Proceedings Paper:
Peng, J., Fang, J., Liu, J. et al. (5 more authors) (Accepted: 2023) Optimizing MPI Collectives on Shared Memory Multi-cores. In: Proceedings of SC23: The International Conference for High Performance Computing, Networking, Storage, and Analysis. SC23: The International Conference for High Performance Computing, Networking, Storage, and Analysis, 12-17 Nov 2023, Denver, USA. ACM . (In Press)

Optimizing MPI Collectives on Shared Memory Multi-Cores

Jintao Peng[1], Jianbin Fang[1,*], Jie Liu[1], Min Xie[1], Yi Dai[1], Bo Yang[1], Shengguo Li[1], Zheng Wang[2]
1 College of Computer Science and Technology, National University of Defense Technology, China
2 School of Computing, University of Leeds, United Kingdom
jintaopengcs@gmail.com, {j.fang, liujie, xiemin, daiyi, yangbo78, nudtlsg}@nudt.edu.cn, z.wang5@leeds.ac.uk

ABSTRACT

Message Passing Interface (MPI) programs often experience performance slowdowns due to collective communication operations, like broadcasting and reductions. As modern CPUs integrate more processor cores, running multiple MPI processes on shared-memory machines to take advantage of hardware parallelism is becoming increasingly common. In this context, it is crucial to optimize MPI collective communications for shared-memory execution. However, existing MPI collective implementations on shared-memory systems have two primary drawbacks. The first is extensive redundant data movements when performing reduction collectives, and the second is the ineffective use of non-temporal instructions to optimize streamed data processing. To address these limitations, this paper proposes two optimization techniques that minimize data movements and enhance the use of non-temporal instructions. We evaluated our techniques by integrating them into the OpenMPI library and tested their performance using micro-benchmarks and real-world applications running on two multi-core clusters. Experimental results show that our approach significantly outperforms existing techniques, yielding a 1.2-6.4x performance improvement.

CCS CONCEPTS

• **Software and its engineering** ; • **Software notations and tools**; • **Software libraries and repositories**;

KEYWORDS

MPI, Collective Communication, Memory Access, Optimization

1 INTRODUCTION

Many parallel applications in high-performance computing (HPC) rely on collective operations provided by the Message Passing Interface (MPI) for distributed communications, such as broadcasting and reduction message exchanges. Studies in production HPC environments show that MPI collectives can account for over two-thirds of the MPI application communication time and are often responsible for the performance bottlenecks [18].

Various efforts have been made to optimize MPI collective communications. Prior works in this area include optimizing the send- and receive-based collectives across computing nodes [15, 16, 45, 50], exploiting operating system kernel-level optimizations like "zero-copying" for point-to-point and communication fusion optimizations [14, 17, 30], as well as using shared-memory optimization directly within a single node [39, 44, 60]. As modern processors increasingly integrate more processor cores with shared memory onto a single chip, it is common to run multiple MPI processes within a single computing node [21, 22]. Under such settings, optimising MPI collectives within a single node is increasingly important.

∗ Corresponding author.

This work aims to improve the optimization techniques for intra-node (shared memory) MPI collective operations. Our techniques are designed to overcome two prominent limitations observed in the existing solutions. Firstly, we observe a large amount of redundant data movements when performing reduction collectives - a key operation of MPI collectives. Even with shared memory, current implementations require copying data from the sending buffer to a shared memory buffer to perform the reduction [13, 34]. We empirically show that such redundant data movements can account for 40% of the total data accesses, but many of such data movements can be eliminated through a better MPI shared-memory reduction algorithm.

Secondly, we found that the widely used pipelined intra-node collective implementation can lead to substantial cache load overheads during memory writes, potentially causing wastage of precious memory bandwidth resources. Specifically, within a write-allocate cache framework [26, 37, 55, 59], a write instruction incurring a cache miss triggers a Request For Ownership (RFO) mechanism. This mechanism requires reading the cache line into the local cache, invalidating other cached copies, and writing the data to the local cache. However, this mechanism is ill-suited for performing write operations on a large sequence of data that exceed the cache capacity and are unlikely to be reused in the immediate future [33, 38, 48]. Under this scenario, the RFO overhead cannot be amortized by the performance improvement of future cache hits but can waste the memory bandwidth. Prior work addresses this problem by using non-temporal (NT) instructions in the memory copy operation to store the data directly in the memory [33, 38, 47]. Nonetheless, our findings indicate that this memory copying strategy does not align well with the data access patterns characteristic of pipelined collectives. Pipelined collectives typically divide substantial messages into smaller slices or chunks. However, the size of these slices might not be sufficient to fully leverage the benefits of non-temporal memory accesses, consequently resulting in suboptimal utilization of memory bandwidth resources.

Our work thus aims to address the two aforementioned drawbacks when performing MPI collectives on shared-memory multi-cores. Firstly, to reduce redundant data movements during the reduction process, we propose an enhanced reduction algorithm. This enhancement splits the message buffer and changes the message operating order. This step aims to decrease the overhead of the copy-in data movements. Secondly, we use a twofold strategy to minimize the RFO overhead for pipelined collectives. Our approach implements memory copy operations employing both traditional techniques and those incorporating non-temporal instructions. Moreover, we develop an analytical model for deciding when and where to use the two memory copy strategies.

We implemented and integrated our techniques as a collective library named YHCCL, which is integrated with the widely used

Open MPI library. We evaluate YHCCL on three multi-core clusters with 64, 48 and 24 cores per node. Experimental results show that YHCCL provides a significant speedup compared to the state-of-the-art collective implementations, with improvements ranging from 1.2× to 6.4× on large messages. When applying our optimization to two real-life workloads, Adaptive Mesh Refinement Mini-App (MiniAMR) and the training of distributed convolutional neural networks (CNNs), our approach achieves a speedup of 1.3×-1.7× on MiniAMR and improves the CNN training throughput by 1.8×-2.0×. Besides, YHCCL has been deployed onto production supercomputer systems to support a wide range of MPI workloads.

This paper makes the following contributions:

- It presents a novel intra-node reduction algorithm to eliminate the redundant data movements for MPI collectives.
- It proposes a fine-grained heuristic to better utilize the non-temporal instructions to optimize pipelined MPI collectives.
- It provides a set of analytical models to guide the optimization of intra-node MPI collectives.

2 BACKGROUND AND MOTIVATION

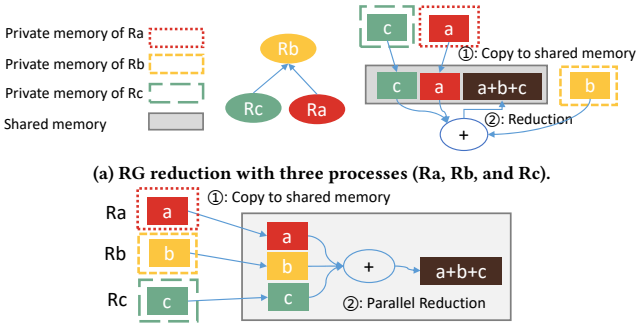
2.1 Background

On shared memory nodes, the performance of MPI collective is heavily impacted by memory access, as their arithmetic intensity is typically low [58]. For this reason, our work focuses on optimizing the memory access of intra-node MPI collectives at both the algorithm and implementation levels, with particular attention to reduction and pipelined collectives such as `reduce-scatter`, `reduce`, `all-reduce`, `broadcast`, and `all-gather`.

To measure the impact of memory access, we consider two important metrics: **data access volume** and **data access bandwidth**. Data access volume (DAV) represents the amount of data loaded and stored during a collective operation on a computing node, while data access bandwidth (DAB) indicates the data loaded and stored per unit of time.

Most operating systems implement **inter-process address isolation**, which isolates the memory space among processes. In this paper, we use the term **local memory** to refer to the memory buffer that is located in a process's own address space. Conversely, **remote memory** refers to memory buffers that are in other processes' address spaces. One approach to enable processes to share data within their own memory space is to use a **shared memory** mechanism provided by the operating system. With shared memory, multiple processes can communicate with each other by reading from or writing to shared memory locations. In **shared memory** collectives, there are terms **copy-in** and **copy-out** to describe the operations of moving data between a process' local and the shared memory. Specifically, **copy-in** refers to the transfer of data from local memory to shared memory, while **copy-out** describes the opposite process of moving data from shared memory back to a process's local memory. We use **V** to represent the data access volume of the copy operation, including both copy-ins and copy-outs.

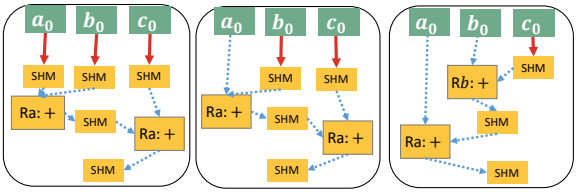
As an alternative to the shared memory approach, the kernel-assisted single-copy mechanisms (e.g., LiMiC [36], KNEM [27], Cross Memory Attach (CMA) [54], and XPMEM [56]) have been investigated to design MPI collective operations. These schemes use specialized kernel modules to map a process's address space into other processes. In this way, each process can directly access the address space of other processes. But the privileged installation, security issues, and registration overhead limit their deployments on production systems [19, 27, 34]. Thus, we focus on optimizing collectives based on the shared memory approach.



(a) RG reduction with three processes (Ra, Rb, and Rc).

(b) DPML reduction with three processes (Ra, Rb, and Rc).

Figure 1: Workflows of MPI reduction collectives.



(a) DPML design (b) RG design (c) A better design

Figure 2: Different process-reduction binding of the reduction of slices $\{a_0, b_0, c_0\}$ on shared memory. $Ra : +$ represents a reduction operation executed by rank a. The red arrow represents a data copy from local memory to shared memory, the blue dotted arrow represents a data load or store, and the *SHM* denotes the shared memory buffers.

2.2 Observations and Motivations

Our work is motivated by two observations, the first is the redundant data movements for the commonly used MPI reduction operation, and the second is the ineffective use of non-temporal instructions for memory optimizations. The two issues limit the memory access performance of current state-of-the-art collectives.

Redundant data movements. Reduction operations on shared memory are used in MPI collective implementations of mainstream MPI libraries, including MPICH and Open MPI [6, 8]. Before the reduction calculation, a *copy-in* procedure is performed to copy data from the sending buffer to shared memory. For the RG pipelined tree reduction algorithm [34] shown in Figure 1a, the children processes copy the entire sending buffers to the shared buffer before the reduction step. Because the reduction rank b cannot directly access the data in the sending buffers of ranks a and c, ranks a and c first have to copy data into shared memory. Similarly, for the DPML reduction algorithm [13] in Figure 1b, three processes reduce the data concurrently, each requiring the data from the other two processes. We find the redundant data copies are caused by the fact that the reduction processes use the data from the other process's sending buffers, and should be minimized whenever possible.

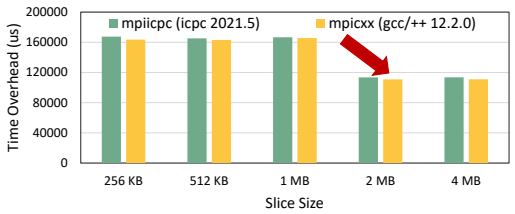


Figure 3: The copy-out overhead for reduction (NODEA, 64 cores).

Figure 2 shows the redundant data movements and optimization opportunities. For three processes shared memory reduction, we divide each sending buffer a, b, and c into three slices of size I and show the different reduction orders of the first slice $\{a_0, b_0, c_0\}$. For the reduction of other slices, the situation is the same. Figure 2a represents the DPML parallel reduction [13], where there are three red arrows for this design, with $V = 6 \cdot I$. Figure 2b represents the pipelined RG tree reduction, where there are two red arrows, with $V = 4 \cdot I$. Figure 2c shows a better design that ranks b and a collaborate to finish the reduction of the slice 0. In the first step, rank c copies the slice c_0 to shared memory, and then rank b performs $b_0 + c_0$. As b_0 is suited in the private memory of rank b, we have not to copy this slice. After that, rank a reads the operands in shared memory and takes a_0 for reduction. Thus, this design has only one red arrow, with $V = 2 \cdot I$. Our work aims to find a shared memory reduction algorithm for minimizing the *copy-in* overhead.

Ineffective use of non-temporal instructions. For pipelined collectives, a large message is chunked into slices and then moved from one process to another in a pipelined fashion [14, 25, 50]. On shared-memory multi-cores, the message passing between processes depends on the *data copy* primitive [25], which is typically implemented as *memmove* or *memcpy* in the C library [34, 51].

As we have mentioned, the collective operations based on shared memory have operations of *copy-in* and *copy-out*. For *copy-out* in all-reduce, we have to copy the reduction result from shared memory to receiving buffers [13, 34]. Figure 3 shows the copy-out overhead, where each rank copies the 256MB data from shared memory to its private buffer with different slice sizes. We see that, for the slice smaller than 2MB, the overhead is significantly larger than using a larger slice with both ICC and GCC compilers due to the *copy-out* with *memmove* does not use non-temporal memory access [38, 46].

For large messages, it is straightforward to use NT stores, but we note that it is not the case for the *copy-in* operation. After moving the data slice onto the shared memory (i.e., *copy-in*), the data will be used again in the near future. In this case, bypassing the cache with NT stores cannot utilize the cache bandwidth when loading the data again. Therefore, the prior pipelined collectives will suffer a loss in performance for either small or large data slices. This performance gap results from the ineffective use of non-temporal instructions, which motivates us to design a new adaptive strategy to use them.

2.3 YHCCL Overview

In this work, we redesign the intra-node collectives from the perspective of memory access at the algorithm and implementation level. Figure 4 shows the overall design of YHCCL. At the algorithm level, we minimize the copy overhead of the shared memory

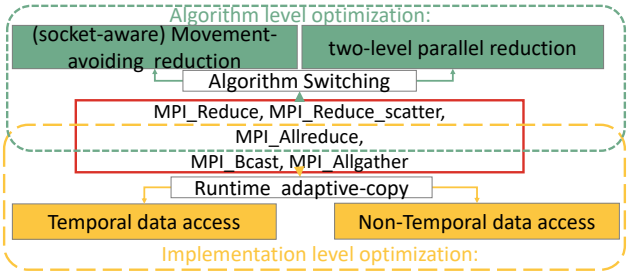


Figure 4: YHCCL overview.

reductions, eliminate the inter-NUMA memory, reduce the impact of synchronization overhead (Section 3), and switch between algorithms on different cases (Section 5.1). At the implementation level, we design an adaptive non-temporal hint implementation for pipelined collective to lower the impact of the RFO overhead (Section 4). Putting it all together, we integrate the two-level designs into our collective communication library (namely YHCCL) to improve the MPI collectives.

3 MOVEMENT-AVOIDING REDUCTION ALGORITHMS

This section addresses the issues of redundant data movements and expensive inter-socket memory accesses by formalizing the optimization problem and instantiating our reduction algorithms.

3.1 Sliced Reduction Problem

Assuming there are p processes, we formalize a sliced reduction problem based on shared memory where the sending buffer is chunked into p slices. We define $s_{i,j}$ to represent the jth slice within process i's sending buffer, where $i, j \in \{1, 2, ..., p\}$, and define a group of slices $G_i = \{s_{1,i}, s_{2,i}, ..., s_{p,i}\}$. Each data slice sizes $I = s/p$. The reduction result of the ith slice is the sum of slices in G_i. Since the reduction operation has two inputs and one output, any reduction combination of p slices is a binary tree. Thus, any reduction combination on G_i can be defined as a reduction tree $T_i = [T_{i,1}, T_{i,2}, ..., T_{i,p-1}]$. In T_i, there are $p-1$ nodes that represent a reduction sequence. Node $T_{i,j} = [r, a, b]$ is a reduction operation where $r \in \{1, 2, .., p\}$, $a, b \in \{1, 2, ..., j-1\} \cup G_i$, and $a \neq b$. The process r executes the reduction, a, b represent the operands, and the reduction result is stored on the shared memory. If $a \equiv s_{x,i} \in G_i$, it means one operand of the reduction is the ith slice in the sending buffer of process x. If $a \in \{1, 2, .., j-1\}$, it means one operand of the reduction is the result of the reduction $T_{i,a}$ in the shared memory.

Then, the copy data access volume required by $T_{i,j} = [r, a, b]$ can be calculated.

$$V(T_{i,j}) = \begin{cases} 0 & (a \notin G_i | a = s_{r,i}) \quad \& (b \notin G_i | b = s_{r,i}) \\ 2 \cdot I & (a \in G_i \& a \neq s_{r,i}) \quad \& (b \notin G_i | b = s_{r,i}) \\ 2 \cdot I & (a \notin G_i | a = s_{r,i}) \quad \& (b \in G_i \& b \neq s_{r,i}) \\ 2 \cdot I + 2 \cdot I & (a \in G_i \& a \neq s_{r,i}) \quad \& (b \in G_i \& b \neq s_{r,i}) \end{cases} \tag{1}$$

where condition $(a \notin G_i | a = s_{r,i})$ denotes that one operand of reduction $T_{i,j}$ comes from shared memory, which is the result of its previous reduction, or the operand is the slice i in the sending buffer of process r, and process r is responsible for executing $T_{i,j}$. In this case, the reduction process r can directly read the operand

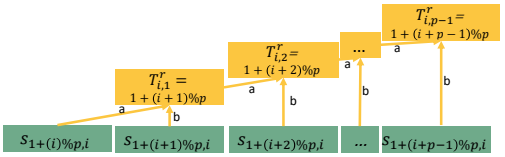


Figure 5: Reduction tree T_i of proposed algorithm $[T_1, T_2, ..., T_p]$.

without extra copies. $(a \in G_i \& a \neq s_{r,i})$ denotes that one operand of $T_{i,j}$ is the ith slice in the sending buffer that does not belong to the process r. Thus, the slice must be copied to shared memory before process r uses it. Each copy operation has one load and store operation and requires a DAV of $2 \cdot I$.

Next, we define any algorithm for sliced reduction based on shared memory as a sequence of p reduction trees $[T_1, T_2, ..., T_p]$, which meets the following constraints:

$$C = \begin{cases} j \in \{1, 2, ..., p-1\} \\ i, (T^r_{i,j}) \in \{1, 2, ..., p\} \\ \forall i \forall j (T^a_{i,j}), (T^b_{i,j}) \in \{1, 2, ..., j-1\} \cup G_i \\ \forall i (\cap_{j=1}^{p-1} (T^a_{i,j} \cup T^b_{i,j})) = \phi \end{cases} \tag{2}$$

where $T^r_{i,j}$, $T^a_{i,j}$, and $T^b_{i,j}$ are the elements of reduction node $T_{i,j} \equiv [r, a, b]$. The third constraint means that both operands of each reduction operation come from a previous reduction's result or a slice in sending buffer. The fourth constraint means that for each reduction tree, all reduction operands (number $2 \cdot (p-1)$) are different from each other. As the length of set $\left|\{1, 2, ..., (p-1)-1\} \cup G_i\right| = p-2+p = 2 \cdot (p-1)$, the fourth constraint ensures that there are $2 \cdot (p-1)$ non-roots in a binary tree. With the root node, there are a total of p leaves and $p-1$ non-leaves, where p is the number of slices in the sending buffer and $(p-1)$ is the number of reduction nodes. The constraints can express any reduction algorithm. For example, the DPML reduction [13] can be formalized as $[T^{\text{DPML}}_1, T^{\text{DPML}}_2, ..., T^{\text{DPML}}_p]$, where $T^{\text{DPML}}_i = \left[[i, s_{1,i}, s_{2,i}], [i, 1, s_{3,i}], ..., [i, p-2, s_{p,i}]\right]$.

The optimization problem is to find a solution X satisfying,

$$MIN \sum_{i=1}^{p} \sum_{j=1}^{p-1} V(T_{i,j}). \\ S.t.\ \forall X \equiv [T_1, T_2, ..., T_p] \rightarrow C(X) = True \tag{3}$$

The optimization goal is to minimize the total copy of all reduction nodes in all trees, which is important for large messages. Figure 2 has shown that the core idea of our design is to change the reduction order and let each reduction node use the operands from shared memory or *local memory* as much as possible. Then, we have to solve the optimization problem and reach an optimal solution.

3.2 Solving the Optimization Problem

This section proposes our algorithm $A' = [T_1, T_2, ..., T_{p-1}]$, reaching a reduction tree of T_i (Figure 5). The reduction has $p-1$ steps. For $T_{i,1}$, one operand comes from the *local memory* of process $T^r_{i,1}$, and the other operand is copied to shared memory before process $T^r_{i,1}$ reads it. For $T_{i,\{2,3..,p-1\}}$, their operands come either from shared memory or the *local memory* of the reduction process. For each tree in A', there is only one slice copy, and we can calculate the copy

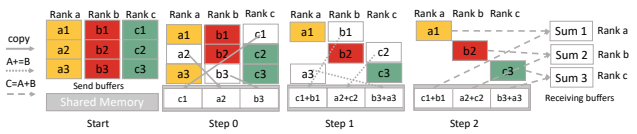


Figure 6: Movement-avoiding reduce-scatter among three processes (Ra, Rb, and Rc).

DAV of the algorithm ($V_{A'} = 2 \cdot I \cdot p = 2 \cdot s$). Next, we prove that $2 \cdot s$ is the minimal copy DAV of the sliced reduction problem.

Theorem 3.1. *For any reduction algorithm* $X \equiv [T_1, T_2, .., T_p]$ *satisfying the constraints* $C(T_{i,j}) =$ **True**, *there is*

$$\forall T_i \in X \rightarrow (\sum_{j=1}^{p-1} V(T_{i,j}) \geq 2 \cdot I)$$

.

Proof. From Equation 1, $V(T_{i,j}) \in \{0, 2 \cdot I, 4 \cdot I\}$. For any reduction tree T_i, there are $p-1$ reduction operations. Thus, the minimum value of $\sum_{j=1}^{p-1} V(T_{i,j})$ is 0. The second minimum value is $2 \cdot I$. If $\sum_{j=1}^{p-1} V(T_{i,j}) \neq 0$, then we have $\sum_{j=1}^{p-1} V(T_{i,j}) \geq 2 \cdot I$.

We assume that there exists an algorithm $\hat{X} \equiv [\hat{T}_1, \hat{T}_2, .., \hat{T}_p]$ that satisfies the constraints and makes $\exists \hat{T}_i \in \hat{X} \rightarrow (\sum_{j=1}^{p-1} V(\hat{T}_{i,j}) = 0)$ hold. For all reductions in $\hat{T}_i$, their operands come from shared memory or the *local memory* of the reduction process. But in the first reduction $\hat{T}_{i,1}$, the shared memory has no data. Thus, both operands come from the *local memory* of process $\hat{T}^r_{i,1}$ to meet $V(\hat{T}_{i,1}) = 0$, which means that $\hat{T}^a_{i,1} = \hat{T}^b_{i,1}$. This violates the fourth constraint of the sliced reduction algorithm. And there is no sliced reduction algorithm $\hat{X}$ that makes $\exists \hat{T}_i \in \hat{X} \rightarrow (\sum_{j=1}^{p-1} V(\hat{T}_{i,j}) = 0)$ hold. Therefore, for all sliced reduction algorithms X, we have $\forall T_i \in X \rightarrow (\sum_{j=1}^{p-1} V(T_{i,j}) \geq 2 \cdot I)$. The theorem is proved. □

According to the theorem, any reduction tree T_i has $\sum_{j=1}^{p-1} V(T_{i,j}) \geq 2 \cdot I$. Thus any algorithm A with p trees has larger copy DAV than $p \cdot 2 \cdot I = 2 \cdot s$, and thus the proposed A' is one of the optimal algorithms for solving the sliced reduction problem. Note that the MPI Ring reduction based on send/recv is proved to be bandwidth-optimal reduction algorithm [45]. However, in a shared memory node, its copy DAV is not minimal. This is because each MPI_send/_recv in each step still involves a least one data copy. Rabenseifner reduction are also based on send/recv operations. Thus, there is $V_{\text{Ring}} \geq 2 \cdot (p-1) \cdot s$. While in algorithm A', only the first step has data copy, other steps perform only computation. Next, we apply this to redesign the MPI reduction algorithms, including `MPI_Reduce_scatter` (Section 3.3), `MPI_Allreduce` (Section 3.4), and `MPI_Reduce` (Section 3.5).

3.3 Movement-avoiding MPI Reduce-scatter

Based on algorithm A', we instantiate the movement-avoiding (MA) reduce-scatter algorithm in Figure 6. The sending buffer is chunked into p slices. There is a global shared memory sizing s (i.e., the entire message), and I is the size of each slice ($I = \frac{s}{p}$).

Our reduction algorithm has three primary operations: ① copy, ② $A+=B$, and ③ $C = A+B$, and requires $1+p-1 = p$ steps. Figure 6

Table 1: Comparing DAV of reduce-scatter algorithms

Reduce-scatter algorithms	Data Access Volume per node
Ring [45]	$5 \cdot s \cdot (p-1)$
Rabenseifner [50]	$5 \cdot s \cdot p \cdot (\frac{1}{2} + \frac{1}{4} + ... + \frac{1}{p})$
DPML [13]	$s \cdot (5 \cdot p - 1)$
YHCCL (proposed)	$s \cdot (3 \cdot p - 1)$

exemplifies that we have three processes, and thus require three steps (S0-S2) for reduce-scatter. The three reduction steps are: (S0) rank a/b/c copies the data slice of 2/3/1 into the shared memory; (S1) the rank a/b/c reduces (using $A+=B$) slice 3/1/2 to the shared memory; (S2) the rank a/b/c reduces (using $C=A+B$) slice 1/2/3 of the shared memory to the receiving buffer. Note that each rank operates its own receiving buffer in the operations $C=A+B$ during step S2. This is because the rank a/b/c will copy slice 2/3/1 to shared memory in S0, and the process-slice binding is fixed. When dealing with a much larger message, our design performs reduce-scatter multiple times to keep the data slice sufficiently small to be cached.

From the perspective of data access volume (DAV), all algorithms for the sliced reduction problem contain $p \cdot (p-1)$ slice reduction operations; each has two loads and one store, which requires a DAV of $3 \cdot I$. Thus we get the DAV of the algorithm A' as $3 \cdot I \cdot p \cdot (p-1) + \sum_{i=1}^{p} \sum_{j=1}^{p-1} V(T_{i,j}) = s \cdot (3 \cdot p - 1)$. We compare the DAV of the prior MPI reduce-scatter algorithms in Table 1. Note that both Ring and Rabenseifner are built on the send/recv communication primitive. For the kernel-assisted MPI, there is a synchronization and single memory copy in a point-to-point communication [25]. Typical kernel modules (CMA [54] and KNEM [27]) perform data copy in kernel space, while shared memory copy is a user space operation, thus involves more system overhead. In a step of the ring reduce-scatter, there is $2 \cdot I$ or $3 \cdot I$ DAV in one send/recv or reduction. The Rabenseifner reduce-scatter half the message size in each step, and the number of steps is the logarithm of p. For DPML, it copies the whole sending buffer to shared memory which needs at least $2 \cdot s \cdot p$. Compared to DPML, YHCCL can eliminate around 40% unnecessary data movements and consumes only $p \cdot I$ shared memory. To summarize, the DAV of YHCCL is smaller than the other shared memory algorithms when using over two processes. This is due to the fact that our algorithm minimizes unnecessary data movements. Although the synchronization overhead of MA reduction can grow with the number of parallel processes (e.g., on future architectures with more cores), the reduction in memory overhead given by our approach (up to 40% reduction over DPML and RG) should amortize the synchronization overhead on large messages.

For large messages, the shared memory, sizing $p \cdot I$, may exceed the cache capacity. We prefer using a small slice size I to perform reduction multiple times to keep the shared memory in the cache. Besides, on a multi-NUMA system, as the MA reduction uses only *local memory* and small slices of shared memory (suited in the cache), it can avoid accessing remote NUMA's physical memory.

Socket-aware optimization. For our MA reduction, using more processes will often endure an expensive synchronization overhead. This is because there exists a synchronization between every two neighboring steps. In practice, we use atomic operations to update a flag held by each process for synchronization and signaling between children and parents in the reduction tree. Thus, we present a socket-aware MA reduction algorithm to mitigate this overhead.

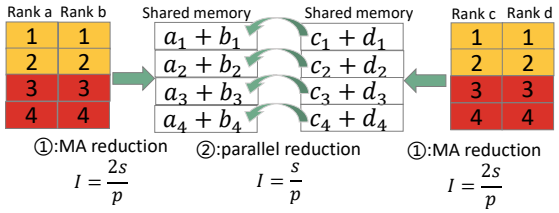


Figure 7: Two-level reduce-scatter on 4 processes (m=2 sockets, each has n=2 processes).

Figure 7 demonstrates the socket-aware MA reduction on $p=4$ processes (divided into $m=2$ sockets), where the sending buffer is chunked into four slices. In the socket-aware design, the first step is performing MA reduction with slice size $I = 2 \cdot s/p$, and we get the local reduction result of each socket. There need $p/m - 1$ times synchronizations. In the second step, each rank performs reductions on the corresponding slices and then stores the final result in shared memory or its receiving buffer. There requires one synchronization.

In the first step, there are m intra-socket MA reductions, and each works on p/m processes and message size s. So there is a DAV of $m \cdot s \cdot (3 \cdot \frac{p}{m} - 1) = s \cdot (3 \cdot p - m)$. In the second step, m socket incurs $m-1$ reduction, and thus there is a DAV of $(m-1) \cdot (3 \cdot \frac{s}{p} \cdot p) = 3 \cdot s \cdot (m-1)$. The total DAV is $s \cdot (3 \cdot p - m) + 3 \cdot s \cdot (m-1) = s \cdot (3 \cdot p + 2 \cdot m - 3)$. Notice that the socket-aware design has a slightly larger DAV but fewer synchronization (p/m) than the pure MA reductions ($p-1$). The synchronization overhead of socket-aware MC can grow with the number of parallel processes (e.g., on future architectures with more cores). However, the reduction in memory overhead given by our approach (up to 40% reduction over DPML and RG) should amortize the synchronization overhead.

3.4 Movement-avoiding MPI All-reduce

Following the design of our reduce-scatter algorithm, we provide our design of movement-avoiding all-reduce algorithm. Our algorithm first performs reduce-scatter with MA reduce-scatter and stores the result on the shared memory, and then each rank copies the result to their receiving buffers. Copying the reduction result to the receiving buffer requires a DAV of $2 \cdot s \cdot p$ per node. As the DAV of the socket-aware reduce-scatter is $s \cdot (3 \cdot p + 2 \cdot m - 3)$, we get the DAV of the all-reduce algorithm as $s \cdot (5 \cdot p + 2 \cdot m - 3)$.

Table 2 compares the DAV of different all-reduce algorithms. The RG all-reduce is a pipelined tree algorithm, and parameter k is the branching degree of reduction. In the first step of the RG all-reduce, the children processes copy data to the shared memory, and then the parent rank reduces the data. For other steps, the parent directly reduces the data on the shared memory. After reduction, all ranks copy their results to receiving buffer. Thus, the DAV of the RG all-reduce algorithm is $(2 \cdot s \cdot k + 3 \cdot s \cdot k)(\frac{p}{k+1}) + (3 \cdot s \cdot k(\frac{p}{(k+1)^2} + \frac{p}{(k+1)^3} + ... + \frac{p}{p})) + (2 \cdot s \cdot p)$. When $p \geq 4$, our all-reduce has a smaller DAV than the other algorithms.

Table 2: Comparing DAV of the all-reduce algorithms

All-reduce algorithms	Data Access Volume per node
Ring [45]	$7 \cdot s \cdot (p-1)$
Rabenseifner [50]	$7 \cdot s \cdot p \cdot (\frac{1}{2} + \frac{1}{4} + ... + \frac{1}{logp})$
DPML [13]	$s \cdot (7 \cdot p - 1)$
RG [34]	$s \cdot p \cdot (\frac{5 \cdot k}{k+1} + \frac{3 \cdot k}{(k+1)^2} + ... + \frac{3 \cdot k}{p} + 2)$
YHCCL (MA reduction)	$s \cdot (5 \cdot p - 1)$
YHCCL (socket-aware MA)	$s \cdot (5 \cdot p + 2 \cdot m - 3)$

Table 3: Comparing DAV of reduce algorithms

Reduce Algorithms	Data Access Volume per node
DPML [13]	$s \cdot (5 \cdot p + 1)$
RG [34]	$s \cdot p \cdot (\frac{5 \cdot k}{k+1} + \frac{3 \cdot k}{(k+1)^2} + ... + \frac{3 \cdot k}{p})$
YHCCL (MA reduction)	$s \cdot (3 \cdot p + 1)$
YHCCL (socket-aware MA)	$s \cdot (3 \cdot p + 2 \cdot m - 1)$

3.5 Movement-avoiding MPI Reduce

Based on our reduce-scatter design, we present the movement-avoiding reduce algorithm. Our algorithm first performs a socket-aware MA reduce-scatter to shared memory, and then the root rank copies the result from the shared memory to the receiving buffer. The DAV of our reduce algorithm is $(s \cdot (3 \cdot p + 2 \cdot m - 3) + 2 \cdot s = s \cdot (3 \cdot p + 2 \cdot m - 1))$. Table 3 compares the DAV of different reduce algorithms. When $m << p$, we see that our reduce algorithm has a smaller DAV when $p \geq 4$.

4 ADAPTIVE COLLECTIVES WITH FINE-GRAINED NON-TEMPORAL STORES

This section addresses the issue that existing pipelined collectives cannot effectively utilize non-temporal (NT) stores.

4.1 The Idea

To further investigate the difference between the temporal and non-temporal (NT) memory accesses, we redesign the STREAM COPY benchmark with two data copy operations: `t-copy`, which uses prefetched loads and regular temporal stores, and `nt-copy`, which uses prefetched loads but non-temporal stores. To simulate the slice data copy of the pipelined collectives, we chunk a 16 GB array into data slices and use slice as the copy granularity. Table 4 shows the bandwidth differences between `memmove`, `t-copy`, and `nt-copy`.

We see that using `nt-copy` can achieve a 50% bandwidth improvement compared to using `t-copy` on sliced large data copy. This is because `t-copy` writes allocate the cache and incur the RFO overhead. When the cache is fully filled, the cache misses may evict the dirty cacheline to the main memory, resulting in a waste of memory bandwidth. By contrast, `nt-copy` can avoid the overhead by hinting the CPU not to cache data. The experiment reveals that for the sliced large data copy where the stored data is not to be used soon, we should use `nt-copy`. If the stored data is going to be used soon (e.g., the **copy-in** operations in pipelined collectives), or if the work data size is small, we should use `t-copy` rather than *nt-copy* to leverage the cache.

Table 4: The memory bandwidth (in MB/s) when using slice copy to copy a 16GB array on NodeA.

	memmove	t-copy	nt-copy
512 KB	147361.4	151731.1	236571.3
1 MB	149686.3	152558.9	239518.3
2 MB	232060.8	158386.0	237662.7

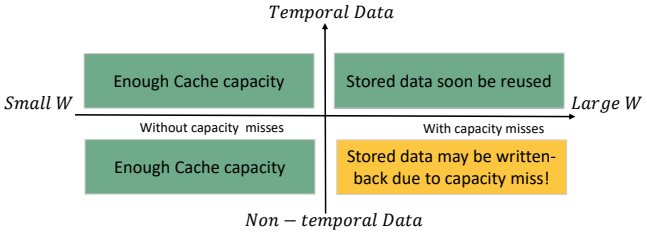


Figure 8: Characterizing the slice copy in pipelined collectives.

Algorithm 1: Memory Copy with adaptive NT Store

```
Data: source buffer a, dest buffer b, copy size τ, non-temporal flag t,
      available cache capacity C, work data size W
C = {c', c' + p · c''};
Function adaptive-copy(a, b, τ, t, C, W):
    if t == true and W > C then
        t-copy(a, b, τ);
    else
        nt-copy(a, b, τ);
    return;
```

The *memmove* switches on or off NT instructions by simply comparing copy size with a threshold [38, 46], which fails to capture the complex memory access patterns of the current pipelined collectives. Thus, we aim to characterize the slice data copy in the pipelined collectives and pass the information to the reimplemented copy function, which can adaptively enable the NT data accesses.

4.2 Modeling Adaptive NT Stores

Figure 8 shows that we characterize the slice copy in pipelined collectives in two dimensions: ① the work data size (W) of the collective algorithm, and ② whether the stored buffer of copy is temporal or non-temporal. We define the work data size of the collective algorithm (W) as the sum of the sending buffer, receiving buffer, and other auxiliary buffers (e.g., shared memory) used in the algorithm. The temporal data is referred to be the data that are frequently used in the algorithm, while the non-temporal data means the data is rarely used [10, 38] in a period. For an algorithm with large W and the slice copy storing non-temporal data, there will occur capacity cache misses and write allocation. As the non-temporal data are rarely used in a period, we choose to use the NT store to write the data directly into the memory to avoid wasting the memory bandwidth. On the other hand, if the stored data is temporal data, whatever the size of W is, writing the data to the cache rather than the main memory will utilize the cache for the next reference. When W is small, there will be no capacity cache misses. Thus, we use the temporal store to write and allocate data into the cache. As a consequence, it requires an adaptive copy that uses `nt-copy` to bypass the cache and `t-copy` to utilize the cache.

Algorithm 2: MA all-reduce with *adaptive-copy*

```
Data: sending buffer sb, receiving buffer rb, shared memory shm, message
      size s, slice size I, my rank r, process number p, reduce operation op
1  Function MA-allreduce(sb, rb, s, op):
2      W ← s·p + s·p + p·I;
3      for (ss = 0; ss < sz; ss+ = I·p) do
4          for j = 0 to p do
5              l ← I·((j + r + 1)%p);
6              i ← ss + l;
7              if j == 0 then
8                  adaptive-copy(sb[i], shm[l], I, 0, C, W)
9              else
10                 op(sb[i], shm[l], I)
11             if j > 0 and i < p − 1 then
12                 Sync-with-neighbor
13         Sync-intra-node;
14         for j = 0 to p do
15             i ← ss + I·j;
16             adaptive-copy(shm[I·j], rb[i], I, 1, C, W)
```

Algorithm 1 shows the pseudocode of `adaptive-copy`, which extends prior copy operations (e.g., `memmove`) with six arguments. The additional three parameters (t, C, W) represent the characteristics of the pipelined algorithm and the system cache information. `adaptive-copy` use the parameters to identify the orange area of Figure 8. Some recent CPU architectures (e.g., Intel's skylake) use a non-inclusive L3 cache [11, 53], and the data in L1 and L2 is not located in L3. Thus, the available cache capacity is larger than L3, and the available cache is $C = c' + p \cdot c''$, where c' is the size of the last level cache, and c'' is the size of the second last level cache per core. When the last-level cache is inclusive, we have $C = c'$.

4.3 Adaptive Collective Implementations

4.3.1 How to use adaptive-copy in MA all-reduce algorithm. Algorithm 2 shows how we use `adaptive-copy` in the MA all-reduce algorithm (Section 3). It also includes the operations of copy-ins and copy-outs. The work data size of the MA all-reduce on p processes is $s \cdot p + s \cdot p + p \cdot I$, which includes the size of sending/receiving buffer and shared memory. The first `adaptive-copy` in line 8 stores the data in the shared memory. In this case, the stored slice is used in the next reduction, and the shared memory is frequently used during the reduction. Thus we set the non-temporal flag to 0. The second slice copy in line 16 is that each rank copies the result to the receiving buffer. The receiving buffer is used once during the all-reduce, and thus we set the non-temporal flag to 1. For the socket-aware MA all-reduce, the work data size is W. The work data size for a m socket system is $s \cdot p + s \cdot p + m \cdot p \cdot I$.

4.3.2 How to use adaptive-copy in pipelined broadcast algorithm. Algorithm 3 shows the classical pipelined broadcast algorithm with adaptive-copy based on shared memory [28, 34]. In each step, the root rank copies a slice to shared memory, and non-root processes copy the previous data to receiving buffers. The work data size of the algorithm is $s + s \cdot (p-1) + 2 \cdot I$. As the message is chunked into slices, for each slice, the root process copies it to the shared memory and then the data is copied to the receiving buffer by the non-root processes. We see that the shared memory is temporal data, and the receiving buffer is non-temporal data.

Algorithm 3: Pipelined broadcast with *adaptive-copy*

```
Data: root process R
1  Function pipe-bcast(sb, rb, s, R):
2      W ← s + s·(p − 1) + 2·I;
3      for (ss = 0; ss < sz; ss+ = I) do
4          if r == R then
5              adaptive-copy(sb[ss], shm[(ss/I %2)], I, 0, C, W)
6          else
7              s1 ← ss − I;
8              if s1 ≥ 0 then
9                  adaptive-copy(shm[(s1/I %2)], rb[s1], I, 1, C, W)
10         Sync-intra-node;
11     if r ≠ R then
12         adaptive-copy(shm[((ss−I)/I %2)], rb[ss − I], I, 1, C, W)
```

Algorithm 4: Pipelined all-gather with *adaptive-copy*

```
1  Function pipe-all-gather(sb, rb, s):
2      W ← s·p + s·p² + 2·p·I;
3      for (ss = 0; ss < sz; ss+ = I) do
4          adaptive-copy(sb[ss], shm[r·2·I + (ss/I %2)·I], I, 0, C, W);
5          s1 ← ss − I;
6          if s1 ≥ 0 then
7              for a = 0 to p do
8                  adaptive-
                       copy(shm[a·2·I + (s1/I %2)·I], rb[s1], I, 1, C, W)
9          Sync-intra-node;
10     for a = 0 to p do
11         adaptive-copy(shm[a·2·I + ((ss−I)/I %2)·I], rb[ss − I], I, 1, C, W)
```

4.3.3 How to use adaptive-copy in pipelined all-gather algorithm. Algorithm 4 shows the pipelined all-gather algorithm based on shared memory [28, 43]. For each step, all processes copy a slice to shared memory. Then processes copy the slices from shared memory to receiving buffers. For the first copy in line 4, the stored data on shared memory will soon be used. For the second copy in lines 8 and 11, the stored slices in receiving buffers are used only after the all-gather. As a result, shared memory is temporal data, and receiving buffer is non-temporal data. The work data size for the algorithm is $s \cdot p + s \cdot p^2 + 2 \cdot p \cdot I$.

5 PERFORMANCE EVALUATION

This section introduces the implementation and compares the movement-avoiding collectives with state-of-the-art algorithms, and then evaluates adaptive collectives with fine-grained NT stores. It also compares YHCCL with state-of-the-art MPIs and real-life applications.

5.1 Implementation Details

We have implemented our optimization techniques as a component of the Open MPI MCA coll framework. Our framework, namely YHCCL[1], is released as an open-source library. We also developed a profiling tool (PMPI) to support MPI collective performance profiling. MPI programmers can use an environment variable to enable our optimization via `OMPI_MCA_COLL_YHCCL_priority=100`.

For the MA reduction algorithm, we use the maximum and minimum slice sizes I_{max} and $I_{min} = Cache\ Line$ respectively to fit the

[1]The code and data are available at: https://github.com/pengjintao/yh-ccl.

shared memory into the cache and avoid the cache line's false sharing [32]. Therefore, the actual slice size is $I = max(min(\frac{s}{p}, I_{max}), I_{min})$.

Even though the socket-aware design mitigates the synchronization overhead, the overhead can become prohibitively expensive as messages get smaller. Some current parallel reduction algorithms like DPML [13] offers one synchronization with coping all sending buffers to shared memory, but it does not utilize socket-aware and cache hierarchy. As a result, the two-level parallel reduction in Figure 4 is that YHCCL optimizes the current DPML algorithm with a two-level hierarchy and switches the reduction algorithm to it when the message is too small (e.g., $s \leq 256$ KB) to benefit from MA reduction at the algorithm level.

5.2 Experimental Setup

5.2.1 Hardware platforms. We evaluate YHCCL on three computing clusters with shared memory multi-cores.

NodeA. Each node has 2x 32-core AMD EPYC 7452 CPUs (64 cores in total). Each CPU has a collective 256 MB non-inclusive L3 cache, complemented by an inclusive L2 cache of 512 KB per core [53]. Moreover, the CPUs are equipped with 16 DDR4-3200 memory channels and are interlinked through a quartet of 16 GT/s xGMI buses.

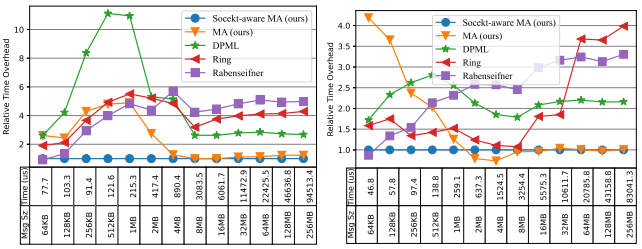
NodeB. Each node of this system has two 24-core Intel Xeon Platinum 8163 CPUs (48 cores in total). Each CPU incorporates a collective 66 MB non-inclusive L3 cache, accompanied by an inclusive 1 MB L2 cache per core [11]. Moreover, these CPUs feature 12 DDR4-2666 memory channels and are interconnected by 3x 10.4 GT/s UPI buses.

Cluster C. Each node has of this cluster two 12-core Intel Xeon E5-2692 v2 (24 cores in total), where each CPU has a shared 60 MB inclusive L3 cache.

5.2.2 Software and Workloads. Our evaluation uses Open MPI v4.1.3 based on UCX v1.8.0, GCC v10.2.0, Pytorch v1.10, and Horovod v0.20.1. As the application workloads, we use both the OSU MPI benchmark suite [9] and real-life applications to evaluate YHCCL. The two applications are Adaptive Mesh Refinement Mini-App (MiniAMR) and the training of distributed deep neural networks (DNNs) applications.

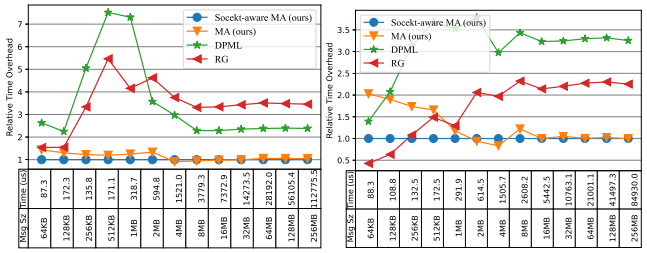
5.3 Evaluating Movement-avoiding Collectives

Figure 9 compares the (socket-aware) MA reduce-scatter with Ring [45], Rabenseifner [50], and DPML [13] reduce-scatter over various message sizes. We have turned these algorithms' parameters (slice size and branch degree) to have their best performance on the NodeA. For example, we select the best reduction size to be 8KB for DPML. Our MA reduction has a maximum slice size of 256KB on NodeA and 128KB on NodeB. Figure 9 shows the algorithms' relative time overhead to the socket-aware MA algorithm. We see that the MA reduce-scatter and socket-aware design have significant performance advantages for messages larger than 64KB. The socket-aware MA reduce-scatter shows an average 4.18x, 3.8x, 3.6x speedup on NodeA and 2.21x, 1.8x, 2.47x on NodeB compared to DPML, Ring, Rabenseifner. In some cases, MA reduction performs better than



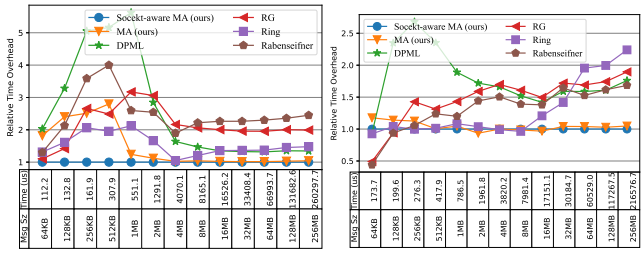
(a) NodeA, p=64 (b) NodeB, p=48

Figure 9: Reduce-scatter algorithm comparison.



(a) NodeA, p=64 (b) NodeB, p=48

Figure 10: Reduce algorithm comparison.

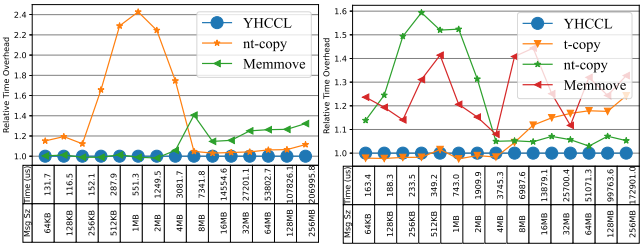


(a) NodeA, p=64 (b) NodeB, p=48

Figure 11: All-reduce algorithm comparison.

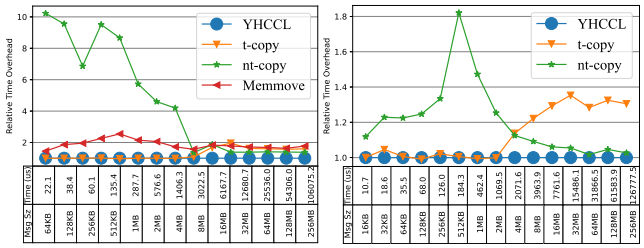
socket-aware MA. The socket-aware MA incurs less synchronization compared to MA, but it requires an extra buffer and additional data movements. In this case, when the socket-aware MA buffer cannot be fitted into a smaller cache, it may perform worse than MA reduction due to cache misses. These performance improvements are due to the MA reduction minimizing the data copy, which can use private data and a small volume of shared memory. Others like Ring and Rabenseifner are based on point-to-point communication, which involves much data copy as shown in Table 1 and 2, even with kenel-assited MPI. For small messages, the MA designs suffer from synchronization, and the Rabenseifner algorithm has an advantage because it has a logarithmic step number.

Figure 11 shows how our socket-aware MA all-reduce performs compared with Ring [45], Rabenseifner [50], DPML [13] and RG [34]. For the RG all-reduce, the branching degree in a socket is set to 2, and the slice size is set to 128KB. We see that our algorithm has a significantly smaller overhead for large messages. RG and Rabenseifner all-reduce have logarithmic steps and perform well for messages smaller than 128KB. Figure 10 shows how our socket-aware MA reduce compared with DPML [13] and RG [34]. The proposed algorithm has an advantage for messages larger than 64KB on NodeA and 128KB on NodeB.



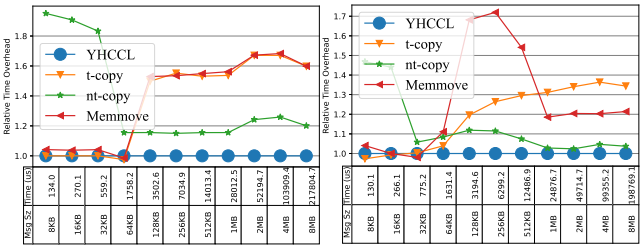
(a) NodeA, p=64, I_{max} =256KB **(b) NodeB, p=48, I_{max} =128KB**

Figure 12: Evaluate adaptive all-reduce with adaptive-copy.



(a) NodeA, p=64, I_{max} =1MB **(b) NodeB, p=48, I_{max} =1MB**

Figure 13: Evaluate adaptive broadcast with adaptive-copy.



(a) NodeA, p=64, I_{max} =1MB **(b) NodeB, p=48, I_{max} =1MB**

Figure 14: Evaluate adaptive all-gather with adaptive-copy.

To summarize, YHCCL outperforms other collectives on memory-intensive cases. This is mainly because YHCCL can eliminate redundant data copies and avoid inter-NUMA memory accesses.

5.4 Evaluating Adaptive Collectives

Figure 12 shows the relative time overhead of the optimized socket-aware MA all-reduce (YHCCL) compared to other implementations. Comparing with *nt-copy*, we observe that *t-copy* performs better on small messages when the cache is sufficient for collective operations. These are small messages that can be entirely cached, but nt-copy guides the cache line to be written back to the memory for each store operation, incurring unnecessary overhead. But for large messages, there will be a cache capacity miss. In this case, *nt-copy* performs better. This is because it writes the data directly to memory, avoiding wasting memory bandwidth on cache loading and dirty data write-back. On one hand, YHCCL is consistent with the *t-copy* for small messages, because YHCCL uses *t-copy* when the work data set is small. One the other hand, YHCCL performs better than *t-copy* and *nt-copy* on large messages, because YHCCL can properly identify the case to use the NT store for copy-ins and copy-outs.

The *memmove* with threshold based on copy size performs worse than *nt-copy* for large messages because it cannot capture the memory access patterns of pipelined collectives. Our results show that YHCCL performs better than *memmove* for large messages on NodeA and NodeB. When it comes to the data access bandwidth, the socket-aware all-reduce has a DAV of $s \cdot (5 \cdot p + 2 \cdot m - 3)$. Thus for *memmove* at 256 MB message on NodeA and NodeB, the original socket-aware MA all-reduce has a DAB of 314.7 and 281.8 GB/s. After enabling the adaptive NT stores, the DAB is improved to 416.2 and 374.7 GB/s. Although pipelined collectives with *memmove* can switch between temporal and non-temporal instructions, the *memmove* only utilizes the information of copy data size. As a result, our adaptive design outperforms *memmove* and other implementations.

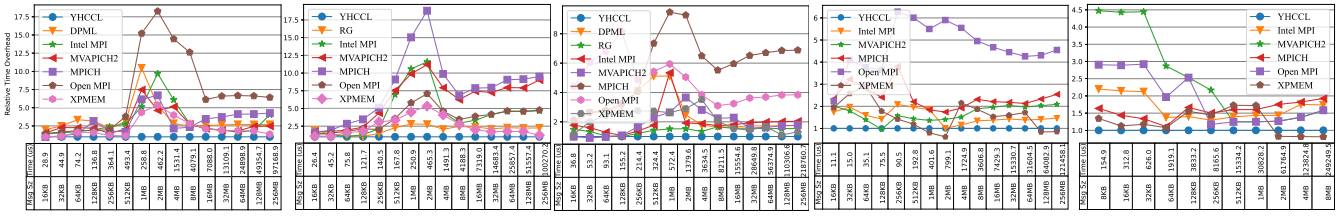
YHCCL switches from *t-copy* to *nt-copy* when $W > C$ and non-temporal flag $t == 1$ (shown in Algorithm 1 and 2). For socket-aware MA all-reduce, the work data set $W = 2 \cdot s \cdot p + m \cdot p \cdot I_{max}$. After solving $W > C$, we get $s > \frac{C - m \cdot p \cdot I_{max}}{2 \cdot p}$. For NodeA and NodeB with non-inclusive L3 cache capacity 256MB and 66MB and with a private L2 cache of 0.5MB and 1MB per core, we get $C = c' + p \cdot c'' = 294912/116736$ KB, and $\frac{C - m \cdot p \cdot I_{max}}{2 \cdot p}$ =2176/1152 KB. That is, on NodeA/NodeB, when the message size is larger than 2176/1152KB, YHCCL starts to use *nt-copy* for non-temporal data. We see that there are improvements of YHCCL compared to *t-copy* starting from 2MB on NodeA and 1MB on NodeB in Figure 12. Thus, YHCCL identifies the right time to switch to the NT store.

Figure 13 and Figure 14 show the relative timer overhead of adaptive pipelined broadcast/all-gather (YHCCL) compared to *t-copy*, *nt-copy*, and *memmove*. For all-gather in Figure 14, the start size is 8KB-8MB, and the aggregated data size is p times larger. YHCCL outperforms *memmove* significantly because broadcast and all-gather have no computation. For small messages, *nt-copy* does not offer advantages, and for large messages, *t-copy* is not beneficial. YHCCL outperforms them in both cases due to the appropriate NT switching. Compared with the current implementation with *memmove*, YHCCL performs better for large messages via adaptive NT store, while the current *memmove*-based implementation failed to identify the cases to use NT instruction.

5.5 Comparing YHCCL With State-of-the-arts

Single node performance. Figure 15 compares YHCCL with other state-of-the-art collectives, including DPML [13], RG [34], MVAPICH2 2.3.7 [7], Intel ONEAPI 2022 [5], Open MPI 4.1.1 [8], MPICH 4.1 [6], and Hashmi's XPMEM-based collectives [30, 31]. To simulate the application's behavior, we update the sending and receiving buffers before each iteration. Thus, some collectives with kernel modules may not benefit from the local cache. For unbalanced collectives (reduce and broadcast), we show the maximum overhead of processes instead of the average because some processes of tree-based collectives finish the collective quickly; they will amortize the slowest overhead. Open MPI and Intel MPI has configured with CMA [54] module, while MVAPICH2 is configured with socket-aware collective optimization [3].

We see that the reduce-scatter, reduce, all-reduce, broadcast, and all-gather have an average speedup of 1.9-5.0x, 2.0-6.4x, 1.4-5.2x, 1.4-4.5x, and 1.2-2.2x, compared to state-of-the-arts over various



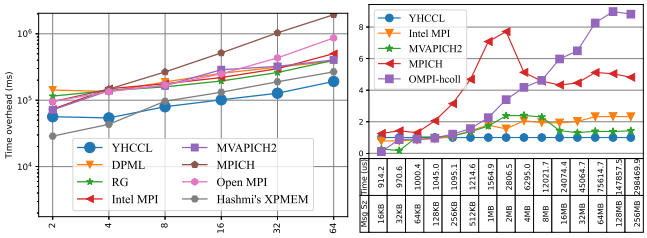
(a) Reduce-scatter, p=64 (b) Reduce, p=64 (c) All-reduce, p=64 (d) Broadcast, p=64 (e) All-gather, p=64

Figure 15: Performance comparison between redesigned collectives and state-of-the-arts.

message sizes. These improvements are from the movement-avoid reduction design and the usage of adaptive copies. For reduction collectives (reduce, reduce-scatter, and all-reduce), Hashmi's XPMEM-based reduction [30] directly accesses remote data, which causes inter-NUMA memory access on large messages. On smaller messages, it directly referent *remote memory* of other processes and thus causes inter-socket access in the real application. Other shared memory reductions and send-/recv- based reductions have a larger DAV than MA reduction. For collectives with significant data storing (all-reduce, broadcast, and all-gather), the RFO overhead and cache write-back downgrade their performance. As evidence, for messages larger than 128 MB, we see the performance of Hashmi's all-reduce has improved due to the slice size of copy $\frac{s}{p} = 2$ MB being larger than the threshold of *memmove* (2 MB). Similarly, for broadcast and all-gather, Hashmi's XPMEM-based algorithm is free from the RFO and cache write-back overhead after 128/2 MB message. Thus, after 128/2 MB, Hashmi's XPMEM-based broadcast and all-gather outperform YHCCL.

Single node scalability. We take all-reduce as an example because all-reduce utilizes all optimizations. Figure 16a shows the scalability of YHCCL on a single node, where YHCCL all-reduce uses MA reduction with a 128 KB slice. It exceeds the other designs after 8 processes due to the lowest copy among shared memory algorithms; it avoids inter-NUMA memory access and has an adaptive non-temporal store. MA all-reduce has a DAV of $s \cdot (5 \cdot p - 1)$, while Hashmi's XPMEM-based all-reduce is $5 \cdot s \cdot (p-1)$. As a result, the gap in DAV becomes more apparent as p becomes smaller. This causes Hashmi's algorithm to perform better on two and four processes. Compared to the all-reduce implementation of DPML, RG, Intel MPI, MVAPICH2, MPICH, Open MPI, Hashmi's XPMEM, YHCCL has a maximum speedup of 2.5x, 2.6x, 2.8x, 2.8x, 10.1x, 4.5x, 1.5x.

Multi-node performance evaluation. For all-reduce, Figure 16 shows the comparison between YHCCL and other MPI implementations on the system of NodeA. All these MPI implementations are configured carefully to have their best performance on the InfiniBand networks [1–4]. The *OMPI-holl* is the Open MPI with Hcoll collective library [4], which is optimized for the InfiniBand network. We implement a hierarchical approach by first applying the proposed reduce-scatter within a node, then ring all-reduce across multiple nodes before performing all-gather within a node. Compared to other implementations, we see that YHCCL has a 1.4-8.8x speedup on large messages compared to other designs when using 1024 processes. On small messages, MVAPICH2 and OpenMPI-hcoll use tree-based implementations, which have advantages over the ring-based strategy used by YHCCL on small messages. This is because YHCCL not only has advantages within the node but also



(a) Single node scalability (b) Multi-node performance

Figure 16: Large-scale performance evaluation of all-reduce.

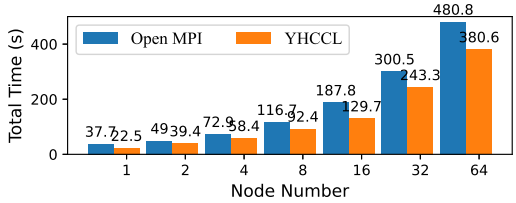


Figure 17: Time overhead of Mini-AMR on multiple node with 64 processes for each node (smaller is better).

uses multiple processes to communicate simultaneously between nodes to saturate the network [52].

5.6 Evaluating YHCCL on Real Applications

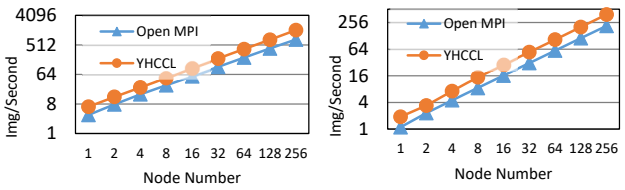
We have implemented our techniques as an Open MPI component, without having to modify users' codes. This section shows the all-reduce performance with two real applications.

Mesh Refinement (AMR) Application. On NodeA, we compare the performance between the original Open MPI and YHCCL with Adaptive Mesh Refinement Mini-App (Mini-AMR). Mini-AMR is a 3D stencil calculation with Adaptive Mesh Refinement. It is one of the Exascale Computing Project (ECP) Proxy applications. All-reduce is frequently used in Mini-AMR, and the message length is proportional to the number of refines. We use 1 to 64 nodes on the cluster of NodeA and set refine times to 40000. The results are shown in Figure 17. YHCCL reduces the execution time of Mini-AMR by 1.26x-1.67x over different processes compared to the default settings of Open MPI based on CMA kernel modules.

Despite being kernel-assisted, the CMA copy technique performs data copying on a per-page basis and lacks the utilization of non-temporal instructions for efficient large data transfer, as reported by Linux sources [12]. Table 5 presents a comparison between the DMA copy (process_vm_readv) and the `adaptive-copy` method for two copy patterns involving 32 MB data per message. The sending buffers are allocated in shared memory using MPI_Win_allocate_shared, while the receiving buffers are allocated in private space. In both the one-to-all and ring copy scenarios,

Table 5: CMA copy compared to `adaptive-copy` (second).

Pattern	DMA copy	`adaptive-copy`
One-to-all: rank 0 to rank i	0.061	0.014
Ring: rank i to rank $(i+1)\%p$	0.027	0.017



(a) Resnet 50 **(b) VGG-16**

Figure 18: Evaluating DNN training with 1 to 256 nodes on Cluster C (with 24 processes per node, larger is better).

the `adaptive-copy` method exhibits significantly improved performance compared to the CMA copy technique, achieving speedups of 4.35x and 1.58x, respectively. These results indicate that the current CMA mechanism suffers from excessive cache load during storage operations. Furthermore, the one-to-all copy case using CMA demonstrates slower performance compared to the ring copy cases. This discrepancy arises from the lock contention that occurs when multiple processes access the same pages [17].

Convolutional Neural Network (CNN) Training. CNN training consists of forwarding propagation, backward propagation, and parameter update [35]. The parameter update stage is implemented by a characteristic optimizer such as SGD [61]. Data parallelism is a popular way of distributed training. Each worker has a copy of the model and processes a portion of the dataset. In the parameter update, the large message all-reduce is frequently used. We measure the performance of 2 different CNN models, i.e., Resnet-50 [49] with 25.6M parameters, and VGG-16 [20] with 138.4M parameters. Figure 18 shows the results. The improvement of proposed designs compared to the original Open MPI on 6144 cores is 1.94x (Resnet-50) and 1.80x (VGG-16). Due to the relatively weaker CPU, the computation dominates the end-to-end execution time on this specific platform (Cluster C), and our optimization in hiding communication with computation for inter-node all reduce. Thus the figures show fixed improvement.

6 RELATED WORK

Our work explicitly targets MPI collective optimization on shared-memory multi-core CPUs. There are three typical ways of implementing collective operations on a shared memory node: sent and received (recv), direct shared memory, and direct kernel-assisted collectives.

Many mainstream MPI libraries (e.g., Open MPI [8], MPICH [6]) employ the send/recv paradigm for collective operations. In the early iterations of these libraries, shared memory was harnessed for the on-node send/recv operations, as demonstrated by designs like the ring-based approach [45], Rabenseifner's technique [50], and node-conscious frameworks [15]. In this paradigm, for each MPI_Send and MPI_Recv pair, the sending process duplicates the data into a previously allocated shared memory space. Subsequently, the receiving process engages in a pipelined procedure, transferring the data to its designated buffer. This intricate process requires two distinct data copies [29]. While the send/recv-driven MPI collectives enhance portability and are easy to deploy, they often deliver sub-optimal performance [8], leaving much room for performance improvement.

To mitigate the issue of extensive data copies of share-memory MPI sent/recv, kernel modules like KNEM [27], CMA [54], LiMiC [36], and XPMEM [56], are used to achieve "zero-copy" for send/recv. For example, XPMEM is a kernel module that enables a process to expose its virtual address to other processes, allowing user-level unlimited data access and better performance on reduction collectives than other kernel modules [30], which only support kernel-level single-copy. Nonetheless, these kernel modules are required to collect all physical pages of the user buffer, which causes system overhead. Further, deploying such modules requires extra effort from the administrator and is not always available on target systems [34]. For instance, in a multi-tenant HPC system, administrators could have security concerns when installing external kernels [19], where some kernels lack security checks [27]. We also note that some kernel solutions fail to capture the cross-socket memory accesses.

In contrastive to sent/recv and kernel-based solutions, direct shared memory reduction [13, 34] creates a global shared memory for caching intermediate results. This technique is implemented using POSIX or SYSTEM V shared memory but incurs redundant copy-in overhead. Some prior studies leverage NUMA/Cache/socket information for optimizing collectives [34, 40, 42, 57, 60]. Hybrid MPI [40] uses threads to simulate processes to implement MPI and achieves direct data access between processes, but there are engineering challenges in real-world applications. Therefore, the shared heap [23, 39, 41, 56] is proposed to preallocate a large section of shared memory and reimplement the *malloc* to allocate data in shared memory. By using the buffers in the shared heap, MPI can directly copy data between processes. Besides this, Ownership Passing Interface [24] also achieves direct copy through preallocated shared memory and ownership passing. In this case, If the user uses MPI_Win_allocate_shared to allocate memory mapped from IPC, this essentially implements a "zero-copy" mechanism. For this, our first optimization of MA reduction won't provide additional benefits. However, our other optimizations of NUMA-aware communications and cache store optimization are still applicable. Additionally, cache-oblivious algorithms [41] have been proposed to improve the locality of collectives. But these collective designs do not consider the cache load and dirty data write-back overhead of storing instructions, which can degrade memory access performance. Different from the prior studies, our work focuses on the shared-memory collectives to address the issues of redundant data movements and inefficient usage of NT stores.

7 CONCLUSION

We have presented YHCCL, an optimized collective communication library for MPI programs, explicitly targeting shared memory multi-cores. The fundamental driving force behind the creation of YHCCL was the recognition of the redundant data transfers inherent in MPI reduction operations within shared memory contexts. Moreover, prevalent pipelined collective operations suffered substantial overheads for cache load and data write-back inefficiencies

during memory accesses. To overcome these hurdles, YHCCL introduces a novel reduction algorithm that minimizes data duplication and an adaptive approach for using non-temporal instructions to optimize cache performance for pipelined collective tasks.

We have implemented a working prototype of YHCCL and integrated it with the OpenMPI framework. YHCCL operates as a user-mode library, removing the need for privileged system access. We evaluate YHCCL across two multi-core clusters with 64 and 48 cores per node. Experimental results show YHCCL's considerable performance enhancements over state-of-the-art collective implementations, with improvements ranging from 1.2x to 6.4x on large messages. Moreover, when applying to real-world workloads, YHCCL yields speedups range from 1.3x to 2.0x.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive feedback. This work was supported in part by the National Key Research and Development Program of China under Grant No. 2021YFB0300101 and the National Natural Science Foundation of China under Grant No. 61972408. For the purpose of open access, the author has applied a Creative Commons Attribution (CCBY) licence to any Author Accepted Manuscript version arising from this submission.

REFERENCES

[1] [n. d.]. Build OpenMPI with UCX. https://openucx.readthedocs.io/en/master/running.html
[2] [n. d.]. Improve Performance and Stability with Intel® MPI Library on InfiniBand. https://www.intel.com/content/www/us/en/developer/articles/technical/improve-performance-and-stability-with-intel-mpi-library-on-infiniband.html
[3] [n. d.]. MVAPICH2 2.3.7 User Guide. http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-userguide.pdf
[4] [n. d.]. NVIDIA: Enabling HCOLL in Open MPI. https://docs.nvidia.com/networking/display/HPCXv29/HCOLL
[5] 2023-01-02. Intel MPI. https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html
[6] 2023-01-02. MPICH. https://www.mpich.org/
[7] 2023-01-02. MVAPICH Home. https://mvapich.cse.ohio-state.edu
[8] 2023-01-02. Open MPI. https://www.open-mpi.org/
[9] 2023-01-02. OSU Micro-Benchmarks. https://mvapich.cse.ohio-state.edu/benchmarks/
[10] 2023-03-14. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
[11] 2023-03-14. Intel® Xeon® Scalable Processor: The Foundation of Data Centre Innovation. https://simplecore-ger.intel.com/swdevcon-uk/wp-content/uploads/sites/5/2017/10/UK-Dev-Con_Toby-Smith-Track-A_1000.pdf
[12] 2023-y-02. Linux kernel v6.4.2. https://elixir.bootlin.com/linux/latest/source/mm/process_vm_access.c
[13] Mohammadreza Bayatpour, Sourav Chakraborty, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K Panda. 2017. Scalable reduction collectives with data partitioning-based multi-leader design. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.
[14] Mohammadreza Bayatpour, Jahanzeb Maqbool Hashmi, Sourav Chakraborty, Hari Subramoni, Pouya Kousha, and Dhabaleswar K Panda. 2018. Salar: Scalable and adaptive designs for large message reduction collectives. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 12–23.
[15] Amanda Bienz, Luke Olson, and William Gropp. 2019. Node-aware improvements to allreduce. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*. IEEE, 19–28.
[16] L. Chai, A. Hartono, and D. K. Panda. 2006. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *IEEE International Conference on Cluster Computing*.
[17] Sourav Chakraborty, Hari Subramoni, and Dhabaleswar K Panda. 2017. Contention-aware kernel-assisted MPI collectives for multi-/many-core systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 13–24.
[18] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. 2018. Characterization of mpi usage on a production supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 386–400.
[19] Sylvain Didelot et al. 2014. Improving MPI communication overlap with collaborative polling. *Computing* 96, 4 (2014), 263–278.
[20] Abhishek Dutta, Ankush Gupta, and Andrew Zissermann. 2016. VGG image annotator (VIA). *URL: http://www. robots. ox. ac. uk/vgg/software/via* 2 (2016).
[21] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. 2020. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Trans. High Perform. Comput.* 2, 4 (2020), 382–400. https://doi.org/10.1007/s42514-020-00039-4
[22] Jianbin Fang, Xiangke Liao, Chun Huang, and Dezun Dong. 2021. Performance Evaluation of Memory-Centric ARMv8 Many-Core Architectures: A Case Study with Phytium 2000+. *J. Comput. Sci. Technol.* 36, 1 (2021), 33–43. https://doi.org/10.1007/s11390-020-0741-6
[23] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. 2013. Hybrid MPI: efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.
[24] Andrew Friedley, Torsten Hoefler, Greg Bronevetsky, Andrew Lumsdaine, and Ching-Chen Ma. 2013. Ownership passing: Efficient distributed memory programming on multi-core systems. *ACM SIGPLAN Notices* 48, 8 (2013), 177–186.
[25] Juan Antonio Rico Gallego. 2016. t-Lop: Scalably and accurately modeling contention and mapping effects in multi-corec clusters. (2016).
[26] Sabela Ramos Garea and Torsten Hoefler. 2013. Modelling communications in cache coherent systems. *Technical Report* (2013).
[27] Brice Goglin and Stephanie Moreaud. 2013. KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework. *J. Parallel and Distrib. Comput.* 73, 2 (2013), 176–188.
[28] Richard L Graham and Galen Shipman. 2008. MPI support for multi-core architectures: Optimized shared memory collectives. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland, September 7-10, 2008. Proceedings 15*. Springer, 130–140.
[29] William Gropp. 2002. MPICH2: A new start for MPI implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting Linz, Austria, September 29–Oktober 2, 2002 Proceedings 9*. Springer, 7–7.
[30] Jahanzeb Maqbool Hashmi, Sourav Chakraborty, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K Panda. 2018. Designing efficient shared address space reduction collectives for multi-/many-cores. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1020–1029.
[31] Jahanzeb Maqbool Hashmi, Sourav Chakraborty, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K Panda. 2019. Design and characterization of shared address space mpi collectives on modern architectures. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 410–419.
[32] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.
[33] Johannes Hofmann, Dietmar Fey, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2016. Analysis of intel's haswell microarchitecture using the ecm model and microbenchmarks. In *International conference on architecture of computing systems*. Springer, 210–222.
[34] Surabhi Jain, Rashid Kaleem, Marc Gamell Balmana, Akhil Langer, Dmitry Durnov, Alexander Sannikov, and Maria Garzaran. 2018. Framework for scalable intra-node collective operations using shared memory. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 374–385.
[35] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 463–479.
[36] H-W Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K Panda. 2005. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *2005 International Conference on Parallel Processing (ICPP'05)*. IEEE, 184–191.
[37] Norman P Jouppi. 1993. Cache write policies and performance. *ACM SIGARCH Computer Architecture News* 21, 2 (1993), 191–201.
[38] Giorgos Kappes and Stergios V Anastasiadis. 2021. Asterope: A Cross-Platform Optimization Method for Fast Memory Copy. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*. 9–16.
[39] Shigang Li, Torsten Hoefler, Chungjin Hu, and Marc Snir. 2014. Improved MPI collectives for MPI processes in shared address spaces. *Cluster computing* 17, 4 (2014), 1139–1155.
[40] Shigang Li, Torsten Hoefler, and Marc Snir. 2013. NUMA-aware shared-memory collective communication for MPI. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. 85–96.
[41] Shigang Li, Yunquan Zhang, and Torsten Hoefler. 2017. Cache-oblivious MPI all-to-all communications based on Morton order. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2017), 542–555.

[42] Amith R Mamidala, Rahul Kumar, Debraj De, and Dhabaleswar K Panda. 2008. MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE, 130–137.
[43] Amith R Mamidala, Abhinav Vishnu, and Dhabaleswar K Panda. 2006. Efficient shared memory and RDMA based design for mpi_allgather over InfiniBand. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI User's Group Meeting Bonn, Germany, September 17-20, 2006 Proceedings 13*. Springer, 66–75.
[44] Benjamin S Parsons. 2015. Accelerating MPI collective communications through hierarchical algorithms with flexible inter-node communication and imbalance awareness. (2015).
[45] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.
[46] Silvius Rus, Raksit Ashok, and David Xinliang Li. 2011. Automated locality optimization based on the reuse distance of string operations. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 181–190.
[47] Andreas Sandberg, David Eklöv, and Erik Hagersten. 2010. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.
[48] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. 2015. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 207–216.
[49] Sasha Targ, Diogo Almeida, and Kevin Lyman. 2016. Resnet in resnet: Generalizing residual architectures. *arXiv preprint arXiv:1603.08029* (2016).
[50] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.
[51] Vinod Tipparaju, Jarek Nieplocha, and Dhabaleswar Panda. 2003. Fast collective operations using shared and remote memory access protocols on clusters. In *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 10–pp.
[52] Jesper Larsson Träff and Sascha Hunold. 2020. Decomposing MPI collectives for exploiting multi-lane communication. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 270–280.
[53] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. 2022. Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 165–175.
[54] Jerome Vienne. 2014. Benefits of cross memory attach for mpi libraries on hpc clusters. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. 1–6.
[55] Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. 2013. Comparison of different propagation steps for lattice Boltzmann methods. *Computers & Mathematics with Applications* 65, 6 (2013), 924–935.
[56] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. 2005. The SGI® AltixTM 3000 global shared-memory architecture. *Silicon Graphics, Inc* 44 (2005).
[57] Meng-Shiou Wu, Ricky A Kendall, and Kyle Wright. 2005. Optimizing collective communications on SMP clusters. In *2005 International Conference on Parallel Processing (ICPP'05)*. IEEE, 399–407.
[58] Charlene Yang. 2020. Hierarchical Roofline Analysis: How to Collect Data using Performance Tools on Intel CPUs and NVIDIA GPUs. *CoRR* abs/2009.02449 (2020). arXiv:2009.02449 https://arxiv.org/abs/2009.02449
[59] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P Jouppi, and Mattan Erez. 2011. FREE-p: Protecting non-volatile memory against both hard and soft errors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 466–477.
[60] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K Panda. 2017. Designing locality and NUMA aware MPI runtime for nested virtualization based HPC cloud with SR-IOV enabled InfiniBand. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 187–200.
[61] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. 2010. Parallelized stochastic gradient descent. *Advances in neural information processing systems* 23 (2010).

A ARTIFACT DOI

https://zenodo.org/record/8200161

B ABSTRACT

This artifact illustrates the procedure to reproduce the experimental results presented in the "Optimizing MPI Collectives on Shared Memory Multi-Cores" paper. This paper presents YHCCL, a high-performance collective communication library for shared memory nodes. YHCCL can be integrated into Open MPI as a component of the MCA coll framework. It can also run independently based on other MPI implementations by PMPI. Both methods are binary compatible for applications. The following parts provide instructions for the library deployment and experiment evaluation.

C REPRODUCIBILITY OF EXPERIMENTS

Running the experiment workflow (including deployment and evaluation) may take about four hours.

C.1 Environment

Hardware Requirement: CPU with non-temporal load/store instructions. like the system listed in Section 6 (AMD EPYC 7452, Intel Xeon Platinum 8163 or Intel Xeon E5-2692 v2). Software Requirement: CentOS Linux release 7.6, GCC-10.2.0, UCX 1.8.0 (optional), Open MPI 4.1.3, OSU MPI benchmark, Pytorch 1.10+torchvision, Horovod 0.20.1.

C.2 Deployment Workflow

S0: Download YHCCL source code from https://github.com/pengjintao/yh-ccl
S1: Download the Open MPI 4.1.3 source code
S2: Copy yhccl into directory openmpi-4.1.3/ompi/mca/coll. The directory name must be `yhccl`.
S3: Go back to openmpi-4.1.3 and run "./autogen.pl –force"
S4: ./configure –prefix=ompi-build-dir –enable-mpi-cxx CFLAGS="-O3 CXXFLAGS="-O3 -fpermissive -std=c++11" –enable-mpi1-compatibility
S5: make -j 16 & make install
S6: Use the following environment variable settings: dir=ompi-build-dir
C_INCLUDE_PATH=$C_INCLUDE_PATH:$dir/include
CPLUS_INCLUDE_PATH=$C_INCLUDE_PATH:$dir/include
PATH=$dir/bin:$PATH
LD_LIBRARY_PATH=$dir/lib:$LD_LIBRARY_PATH
S7: Check whether the building is successful: which mpicc & ompi_info
S8: Check lscpu, and make sure the process-core binding is in the right order.
S9: Building OSU MPI Benchmark with " ./configure CC=mpicc CXX=mpicxx & make -j16".

C.3 Micro-benchmark Workflow

This section introduces the same evaluation steps as the paper, which include microbenchmark, MA reduction, adaptive NT store, and application evaluations. For benchmark evaluation, the competing baselines include:

0: Intel MPI 2021.6:
1: MPICH 3.1.4: https://www.mpich.org/
2: Open MPI 4.1.3: https://www.open-mpi.org/
3: MVAPICH 2.3.7: http://mvapich.cse.ohio-state.edu/downloads/
4: DPML: https://dl.acm.org/doi/abs/10.1145/3126908.3126954
5: RG: https://ieeexplore.ieee.org/abstract/document/8665755
6: XPMEM all-reduce: https://ieeexplore.ieee.org/abstract/document/8425255

We take `all-reduce` as an example and run it with 64 processes on a node. We use a similar experiment flow for other collective operations.

S0: Enable yhccl: export OMPI_MCA_coll_yhccl_priority=100
S1: Disable yhccl: export OMPI_MCA_coll_yhccl_priority=0
S2: Verification Test: mpiexec -n 64 ./osu_allreduce -c -m 65536:268435456
S3: Performance evaluation compared with original Open MPI, and other collective implementations
Disable yhccl and mpiexec -n 64 ./osu_allreduce -c -m 65536:268435456
Enable YHCCL and mpiexec -n 64 ./osu_allreduce -c -m 65536:268435456

Expected results: We see that the YHCCL has 2.38x,1.40x,1.83x,1.73x, 5.21x, 4.60x, and 2.00x speedup on geometry average compared to DPML, RG reduction, Intel MPI 2021.6, MVAPICH2 2.3.7, MPICH 3.1.4, Open MPI 4.1.3, Hashmi's XPMEM on 64-256KB (as shown in Figure 15c).

C.4 Evaluating MA reduction & Adaptive-NT store

S0: Modify the yhccl_allreduce.cc, find the function yhccl_allreduce, change option variable mulit_leader_algorithm to choose different reduction algorithms to evaluate.
S1: Modify option variable using_non_temporal to choose whether to use *memmove* or *adaptive-copy*.
S2: Run the microbenchmark in the directory "test/allreduce.cc".

Expected results: The MA reduction algorithm has 1.50x,2.20x,2.08x, and 2.37x speedup compared to Ring, DPML, Intel RG, and Rabenseifner all-reduce on 64-256KB (as shown in Figure 10a). This is because MA reduction has a smaller *copy-in* overhead and accesses only local NUMA and shared memory. The MA reduction with *adaptive-copy* improves the performance by 28.89% at max (4MB message, as shown in Figure 13a) compared to the one using *memmove* all-reduce message and has no loss on smaller messages. This is because *adaptive-copy* can accurately switch between temporal and non-temporal instructions.

C.5 CNN Application Evaluation

S0: Install Pytorch binary with "conda install pytorch torchvision torchaudio cpuonly -c pytorch"
S1: Install Horovod:
Setting up Open MPI with yhccl.
HOROVOD_WITH_PYTORCH=1
HOROVOD_WITH_MPI=1
pip install horovod
S2: Evaluate CNN training with Pytorch based on Horovod:
Enable/Disable yhccl and run pytorch_synthetic_benchmark.py

Expected results: When running CNN training on two Xeon E5-2692v2 CPUs with 24 processes, YHCCL shows a 1.62x speedup compared to the default Open MPI (as shown in Figure 18a). On multiple nodes, the speedup is almost holding due to the limited performance of the CPU system, and the inter-node communication is overlapped with intra-node communication.

C.6 Mini AMR Application Evaluation

S0: Install Mini AMR from https://github.com/Mantevo/miniAMR.git
S1: Run with srun -N 64 -n 4096 miniAMR.x –num_refine 40000 –num_tsteps 20 –refine_freq 1 –npx 16 –npy 16 –npz 16

Expected results: We set refine times to 40000 to get a large message all-reduce which is the major overhead of the application. The –num_tsteps is the loop time. Finally, we see up to 1.67x speed in application performance (as shown in Figure 17).

D OVERALL RESULTS

We show that YHCCL outperforms the competing baselines in most test cases for both benchmark and application evaluations. But in small messages ($\leq$ 64 KB), YHCCL fails to achieve satisfying performance.