considered as a private cache for each node.
4) We propose an ordered-buffer approach combined with Morton order to exploit the data locality for neighborhood collectives.
5) We perform benchmark and application studies on different many-core machines to assess the benefit of the proposed cache-oblivious algorithms.
6) We compare our analytic bounds with the measured cache misses to demonstrate that the performance advantage is due to better data locality.

In the next section, we discuss our cache-oblivious algorithms for all-to-all style collectives. Sections 2.1 and 2.2 present the cache-oblivious algorithms of MPI_Alltoall and MPI_Allgather based on Morton order. Section 2.3 analyzes their cache complexity. Sections 2.4 and 2.5 discuss the improved algorithms for NUMA architectures and multi-node variants. Section 3 discusses an ordered-buffer approach combined with Morton order for neighborhood collectives. Experimental results are presented in Section 4. Section 5 discusses related work, and Section 6 concludes.

2 Cache-Oblivious Algorithms for All-to-All Style Collectives

For intra-node collectives, our algorithm design is based on a shared heap [9], [19]. We briefly explain how to establish the shared heap. First, we use POSIX shared memory APIs to create and open a shared memory object, and map the object to the virtual address space shared by the processes. Then, the shared address space is equally partitioned among the processes within a node. Next, we override the dynamic memory allocation functions provided by the system, so that each process can allocate and release memory on its own partition. In this way, each process can directly access the data allocated on the partitions of other processes. All the send/receive buffers and the auxiliary arrays shown in the following, are allocated on the shared heap. Note that any MPI library can do this legally following the specification. A program with buffers on the stack cannot benefit from the shared heap solution, unless the code is modified to allocate the buffers on the shared heap.

2.1 MPI_Alltoall and MPI_Allgather Based on Morton Order

For MPI_Alltoall, also known as all-to-all personalized exchange, every process sends a distinct data block to every other process. Processes can view all send buffers as a 2D matrix, of which each dimension's size is equal to the number of processes involved and each element represents a data block; and so do the receive buffers. We name these two matrices as 'send-buffer matrix' and 'recv-buffer matrix', respectively. An all-to-all personalized exchange is equivalent to transposing the send-buffer matrix and writing the results to the recv-buffer matrix.

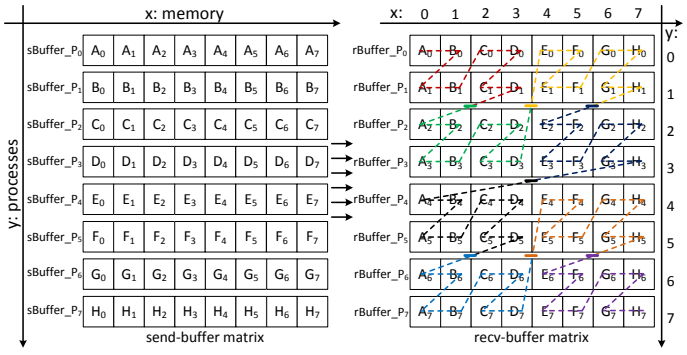


Fig. 2. MPI_Alltoall with 8 processes based on Morton order. The Z-shaped curve is equally divided into 8 segments as indicated by the colors and each one is handled by a different process.

In a naive implementation of MPI_Alltoall, each process copies a column of the send-buffer matrix into a row of the recv-buffer matrix. The access to the send-buffer matrix exhibits poor spatial locality because of the row-major property of the matrix. To have good spatial locality for both send-buffer and recv-buffer matrices, we use Morton order

[18] (also known as Z-order) to sort the elements of the recv-buffer matrix. For MPI_Alltoall with P processes, the recv-buffer matrix has 2 dimensions (x and y) with integer coordinates $0 \leq x \leq P-1$ and $0 \leq y \leq P-1$. By interleaving the bits of the binary values of x and y, we get a set of values (called 'Z-values' here). Taking $x = 2 = 010_2$ and $y = 1 = \mathbf{001}_2$ as an example, we get the Z-value = $\mathbf{0}0\mathbf{1}\mathbf{1}0_2 = 6$. The pairs of coordinates are sorted in the numerical order of their corresponding Z-values, and then stored sequentially in a 2-tuple array of length P^2 (for a total of P^2 pairs of coordinates). This 2-tuple array is logically partitioned into P segments. Each process sequentially accesses one segment of the 2-tuple array, and copies the data block with coordinates (x, y) of the send-buffer matrix into the data block with coordinates (y, x) of the recv-buffer matrix. If we connect the coordinates in the numerical order of the Z-values, we get a recursively Z-shaped curve, as shown in Figure 2. The data blocks, which are close to each other in the 2D matrix, are also close to each other in the Z-shaped curve. Thus, following the Z-shaped curve, the spatial locality of both send-buffer and recv-buffer matrices is exploited.

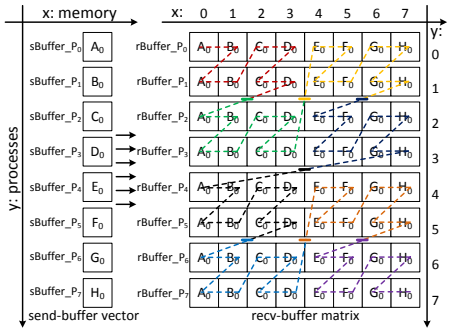
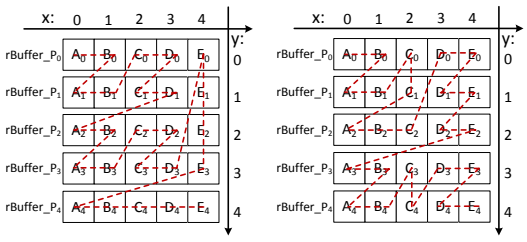


Fig. 3. MPI_Allgather with 8 processes based on Morton order. The Z-shaped curve is equally divided into 8 segments as indicated by the colors and each one is handled by a process.

For MPI_Allgather, also known as all-to-all broadcast, each process sends the same data block to all other processes. Each process can view the send buffers as a vector (we call it "send-buffer vector") and the receive buffers as a matrix (we call it "recv-buffer matrix"). In a naive implementation of MPI_Allgather, each process copies the send-buffer vector into its receive buffer (a row of the recv-buffer matrix). There is no data reuse when each process accesses the send-buffer vector, which exhibits poor temporal locality. As for MPI_Alltoall, we use Morton order to sort the coordinates of the recv-buffer matrix and then generate the 2-tuple array. Each process sequentially reads the coordinates stored in one segment of the 2-tuple array, and then copies the x-th block in the send-buffer vector into the block with coordinates (y, x) in the recv-buffer matrix. Using our 2-tuple array, the Z-shaped curve is drawn in Figure 3. Once a data block in the send-buffer vector has been accessed, it will soon be accessed again following the Z-shaped curve, which exhibits good temporal locality.

2.2 Metadata Generation

When the number of processes P is a power-of-two, we can avoid the explicit 2-tuple array and generate the coordinates on the fly using high-performance bit manipulation instructions [12]. The intrinsic we use is $\_pext\_u32(source, mask)$ supported by Intel Haswell architecture. For each bit set in the $mask$, the intrinsic extracts the corresponding bits from the $source$ operand and writes them into contiguous lower bits of the return value, with the remaining upper bits of the return value set to 0. For a process with rank i, the Morton codes it deals with are the integers in the range of $[P * i, P * (i + 1) - 1]$. Each process sequentially selects a $code$ in the range, and computes the corresponding coordinates (x, y) as (x = $\_pext\_u32(code$, 0x55555555), y = $\_pext\_u32(code$, 0xAAAAAAAA)). In this case, there is no metadata generation overhead.



(a) Imbalanced curve (b) Balanced curve

Fig. 4. Morton order example where the number of processes is not a power-of-two.

When the number of processes P is not a power-of-two, using the bit-interleaving method discussed above, we may generate a 2-tuple array corresponding to an imbalanced Z-shaped curve. Figure 4(a) shows the Z-shaped curve for $P = 5$, in which the long vertical line (in the right border) and the long horizontal line (in the bottom border) lead to reduced locality. Thus, if P is not a power-of-two, we generate a more balanced Z-shaped curve using binary search. For a 2D matrix M, by dividing the longer dimension by two, we partition M into two submatrices. We call the top or the left submatrix M_0, and the other one M_1. Suppose M_0 has n_0 elements. Then the Z-value for coordinates (x, y) in M is given by the following binary recursion:

$$Zvalue(M, x, y) = \begin{cases} 0 & \text{if } M \text{ only contains } (x, y), \\ Zvalue(M_0, x, y) & \text{if } (x, y) \in M_0, \\ n_0 + Zvalue(M_1, x, y) & \text{if } (x, y) \in M_1. \end{cases}$$

Connecting the coordinates in the numerical order of the Z-values calculated by the recursion, we get a more balanced Z-shaped curve, as shown in Figure 4(b). The coordinates calculation is equivalent to the depth-first search of a binary tree with P^2 leaf nodes. To parallelize the calculation among the P processes, each process calculates the Z-values for the P pairs of coordinates of its own receive buffer. Alternatively, each process (with rank i) calculates the P pairs of coordinates for the Z-values in the range of $[P*i, P*(i+1)-1]$ using a similar binary recursion method (i.e., recursively searching for the coordinates for a given Z-value). We use the latter method since the coordinates to be used by each process are calculated by itself and stored locally. The storage overhead of the 2-tuple array for each process is $2P$, and the computation overhead for each process is $2 \log_2 P * P$, where $2 \log_2 P$ is the height of the tree.

Since MPI_Alltoall and MPI_Allgather use the same 2-tuple array, we only generate one array for both when P is not a power-of-two. The 2-tuple array is created together with the *communicator* structure [1], which contains the information to provide the appropriate scope for all communication operations in MPI. The 2-tuple array can be reused whenever a collective function (such as MPI_Alltoall, MPI_Allgather, or their irregular counterparts) related to the *communicator* is called. Thus, the metadata generation overhead can be amortized across the collective calls on the communicator, but at a storage overhead of $O(P)$.

2.3 Cache Complexity Analysis

We discuss the cache complexity (the number of cache misses) of the proposed algorithms for MPI_Alltoall and MPI_Allgather in this section. Assume an *ideal distributed-cache model* [15] for parallel machines. The model defines a computer with a two-level memory hierarchy. Each core has a private *ideal cache* [14] connected to an arbitrarily large shared main memory. Each private cache is partitioned into cache lines, and L is the number of words in each cache line[1]. Each private cache contains Z words, where $Z = \Omega(L^2)$. Since there is no data dependency between processes in the algorithms to be analyzed, we assume the number of cache misses incurred by each process can be analyzed independently. We use Q for the cache complexity of an algorithm, P for the number of processes, and B for the number of words of each data block, namely the block size.

2.3.1 *Analysis for MPI_Alltoall*

For MPI_Alltoall, each dimension of send- and recv-buffer matrices is of size P. It is not consecutive between two adjacent rows in these two matrices, since each process allocates its own buffers. For the naive implementation discussed in Section 2.1, the access to a row of the recv-buffer matrix is consecutive, which incurs $\lceil PB/L \rceil$ cache misses for each process. However, the access to a column of the send-buffer matrix is not consecutive, which incurs $P\lceil B/L \rceil$ cache misses for each process. Thus, the cache complexity of the naive implementation of MPI_Alltoall is

$$\begin{aligned} Q_{alltoall-naive} &= P\lceil PB/L \rceil + P^2\lceil B/L \rceil \\ &< 2P^2B/L + P^2 + P = O(P^2B/L + P^2). \end{aligned} \quad (1)$$

Whether P^2B/L or P^2 is the dominant term depends on the block size B. Thus, we keep both terms in Equation (1).

Next, we prove that the cache complexity of the sequential algorithm for MPI_Alltoall based on Morton order is asymptotically optimal. We recursively divide the longer dimension of the recv-buffer and send-buffer matrix (or submatrix) by 2, which halves the work recursively and forms a 2-ary task tree. This procedure is called k-*recursive decomposition* [15] which forms a k-ary task tree (here k = 2). Morton order is equal to a post-order traversal of the task tree. The working set of each task corresponds to two submatrices of size $m \times n$ and $n \times m$. Let λ be a constant sufficiently small so that two submatrices, where $\max\{mB, nB\} \leq \lambda L$, fit completely in cache. If $B > Z/2$, such λ does not exist. In this case, the data block size is so large that the algorithm execution is equal to streaming the two matrices, which incurs $2P\lceil PB/L \rceil$ cache misses. If λ exists, there are three cases:

Case 1. $\max\{mB, nB\} \leq \lambda L$. Both matrices fit in the cache. The cache complexity is equal to the number of cache lines of the two matrices, namely $n\lceil mB/L \rceil + m\lceil nB/L \rceil$.

Case 2. $nB \leq \lambda L < mB$. Since m is the larger dimension, it is recursively divided by 2 by traversing the task tree. When mB falls into the range of $[\lambda L/2, \lambda L]$, the working set of the current task fits in cache. Then, we have the recursion

$$Q(m,n) \leq \begin{cases} n\lceil mB/L \rceil + m\lceil nB/L \rceil & \text{if } mB \in [\lambda L/2, \lambda L], \\ 2Q(m/2, n) & \text{otherwise;} \end{cases}$$

whose solution is $Q(m,n) = n\lceil mB/L \rceil + m\lceil nB/L \rceil$. Similarly, we have the same solution for $mB \leq \lambda L < nB$.

Case 3. $mB, nB > \lambda L$. The working set is recursively divided by 2 until both mB and nB fall into the range of $[\lambda L/2, \lambda L]$, and the working set of the current task occupies $n\lceil mB/L \rceil + m\lceil nB/L \rceil$ cache lines. By solving the recursion, we get $Q(m,n) = n\lceil mB/L \rceil + m\lceil nB/L \rceil$ for this case.

For MPI_Alltoall, we have m=n=P. Thus, the cache complexity of the sequential cache-oblivious algorithm based on Morton order for MPI_Alltoall is

$$\begin{aligned} Q_{alltoall-co-seq} &= 2P\lceil PB/L \rceil < 2(P^2B/L + P) \\ &= O(P^2B/L + P). \end{aligned} \quad (2)$$

Since the send-buffer and recv-buffer matrices occupy at least $2P\lceil PB/L \rceil$ cache lines, the sequential cache-oblivious algorithm is asymptotically optimal. However, low cache complexity for sequential algorithms does not mean the same for parallel algorithms [20]. We utilize Theorem 2.1 from Frigo and Strumpen [15] to analyze the cache complexity of the parallel cache-oblivious algorithm.

Theorem 2.1. *(Frigo and Strumpen [15]) Let T be a trace (the sequence of instructions in program order) of a parallel computation in P processes. T is partitioned into S segments and the segments are executed on an ideal distributed-cache machine. For any segment A of T, let f be a concave function such that $Q(A) \leq f(|A|)$ holds, where $|A|$ is the length of A. Then, the total number $Q_P(T)$ of cache misses incurred by the parallel execution of the trace is bounded by $Q_P(T) \leq Sf(|T|/S)$.*

To use Theorem 2.1, one should prove that the cache misses of all the segments A are bounded by $Q(A) \leq f(|A|)$. Let T be the trace of the post-order traversal of a k-ary task tree. One can easily find a nondecreasing function f such that $Q(A) \leq f(|A|)$ holds for any segment A corresponding to a complete subtree. For such f, it has been proved that $Q(A) \leq 2f(k|A|)$ holds for all A of T, not only for those corresponding to complete subtrees [15]. Then, we obtain the following corollary based on Theorem 2.1.

Corollary 2.2. *Let T be the trace of the post-order traversal of a k-ary task tree formed by k-recursive decomposition. If f is a nondecreasing concave function so that $Q(A) \leq f(|A|)$ holds for any segment A of T corresponding to a complete subtree, the cache complexity incurred by a parallel execution of T in P processes is $Q_P(T) = O(Sf(k|T|/S))$, where S is the number of segments.*

Recall that the cache complexity of the sequential cache-oblivious algorithm for MPI_Alltoall is $O(P^2B/L + P)$, and

1. If streaming prefetchers [12] are triggered on some processor, L is the total size of the multiple cache lines prefetched at a time.

P^2B is the trace length. Let $|A|$=P^2B. Then, we have a non-decreasing concave function $f(|A|) \in O(|A|/L + \sqrt{|A|/B})$, and $Q(A) \leq f(|A|)$ holds for any A corresponding to a complete subtree, which can be proved by induction on the complete subtrees. For the parallel cache-oblivious algorithm, the trace T is equally partitioned into P segments for P processes. Using Corollary 2.2, we have $Q_P(T) = O(Pf(2|T|/P))$, where $|T| = P^2B$. Thus, the cache complexity of the parallel cache-oblivious algorithm for MPI_Alltoall is

$$Q_{alltoall-co-par} = O(P^2B/L + P^{\frac{3}{2}}). \quad (3)$$

In practice, P is large enough so that $P > L/B$ is commonly satisfied. Then both Equation (2) and Equation (3) can be simplified to $O(P^2B/L)$, from which we find that cache complexities of the parallel and the sequential cache-oblivious algorithms for MPI_Alltoall are asymptotically equal, and also asymptotically optimal. Comparing Equation (1) with Equation (3), we find that the smaller the value of B/L (i.e., the smaller the value of the block size B), the larger advantages the parallel cache-oblivious algorithm has over the naive algorithm; when B is so large that Equation (1) can be simplified to $O(P^2B/L)$, the two algorithms perform equally.

Equation (3) gives an upper bound of the cache complexity for the parallel cache-oblivious algorithm in general cases. To obtain the exact number of cache misses is difficult. However, both the sequential and parallel cache-oblivious algorithms incur $2P^2B/L$ cache misses if the following two conditions are satisfied: (1) The workload of each process is a complete task subtree; (2) PB is an integral multiple of L. This can be proved using Theorem 2.1.

2.3.2 *Analysis for MPI_Allgather*

For MPI_Allgather, each dimension of the recv-buffer matrix is P. The length of the send-buffer vector is P. Each element of the send-buffer vector represents a data block to be sent. Since each process allocates its own send buffer, it is not consecutive between two adjacent elements in the send-buffer vector. For the naive implementation, each process copies the send-buffer vector into a row of the recv-buffer matrix (its own receive buffer). Thus, the cache complexity of the naive implementation of MPI_Allgather is

$$\begin{aligned} Q_{allgather-naive} &= P\lceil PB/L \rceil + P^2\lceil B/L \rceil \\ &= O(P^2B/L + P^2). \end{aligned} \quad (4)$$

To analyze the cache complexity of the parallel algorithm based on Morton order for MPI_Allgather, there are 3 cases:

$Case$ 1. $B > Z/2$. In this case, the size of each data block is too large that two data blocks would exceed the cache capacity, no temporal locality can be exploited for the send-buffer vector. Thus, the sequential algorithm based on Morton order for MPI_Allgather incurs the same cache complexity (shown in Equation (4)) as the naive implementation.

Using Corollary 2.2, the cache complexity of the parallel cache-oblivious algorithm for MPI_Allgather is

$$Q_{allgather-co-par} = O(P^2B/L + P^2). \quad (5)$$

In this case, B/L is much larger than one, and both Equation (4) and Equation (5) can be simplified to $O(P^2B/L)$. Thus, for very large block size, the parallel cache-oblivious algorithm and the naive algorithm for MPI_Allgather perform equally.

$Case$ 2. $B \leq Z/2$ and $P \geq \sqrt{Z/B}$. In this case, the total size of send-buffer vector and recv-buffer matrix exceeds the cache capacity. The temporal locality of the send-buffer vector can be exploited using Morton order. Using a similar analysis to the one we used for MPI_Alltoall, we obtain the cache complexity of the sequential algorithm based on Morton order for MPI_Allgather as

$$\begin{aligned} Q_{allgather-co-seq} &= P^2\lceil B/L \rceil / \sqrt{Z/B} + P\lceil PB/L \rceil \\ &= O(P^2B/L + P^2/\sqrt{Z/B}). \end{aligned} \quad (6)$$

Using Corollary 2.2, the cache complexity of the parallel cache-oblivious algorithm for MPI_Allgather is

$$Q_{allgather-co-par} = O(P^2B/L + P^2/\sqrt{Z/B}). \quad (7)$$

Comparing Equation (4) with Equation (7), we find that the larger the value of $\sqrt{Z/B}$ (i.e., the smaller the value of B, for Z is a constant), the larger the advantages the parallel cache-oblivious algorithm has over the naive implementation.

$Case$ 3. $B \leq Z/2$ and $P < \sqrt{Z/B}$. In this case, the size of send-buffer vector and the recv-buffer matrix is small enough to fit in the cache. Thus, we have

$$\begin{aligned} Q_{allgather-co-seq} &= P\lceil B/L \rceil + P\lceil PB/L \rceil \\ &= O(P^2B/L + P). \end{aligned} \quad (8)$$

Using Corollary 2.2, the cache complexity of the parallel cache-oblivious algorithm for MPI_Allgather is

$$Q_{allgather-co-par} = O(P^2B/L + P^{\frac{3}{2}}). \quad (9)$$

In practice, Equation (8) and Equation (9) can be simplified to $O(P^2B/L)$. Comparing Equation (4) with Equation (9), we find that the smaller the value of B/L (i.e., the smaller the value of the block size B), the larger the advantages of the parallel cache-oblivious algorithm.

The send-buffer vector and recv-buffer matrix occupy at least $P\lceil B/L \rceil + P\lceil PB/L \rceil$, i.e., $O(P^2B/L + P)$, cache lines, which is asymptotically lower than the cache complexity of the parallel cache-oblivious algorithm in $Case$ 2. We do not know if there are other algorithms for MPI_Allgather which incur a lower cache complexity. For the cache-aware algorithm, the recv-buffer matrix is tiled to exploit the data locality. In $Cases$ 1 and 3, it is straightforward to observe that the cache-aware algorithm incurs the same cache complexity as the cache-oblivious algorithm. In $Case$ 2, the total size of the recv-buffer submatrix and the send-buffer subvector in the cache-aware algorithm is tuned to fit in the cache. Thus, the size of each dimension of the recv-buffer submatrix is $\sqrt{Z/B}$. In this case, the cache-aware algorithm also incurs the same cache complexity (shown in Equation (6)) as the cache-oblivious algorithm.

2.4 Improved Algorithms for NUMA Architectures

In NUMA architectures, a processor accesses local memory faster than remote memory. However, our previously proposed algorithms assume all the processes are equal and

ignore the NUMA features. Guided by the Z-shaped curve, a process may copy a remote data block to another remote data block, which leads to a performance penalty. Thus, we propose improved algorithms for NUMA architectures.

We discuss the algorithm for MPI_Alltoall first. Assume that a NUMA system has s processors and each processor has q cores. Each process allocates its send and receive buffers in its local memory. The distance between a process and a local data block is d_l, while the distance between a process and a remote data block is d_r, where $d_r > d_l$. To fill a receive buffer, it needs to read at least $q(s-1)$ remote data blocks, plus reading q local blocks and writing qs local data blocks, which leads to the total distance of $q(s-1)d_r + q(s+1)d_l$ (the minimal distance).

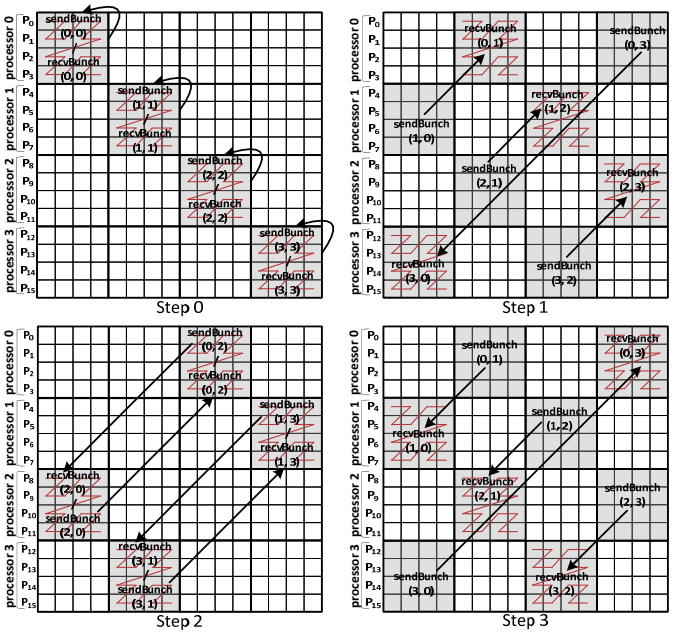


Fig. 5. NUMA-aware MPI_Alltoall combined with Morton order on a NUMA system with 4 processors. Each processor has 4 cores. A total of 16 processes are ranked sequentially in the 4 processors.

To achieve the minimal distance of data transfers, we propose a NUMA-aware algorithm combined with Morton order for MPI_Alltoall, which guarantees that each process only writes into its local receive buffers. For processor r, the local data blocks to be sent to processor x form a bunch that we call $sendBunch(r, x)$; the local data blocks to be received from processor y form a bunch that we call $recvBunch(r, y)$. Here $r, x, y \in [0, s\text{-}1]$. The algorithm needs s steps. In step i ($i \in [0, s\text{-}1]$), processor r copies $sendBunch((r+i)\%s, r)$ into $recvBunch(r, (r+i)\%s)$; and the q processes within processor r copy the data blocks in parallel following Morton order, as discussed in Section 2.1. An instance of the algorithm for MPI_Alltoall is illustrated in Figure 5. The NUMA-aware algorithm for MPI_Allgather is similar. Using the analysis in Section 2.2, one can show that the NUMA-aware algorithms are cache-oblivious within each processor.

2.5 Algorithms for Multi-Node Machines

The proposed cache-oblivious algorithms can be easily extended to multi-node machines. Taking MPI_Alltoall as an example, the optimized implementation on multi-core clusters [21], [22] typically has three phases: intra-node packing with local transpose, inter-node transpose by node leaders, and intra-node unpacking. Our cache-oblivious algorithms benefit the intra-node transposes as discussed above.

Next, we discuss how the cache-oblivious algorithm also benefits the inter-node transpose. For this, we model DRAM as a private cache for each node, and we model the whole cluster as a distributed cache. Different from the common hardware caches, we cache the adjacent data from remote node in DRAM manually, which is done by sending or receiving large blocks of consecutive data. First, we consider the case if P (the number of nodes) is a power-of-two. For simplicity, we reuse Figure 2 to illustrate how the algorithm for MPI_Alltoall works on an 8-node machine (i.e., P=8). The workload of each process is determined by the corresponding segment of Morton order. The algorithm includes three phases: (1) Each process receives the consecutive data blocks from the corresponding remote node in a single aggregated message. For example, $process_0$ receives the aggregated message $\{A_0, A_1\}$ from itself, $\{B_0, B_1\}$ from $process_1$, $\{C_0, C_1\}$ from $process_2$, and $\{D_0, D_1\}$ from $process_3$. (2) Each process does a local transpose on the received blocks following the Morton order. (3) Each process sends the consecutive transposed data blocks to the corresponding remote node in a single aggregated message. For example, $process_0$ sends the aggregated message $\{A_1, B_1, C_1, D_1\}$ to $process_1$, and sends $\{A_0, B_0, C_0, D_0\}$ to itself. In this way, the spatial locality of the inter-node transpose is exploited. If P=2^n and n is an even number, each process issues $\sqrt{P}$ independent communications in both $phase$ (1) and $phase$ (3). If P=2^n and n is an odd number, each process issues $\sqrt{2P}$ independent communications in $phase$ (1) and $\sqrt{P/2}$ independent communications in $phase$ (2).

We further compare our algorithm for MPI_Alltoall with several traditional algorithms in terms of communication rounds. Suppose P=2^n and n is an even number. The number of communications caused by our algorithm is $2\sqrt{P}$, which is asymptotically lower than the *Isends-Irecvs-Waitall* and *pairwise exchange* algorithms [10] (both algorithms cause P-1 communications). The number of communications caused by *Bruck*'s algorithm [23] is $\log_2 P$, which is lower than our algorithm. However, our algorithm can better utilize the parallelism within the interconnect network (each process can issue up to $\sqrt{P}$ messages simultaneously in both $phase$ (1) and $phase$ (3)). On the contrary, single-ported *Bruck*'s algorithm issues the messages sequentially. *Bruck*'s algorithm can also be implemented as multi-ported to exploit the parallelism of the interconnect network. However, the single-ported version is widely used in the latest MPI libraries, such as MPICH3, MVAPICH2, and Open MPI, to achieve low latency for small messages. In addition, our algorithm transfers less data than *Bruck*'s algorithm: our algorithm transfers $2n$ bytes while *Bruck*'s algorithm transfers $\frac{n}{2}\log_2 P$ bytes, where n is the size of the total receive buffer. We demonstrate that our algorithm has a performance advantage over the traditional algorithms for small messages in Section 4.1.3.

If P is not a power-of-two, we expect the future hardware or software cache for distributed memory to cache

the remote data automatically, instead of caching the remote data manually. The reason is that non-power-of-two nodes lead to irregular sizes of aggregated messages, which makes the manual cache become complicated and have additional overhead.

3 Neighborhood Collectives Based on Morton Order

For the MPI neighborhood collectives [1], a process only communicates with its neighbors in a pre-defined process topology. For example, MPI_Neighbor_alltoall sends a distinct data block to every neighbor process. For a naive implementation based on shared heap, each process directly copies the data blocks from its neighbors to its receive buffer. In the following, we discuss how to use Morton order to exploit data locality for these sparse communication patterns. We will use a 2D Cartesian topology for 9 processes, shown in Figure 6, as an example to elaborate how our approach works, although the same approach works for any process topology, such as a 3D Cartesian and a general graph topology [24].

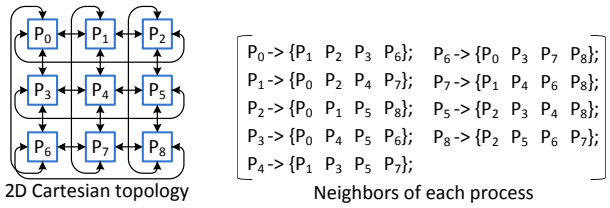
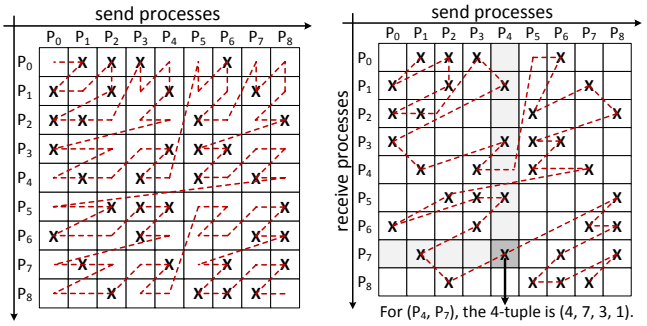


Fig. 6. A 2D Cartesian topology for 9 processes.

3.1 Morton Order for Neighborhood Collectives

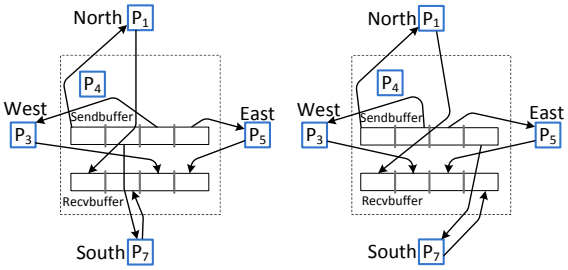
For any process topology, the Cartesian product of all the possible send processes and receive processes produces a 2D coordinate system. Figure 7 presents a 2D coordinate system generated from the 2D Cartesian topology for 9 processes. If a communication happens between a send process and a receive process, we mark the corresponding coordinates with 'X'. We use Morton order to sort all the coordinates, which forms a Z-shaped curve. By directly connecting the adjacent coordinates marked by 'X' in the Z-shaped curve, we get a compact curve shown in Figure 7(b).

We create a 4-tuple for each pair of coordinates in the compact curve, and store the 4-tuples in an array in the same order as that in the compact curve. Each 4-tuple is expressed as (R_s, R_r, B_s, B_r), where R_s denotes the rank of send process, R_r denotes the rank of receive process, B_s denotes the B_s-th block in the send buffer, and B_r denotes the B_r-th block in the receive buffer. R_s and R_r can be derived directly from the coordinates, while B_s is equal to the number of 'X's above the coordinates in the same column and B_r is equal to the number of 'X's on the left of the coordinates in the same row. The 4-tuple array is equally partitioned and each process handles one segment. For MPI_Neighbor_alltoall, the process copies the B_s-th block in the send buffer of process R_s into the B_r-th block in the receive buffer of process R_r. For MPI_Neighbor_allgather, each process sends the same block to its neighbors. Thus,



(a) Z-shaped curve for neighborhood collectives (b) A compact curve for the coordinates with communication

Fig. 7. Morton order for neighborhood collectives in a 2D Cartesian topology, where 'X' denotes a communication between a pair of neighbor processes.



(a) Traditional neighborhood alltoall (b) Neighborhood alltoall with ordered buffers

Fig. 8. Comparison of data-block layouts between the traditional and ordered-buffer MPI_Neighbor_alltoall in a 2D Cartesian topology.

each process copies the data block in the send buffer of process R_s into the B_r-th block in the receive buffer of process R_r.

The irregular neighborhood collectives, including MPI_Neighbor_alltoallv and MPI_Neighbor_allgatherv, can also utilize the 4-tuple array to exploit data locality. These two irregular operations allow one to receive data blocks with different sizes from its neighbors. The data block size and its displacements in the send and receive buffers can be obtained by accessing the block-size and displacement arrays [1] using the 4-tuple. For MPI_Neighbor_allgatherv, the data block size and its displacement in the receive buffer are determined by the B_r-th elements of the block-size and displacement arrays, respectively. For NUMA architectures, we use a similar NUMA-aware algorithm as discussed in Section 2.4 to minimize the data transfer distance.

3.2 Ordered Buffers for Neighborhood Collectives

The algorithm presented in Section 3.1 requires that the data blocks to be sent are placed in the send buffer in the numerical order of the ranks of the destination processes; and the data blocks received are placed in the receive buffer in the numerical order of the ranks of the source processes. However, this is not always true. For the MPI_Neighbor_alltoall in a 2D Cartesian topology shown in Figure 8(a), the four data blocks to be sent by process P_4 are placed in the send buffer in the order of orientations of the four destination processes, i.e., the order of $\{North\ (P_1), South\ (P_7), West\ (P_3), East\ (P_5)\}$. However, we should

change it to $\{P_1, P_3, P_5, P_7\}$ to obtain the ordered send buffer as shown in Figure 8(b). The requirement of ordered buffers may increase the programming complexity, since the programmer needs to build the topology as a general graph [1]. Fortunately, for many scientific applications [25], [26], [27], [28], [29], these structures are only created in the initialization phase and reused for many iterations.

4 Evaluation

Experiments were conducted on three different machines, including an Intel Xeon Phi KNC 5110P, an Intel Xeon E7-8890 v3, and a multiple-node Intel Xeon E5-2680 v3 cluster. The cache line size of all three architectures is 64 bytes. Xeon Phi has 60 cores, and each core has a 32 KB L1 data cache and a 512 KB unified L2 cache. The tag directories for cache coherence [11] and memory controllers are connected by a bidirectional ring, which forms a UMA architecture. Xeon E7-8890 contains 4 processors connected by QPI, and each processor has 18 cores sharing a 45 MB unified L3 cache, which forms a NUMA architecture. Each core of Xeon E7-8890 has a 32 KB L1 data cache and a 256 KB unified L2 cache. We run 72 processes on Xeon E7-8890 and 60 processes on Xeon Phi to utilize all the cores. The multi-node cluster consists of 256 nodes, which are connected by Infiniband. Each node of the cluster has two Xeon E5-2680 v3 processors and each processor has 12 cores.

We compared our cache-oblivious collectives with several state-of-the-art MPI libraries, including MPICH 3.1.4, Intel MPI 5.0, MVAPICH2 2.1, Open MPI 1.10, and MVAPICH2-MIC 2.0. All these libraries provide specific channel for efficient shared-memory communication, such as Shared-Memory-CH3 in MVAPICH2, sm BTL in Open MPI, and Nemesis in MPICH. We also configure MVAPICH2 to use Limic2 [4] for one-copy shared-memory communication. Recall that our algorithms require that the send and receive buffers be allocated in the shared heap whereas traditional MPI libraries work with any buffers. For brevity, we use *SH-Naive* to denote the naive implementations (implemented in [19]) based on shared heap, where each process sequentially copies the data blocks into its receive buffer as discussed in Section 2. To have a fair performance comparison on NUMA architectures, *SH-Naive* is also optimized as NUMA-aware, where each process staggers the data block copies to avoid the inter-socket congestions. We use *SH-CO* to denote the cache-oblivious algorithms based on shared heap which ignore NUMA features, *SH-NUMA-CO* to denote the NUMA-aware algorithms which keep cache-oblivious within each processor, and *MN-CO* to denote the cache-oblivious algorithms on multi-node machines. We define the speedup S as $S = \frac{T_{\text{ref}}}{T}$. This means an optimized operation which runs in 50% of the latency (time) of the reference operation is said to have a speedup of 2 (denoted as 2X). When mentioning *average speedup*, we mean the *geometric mean* of the speedups across different problem sizes.

4.1 Benchmark Evaluation

We use benchmarks to test the latency of the collectives. Each collective is run for 256 times and we present the average latency in the following figures. The difference between the average value and the latency measured in each iteration is within 5%. On shared memory machines, we design our own micro-benchmarks, which makes sure that the send and receive buffers of each process are only in its local cache before each time of running. In this way, we prevent that the whole send-buffer and receive-buffer matrices are cached locally for very small block sizes. Figure 9 shows the pseudo code of the benchmarks we use on shared memory machines. On multi-node machines, we use the OSU micro-benchmarks [30] to test the latency. The OSU micro-benchmarks are the same as our micro-benchmarks used on shared memory machines except that lines 3-7 in Figure 9 are removed in the OSU micro-benchmarks.

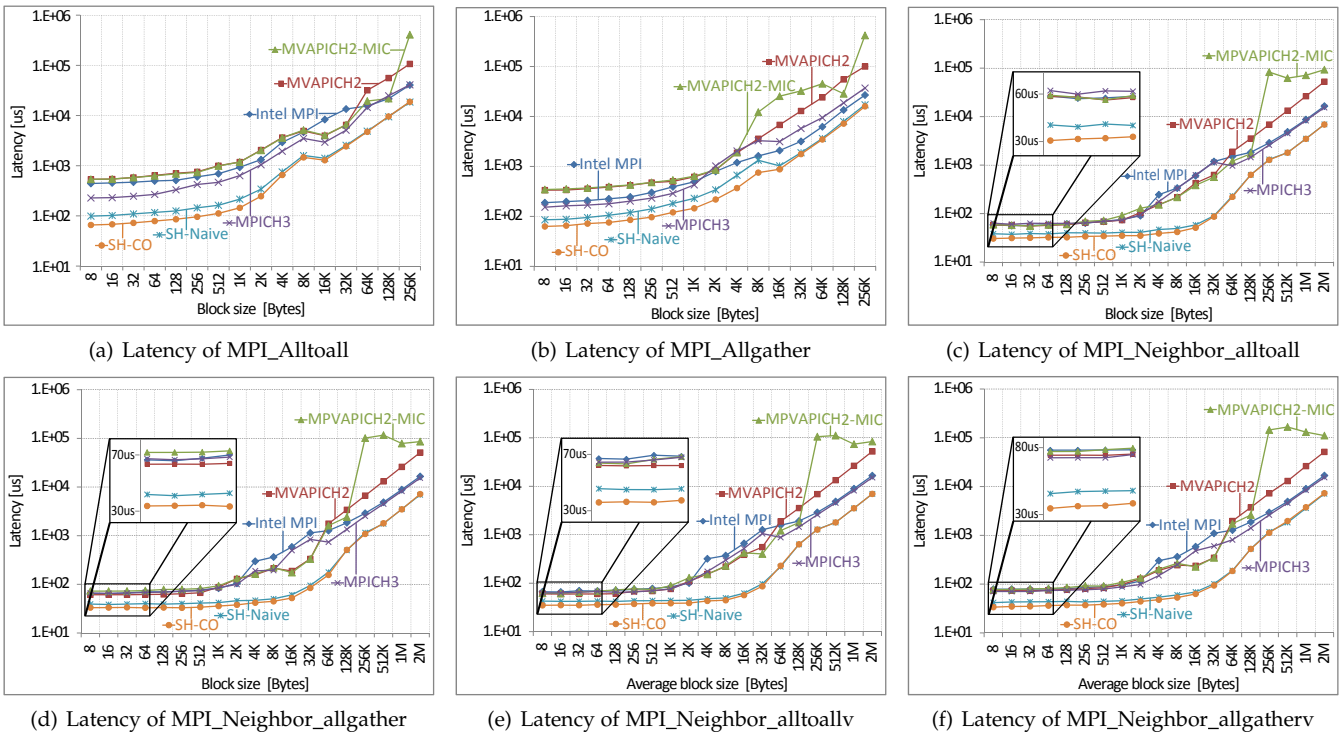
```
1  double begin, totalTime, avgTime=0.0;  int i;  long long missNum=0;
2  for(i=0; i<ITER; i++) {
3     Invalidate all the cache lines;
4
5     /*** bring buffers into local cache or memory ***/
6     Access the send buffer sequentially;
7     Access the receive buffer sequentially;
8     MPI_Barrier();
9
10    /** test the latency or the cache misses of the operation **/
11    Begin = MPI_Wtime();  or  PAPI_start();
12    Call the MPI all-to-all operation;
13    totalTime += MPI_Wtime() - begin;  or  missNum += PAPI_stop();
14 }
15 avgTime = totalTime/ITER;  or  missNum =  missNum/ITER;
```

Fig. 9. Pseudo code of the benchmark on shared memory machines.

4.1.1 Results on Xeon Phi

Figure 10(a) shows that *SH-Naive* for MPI_Alltoall outperforms all the traditional MPI libraries, including Intel MPI, MVAPICH2, MVAPICH2-MIC, and MPICH3. This is because the shared heap incurs less memory copies than the traditional MPI [7]. Compared with MPICH3 which performs best among the traditional MPI libraries, *SH-CO* for MPI_Alltoall achieves on average 3.11X speedup for all the block sizes. Compared with *SH-Naive*, *SH-CO* for MPI_Alltoall achieves on average 1.40X speedup when the block size is less than 16 KB (small and medium block sizes), and performs equally when the block size gets larger. This is consistent with the cache complexity analysis for MPI_Alltoall in Section 2.3, which shows that *SH-CO* has advantages over *SH-Naive* for smaller value of B/L, where B is block size. Here, the cache line length is 64 bytes. However, modern processors provide streaming prefetchers [12], which prefetch multiple consecutive cache lines at a time for consecutive references. In this case, the value of L is considered as the size of multiple cache lines prefetched, which is larger than 64 bytes. This is the reason why *SH-CO* still outperforms *SH-Naive* for medium block size.

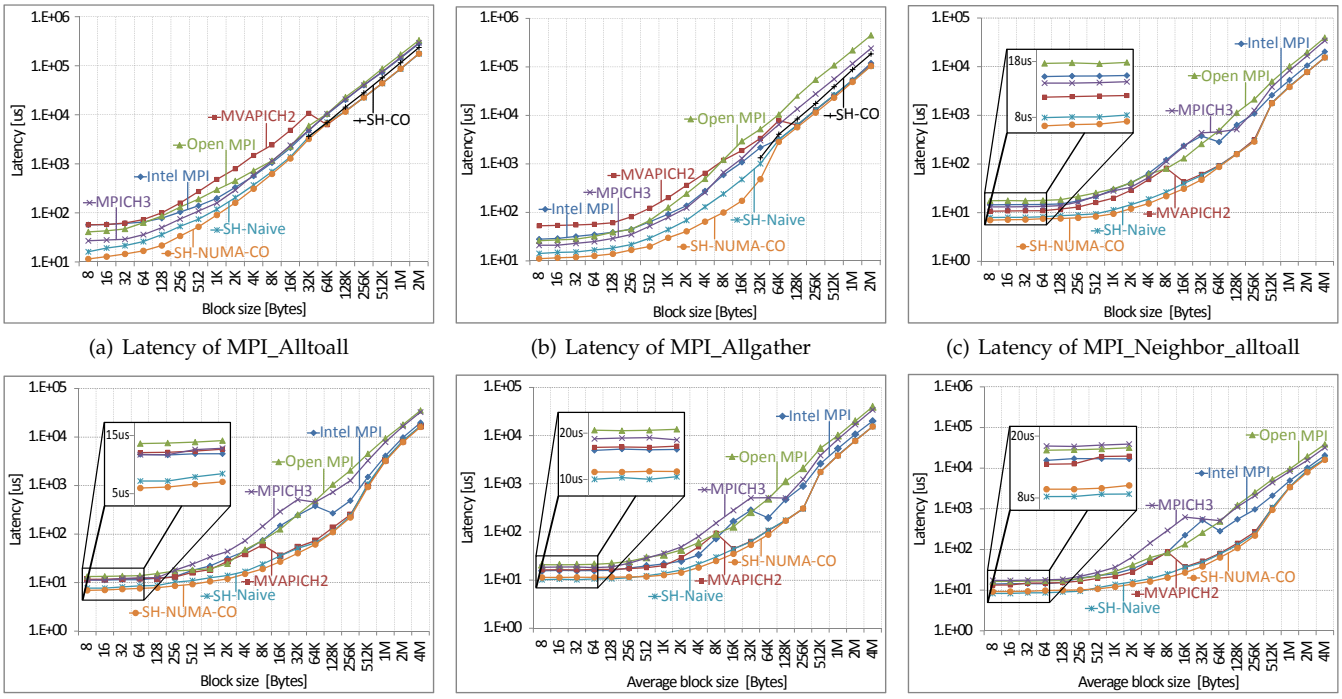
Figure 10(b) shows that *SH-Naive* for MPI_Allgather outperforms all the traditional MPI libraries due to the shared heap. For all the block sizes, *SH-CO* for MPI_Allgather achieves on average 2.90X speedup and 2.57X speedup over MPICH3 and Intel MPI, respectively. Compared with *SH-Naive*, *SH-CO* achieves on average 1.49X speedup when the block size is less than 32 KB, and achieves on average 1.09X speedup for block sizes between 32 KB and 128 KB.



(a) Latency of MPI_Alltoall (b) Latency of MPI_Allgather (c) Latency of MPI_Neighbor_alltoall

(d) Latency of MPI_Neighbor_allgather (e) Latency of MPI_Neighbor_alltoallv (f) Latency of MPI_Neighbor_allgatherv

Fig. 10. Latency of all-to-all style collective operations on Intel Xeon Phi KNC.



(a) Latency of MPI_Alltoall (b) Latency of MPI_Allgather (c) Latency of MPI_Neighbor_alltoall

(d) Latency of MPI_Neighbor_allgather (e) Latency of MPI_Neighbor_alltoallv (f) Latency of MPI_Neighbor_allgatherv

Fig. 11. Latency of all-to-all style collective operations on Xeon E7-8890.

This is consistent with the cache complexity analysis for MPI_Allgather in Section 2.3, namely *SH-CO* has advantages over *SH-Naive* when the block size is less than half of the cache capacity (i.e., $B \leq Z/2$), and the advantages of *SH-CO* are larger when the block size B is smaller.

Figure 10(c) and Figure 10(d) show the performance for neighborhood collectives in a 2D Cartesian topology. We find that *SH-Naive* significantly outperforms all the traditional MPI libraries due to the shared heap. For all the block sizes, *SH-CO* for MPI_Neighbor_alltoall and MPI_Neighbor_allgather achieve on average 3.05X speedup and 2.91X speedup over MPICH3, respectively. When the block size is less than 64 KB, *SH-CO* for MPI_Neighbor_alltoall and MPI_Neighbor_allgather

achieve on average 1.18X speedup and 1.17X speedup over *SH-Naive*, respectively. This demonstrates that our cache-oblivious algorithms have advantages even for sparse communication patterns. Figure 10(e) and Figure 10(f) show similar results for irregular MPI neighborhood collectives.

4.1.2 Results on Xeon E7-8890

On Intel Xeon E7-8890, we configure MVAPICH2 to use Limic2 [4] for one-copy shared-memory communication. Although Open MPI and MPICH3 can also be configured with KNEM [5] to support one-copy shared-memory communication, we only use MVAPICH2 with Limic2 as an example for performance comparison. Figure 11(a) shows that *SH-Naive* for MPI_Alltoall outperforms all the traditional MPI libraries. One special case is MVAPICH2, which performs equally with *SH-Naive* for large block size. This is because MVAPICH2 switches to use Limic2 when the block size is larger than 64 KB. *SH-CO* performs worse than *SH-Naive*, *SH-NUMA-CO*, and MVAPICH2 for large block size, because *SH-CO* ignores the NUMA feature of Xeon E7-8890 and causes more cross-chip data transfers. *SH-NUMA-CO* performs equally with MVAPICH2 when Limic2 is triggered (i.e., for the block sizes from 64 KB to 2 MB), and significantly outperforms MVAPICH2 for the block sizes smaller than 64 KB. Overall, *SH-NUMA-CO* achieves on average 3.03X speedup over MVAPICH2 for all the block sizes. Compared with *SH-Naive*, *SH-NUMA-CO* achieves on average 1.41X speedup when the block size is smaller than 16 KB, and performs equally when the block size gets larger. This is consistent with the cache complexity analysis in Section 2.3. Since *SH-Naive* is also optimized as NUMA-aware, the advantage of *SH-NUMA-CO* comes from the lower cache complexity of the cache-oblivious algorithm.

Figure 11(b) shows that *SH-NUMA-CO* for MPI_Allgather achieves on average 1.62X speedup over *SH-Naive* when the block size is smaller than 64 KB, and achieves on average 1.12X speedup over *SH-Naive* for block sizes from 64 KB to 2 MB. These results are consistent with the cache complexity analysis for MPI_Allgather in Section 2.3. Compared with MVAPICH2, *SH-NUMA-CO* achieves on average 1.09X speedup when Limic2 is triggered in MVAPICH2 (i.e., for the block sizes from 128 KB to 2 MB), and achieves on average 6.05X speedup for the block sizes less than 128 KB. Because of ignoring the NUMA features, *SH-CO* performs worse than *SH-Naive*, *SH-NUMA-CO*, and MVAPICH2 for large block sizes.

To further explain these phenomena on Xeon E7-8890, we present the cache misses of different algorithms for MPI_Allgather and MPI_Alltoall in Table 1 and Table 2, respectively. First, we demonstrate that the predicted values discussed in Section 2.3 are consistent with the measured values. We measure the value of L using a simple benchmark, in which the data in one buffer is consecutively copied from one buffer into another buffer and the cache misses on each cache level during the memory copy are measured. We use N_{miss} to denote the number of cache misses on each cache level, and N_{total} to denote the number of cache lines that the two buffers occupy. Then, the value of L is estimated as $L = (N_{total}/N_{miss}) * cacheLineSize$. We find that the value of L is different on different cache levels, and also different for different buffer sizes. Here, we use an example, the number of L2 cache misses of MPI_Allgather when the block size is 4 KB, to show the consistency. When comparing the values, we use the number of cache misses predicted for the sequential cache-oblivious algorithm to estimate the number of cache misses incurred the parallel cache-oblivious algorithm. In the example, the L2 cache capacity Z=256 KB, the block size B=4 KB, and the number of processes P=72. We measured L=70.4 for the buffer size around 4 KB and $L^{'}$=80.6 for the buffer size around 16 KB on L2 cache. Since $B \leq Z/2$ and $P \geq \sqrt{Z/B}$, Equation (6) is used to predict the cache misses for the cache-oblivious algorithm. The number of cache misses incurred by the cache-oblivious algorithm for each process is $P\lceil B/L \rceil / \sqrt{Z/B} + \lceil PB/L^{'} \rceil$=4190 ($L^{'}$ is used here since consecutive data blocks in the receive-buffer matrix are accessed under Morton order, and the size of the consecutive data blocks is about 16 KB). Similarly, using Equation (4), the number of cache misses incurred by the naive algorithm for each process is $\lceil PB/L^{'} \rceil + P\lceil B/L \rceil$=7907. We can see that the predicted values are approximately consistent with the measured values (4888 and 8134 for *SH-NUMA-CO* and *SH-Naive*, respectively) shown in Table 1. We summarize that other predicted values are also approximately consistent with the measured values. The reasons for the error between the predicted and the measured values include: (1) The value of L is not very accurate since it varies for different buffer sizes; (2) The instructions and other data structures would take up part of the cache capacity.

Next, we discuss how the cache miss statistics in Table 1 explain the performance data of MPI_Allgather in Figure 11(b). For the block sizes less than 32 KB, *SH-CO* and *SH-CO-NUMA* incur less cache misses than *SH-Naive* in both the L1 and L2 cache, due to better locality for the private caches. This explains why *SH-CO-NUMA* outperforms *SH-Naive* for small block sizes. However, *SH-CO* incurs more L3 cache misses than *SH-NUMA-CO* and *SH-Naive* for all block sizes, because *SH-CO* is not NUMA-aware and causes more remote (cross-chip) accesses. This explains why *SH-CO* performs worse than *SH-Naive* and *SH-NUMA-CO*. The cache misses in Table 2 also explain the performance data of MPI_Alltoall in Figure 11(a). The cache miss statistics, together with the cache complexity analysis in Section 2.3, provide concrete evidence that the performance advantage of our cache-oblivious algorithms is due to better data locality.

Figure 11(c) and Figure 11(d) show the performance for neighborhood collectives in a 3D Cartesian topology. *SH-NUMA-CO* for MPI_Neighbor_alltoall and MPI_Neighbor_allgather achieve on average 1.17X speedup and 1.15X speedup over *SH-Naive* when the block size is less than 64 KB, respectively; and achieve 109.6 GB/s and 104.8 GB/s total bandwidth when the block size is 4 MB, respectively. Figure 11(e) and Figure 11(f) show that *SH-NUMA-CO* performs slightly worse than *SH-Naive* for small blocks for irregular neighborhood collectives, which is caused by the indirect data accesses to the message-size and displacement arrays.

4.1.3 Results on the Xeon E5-2680 cluster

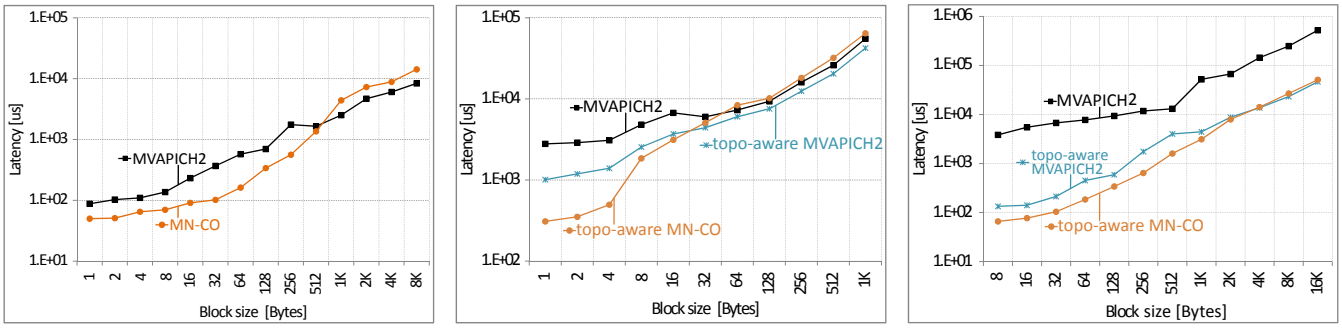
Figure 12(a) presents the latency of MPI_Alltoall on the 256-node Xeon E5-2680 cluster (using one core on each node).

TABLE 1
Cache misses for MPI_Allgather on Xeon E7-8890. Measured with PAPI 5.4.1 using the benchmark in Figure 9.

Block size (bytes)	L1 data cache misses			L2 data cache misses			L3 cache misses		
	SH-Naive	*SH-CO*	*SH-NUMA-CO*	*SH-Naive*	*SH-CO*	*SH-NUMA-CO*	*SH-Naive*	*SH-CO*	*SH-NUMA-CO*
8	292	149	204	250	86	141	12	14	12
64	385	239	301	323	133	208	15	33	17
512	1557	967	1077	925	543	840	43	163	42
4K	9644	6667	7254	8134	4260	4888	108	1078	111
32K	75052	75152	75010	56759	36232	37314	3317	5951	2931
2M	4736648	4735659	4736010	4684743	4708920	4692626	682473	1039694	705242

TABLE 2
Cache misses for MPI_Alltoall on Xeon E7-8890. Measured with PAPI 5.4.1 using the benchmark in Figure 9.

Block size (bytes)	L1 data cache misses			L2 data cache misses			L3 cache misses		
	SH-Naive	*SH-CO*	*SH-NUMA-CO*	*SH-Naive*	*SH-CO*	*SH-NUMA-CO*	*SH-Naive*	*SH-CO*	*SH-NUMA-CO*
8	296	189	228	274	111	148	18	31	18
64	437	313	341	354	210	259	64	70	59
512	1681	1279	1338	1109	846	886	446	482	349
4K	10508	9228	9302	8419	7076	7221	1565	1739	1256
32K	75209	75184	75351	60318	58944	57524	17620	19899	15693
2M	4724670	4728056	4699975	4644480	4625977	4450142	1391704	1452379	1399442



(a) MPI_Alltoall on 256-node Xeon E5-2680 v3, using one core on each node.
(b) MPI_Alltoall on 128-node Xeon E5-2680 v3, using all 24 cores on each node.
(c) MPI_Allgather on 256-node Xeon E5-2680 v3, using all 24 cores on each node.

Fig. 12. The latency of MPI_Alltoall and MPI_Allgather on multi-node machines.

For MVAPICH2, it uses *Bruck*'s algorithm [23] for block sizes less than 512 bytes, and uses the *Isend-Irecv* algorithm for the larger block sizes. Note that both MVAPICH2 and *MN-CO* use MPI point-to-point communications to implement the collective operations on multi-node machines. *MN-CO* achieves on average 2.23X speedup over MVAPICH2 for block sizes less than 1 KB, and performs worse than MVAPICH2 for larger block sizes. Compared with *Bruck*'s algorithm, our algorithm better utilizes the parallelism within the interconnect network and transfers less data, as discussed in Section 2.5. This is the reason why *MN-CO* outperforms MVAPICH2 for small messages. However, as the block size becomes larger, MVAPICH2 switches to the *pairwise exchange* algorithm. Although *MN-CO* causes less communications than the *pairwise exchange* algorithm, the amount of data to be transferred is two times as much as the *pairwise exchange* algorithm. This is the reason why *MN-CO* performs worse than MVAPICH2 for larger block sizes. At last, it demonstrates that the cache-oblivious algorithm has a significant advantage over *Bruck*'s algorithm for small messages by exploiting data locality in DRAM.

Figure 12(b) presents the latency of MPI_Alltoall on the 128-node Xeon E5-2680 cluster (using all 24 cores on each node). The label *topo-aware* in Figure 12(b) means the corresponding implementation is topology-aware, which consists of three phases: (1) intra-node packing with local transpose, (2) inter-node transpose by node leaders, and (3) intra-node unpacking. The difference between the different topology-aware implementations lies in the phase of the inter-node transpose by node leaders. For the block sizes less than 64 B, *topo-aware MN-CO* achieves on average 1.88X speedup over *topo-aware* MVAPICH2. This is because *topo-aware MN-CO* has a performance advantage in the phase of the inter-node transpose for small block sizes. *Topo-aware MN-CO* achieves on average 3.81X speedup over MVAPICH2 without topology-aware optimization for the block sizes less than 64 B. We also present the latency of MPI_Allgather on the 256-node Xeon E5-2680 cluster in Figure 12(c). For block sizes less than 4 KB, *topo-aware MN-CO* achieves on average 1.91X speedup over *topo-aware* MVAPICH2.

We also implement the two topology-aware MPI_Allgather algorithms proposed by Mamidala et al. [22] and Ma et al. [31], and compare them with our algorithm in Figure 13. For the former one, MPI_Allgather is implemented as inter-node *recursive-doubling* overlapped with intra-node unpacking, labeled as *recursive-doubling with overlap*. For the latter one, MPI_Allgather is implemented as inter-node *ring* algorithm overlapped with intra-node

unpacking, labeled as *ring with overlap*. For the block sizes less than 2 KB, *topo-aware MN-CO* achieves on average 1.88X and 1.82X speedups over *recursive-doubling with overlap* and *ring with overlap*, respectively, since our algorithm utilizes the parallelism of the interconnect network better. These results demonstrate that our cache-oblivious algorithms also benefit multi-core clusters.

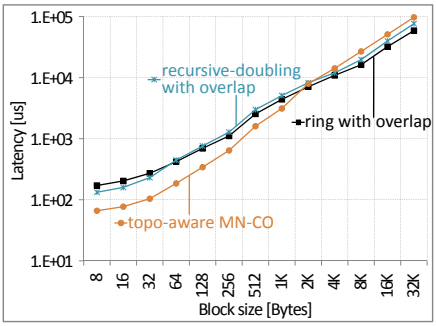
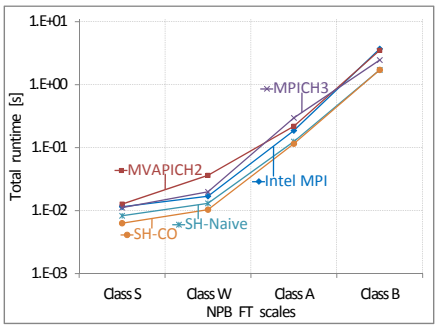


Fig. 13. MPI_Allgather with intra- and inter-node communications overlap on 256-node Xeon E5-2680 v3, using all 24 cores on each node.

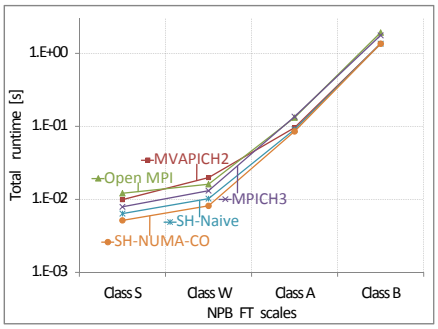
4.2 Application Evaluation

We test the total runtime of 3D FFT from NPB3.2 [32] using the classes S, W, A, and B, where MPI_Alltoall is intensively used for global transpose. The workload of each class is slightly changed since the number of processes (in 1D layout) on Xeon Phi and Xeon E7-8890 is not a power-of-two. The data block sizes of MPI_Alltoall for classes S, W, A, and B are 1 KB, 2 KB, 32 KB, and 128 KB, respectively. Results in Figure 14(a) and Figure 14(b) show that *SH-CO* has a performance improvement over *SH-Naive* at small scales (S and W). On Xeon Phi, *SH-CO* achieves on average 1.16X speedup over *SH-Naive* at all scales, and achieves on average 1.80X speedup over Intel MPI at all scales. On Xeon E7-8890, *SH-NUMA-CO* achieves on average 1.14X speedup over *SH-Naive* at all scales, and achieves on average 1.52X speedup over MVAPICH2 with Limic2 at all scales.

Heat transfer simulations on 2D grid and 3D grid of different sizes are carried out on Xeon Phi and Xeon E7-8890, respectively. Both simulations are run for 1,024 iterations. On Xeon Phi, we run 60 processes, which are arranged in a 2D Cartesian topology (6×10). As labeled in Figure 15(a), the size of the 2D grid is N*N. Thus, each process is responsible for a (N/6)*(N/10) rectangular region. The ghost data to be exchanged for each process are the four edges, with the lengths of N/10, N/10, N/6, and N/6, respectively. MPI_Neighbor_alltoallv dominates the communication time; *SH-CO* achieves on average 1.15X speedup and 1.85X speedup over *SH-Naive* and MVAPICH2 at all scales, respectively, as shown in Figure 15(a). On Xeon E7-8890, we run 72 processes, which are arranged in a 3D Cartesian topology (3×4×6). As labeled in Figure 15(b), the size of the 3D grid is 3N*4N*6N. Thus, each process is responsible for an N*N*N cubic region. The ghost data to be exchanged for each process are the six facets of the cubic region, with the area of N*N. MPI_Neighbor_alltoall dominates the communication time; *SH-NUMA-CO* achieves on average 1.27X speedup and 1.79X speedup over *SH-Naive*

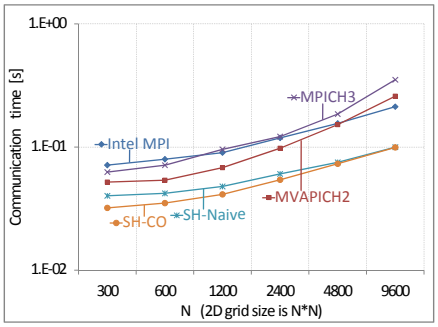


(a) 3D FFT on Xeon Phi

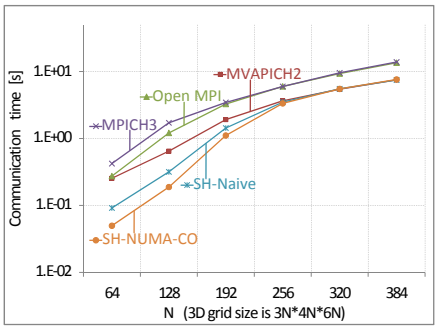


(b) 3D FFT on Xeon E7-8890

Fig. 14. 3D FFT on Xeon Phi and Xeon E7-8890.



(a) 5-point 2D stencil on Xeon Phi



(b) 7-point 3D stencil on Xeon E7-8890

Fig. 15. Heat transfer simulation on Xeon Phi and Xeon E7-8890.

and MVAPICH2 with Limic2 at all scales, respectively, as shown in Figure 15(b). Results on real applications verify the advantages of the cache-oblivious collective operations.

5 Related Work

To improve the efficiency of data movement, several techniques [4], [5], [6], [8], [19] have been developed for single-

copy shared-memory MPI communication. Although the underlying communication channel has been well tuned, both intra-node and inter-node collectives are commonly implemented using the traditional algorithms [10]. These algorithms are originally designed to minimize the latency or bandwidth overhead in a network, but do not fully exploit the data locality in the memory hierarchy.

Topology-aware implementations for MPI collective and I/O operations have been intensively studied. Mamidala et al. [22] proposed a shared memory and RDMA based design for MPI_Allgather. They used a common memory segment for both intra- and inter-node communications. Based on this, the number of inter-node messages is reduced, and also the intra- and inter-node communications can be overlapped. Ma et al. [31] proposed HierKNEM, which is a kernel-assisted topology-aware collective framework and enables overlap of intra- and inter-node communications. Karonis et al. [33] proposed an implementation of topology-aware collective operations, which exploited the hierarchy in a multi-layer network. To improve the data locality for collective I/O operation, Filgueira et al. [34] employed the linear assignment problem for finding the optimal distribution of data to processes.

To exploit the data locality, Frigo et al. [14] have presented cache-oblivious algorithms for matrix transpose, FFT, sorting, and matrix multiplication, where the problem is divided recursively and eventually reaches a subproblem size that fits into cache. These algorithms achieve asymptotically optimal cache complexity without tuning any parameter. However, these algorithms rely heavily on recursive function calls. Besides, one should use scheduling strategies, like work stealing [20], to parallelize these algorithms, which incurs scheduling overhead. Alternatively, this paper uses a Z-shaped curve to implement the parallel cache-oblivious algorithms without any scheduling overhead. Chatterjee and Sen [35] investigated the memory system performance of several memory-efficient algorithms under different memory models, including the cache-oblivious algorithm, for matrix transposition. Frigo et al. [15] analyzed the cache complexity of parallel cache-oblivious algorithms executed by the Cilk work-stealing scheduler [36], which inspired us to analyze the proposed cache-oblivious MPI collectives.

Space-filling curves, such as Morton order, Peano, and Hilbert, have been utilized to implement cache-oblivious algorithms. Bader et al. [16] proposed a cache-oblivious scheme combined with hand-tuned kernels for matrix multiplication and LU decomposition based on Peano curves. Frens et al. [37] presented a cache-oblivious algorithm for QR factorization based on Morton-ordered quadtree matrices. Yzelman and Bisseling [38] proposed a cache-oblivious sparse matrix-vector multiplication scheme using Hilbert curve. Martone et al. [39] presented a recursive sparse matrix storage format based on an improved Morton order for matrices with non-power-of-two dimensions. These works motivate us to implement cache-oblivious MPI collectives based on Morton order.

6 Conclusion

As supercomputers evolve into the exascale era, the number of cores keeps increasing while the amount of memory per core is decreasing. Data movement is increasingly expensive in terms of runtime and power consumption. Thus, it is critical for parallel programming languages and libraries to take advantage of memory hierarchies and provide communication operations with high cache efficiency. In this paper, we propose cache-oblivious algorithms for MPI, the most popular library for high-performance computing, to improve the performance of all-to-all style collectives.

For MPI_Alltoall and MPI_Allgather, we design cache-oblivious algorithms based on Morton order, and prove their optimality. We further optimize the cache-oblivious algorithms for NUMA architectures to minimize the distance of data transfer. We extend the cache-oblivious algorithms for multi-node machines, regarding DRAM as a private cache for each node. For neighborhood collectives, we propose an ordered-buffer approach combined with Morton order to exploit data locality.

Experimental results show that our cache-oblivious algorithms based on shared heap achieve portable performance improvement on both single- and multi-node machines. Our implementation for MPI_Alltoall achieves on average 3.11X speedup over MPICH3 on the UMA-architecture Xeon Phi; achieves on average 3.03X speedup over MVAPICH2 on the NUMA-architecture Xeon E7-8890; and achieves on average 2.23X speedup over MVAPICH2 on a 256-node Xeon E5-2680 for the block sizes less than 1KB. Architecture trends indicate that deep memory hierarchies will be necessary. We foresee that the benefit of our cache-oblivious algorithms will be more significant on such future machines. Our developed algorithms and cache complexity analysis approach form a basis for parallel communication algorithms design on future exascale systems.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant No. 61502450, Grant No. 61432018, and Grant No. 61521092; National Key R&D Program of China under Grant No. 2016YFB0200800, and Grant No. 2017YFB0202302. Shigang Li is the corresponding author.

References

[1] MPI Forum, "MPI: A Message-Passing Interface standard. Version 3.1," June 2015.

[2] S. Ramos and T. Hoefler, "Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi," in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*. ACM, 2013, pp. 97–108.

[3] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda, "Virtual machine aware communication libraries for high performance computing," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 2007, p. 9.

[4] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems," in *Proceedings of the 2007 IEEE International Conference on Cluster Computing*. IEEE, 2007, pp. 446–451.

[5] B. Goglin and S. Moreaud, "KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176–188, 2013.

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

[7] S. Li, T. Hoefler, and M. Snir, "NUMA-aware shared-memory collective communication for MPI," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, 2013, pp. 85–96.
[8] M. Woodacre, D. Robb, D. Roe, and K. Feind, "The SGI® AltixTM 3000 global shared-memory architecture," *Technical Whitepaper, Silicon Graphics, Inc.*, 2003.
[9] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine, "Hybrid MPI: efficient message passing for multi-core systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 18.
[10] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.
[11] G. Chrysos, "Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture," *Technical Whitepaper, Intel Corporation*, 2012.
[12] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual," January 2016.
[13] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. González-Domínguez, A. Gómez, and B. Wibecan, "Scalable PGAS collective operations in NUMA clusters," *Cluster computing*, vol. 17, no. 4, pp. 1473–1495, 2014.
[14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Transactions on Algorithms (TALG)*, vol. 8, no. 1, p. 4, 2012.
[15] M. Frigo and V. Strumpen, "The cache complexity of multithreaded cache oblivious algorithms," *Theory of Computing Systems*, vol. 45, no. 2, pp. 203–233, 2009.
[16] M. Bader, R. Franz, S. Günther, and A. Heinecke, "Hardware-oriented implementation of cache oblivious matrix operations based on space-filling curves," in *Parallel Processing and Applied Mathematics*. Springer, 2008, pp. 628–638.
[17] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Low depth cache-oblivious algorithms," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 189–199.
[18] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*. New York: International Business Machines Company, 1966.
[19] S. Li, T. Hoefler, C. Hu, and M. Snir, "Improved MPI collectives for MPI processes in shared address spaces," *Cluster Computing*, vol. 17, no. 4, pp. 1139–1155, 2014.
[20] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2000, pp. 1–12.
[21] Y. Qian and A. Afsahi, "Efficient shared memory and RDMA based collectives on multi-rail QsNetII SMP clusters," *Cluster Computing*, vol. 11, no. 4, pp. 341–354, 2008.
[22] A. R. Mamidala, A. Vishnu, and D. K. Panda, "Efficient shared memory and RDMA based design for MPI_Allgather over infiniband," in *Proceedings of the European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2006, pp. 66–75.
[23] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1143–1156, 1997.
[24] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Träff, "The scalable process topology interface of MPI 2.2," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, pp. 293–310, 2011.
[25] P. W. Jones, P. H. Worley, Y. Yoshida, J. White, and J. Levesque, "Practical performance portability in the Parallel Ocean Program (POP)," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 10, pp. 1317–1327, 2005.
[26] B. Wu, S. Li, Y. Zhang, and N. Nie, "Hybrid-optimization strategy for the communication of large-scale Kinetic Monte Carlo simulation," *Computer Physics Communications*, vol. 211, pp. 113–123, 2017.
[27] F. Gygi, "Large-scale first-principles molecular dynamics: moving from terascale to petascale computing," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 268, 2006.
[28] S. Plimpton, A. Thompson, and A. Slepoy, "SPPARKS Kinetic Monte Carlo simulator," http://spparks.sandia.gov/index.html, 2012.
[29] J. H. Ferziger and M. Peric, *Computational methods for fluid dynamics*. Springer Science & Business Media, 2012.
[30] OSU Network-Based Computing Laboratory, "OSU Micro-Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks, 2017.
[31] T. Ma, G. Bosilca, A. Bouteiller, and J. Dongarra, "HierKNEM: an adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters," in *Proceedings of the 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 970–982.
[32] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.
[33] N. T. Karonis, B. R. De Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, "Exploiting hierarchy in parallel computer networks to optimize collective operation performance," in *Proceedings of the 14th International Parallel and Distributed Processing Symposium*. IEEE, 2000, pp. 377–384.
[34] R. Filgueira, D. Singh, J. Pichel, F. Isaila, and J. Carretero, "Data locality aware strategy for two-phase collective I/O," *High Performance Computing for Computational Science-VECPAR 2008*, pp. 137–149, 2008.
[35] S. Chatterjee and S. Sen, "Cache-efficient matrix transposition," in *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*. IEEE, 2000, pp. 195–205.
[36] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.
[37] J. D. Frens and D. S. Wise, "QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism," *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 144–154, 2003.
[38] A.-J. N. Yzelman and R. H. Bisseling, "A cache-oblivious sparse matrix-vector multiplication scheme based on the Hilbert curve," in *Progress in Industrial Mathematics at ECMI 2010*. Springer, 2012, pp. 627–633.
[39] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, and M. Ganzha, "Utilizing recursive storage in sparse matrix-vector multiplication-preliminary considerations," in *Proceedings of the 25th International Conference on Computers and Their Applications*, 2010, pp. 300–305.

Shigang Li is currently an Assistant Professor of Computer Science at Institute of Computing Technologies, Chinese Academy of Sciences. He received the B.S. and Ph.D degrees in computer architecture from University of Science and Technology Beijing, China, in 2009 and 2014, respectively. He studied in University of Illinois at Urbana-Champaign as a visiting Ph.D student from Sept. 2011 to Sept. 2013. His research interests are in the areas of high-performance computing, focusing on message passing communications, large-scale parallel algorithms, heterogeneous computing, performance modeling, and high-performance deep learning platforms. His efforts are published in HPDC, IEEE TPDS, TACO, PPoPP, etc.

Yunquan Zhang is a Full Professor of computer science at Institute of Computing Technology, Chinese Academy of Sciences. His research interests are in the areas of high performance parallel computing, focusing on parallel programming models, high-performance numerical algorithms, and performance modeling and evaluation for parallel programs.

Torsten Hoefler is an Associate Professor of Computer Science at ETH Zurich. He is a member of the MPI Forum where he chairs the Collective Operations and Topologies group. His research interests revolve around the central topic of Performance-Centric Software Development and include scalable networks, parallel programming techniques, and performance modeling.