cMPI: Using CXL Memory Sharing for MPI One-Sided and Two-Sided Inter-Node Communications

Xi Wang*
University of California, Merced
Merced, CA, USA
swang166@ucmerced.edu

Bin Ma
University of California, Merced
Merced, CA, USA
bma100@ucmerced.edu

Jongryool Kim
SK hynix
San Jose, CA, USA
jongryool.kim@sk.com

Byungil Koh
SK hynix
San Jose, CA, USA
byungil.koh@sk.com

Hoshik Kim
SK hynix
San Jose, CA, USA
hoshik.kim@sk.com

Dong Li
University of California, Merced
Merced, CA, USA
dli35@ucmerced.edu

Abstract

Message Passing Interface (MPI) is a foundational programming model for high-performance computing. MPI libraries traditionally employ network interconnects (e.g., Ethernet and InfiniBand) and network protocols (e.g., TCP and RoCE) with complex software stacks for cross-node communication. This paper presents cMPI, the first work to optimize MPI point-to-point communication (both one-sided and two-sided) using CXL memory sharing on a real CXL platform, transforming cross-node communication into memory transactions and data copies within CXL memory, bypassing traditional network protocols. We analyze performance across various interconnects and find that CXL memory sharing achieves 7.2×-8.1× lower latency than TCP-based interconnects deployed in small- and medium-scale clusters. We address challenges of CXL memory sharing for MPI communication, including data object management over the dax representation [50], cache coherence, and atomic operations. Overall, cMPI outperforms TCP over standard Ethernet NIC and high-end SmartNIC by up to 49× and 72× in latency and bandwidth, respectively, for small messages.

CCS Concepts

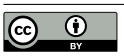
• **Computer systems organization** → **Interconnection architectures**; • **Networks** → *Network performance evaluation.*

Keywords

MPI, Compute Express Link (CXL), high-performance computing (HPC), shared memory, interconnection architectures

ACM Reference Format:
Xi Wang, Bin Ma, Jongryool Kim, Byungil Koh, Hoshik Kim, and Dong Li. 2025. cMPI: Using CXL Memory Sharing for MPI One-Sided and Two-Sided Inter-Node Communications. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3712285.3759816

*This work was done during the first author's internship at SK hynix.



SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759816

1 Introduction

Message Passing Interface (MPI) is the de-facto standard to program distributed memory in high-performance computing (HPC), and has been utilized to enable large-scale process-level parallelism for many scientific and engineering applications. Based upon data copy and message passing, MPI allows processes with separate address spaces to communicate and share data.

Given two MPI processes distributed on two nodes, two types of MPI communication can be employed, either one-sided or two-sided depending on process participation. The MPI communication traditionally employs high-performance interconnect (e.g., Ethernet), and relies on Remote Direct Memory Access (RDMA) or network protocols (e.g., RoCE [59]). The support of high-performance interconnect includes a complicated software stack, including application-level protocol implementation and system-level driver for device registration and interrupt handling. Using the interconnect technologies also requires the MPI library to use special APIs or programming methods (e.g., socket programming).

Recent work [41, 53, 71] reveals low memory utilization in individual nodes in data centers or (supercomputers), and demonstrates the potential of memory disaggregation for improving memory utilization and reducing production cost on supercomputers [21, 40, 71]. The emerging Compute Express Link (CXL) [61] makes memory disaggregation highly feasible. CXL is an interconnect technology designed for high-speed communication among processors, accelerators, and memory devices. CXL allows for memory expansion beyond traditional CPU memory channels by enabling a CPU to communicate with DRAM managed by its own controller over a PCIe channel, hence offering increased capacity and bandwidth without increasing the number of primary CPU memory channels. The CXL memory can be accessed using traditional CPU instructions (e.g., load and store), allowing the existing applications to use it with little changes. The CXL memory has been explored as a memory tier to build HPC systems with larger memory capacity for checkpointing or accommodating scientific applications with larger memory footprint [21, 22, 70, 71, 75].

In this paper, we study a new MPI communication scheme using CXL memory sharing. The CXL memory sharing is based upon the CXL pooled memory platform (a solution for memory disaggregation), shown in Figure 1. Using the CXL memory sharing, the MPI communication across nodes can be transformed into regular memory transactions and data copy within the CXL memory,

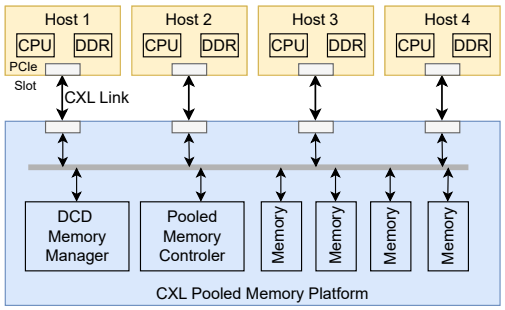


Figure 1: CXL memory sharing using a CXL pooled-memory platform. The CXL memory sharing is based upon the Multi-Headed Device (MHD) and Dynamic Capacity Device (DCD).

without going through the traditional network protocols. There are a couple of benefits of doing so. First, the communication latency can be shorter, compared with network-based approaches such as TCP over Ethernet and RoCEv2. For example, given a message size of 8 bytes, we show that using a *real* CXL pooled-memory platform (not based on hardware simulation or emulation [4, 39, 79]) to access the CXL memory takes only 790 *ns*, while using TCP over a standard Ethernet NIC and TCP over Mellanox (CX-6 Dx) to access data across nodes take 16 μs and 18 μs respectively, about an order of magnitude longer than using the CXL memory sharing (see Section 2 for details). Second, using the CXL memory sharing leads to a simpler software stack, compared to using the traditional network protocols. This is because the CXL protocol to perform CXL memory transactions for data (de)packaging and synchronization between CPU and memory device are done by hardware (not system software).

However, using the CXL memory sharing for MPI communication across nodes faces challenges. First, the representation of the CXL memory sharing in the host (node) creates challenges to manage data objects in the CXL memory. At the node, the CXL memory sharing is exposed as a Direct Access (`dax`) device by the CXL driver and `daxctl` [50]. The `dax` device bypasses the traditional page cache and provides memory-level access. However, the abstract of the `dax` device cannot provide flexibility to create arbitrary data objects and manage their life cycle (e.g., creation and deletion).

Second, the CXL hardware creates difficulty for the MPI library to implement one-sided and two-sided communications because of cache coherence and atomic operations. For high-performance communication, the MPI library often uses an internal buffer to hold messages and enable asynchronous operations. This buffer, when allocated on the CXL memory sharing, must be cache-coherent to ensure execution correctness. The CXL protocol defines hardware-based cache coherence. However, such a cache coherence mechanism is not scalable because of tracking cache line statuses at a large scale and potentially with large CXL memory. Furthermore, the MPI library often uses atomic operations to coordinate concurrent accesses to internal message queues. However, the CXL pooled memory often lacks a mechanism to enforce atomicity across nodes.

To address the above challenges and make feasible the CXL memory sharing for MPI one-sided and two-sided communications across nodes, we introduce *cMPI*. cMPI is an extension to the MPI library and does not require any change to the application and other system software. To address the first challenge, cMPI introduces a data-object management system without changing the `dax` abstract for the CXL memory sharing. By mapping the CXL memory into the virtual address space of MPI process and bookkeeping the information of data objects based on multi-level hashing, cMPI brings flexibility to create arbitrary data objects. To address the second challenge, cMPI introduces software-based cache coherence and enforces it on demand only between communicating processes. This solution is lightweight and does not require hardware changes. To tackle the problem of lacking atomicity for managing message queues, we propose a message queue structure for pair-wise MPI communication across nodes, and hence avoid the needs of atomic operations. Built upon the above design, cMPI further introduces techniques to support message management and synchronization.

Our contributions are summarized as follows.

- We explore the feasibility and benefits of using the CXL memory sharing for MPI one-sided and two-sided communications across nodes on real CXL memory. Our analysis shows that while high-end RDMA-supported devices deliver superior performance, CXL memory sharing shows significant performance potential compared to TCP-based interconnects commonly deployed in small- and medium-scale clusters.
- We develop a library, cMPI, by extending MPICH-4.2.3 [37]. We address the limitation of CXL memory sharing to support MPI communications.
- Our evaluation shows that using the CXL memory sharing for one-sided and two-sided inter-node communications, cMPI outperforms TCP over a standard Ethernet NIC by up to 49× in latency and 72× in bandwidth for all messages, and outperforms TCP over a high-end SmartNIC by up to 48× in latency and 3.7× in bandwidth for small messages.

2 Background and Motivation

We review the background and motivate our work in this section.

2.1 MPI

MPI provides a set of routines offering a portable way to express parallel communication patterns. For inter-node communication (i.e., the communication between processes on different hosts), MPI utilizes network stacks to transfer messages. For intra-node communication (i.e., the communication between processes on the same host), MPI utilizes shared memory to transfer messages. We use the term "MPI process" and "rank" interchangeably in the paper.

One-sided communication is built upon the scheme of Remote Memory Access (RMA). It utilizes an RMA window, which allows an MPI process to directly access data residing in another MPI processes. The process initiating data access is called the origin process, while the process whose data is accessed is called the target process. Unlike the two-sided communication, the one-sided communication requires only the origin process to execute RMA operations such as `MPI_Get` (fetching data from a target process)

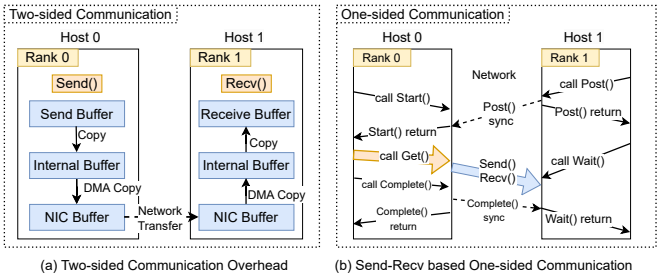


Figure 2: MPI communication across hosts (nodes).

Table 1: Memory access latency and bandwidth over various interconnect and protocols.

Arch Type	Latency	Bandwidth
Main Memory	100 *ns*	132.8 GB/s
TCP over Standard Ethernet NIC (abbreviated as: TCP over Ethernet)	16 μs	117.8 MB/s
TCP over Mellanox (CX-6 Dx)	18 μs	11.5 GB/s
RoCEv2 over Mellanox (CX-6 Dx)	1.6 μs	10.8 GB/s
RoCEv2 over Mellanox (CX-3)	sub-2 μs	7.0 GB/s
InfiniBand over Mellanox (CX-6)	sub-600 *ns*	25.0 GB/s
CXL Memory Sharing (with caching; no cache flushing)	790 *ns*	9.9 GB/s
CXL Memory Sharing (with cache flushing)	2.2 μs	9.5 GB/s

and `MPI_Put` (storing data into a target process). Because of the absence of the coordination between the origin and target processes, there is a need for synchronization between the two processes.

Two-sided communication explicitly involves a sender and a receiver during the communication. It utilizes intermediate buffers (MPI internal buffer and NIC buffer) to transfer data between two MPI processes since a sender process cannot directly access a receiver process's address space. This pattern requires both the sender and receiver participate in the communication by calling matching `MPI_Send` and `MPI_Recv` routines (or their non-blocking version).

Figure 2 shows the overall workflow of traditional MPI one-sided and two-sided inter-node communications.

2.2 CXL

The recent CXL specification (CXL 3.0) introduces CXL memory sharing. Based upon the Multi-Headed Device (MHD) and Dynamic Capacity Device (DCD), a host (node) can connect to the CXL pooled memory platform through a dedicated CXL port, ensuring bandwidth fairness and eliminating added latency of a switch hop, shown in Figure 1. We use a CXL pooled memory platform, Niagara 2.0 [15], for CXL memory sharing.

CXL memory sharing allows multiple processors to access a shared memory region, offering a unique opportunity for efficient communication between processors. First, data can be exchanged between processors without relying on additional communication protocols such as TCP/IP or RoCE. Second, the programming model is simplified, as each processor interacts with the shared memory using CPU instructions (e.g., load and store), mirroring the behavior of local memory accesses. Finally, the memory sharing through CXL can offer lower communication latency, compared to traditional network-based communication mechanisms.

Performance evaluation. To assess the benefits of using CXL memory sharing to improve communication performance across nodes, we compare the following 8 cases.

(1) The traditional main memory attached to the CPU through memory bus (or "main memory" for short);
(2) TCP over a standard Ethernet NIC (or "*TCP over Ethernet*" for short);
(3) TCP over Mellanox CX-6 Dx (a high-end SmartNIC);
(4) RDMA over Converged Ethernet v2 (i.e., RoCEv2) over Mellanox CX-6 Dx;
(5) RoCEv2 over Mellanox CX-3 (a low-end SmartNIC);
(6) InfiniBand over Mellanox CX-6;
(7) CXL memory sharing (with caching, but without using cache flushing for cache coherence);
(8) CXL memory sharing with cache flushing for cache coherence (see Section 3.5).

Section 4.1 has more hardware details. The case (1) studies intra-node communication, providing the best performance. The other cases study inter-node communication on two nodes.

To measure latency and bandwidth, we use Intel MLC [31] for the cases (1) and (7), iPerf [38] for (2) and (3), and Perftest [17] for (4). We choose those different tools (benchmarks) based on whether they are aligned with the specific system features (e.g., RDMA, or TCP over Ethernet). As we do not have Mellanox CX-3 and CX-6, the performances for the cases (5) and (6) are taken from the vendor's product report [52, 66]. For the case (8), since existing tools do not support testing memory access with cache flushing on a `dax` device (the system-level abstraction for CXL memory sharing), we develop a micro-benchmark. This benchmark uses `mmap` to obtain the virtual address space of the `dax` device. For latency tests, this benchmark performs `memset` with cache flushing. During the test, we vary the data size and measure the average latency over 1000 iterations. For bandwidth tests, this benchmark uses multiple threads, and each thread accesses 512 bytes with cache flushing on a `dax` device region. We measures aggregated bandwidth. See Table 1 for results.

Observation 1. The CXL memory sharing (with cache flushing) outperforms TCP over Ethernet and TCP over high-end SmartNIC.

The latency of CXL memory sharing is 7.2×-8.1× lower than that of TCP-based interconnect. Compared with TCP over Ethernet, the bandwidth of CXL memory sharing is 80× higher. The performance benefit of the CXL memory sharing is because it allows direct load/store operations to remote memory, eliminating the need to wrap data in network packets, traverse the full TCP/IP stack, and perform additional context switches.

Observation 2. The CXL memory sharing (with cache flushing) has longer latency but comparable bandwidth than RoCEv2, and performs worse than InfiniBand (on Mellanox CX-6).

Observation 3. Cache flushing in the CXL memory sharing for cache coherence increases latency by 2.8×.

In conclusion, while using high-end RDMA-supported devices for inter-node communication delivers the best performance, using the CXL memory sharing for inter-node communication shows

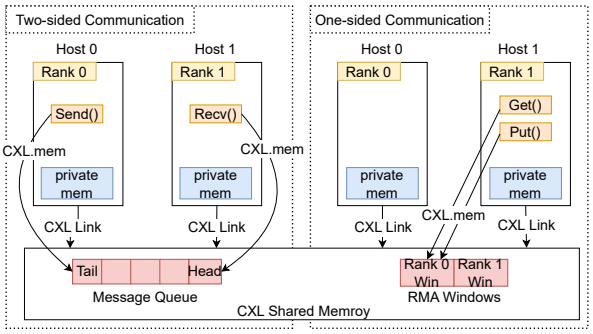


Figure 3: The overview of cMPI.

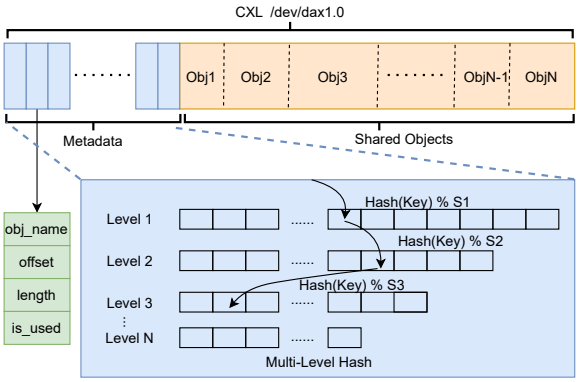


Figure 4: CXL SHM Arena.

great performance potential, compared with TCP-based interconnect which is commonly employed in small- or middle-scale clusters. Furthermore, if the CXL memory sharing can reduce the latency of cache flushing, the latency of using the CXL memory sharing for inter-node communication can be more competitive. We refer to the CXL memory sharing as *CXL SHM* in the remaining sections.

3 Design

The goals of cMPI are to enable MPI processes to freely create shareable memory objects for message passing and to significantly reduce communication overhead across nodes. cMPI does not modify the user-facing MPI routines (e.g., MPI initialization, message-passing, and synchronization interfaces). Figure 3 overviews cMPI. Multiple nodes connect to a CXL pooled memory platform through CXL physical links, sharing data stored in the CXL memory. We design a management layer called the *CXL SHM Arena* to efficiently allocate independent shared memory objects. The nodes involved in MPI communication exchange data by storing and loading directly from the CXL shared memory. For the two-sided communication, the message queues used in the traditional MPI library are created and maintained in the CXL shared memory, allowing nodes to transfer messages via enqueue and dequeue operations on the CXL shared memory. For the one-sided communication, Remote Memory Access (RMA) windows are allocated within the CXL shared memory, allowing processes to execute RMA operations such as `MPI_Get()` and `MPI_Put()` by directly accessing remote ranks' data stored in the CXL shared memory, eliminating network-based data transfers.

3.1 CXL SHM Arena

Management of data objects. For effective management of shared memory objects, we introduce the CXL SHM Arena, a CXL-based shared-memory management library integrated into MPI. While MPI traditionally uses POSIX shared memory (POSIX SHM) for intra-node IPC, the POSIX SHM API is unsuitable for managing CXL shared memory. POSIX SHM operates via memory-mapped files in a tmpfs filesystem (typically mounted at `/dev/shm`), where applications create POSIX SHM objects by naming files. Typical usage involves: (1) using `shm_open` to create/open a shared memory file in `/dev/shm`; (2) mapping this file into the caller's address space using `mmap`; (3) transferring data with `memcpy`; and (4) deleting data object with `shm_unlink` once processes finish their operations.

However, POSIX SHM cannot manage CXL shared memory because of the following two reasons. First, unlike POSIX SHM, which uses a tmpfs filesystem, CXL shared memory is provided as a device in Device Direct Access (`devdax`) mode, also known as a `dax` device. Thus, we cannot create arbitrary SHM objects by creating uniquely-named files in a directory like `/dev/shm`. Although `mmap` can map different regions of a file by specifying file offset and size, the application must explicitly manage offsets. Also, the `dax` device requires the boundary-alignment mapping (e.g., 2MB alignment), limiting the total number of SHM objects. Second, Linux cannot manage the lifecycle (creation, opening, closing, and deletion) of CXL SHM objects as it does with POSIX SHM objects.

To address the above limitation, we design the CXL SHM Arena. Figure 4 generally depicts our design. The CXL SHM Arena maps the entire `dax` device into the process's virtual address space and divides it into two regions: `metadata` and `shm_objects`. The `metadata` region is organized as a fixed-size array of slots, each containing information about a shared memory object (e.g., usage status, object name, offset within the `dax` file, and object size). The `shm_objects` region stores the actual shared data, allocated contiguously.

Multi-level hashing. For efficient SHM object indexing, we employ a hash-based structure rather than a plain array to construct the `meta-data` region. We design the hash structure based on two requirements. First, it should effectively resolve hash collisions while maintaining a fixed capacity to avoid dynamic resizing. Second, it must support parallel insertions and searches. To meet these requirements, we use an existing fixed-capacity multi-level hashing scheme [10], which is a representative of many multi-level hashing techniques [9, 10, 13, 86, 92, 93].

Figure 4 illustrates the design, where buckets are organized into multiple levels, each with a distinct prime bucket-count to evenly distribute entries and reduce collisions. These levels are flattened into a contiguous array within the metadata region. The key lookup involves computing a hash value and examining each level in turn, continuing until the key is found or all levels are checked. The lookup can be parallelized across the levels, enhancing efficiency.

CXL SHM Arena creates SHM objects identified by a unique object name, which serves as the hash key. If the object name is not found in the metadata hashing, a new object is allocated in the free

region of `shm_objects`, with corresponding metadata recorded (including object name, address offset relative to the Arena base address, and size). The application subsequently opens an existing SHM object by searching its name in the metadata hashing to retrieve the offset and size, and then calculates the object's virtual address by adding the offset to the base address of the Arena.

Famfs vs. CXL SHM Arena. Famfs [16] is a shared-memory file system framework recently created by MICRON. Famfs is different from our design from three perspectives. *First*, Famfs lives in the Linux kernel. In contrast, CXL SHM Arena is a user-space library without any kernel modification. *Second*, Famfs uses a client/master architecture, and restricts the creation of shared files exclusively to the master node, limiting its applicability to MPI where any node may need to create SHM objects. In contrast, CXL SHM Arena supports the creation of SHM objects by arbitrary nodes. *Third*, the SHM management APIs in CXL SHM Arena resemble those in the traditional POSIX SHM, which facilitates easy integration into the application, whereas Famfs uses quite different APIs.

3.2 One-Sided MPI Communication over CXL

MPI-3 introduces `MPI_Win_allocate_shared` to create POSIX SHM-based RMA windows for one-sided communication. This call allocates a shared-memory segment for each MPI process on the same host. Those memory segments are laid out contiguously across all ranks. This means that the first address in the memory segment of the rank i is consecutive with the last address in the memory segment of the rank $i-1$. This layout allows an MPI process to calculate the address of a RMA window using only local information.

CXL SHM-based RMA window. We extend `MPI_Win_allocate_shared` to create CXL SHM-based RMA window for ranks across nodes. During MPI initialization, each rank maps the CXL `dax` device into its own virtual address space using `mmap`. This gives the rank a base address, called `meta_addr`. To create a CXL SHM-based RMA window for a specific communicator, the root rank of the communicator creates a CXL SHM object in CXL SHM Arena. The object size is equal to the number of ranks multiplied by the size of the RMA window for each rank. The root rank then broadcasts the object name to all other ranks. Each rank uses this name to open the object (i.e., searching for the object in the CXL SHM Arena) and read the object metadata. Each rank gets the virtual address of its memory region by adding the metadata's offset to `meta_addr`. Since the memory segments are contiguous across ranks, each process can determine the virtual address of any other rank's window using its own SHM object address and the other rank's ID.

To facilitate one-sided communication over CXL, we bypass network communication by allowing a rank to treat the communicating peer as a one on the same host. This allows RMA operations to proceed fully through CXL SHM. For `MPI_Put()`, the origin rank directly writes data to the target rank's RMA window from the origin rank's address space. For `MPI_Get()`, the origin rank reads data directly from the target rank's RMA window.

3.3 Two-Sided MPI Communication over CXL

In the current MPI implementations (e.g., MPICH and OpenMPI), the shared memory is commonly utilized for point-to-point (pt2pt) communication between processes on the same host. Within the shared memory, lock-free message queues are created as a channel for message passing. Each process maintains a receive queue, into which other processes on the same host can enqueue messages. Those queues are implemented as linked-list structures, containing head and tail pointers pointing to individual message cells. Those message cells store the actual message data and are linked together to form a *message pool*. To send a message, the sending process copies the message header and data into a message cell and then enqueues the cell into the target process's receive queue. To receive a message, the receiving process polls its own receive queue and retrieves incoming messages by dequeuing those message cells.

CXL SHM-based message queue. We extend the SHM-based pt2pt communication by enabling processes across different hosts (not just processes on the same host) to send messages via CXL SHM. Specifically, instead of creating a POSIX SHM-based message queue only for processes on the same host, we create a CXL SHM-based message queue accessible to all processes across the hosts. Similar to the SHM-based one-sided window creation, these queues are laid out contiguously within the CXL SHM, forming a message queue region. This contiguous arrangement allows any process to locate other processes' message queues based on the base address of the message queue region and the index of each message queue.

The current MPI implementations often create only one receive queue per host, employing either a Multi-Producers Single-Consumer (MPSC) or a Multi-Producers Multi-Consumer (MPMC) model. When using the sender-side message pool, an MPSC receive queue is employed, wherein multiple sender processes simultaneously dequeue available cells from their individual message pools and enqueue messages into the receiver's queue (a single queue). In this scenario, multiple processes concurrently enqueue messages, but only a single receiver process dequeues messages from the receiver's queue. When using the receiver-side message pool, an MPMC queue is employed, wherein multiple sender processes simultaneously dequeue free message-cells from the receiver's message pool, fill the message cells with the message, and concurrently enqueue filled cells into the receiver's receive queue. In this scenario, multiple processes can simultaneously perform both enqueue and dequeue operations on the same queue.

Both the MPSC queue and MPMC queue are prone to race conditions when multiple processes attempt to enqueue or dequeue simultaneously. Although some MPI implementations (e.g., MPICH) employ lock-free queues using atomic operations to address the race condition, atomic operations are not effective in CXL SHM across multiple hosts (depicted in Section 3.5).

To implement lock-free queues without relying on atomic operations, we design a Single-Producer Single-Consumer (SPSC) message ring queue. Specifically, we create a matrix of message ring queues for pt2pt communication. Unlike the traditional design where each process maintains a single receive queue for messages from all other processes, we establish separate ring queues for each pair of sender and receiver processes. These message queues follow the SPSC model and eliminate the need for atomic operations, as enqueue and dequeue operations are handled by a single producer and a single consumer, respectively. The collection of those message queues forms a message queue matrix, indexed by receiver rank ID and sender rank ID. To send a message, the sender locates the appropriate message queue and enqueues the message. To receive

messages, the receiver polls ring queues associated with its ID and dequeues messages from each queue.

3.4 Synchronization

We optimize the most common synchronization mechanisms for CXL SHM, but keep other synchronization mechanisms unchanged.

Synchronization in the one-sided communication is used to permit processes to access a target window. There are two common synchronization methods for the one-sided communication: Post-Start-Complete-Wait (PSCW) and Lock-Unlock. We leverage CXL SHM to reduce the one-sided synchronization overhead. PSCW enables a selected group of processes to participate in synchronization, using an access epoch and an exposure epoch. The target process invokes `Post` and `Wait` to define an exposure epoch, where it gives the origin process the availability of the RMA window. The origin processes invoke `Start` and `Complete` to define an access epoch, where RMA operations can be executed. The MPI implementation typically sends messages over the network to notify the origin and target processes of the epoch-status update. To reduce the PSCW overhead, we allocate a shared synchronization array during RMA window creation in CXL. Similar to the RMA window, the synchronization array is allocated contiguously across all processes. Each element in the array contains each process's shared Post-Wait flag and shared Start-Complete flag. The origin process's Post-Wait flag is directly set by the target process and is reset by the origin. Conversely, the target process's Start-Complete flag is directly set by the origin process and is reset by the target. This method eliminates the synchronization message transfer over network.

Similarly, Lock-Unlock synchronization can be optimized by placing the window lock in CXL SHM, thereby eliminating the network round trip for lock/unlock request-acknowledge.

Synchronization in the two-sided communication includes `MPI_Wait` and `MPI_Test`. The `MPI_Wait` call blocks until a given request completes, whereas `MPI_Test` returns immediately and indicates whether the request has finished. Both calls monitor the status of outstanding MPI send/receive requests and advance local progress for any pending message passing. Since the message paths have been optimized, we do not use CXL SHM for both calls due to little performance benefit.

Initialization barrier is used for the processes synchronization in the same node during the MPI initialization phase (e.g., socket creation, broadcast table creation, and shared message queue creation). The current MPI library implements the initialization barrier using a sense-reversing algorithm where each MPI process maintains a local `sense` variable. As the MPI processes enter the barrier, they increment a shared counter; the last arriving process resets this counter and toggles a shared `wait` flag to indicate the barrier completion. The remaining processes spin-wait until the `wait` flag matches their local `sense`, after which all processes flip their `sense` in preparation for the next barrier. cMPI refactors the initialization barrier to support synchronization across nodes for the creation of shared message queue while avoiding atomic operations (i.e., atomic increment). Specifically, each process maintains a local barrier sequence number, and all processes share a barrier array to store those numbers. When entering the barrier, a process increments its local sequence number and updates the barrier array entry corresponding to its rank. The process then checks the sequence numbers of other ranks in the barrier array and spin-waits until all other processes' sequence numbers are equal to or greater than its own local sequence number.

3.5 Memory Coherence

CXL SHM allows multiple hosts to access the same memory region. However, our CXL SHM lacks the cache coherence support, which causes the updates made by one host to be invisible to others. To maintain data coherence, we must rely on either non-cacheable memory sharing or a software-based cache coherence mechanism. While CXL 3.0 introduces support for Back-Invalidate (BI) to enforce hardware coherence across hosts, the anticipated growth in CXL memory size makes impractical a precise snoop filter that tracks each cache line [32]. In contrast, the software-based cache coherence has been shown to succeed to optimize performance in large shared address space [50, 64]. We discuss non-cacheable memory sharing and software-based cache coherence as follows.

Non-cacheable memory sharing. Modern x86 processors feature Memory Type Range Registers (MTRRs), which allow the CPU to access specific memory address ranges with different cacheability attributes, such as write-back, write-combining, write-through, write-protect, and uncachable. We configure `/proc/mtrr` to set the physical memory region of the CXL `dax` device as uncachable, thereby enabling CXL SHM accesses to bypass CPU caching. However, as demonstrated in Section 4.5, uncachable access to CXL SHM results in significantly higher latency (exceeding 4,000 μs) when the data size exceeds 2 KB.

Software-based cache coherence. We use cache flush and memory fence. Cache flush instructions, such as `clwb`, `clflush`, and `clflushopt`, invalidate the cache line for specific addresses from each level of the cache hierarchy in the coherence domain; if that cache line contains modified data, that data is written back to memory [26]. Memory fence instructions, such as store fence (`sfence`) and load fence (`lfence`), enforce ordering constraints on memory operations, ensuring that all operations issued prior to the fence are completed before those issued after the fence. We perform cache flush and memory fence when accessing the CXL SHM Arena, CXL SHM-based RMA windows, and CXL SHM-based message queues. Specifically, after every write, we perform a cache flush followed by a memory fence to guarantee that data is written back to memory. Conversely, before every read, we perform a memory fence followed by a cache flush to ensure that stale data, potentially held in caches or due to prefetching, is invalidated.

Given that uncachable access to CXL SHM leads to drastically higher latency for the large data size, we choose the software-based cache coherence. Additionally, we utilize non-temporal stores and loads when operating on integer data, such as the synchronization flags and the head/tail pointers of the message queue, to ensure that data is accessed directly from CXL SHM.

3.6 Discussion

cMPI currently focuses on one-sided and two-sided communications across nodes. There are collective communications such as `Allgather`, `Allreduce`, and `ReduceScatter` that can benefit from CXL SHM. Within an MPI library, these collective communications

Table 2: CXL SHM Arena APIs for cMPI

API Function	Arguments	Description
`cxl_shm_init`	–	Initialize and mmap the CXL SHM Arena
`cxl_shm_finalize`	–	Clean up SHM arena and device memory
`cxl_shm_create`	`name, size, *obj_handle`	Create a new object with the specified size
`cxl_shm_open`	`name, *obj_handle`	Open an existing object by name
`cxl_shm_destroy`	`*obj_handle`	Delete an object from the CXL SHM Arena
`cxl_shm_close`	`*obj_handle`	Close and release an object handle

are often implemented using point-to-point communications based on certain algorithms (e.g., recursive doubling [5] and Bruck's algorithm [20]), hence the collective communications can directly benefit from cMPI. Nevertheless, there are multiple challenges of using cMPI for the collective communicators, such as communicator handling and integrating intra/inter-node point-to-point communications. We leave collective communications as future work.

cMPI exhibits performance benefits in inter-node communication, mostly for small message sizes (16 KB or smaller, shown in Section 4.2). When communicating using CXL SHM, the CPU must handle message transfers primarily using the `mov` instruction (or other load/store-like instructions) to copy data between the local buffer in main memory and the message queue or RMA window in CXL SHM. For small messages, the MPI communication bandwidth using CXL SHM with 8 processes even outperforms that of TCP over Mellanox (CX-6 Dx) with 32 processes by up to 1.3×, due to the lower latency of CXL SHM. However, for large messages, excessive CPU-based memory accesses lead to contention in the memory hierarchy, resulting in degraded communication performance. In contrast, the network-based approaches using high-end SmartNICs offer better scalability for large message transfers, as the actual data transmission occurs over the network with less CPU involvement.

Using CXL SHM to accommodate RMA window for one-sided communication or the message buffer for two-sided communication can lead to performance loss, compared to using the node-local memory. This is because of the performance difference between the local memory and CXL SHM (e.g., 100 *ns* vs. 2.2 μs shown in Table 1). However, such a performance difference is outweighed by the performance benefit of using the CXL SHM when the message size is small. The performance difference is more pronounced when the message size is large.

3.7 More Implementation Details

We implement cMPI by extending MPICH-4.2.3 [24]. In CXL SHM Arena, to enable cMPI to create millions of distinct CXL SHM objects, we configure a 10-level multi-level hash table and set the maximum number of slots at the level one to 200,000. Since the number of slots is constrained to be a prime number, the actual slot counts across the levels 1-10 range from 199,999 to 199,873. This results in a total of 1,999,260 slots across all levels.

To integrate CXL SHM Arena seamlessly with MPI, we design the APIs of CXL SHM Arena to resemble the POSIX SHM API, as shown in Table 2. This design ensures that generating SHM objects only requires API-level changes, without significantly modifying the logic flow of creating and broadcasting SHM objects in MPI.

To support memory consistency, we align each CXL SHM object to the cacheline size. This alignment facilitates efficient cache flushing and supports non-temporal memory accesses.

4 Evaluation

4.1 Evaluation Methodology

Evaluation platform. We evaluate cMPI on two dual-socket servers. Each server is equipped with two Intel Xeon Gold 6530 processors (@2.10 GHz, 32 cores per socket). Each socket has 8×32 GB DDR5-5600 DIMM. We use MPICH-4.2.3 (CH4:OFI) and cMPI on Linux kernel 6.6.0-rc6, which is integrated with the CXL driver and DCD patches. To study the impact of communication across nodes and avoid the NUMA effect within each node, we disable NUMA balancing by setting `/proc/sys/kernel/numa_balancing` to 0.

Real CXL Platform. We use Niagara 2.0 [15], an FPGA-based CXL pooled memory platform. This platform can connect up to 4 nodes without a CXL switch. Each server connects to this platform via an MCIO x8 cable using PCIe 4.0 (16 GT/s per lane). The platform uses four DDR4-2400 DIMM channels, with a total capacity of 1TB and a bandwidth of 19.2 GB/s per channel.

Interconnect. Each node uses a ConnectX-6 Dx SmartNIC (Mellanox Technologies MT2892 Family), supporting Ethernet and RoCEv2. Two nodes are also connected via another standard Ethernet NIC with limited performance (see "TCP over Ethernet" in Table 1).

Baselines. We compare our approach (CXL SHM-based communication) with TCP over Ethernet and TCP over Mellanox (CX-6 Dx). We do not compare with RoCEv2 over Mellanox (CX-6 Dx and CX3) and InfiniBand (Melanox CX-6), as CXL SHM is not competitive with them in terms of latency and bandwidth (see Table 1).

Simulator for scalability study. We use SimGrid [11], an event-based simulator that can study MPI applications with user-specified interconnect latency and bandwidth. Because our CXL platform is constrained to connecting only four hosts, simulation is necessary to study scalability beyond this hardware limitation.

4.2 Microbenchmarks

We use the OSU Micro-Benchmark suite (OMB) [39] to compare performance (i.e., latency and bandwidth) across two nodes using CXL SHM, TCP over Ethernet, and TCP over Mellanox (CX-6 Dx). To evaluate the performance of one-sided communication under multiple processes, we extend the OMB's benchmark for one-sided communication, which originally only supports two processes, to support any number of MPI origin and target processes. To evaluate the performance of two-sided communication, we change the size per message cell to 64KB to improve performance (see Section 4.3). We evaluate the message size from 1B to 8MB, representing small, medium, and large message sizes [39]. For one-sided communication, given n MPI processes, half of them is used as the origin processes, and the other half is used as the target processes.

Bandwidth of one-sided communication. See Figure 5.

TCP over Ethernet vs. CXL SHM. CXL SHM outperforms TCP over Ethernet by up to 71.6×. With CXL SHM, the bandwidth saturates at around 8,600 MB/s with 16 processes when the message size reaches 16 KB. With TCP over Ethernet, adding more processes can saturate the interconnect bandwidth using smaller message sizes, but the bandwidth curves converge near 120 MB/s.

TCP over Mellanox (CX-6 Dx) vs. CXL SHM. For message sizes of 16 KB or smaller, CXL SHM outperforms TCP over Mellanox (CX-6 Dx) by up to 3.7×. CXL SHM achieves higher bandwidth even using fewer processes compared to TCP over Mellanox (CX-6

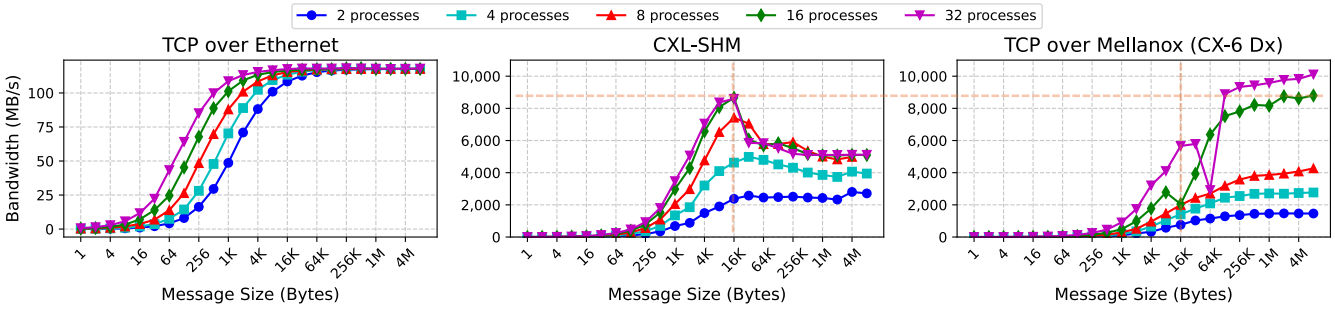


Figure 5: Bandwidth of one-sided MPI communication.

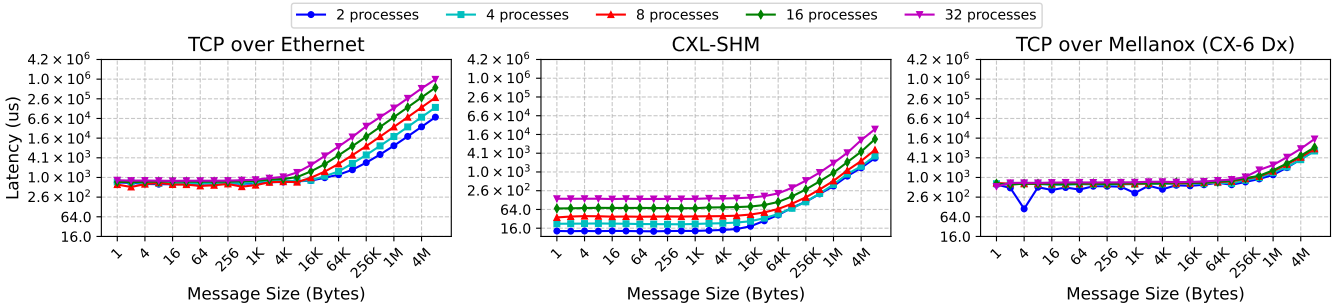


Figure 6: Latency of one-sided MPI communication.

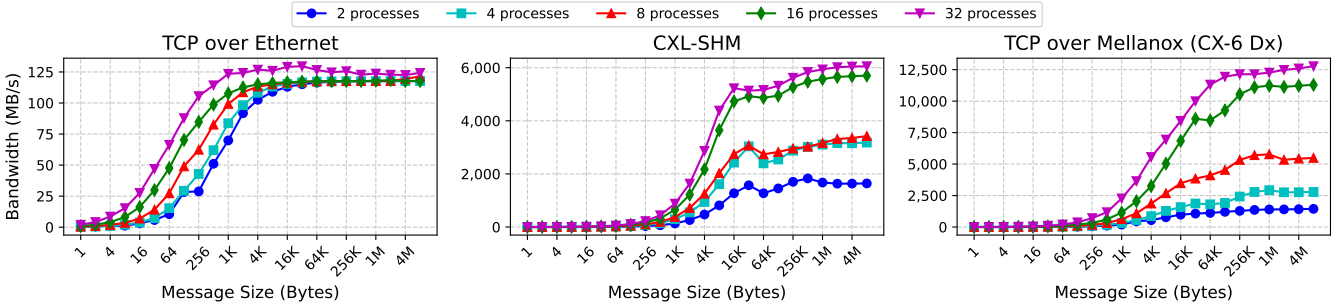


Figure 7: Bandwidth of two-sided MPI communication.

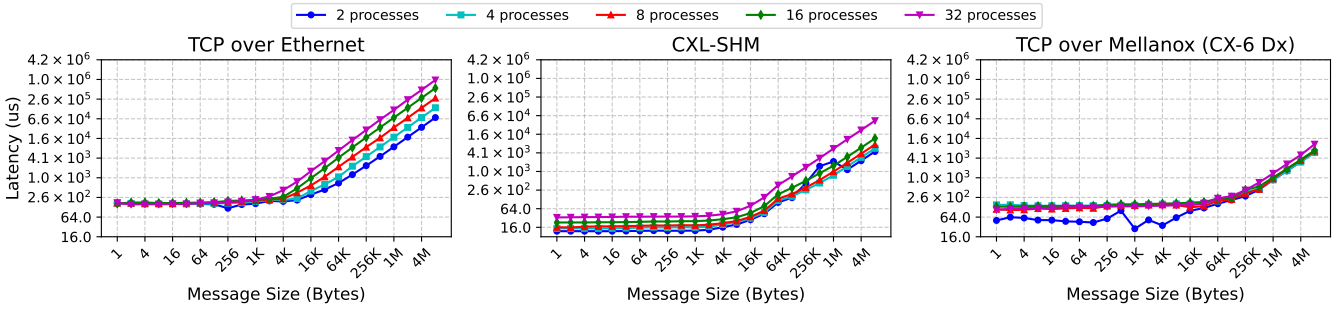


Figure 8: Latency of two-sided MPI communication.

Dx). For example, CXL SHM achieves 7,420 MB/s at 16 KB using 8 processes, whereas TCP over Mellanox only reaches 5,660 MB/s at 16 KB even with 32 processes. The high bandwidth of CXL SHM for small messages stems from the low latency of CPU-based accesses to small data. Beyond 16 KB, however, the CXL SHM bandwidth declines because concurrent accesses to large data from multiple processes induce contention on the memory hierarchy.

For messages larger than 16 KB, TCP over Mellanox (CX-6 Dx) surpasses CXL SHM when using 16 or 32 processes. Using 32 processes, TCP over Mellanox (CX-6 Dx) allows the bandwidth to climb to 10,150 MB/s as the message size grows, which is near the theoretical peak bandwidth (100 Gb/s). Although TCP over Mellanox (CX-6 Dx) initially requires the CPU to package the message, the subsequent network transfer proceeds without further CPU involvement. In contrast, CXL SHM relies on CPU `mov` instruction for the entire one-sided message passing, continuously involving the CPU. Consequently, TCP over Mellanox (CX-6 Dx) has the advantage for larger data transfers, because of less involvement of the CPU.

Latency of one-sided communication. See Figure 6.

TCP over Ethernet vs. CXL SHM. CXL SHM outperform TCP over Ethernet by up to 49.4×. With CXL SHM, the latency remains consistently low (around 12 μs) for message sizes from 1 B to 16 KB, and increases modestly as more processes are used. Conversely, TCP over Ethernet experiences latency in the hundreds of microseconds (approximately 630 μs) even for small messages. Increasing the process count from 2 to 32 shifts the curve only slightly because more processes make the link busier. Ethernet's inherent limitations (e.g., multiple rounds of data copy and deep software stack) cap overall performance.

TCP over Mellanox (CX-6 Dx) vs. CXL SHM. For message sizes of 256 KB or smaller, CXL SHM achieves up to 48.3× lower latency than TCP over Mellanox (CX-6 Dx). Specifically, the latency for TCP over Mellanox (CX-6 Dx) hovers around 620 μs, while CXL SHM has a minimal latency of approximately 12 μs.

For messages larger than 256 KB, TCP over Mellanox (CX-6 Dx) outperforms CXL SHM because its latency increases more slowly for larger data. This advantage stems from offloaded network transfers that do not require continuous CPU intervention. In contrast, the latency with CXL SHM increases proportionally as the data size grows when the data size is larger than 16 KB. This growth is mainly due to contention from concurrent accesses to large data from multiple processes. This observation is aligned with the observations from the one-sided bandwidth evaluation.

Bandwidth of two-sided communication. See Figure 7.

TCP over Ethernet vs. CXL SHM. CXL SHM outperforms TCP over Ethernet by up to 48.2×. With CXL-SHM, the bandwidth saturates at around 6,050 MB/s. With TCP over Ethernet, the bandwidth exhibits a trend similar to its one-sided counterpart and saturates at about 120 MB/s due to the throughput ceiling of Ethernet interface.

TCP over Mellanox (CX-6 Dx) vs. CXL SHM. TCP over Mellanox (CX-6 Dx) outperforms CXL SHM by up to 2.1× when using more than 4 processes. With CXL SHM, the peak bandwidth for the two-sided communication (about 6,050 MB/s) is approximately 30% lower than its one-sided counterpart (8,600 MB/s). This occurs because each two-sided operation involves two message transfers: one from the sender's local buffer to the message queue in CXL SHM, and another from that message queue to the receiver's local buffer. Consequently, the memory access contention doubles at the DIMM and memory bus level in the CXL platform, which leads to lower bandwidth. With TCP over Mellanox, the scalability is stronger than with CXL SHM, as adding more processes continues to boost bandwidth. The bandwidth reaches beyond 12,500 MB/s at 32 processes for large message sizes.

Latency of two-sided communication. See Figure 8.

TCP over Ethernet vs. CXL SHM. CXL SHM outperforms TCP over Ethernet by up to 13.7×. With TCP over Ethernet, the two-sided communication achieves lower latency (approximately 160 μs) than its one-sided counterpart (630 μs) due to extra synchronization round-trips in the one-sided communication. Nevertheless, the overall trend is similar to that of the one-sided communication, with latency constrained by Ethernet. In contrast, CXL SHM maintains a latency of around 12 μs, highlighting its advantage for latency-sensitive communications.

TCP over Mellanox (CX-6 Dx) vs. CXL SHM. For message sizes smaller than 64 KB, CXL SHM achieves up to 9.6× lower latency, compared to TCP over Mellanox (CX-6 Dx). Once the message size reaches 64 KB or larger, the latency with CXL SHM increases in proportion to the message size. This is due to the limited cell size (64 KB) in the message queue, which cannot accommodate a 64 KB message plus its header. As a result, a larger message is split into smaller chunks and sent sequentially through the message queue, causing a linear increase in latency.

With TCP over Mellanox (CX-6 Dx), for messages smaller than 256 KB, the two-sided communication also exhibits lower latency (around 55 μs), compared to its one-sided counterpart (around 620 μs). Beyond 256 KB, however, the latency scales linearly with the message size, reflecting bandwidth saturation and aligning with the results for two-sided communication.

4.3 Message Size Threshold for CXL SHM-based Two-Sided Communication

As described in Section 3.3, the two-sided communication uses a shared message queue in which head and tail pointers point to "message cells" that store both message headers and message payloads. Specifically, a sender copies data from its send buffer into one of those cells, while a receiver copies data out of the cell and into its receive buffer. MPI classifies messages whose size is no larger than a cell's capacity as short messages; a longer message is split into multiple cell-sized chunks sent sequentially. Consequently, the size of message cell impacts parallelism in data transfers.

Figure 9 shows that with the MPI default cell size (16 KB), the highest bandwidth achieved is 3600 MB/s (when the message size is 8 KB and the number of processes is 32). Beyond 16 KB, the message must be split, causing a drop in bandwidth. Increasing the cell size to 32 KB raises bandwidth to 3900 MB/s (when the message size is 16 KB and the number of processes is 32).

Setting the message cell size to 64 KB improves peak bandwidth to approximately 6000 MB/s for messages larger than 64 KB with 32 processes. However, increasing the cell size beyond 64 KB does not continue to improve bandwidth, indicating an upper limit to the benefits of using a larger message cell.

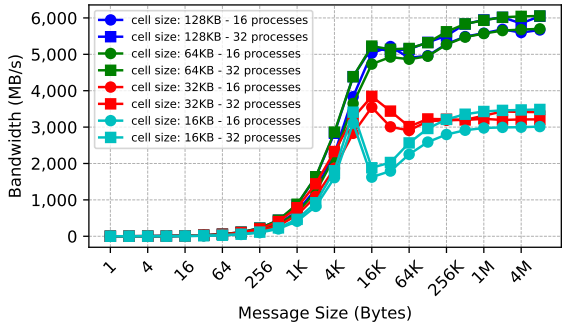


Figure 9: Bandwidth of two-sided communication with various sizes of message cells.

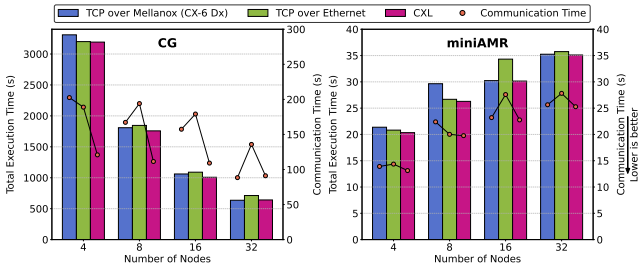


Figure 10: Strong scaling with CG and miniAMR.

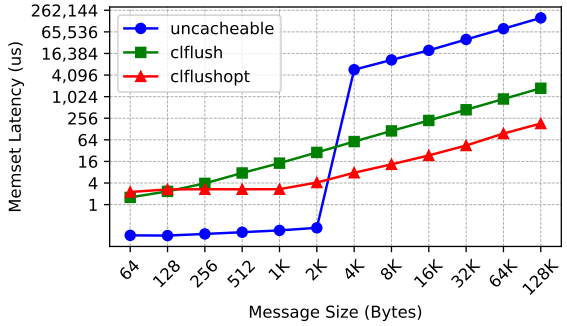


Figure 11: Memset latency with uncacheable memory and cacheable memory (plus cache flushing overhead).

4.4 Scalability Study

We use SimGrid simulator to perform strong scaling evaluation, because our CXL platform does not support large-scale evaluation. The latency and bandwidth of inter-node and intra-node are configured based on our evaluation results in Section 4.2. We use eight MPI processes per node during the evaluation.

We use CG (a conjugate gradient solver) from NAS Parallel Benchmark (NPB) Suite [7] 3.4. For CG, we use Class D as input. We also use miniAMR [69] (a proxy application for adaptive mesh refinement) with the input block size 4 in x, y, z directions. We choose CG and miniAMR for evaluation, because their communications are dominated by MPI two-sided communications [62, 63].

CG results. CXL SHM outperforms other two network-based approaches because of its shorter communication time. Specifically, the communication time using CXL SHM is shorter than that of TCP over Mellanox (CX-6 Dx) and TCP over Ethernet by 25.3% and 37.6%, respectively. This reduction is attributed to the lower inter-node latency provided by CXL SHM. As the number of nodes increases, the communication gap between CXL SHM and TCP over Mellanox (CX-6 Dx) narrows, since the higher bandwidth of Mellanox (CX-6 Dx) benefits message transfers at larger scales. However, because communication accounts for a small portion (less than 15%) of total execution time, the overall performance difference is small.

miniAMR results. CXL SHM achieves an average speedup of 4% and 4.7% in total execution time compared to TCP over Mellanox (CX-6 Dx) and TCP over Ethernet, respectively. Unlike CG, miniAMR has a higher communication portion (more than 62%) of total execution time. As the number of nodes increases, the communication time grows while the computation time remains steady, since miniAMR assigns each process a fixed number of grid blocks and each process performs a constant computational workload. The reduced communication time with CXL SHM—4.9% lower than Mellanox and 9.8% lower than Ethernet—directly contributes to the performance gain. Additionally, TCP over Ethernet outperforms TCP over Mellanox (CX-6 Dx) at 8 nodes or fewer due to its slightly lower latency (16 μs vs. 18 μs), but performs worse beyond 8 nodes because of its significantly lower bandwidth (117.8 MB/s vs. 11.5 GB/s). At small scales, communication time is latency-dominated, while at larger scales, bandwidth becomes the limiting factor.

4.5 Cache Coherence

To maintain CXL memory coherence, we can either use non-cacheable memory sharing or a software-based cache coherence mechanism, discussed in Section 3.5. To compare the two approaches, we use our micro-benchmark (see Section 2) to measure latency across data sizes ranging from 64 B to 128 KB. For the software-based cache coherence, the `memset` latency includes caching flushing overhead.

For the uncacheable approach, we use MTRR to mark the physical memory region of the CXL `dax` device as uncacheable before running the benchmark, causing all subsequent memory accesses to bypass the CPU cache. We then execute `memset` without performing cache flushes. For the `clflush` approach, using MTRR the CXL SHM region is configured to be write-back cacheable. We execute `memset` followed by `clflush` and an `sfence` to ensure proper memory ordering. We use the same approach when evaluating `clflushopt`. Compared to `clflush`, `clflushopt` can flush multiple cache lines in parallel.

Figure 11 shows the results. We observe that when the data size is larger than 2 KB, the uncacheable accesses experience 256× higher latency, compared to using `clflush` and `clflushopt`. Moreover, `clflushopt` outperforms `clflush` by up to 4× when the data size is larger than 64 B. In particular, when the data size ranges from 1 B to 64 B, both `clflush` and `clflushopt` exhibit similar latencies (around 2 μs to 3 μs), since the data size does not exceed a single cache line, requiring only one cache flush. Beyond 64 B, multiple cache flushes are necessary, and `clflushopt` performs better because of its efficient parallel flushing.

When the data size is larger than 2 KB, uncacheable accesses lead to sudden latency spikes (exceeding 4096 μs). This behavior arises from the Maximum Payload Size (MPS) limitation in PCIe, which is typically between 128 B and 4096 B. Once the data size exceeds the MPS, the PCIe link splits it into multiple Transaction Layer Packets, and transfers them sequentially, increasing latency.

5 Related Work

Intra-node MPI communication over shared memory. Optimizing intra-node MPI communication within shared memory has been extensively studied [2, 14, 23, 28, 29, 42, 73, 78]. For example, Adam et al. [2] reveal significant bandwidth differences between cores and sockets in a node and demonstrate that current MPI implementations may not fully utilize the intra-node bandwidth potential. Walia et al. [73] leverage OpenMP for accelerating MPI_Allreduce in deep learning workloads. Cho et al. [14] improve MPI bandwidth performance through on-package memory and huge page support. White and Kale [78] optimize point-to-point communication in AMPI [28] by their locality-aware design in shared memory. Unlike previous works focusing on intra-node communication, our study optimizes inter-node MPI communication over CXL SHM.

Evaluation of CXL. Recent CXL memory evaluations provide insights into this emerging memory technology [44, 64, 64, 65, 75, 79, 90]. For example, Sun et al. [64] demonstrate important differences between real and emulated CXL memory, necessitating reconsideration of previous assumptions on the CXL hardware. Tang et al. [65] evaluate CXL performance and design an Abstract Cost Model to estimate the cost-benefit of using CXL memory in datacenters. Wang et al. [75] provide a comprehensive evaluation of how CXL interacts with GPU in LLM training and inference.

CXL shared memory management. Famfs [6, 16] from Micron is an open source file system for scale-out disaggregated shared memory across multiple nodes. We discuss its difference from cMPI in Section 3.1. Zhang et al. [91] develop a failure-resilient CXL SHM system using reference counting to tolerate partial client failures.

CXL for MPI. OMB-CXL [67] introduces a micro-benchmark suite for evaluating MPI communications over CXL SHM based on emulation. Ahn et al. [3] present the first MPI over CXL SHM implementation beyond benchmarking, utilizing CXL SHM to accelerate MPI_Allgather across nodes. However, this work addresses only a single collective operation and is based on emulation. In contrast, our work is the first to optimize MPI point-to-point communication (both one-sided and two-sided) on real CXL Pooled Memory Platforms, with novel approaches to build DAX abstraction, MPI synchronization, and memory coherence.

Applications of CXL memory. CXL has been explored by various application domains. For databases, Ahn et al. [4] leverage CXL memory expansion to enhance in-memory database systems. For AI workloads, Xie et al. [84] introduce SmartQuant, a CXL-based AI model store supporting runtime weight quantization to reduce inference latency. For approximate nearest neighbor search (ANNS), Jang et al. [33] propose CXL-ANNS, which utilizes CXL-based memory disaggregation and parallel search techniques to efficiently handle large datasets.

Tiered memory systems. Apart from serving as a shared memory pool, CXL can also be used to build a tiered memory system for a host. With a tiered memory system, various memory components [1, 8, 12, 18, 25, 60, 68, 75, 76, 83] exhibit differences in latency, bandwidth, capacity, and cost. A range of solutions [19, 27, 30, 34–36, 43, 45–49, 51, 54–58, 72, 74, 75, 77, 80–82, 85, 87–89] have been proposed to leverage tiered memory systems to improve application performance.

6 Conclusions

In this paper, we explore how to use CXL memory sharing for MPI point-to-point communication across nodes. Our study reveals the performance potential of using CXL memory sharing, compared to certain interconnects. We also study how to customize the MPI library to use CXL memory sharing. Our work opens a door for using CXL techniques in HPC.

Acknowledgments

This work was partially supported by U.S. National Science Foundation (2104116, 2316202 and 2348350). We would like to thank the anonymous reviewers for their feedback on the paper.

References

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 971–985.

[2] Julien Adam, Jean-Baptiste Besnard, Adrien Roussel, Julien Jaeger, Patrick Carribault, and Marc Pérache. 2025. To Share or Not to Share: A Case for MPI in Shared-Memory. In *Recent Advances in the Message Passing Interface*, Claudia Blaas-Schenner, Christoph Niethammer, and Tobias Haas (Eds.).

[3] Hooyoung Ahn, Seonyoung Kim, Yoomi Park, Woojong Han, Shinyoung Ahn, Tu Tran, Bharath Ramesh, Hari Subramoni, and Dhabaleswar K. Panda. 2024. MPI Allgather Utilizing CXL Shared Memory Pool in Multi-Node Computing Systems. In *2024 IEEE International Conference on Big Data (BigData)*.

[4] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *International Workshop on Data Management on New Hardware*.

[5] Omer Arap, Martin Swany, Geoffrey Brown, and Bryce Himebaugh. 2015. Adaptive Recursive Doubling Algorithm for Collective Communication. In *International Parallel and Distributed Processing Symposium Workshop*.

[6] Ramesh Aravind and Groves John. 2024. Study of CXL Memory Sharing with FamFS and its Use cases. In *International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*.

[7] 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, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. 1991. The Nas Parallel Benchmarks. *The International Journal of Supercomputing Applications* (1991).

[8] Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. 2022. Reconsidering os memory optimizations in the presence of disaggregated memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management*. 1–14.

[9] Andrei Broder and Michael Mitzenmacher. 2001. Using multiple hash functions to improve IP lookups. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, Vol. 3. IEEE, 1454–1463.

[10] Andrei Z Broder and Anna R Karlin. 1990. Multilevel adaptive hashing. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. 43–53.

[11] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. 2025. Lowering Entry Barriers to Developing Custom Simulators of Distributed Applications and Platforms with SimGrid. *Parallel Comput.* 123 (2025), 103–125.

[12] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An Extensible Optimizer for Portable Data Placement on GPU. In *IEEE/ACM International Symposium on Microarchitecture*.

[13] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 799–812.

[14] Joong-Yeon Cho, Hyun-Wook Jin, and Dukyun Nam. 2017. Enhanced memory management for scalable MPI intra-node communication on many-core processor. In *Proceedings of the 24th European MPI Users' Group Meeting* (Chicago, Illinois) *(EuroMPI '17)*. Association for Computing Machinery, New York, NY, USA, Article 10, 9 pages. doi:10.1145/3127024.3127035

[15] Jungmin Choi. [n. d.]. Exploring CXL Memory Disaggregation: Use Cases and System Benefits. https://memverge.com/wp-content/uploads/Memory-Fabric-Forum-at-OCP-Global-Summit-2024-%E2%80%93-SK-hynix_Exploring-CXL-Memory-Disaggregation.pdf

[16] CXL Micron Research Kit. 2024. Famfs Shared Memory Filesystem Framework. https://github.com/cxl-micron-reskit/famfs. Accessed: 2025-04-05.

[17] Open Fabrics Enterprise Distribution. [n. d.]. perftest. https://github.com/linux-rdma/perftest.

[18] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[19] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeffrey R. Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. *Proceedings of the Eleventh European Conference on Computer Systems* (2016). https://api.semanticscholar.org/CorpusID:8681081

[20] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. 2022. Optimizing the Bruck Algorithm for Non-Uniform All-to-All Communication. In *International Symposium on High-Performance Parallel and Distributed Computing*.

[21] Yehonatan Fridman, Suprasad Mutalik Desai, Navneet Singh, Thomas Willhalm, and Gal Oren. 2023. CXL Memory as Persistent Memory for Disaggregated HPC: A Practical Approach. In *Proceedings of the SC Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*.

[22] Ellis Giles and Peter Varman. 2024. Towards Continuous Checkpointing for HPC Systems Using CXL. In *IEEE 31st International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*.

[23] Richard L. Graham and Galen Shipman. 2008. MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag.

[24] William Gropp. 2002. MPICH2: A New Start for MPI Implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, Berlin, Heidelberg, 7.

[25] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[26] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: system programming guide, Part* 2, 11 (2011), 0–40.

[27] Mark Hildebrand, Jawad Ali Khan, Sanjeev N. Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems using Integer Linear Programming. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020). https://api.semanticscholar.org/CorpusID:212641763

[28] Chao Huang, Orion Lawlor, and Laxmikant V Kale. 2003. Adaptive mpi. In *International workshop on languages and compilers for parallel computing*. Springer, 306–322.

[29] Wei Huang, Matthew J. Koop, and Dhabaleswar K. Panda. 2008. Efficient One-Copy MPI Shared Memory Communication in Virtual Machines. In *International Conference on Cluster Computing*.

[30] Y. Huang and D. Li. 2017. Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems. In *IEEE International Conference on Cluster Computing (CLUSTER)*.

[31] Intel Corporation. 2019. Intel Memory Latency Checker v3.5. https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[32] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. 2024. Memory Sharing with CXL: Hardware and Software Design Approaches. *arXiv preprint arXiv:2404.03245* (2024).

[33] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 585–600. https://www.usenix.org/conference/atc23/presentation/jang

[34] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS — OS design for heterogeneous memory management in datacenter. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), 521–534. https://api.semanticscholar.org/CorpusID:19189083

[35] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:236992513

[36] Sandeep Kumar, Aravinda Prasad, Smruti Ranjan Sarangi, and Sreenivas Subramoney. 2021. Radiant: efficient page table management for tiered memory systems. *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (2021). https://api.semanticscholar.org/CorpusID:235463147

[37] Argonne National Lab. [n. d.]. MPICH | High-Performance Portable MPI. https://www.mpich.org/

[38] Lawrence Berkeley National Laboratory. 2025. iPerf - The TCP, UDP and SCTP network bandwidth measurement tool. https://iperf.fr/

[39] Network-Based Computing Laboratory. [n. d.]. OSU Micro-Benchmarks. https://mvapich.cse.ohio-state.edu/benchmarks/

[40] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[41] Jie Li, George Michelogiannakis, Brandon Cook, Dulanya Cooray, and Yong Chen. 2023. Analyzing Resource Utilization in an HPC System: A Case Study of NERSC's Perlmutter. In *High Performance Computing*, Abhinav Bhatele, Jeff Hammond, Marc Baboulin, and Carola Kruse (Eds.). Springer Nature Switzerland, Cham, 297–316.

[42] Shigang Li, Torsten Hoefler, and Marc Snir. 2018. NUMA-Aware Shared-Memory Collective Communication for MPI. In *International Symposium on High-Performance Parallel and Distributed Computing*.

[43] Zhe Li and Mingyu Wu. 2022. Transparent and lightweight object placement for managed workloads atop hybrid memories. *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2022). https://api.semanticscholar.org/CorpusID:247108266

[44] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S. Berger, and Huaicheng Li. 2024. Dissecting CXL Memory Performance at Scale: Analysis, Modeling, and Optimization. arXiv:2409.14317 [cs.OS] https://arxiv.org/abs/2409.14317

[45] Jiawen Liu, Dong Li, and Jiajia Li. 2021. Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory. In *International Conference on Supercomputing (ICS)*.

[46] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory. In *Principles and Practice of Parallel Programming*.

[47] Jiawen Liu, Hengyu Zhao, Matheus A. Ogleari, Dong Li, and Jishen Zhao. 2018. Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach. In *IEEE/ACM International Symposium on Microarchitecture*.

[48] Jiaolin Luo, Luanzheng Guo, Jie Ren, Kai Wu, and Dong Li. 2020. Enabling Faster NGS Analysis on Optane-based Heterogeneous Memory. In *Proceedings of Supercomputing '20 (Posters)*.

[49] Bin Ma, Jie Ren, Shuangyan Yang, Benjamin Francis, Ehsan Ardestani, Min Si, and Dong Li. 2025. Machine Learning-Guided Memory Optimization for DLRM Inference on Tiered Memory. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[50] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, et al. 2024. {HydraRPC}:{RPC} in the {CXL} Era. In *USENIX Annual Technical Conference*.

[51] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniela Campello, Andy Rudoff, and Raju Rangaswami. 2022. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022), 925–937. https://api.semanticscholar.org/CorpusID:248865268

[52] NVIDIA. [n. d.]. NVIDIA ConnectX-6 InfiniBand/Ethernet Adapter Cards User Manual. https://docs.nvidia.com/networking/display/connectx6vpi/introduction

[53] Ivy Peng, Ian Karlin, Maya Gokhale, Kathleen Shoga, Matthew Legendre, and Todd Gamblin. 2022. A Holistic View of Memory Utilization on HPC Systems: Current and Future Trends. In *Proceedings of the International Symposium on Memory Systems*.

[54] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021). https://api.semanticscholar.org/CorpusID:239029009

[55] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. 2021. Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy. In *International Conference on Supercomputing (ICS)*.

[56] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, and Hyeran Jeon. 2021. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2021), 598–611. https://api.semanticscholar.org/CorpusID:231620477

[57] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory Systems. In *European Conference on Computer Systems*.

[58] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[59] Niklas Schelten, Fritjof Steinert, Justin Knapheide, Anton Schulte, and Benno Stabernack. 2022. A High-Throughput, Resource-Efficient Implementation of the RoCEv2 Remote DMA Protocol and its Application. *ACM Transactions on Reconfigurable Technologies and Systems*, Article 5 (2022).

[60] Debendra Das Sharma. 2022. Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 5–12.

[61] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. 2023. An Introduction to the Compute Express Link (CXL) Interconnect. arXiv:2306.11227 [cs.AR]

[62] Matthew Small and Xin Yuan. 2009. Maximizing MPI point-to-point communication performance on RDMA-enabled clusters with customized protocols. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS '09)*. Association for Computing Machinery, New York, NY, USA, 306–315. doi:10.1145/1542275.1542320

[63] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. [n. d.]. Understanding the use of message passing interface in exascale proxy applications. *Concurrency and Computation: Practice and Experience* 33, 14 ([n. d.]). doi:10.1002/cpe.5901

[64] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. doi:10.1145/3613424.3614256

[65] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 818–833. doi:10.1145/3627703.3650061

[66] Mellanox Technologies. [n. d.]. HP and Mellanox Benchmarking Report for Ultra Low Latency 10 and 40Gb/s Ethernet Interconnect. https://network.nvidia.com/related-docs/whitepapers/HP_Mellanox_FSI%20Benchmarking%20Report%20for%2010%20%26%2040GbE.pdf

[67] Tu Tran, Mustafa Abduljabbar, Hooyoung Ahn, Seonyoung Kim, Yoomi Park, Woojong Han, Shinyoung Ahn, Hari Subramoni, and Dhabaleswar K. Panda. 2024. OMB-CXL: A Micro-Benchmark Suite for Evaluating MPI Communication Utilizing Compute Express Link Memory Devices. In *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing* (Providence, RI, USA) *(PEARC '24)*. Association for Computing Machinery, New York, NY, USA, Article 27, 8 pages. doi:10.1145/3626203.3670533

[68] Stephen Van Doren. 2019. Hoti 2019: Compute express link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 18–18.

[69] Courtenay T. Vaughan and Richard F. Barrett. 2015. Enabling Tractable Exploration of the Performance of Adaptive Mesh Refinement. In *2015 IEEE International Conference on Cluster Computing*. 746–752. doi:10.1109/CLUSTER.2015.129

[70] Jacob Wahlgren, Maya Gokhale, and Ivy B. Peng. 2022. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. In *IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*.

[71] Jacob Wahlgren, Gabin Schieffer, Maya Gokhale, and Ivy Peng. 2023. A Quantitative Approach for Adopting Disaggregated Memory in HPC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[72] Jacob Wahlgren, Gabin Schieffer, Maya Gokhale, and Ivy Peng. 2023. A quantitative approach for adopting disaggregated memory in HPC systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[73] Pranjal Walia, Ishan Shanware, Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, and Uma M. Natarajan. 2024. LOSM: Leveraging OpenMP and Shared Memory for Accelerating Blocking MPI Allreduce. In *International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*.

[74] Chenxi Wang, Huimin Cui, Ting Cao, John N. Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: holistic memory management for big data processing over hybrid memories. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019). https://api.semanticscholar.org/CorpusID:150372592

[75] Xi Wang, Jie Liu, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. 2025. Performance Characterization of CXL Memory and Its Use Cases. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1048–1061.

[76] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 496–508.

[77] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, M. Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: transparent memory offloading in datacenters. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022). https://api.semanticscholar.org/CorpusID:247026540

[78] Sam White and Laxmikant V Kale. 2020. Optimizing Point-to-Point Communication between Adaptive MPI Endpoints in Shared Memory. *Concurrency and Computation: Practice and Experience* 32, 3 (2020).

[79] Jianbo Wu, Jie Liu, Gokcen Kestor, Roberto Gioiosa, and Dong Li. 2024. Performance Study of CXL Memory Topology. In *10th International Symposium on Memory Systems*.

[80] K. Wu, Y. Huang, and D. Li. 2017. Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis*.

[81] Kai Wu, Jie Ren, and Dong Li. 2018. Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Memory for Task Parallel Programs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*.

[82] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey Vetter, and Sparsh Mittal. 2016. Algorithm-Directed Data Placement in Explicitly Managed Non-Volatile Memory. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[83] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at its On-DIMM Buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 488–505.

[84] Rui Xie, Asad Ul Haq, Linsen Ma, Krystal Sun, Sanchari Sen, Swagath Venkataramani, Liu Liu, and Tong Zhang. 2024. SmartQuant: CXL-based AI Model Store in Support of Runtime Configurable Weight Quantization. arXiv:2407.15866 [cs.LG] https://arxiv.org/abs/2407.15866

[85] Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. 2023. Merchandiser: Data Placement on Heterogeneous Memory for Task-Parallel HPC Applications with Load-Balance Awareness. In *Proceedings of the Symposium on Principles and Practices of Parallel Programming (PPoPP)*.

[86] Zi-Wei Xiong, De-Jun Jiang, Jin Xiong, and Ren Ren. 2023. Dalea: A persistent multi-level extendible hashing with improved tail performance. *Journal of Computer Science and Technology* 38, 5 (2023), 1051–1073.

[87] Dong Xu, Junhee Ryu, Jinho Baek, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *30th USENIX Annual Technical Conference (ATC)*.

[88] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. 2023. Betty: Enabling Large-Scale GNN Training with Batch-Level Graph Partitioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[89] Shuangyan Yang, Minjia Zhang, and Dong Li. 2025. Buffalo: Enabling Large-Scale GNN Training via Memory-Efficient Bucketization. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[90] Yujie Yang, Lingfeng Xiang, Peiran Du, Zhen Lin, Weishu Deng, Ren Wang, Andrey Kudryavtsev, Louis Ko, Hui Lu, and Jia Rao. 2025. Architectural and System Implications of CXL-enabled Tiered Memory. arXiv:2503.17864 [cs.AR] https://arxiv.org/abs/2503.17864

[91] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 658–674. doi:10.1145/3600006.3613135

[92] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 461–476.

[93] Pengfei Zuo, Yu Hua, and Jie Wu. 2019. Level hashing: A high-performance and flexible-resizing persistent hashing index structure. *ACM Transactions on Storage (TOS)* 15, 2 (2019), 1–30.

Appendix: Artifact Description

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C_1 We explore the feasibility and benefits of using CXL memory sharing (CXL SHM) for inter-node communication on real CXL memory.

C_2 We develop a library, cMPI, by extending MPICH-4.2.3 and address the limitations of CXL memory sharing in supporting MPI communications.

C_3 Our evaluation shows that by using CXL memory sharing for one-sided and two-sided inter-node communications, cMPI outperforms TCP over a standard Ethernet NIC by up to 49× in latency and 72× in bandwidth for all messages, and outperforms TCP over a high-end SmartNIC by up to 48× in latency and 3.7× in bandwidth for small messages.

C_4 We evaluate the scalability of CXL memory sharing at large-scales using SimGrid.

A.2 Computational Artifacts

- DOI: https://doi.org/10.5281/zenodo.16938376
- GitHub: https://github.com/skhynix/cMPI.git

A_1 cMPI/cxl_shm_tests
A_2 cMPI/mpich
A_3 cMPI/mpi_run_bench
A_4 cMPI/simulation

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Tables 1 Figure 11
A_2	C_2	Tables 2 Figures 5-9
A_3	C_3	Figure 5-9
A_4	C_4	Figure 10

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

Artifact A_1 provides the implementation of our micro-benchmark to test the latency and bandwidth of CXL memory sharing with various cache ability, supporting contribution C_1.

To measure latency and bandwidth of other communication approaches, We use Intel MLC for Main Memory and CXL SHM (with caching), iPerf for TCP over Ethernet and Mellanox CX-6 Dx, and Perftest for RoCEv2. Results for RoCEv2 over CX-3 and InfiniBand over CX-6 are from vendor reports.

Expected Results

The results will show that CXL memory sharing achieves 7.2×–8.1× lower latency compared to TCP over standard Ethernet NIC and TCP over a high-end SmartNIC. Its bandwidth is 80× higher than TCP over Ethernet and comparable to RoCEv2 (as shown in Table 1).

The results will also present the latency of CXL memory sharing under different cacheability settings (as shown in Figure 11).

Expected Reproduction Time (in Minutes)

All tests in Table 1 take around 20 minutes, and all tests in Figure 11 take around 10 minutes.

Artifact Setup (incl. Inputs)

Hardware.

- Two servers, each equipped with two Intel Xeon Gold 6530 processors (@2.10 GHz, 32 cores per socket). Each socket has 8×32 GB DDR5-5600 DIMMs.
- Two servers are connected to Niagara 2.0, an FPGA-based CXL pooled memory platform.
- Two servers are equipped with a standard Ethernet NIC (1,000 Mb/s) and a Mellanox CX-6 Dx SmartNIC (100 Gb/s).

Software.

- cxl_shm_tests: https://github.com/skhynix/cMPI/tree/main/cxl_shm_tests
- Intel MLC: https://downloadmirror.intel.com/834254/mlc_v3.11b.tgz
- iPerf: https://iperf.fr/iperf-download.php#ubuntu
- Perftest:https://github.com/linux-rdma/perftest

Installation and Deployment.

- Install cxl_shm_tests:
```
cd $HOME
git clone --recursive https://github.com
    /skhynix/cMPI.git
cd cMPI/cxl_shm_tests/memo_ae/src &&
    make
```
- Install Intel MLC:
```
cd $HOME
wget https://downloadmirror.intel.com
    /834254/mlc_v3.11b.tgz
tar -xzvf mlc_v3.11b.tgz
```
- Install iPerf:
```
sudo apt-get install iperf3
```
- Install Perftest:
```
cd $HOME
git clone https://github.com/linux-rdma/
    perftest.git
cd perftest/
./autogen.sh && ./configure
make
```

Artifact Execution

- T_1 Test main memory and CXL Memory Sharing (with caching; without cache flushing):
```
cd $HOME/Linux
```

```
# test latency
./mlc --latency_matrix
# test bandwidth
./mlc --bandwidth_matrix
```

- T_2 Test CXL Memory Sharing (with cache flushing):

```
cd $HOME/cMPI/cxl_shm_tests/memo_ae/
    test_cxl
# Latency
bash test_memset_lat_clflushopt.sh
# Bandwidth
bash test_seq_bw_devdax_read_nt.sh
```

- T_3 Test TCP over Standard Ethernet NIC:

```
# Latency
## on server
iperf3 -s -p 45678
## on client
iperf3 -p 45678 -c server_ip -t 30 -u

# Bandwidth
## on server
iperf3 -s -p 45678
## on client
iperf3 -p 45678 -c server_ip -t 30
```

- T_4 Test TCP over Mellanox (CX-6 Dx):

```
# Latency
## on server
iperf3 -s -p 45678
## on client
iperf3 -p 45678 -c mlx_server_ip -t 30 -
    u

# Bandwidth
## on server
iperf3 -s -p 45678
## on client
iperf3 -p 45678 -c mlx_server_ip -t 30 -
    P 8
```

- T_5 Test RoCEv2 over Mellanox (CX-6 Dx):

```
cd $HOME/perftest
# Latency
## on server
./ib_write_lat --ib-dev=mlx5_1 -s 8 -n
    10000
## on client
./ib_write_lat --ib-dev=mlx5_1 -s 8 -n
    10000  mlx_server_ip

# Bandwidth
## on server
./ib_write_bw --ib-dev=mlx5_1 -s 65536 -
    n 10000
## on client
./ib_write_bw --ib-dev=mlx5_1 -s 65536 -
    n 1000 mlx_server_ip
```

- T_6 Test latency of CXL memory sharing (uncacheable):

```
cd $HOME/cMPI/cxl_shm_tests/memo_ae/
    test_cxl
bash test_memset_lat_uncacheable.sh
```

- T_7 Test latency of CXL memory sharing with `clflush`:

```
cd $HOME/cMPI/cxl_shm_tests/memo_ae/
    test_cxl
bash test_memset_lat_clflush.sh
```

- T_8 Test latency of CXL memory sharing with `clflushopt`:

```
cd $HOME/cMPI/cxl_shm_tests/memo_ae/
    test_cxl
bash test_memset_lat_clflushopt.sh
```

B.2 Computational Artifact A_2

Relation To Contributions

Artifact A_2 provides the implementation of MPI based on CXL SHM, enhancing MPI one-sided and two-sided inter-node communications, supporting C_2.

Expected Results

The expected results will show the latency and bandwidth of one-sided and two-sided communication.

Expected Reproduction Time (in Minutes)

Running one test takes approximately 10 minutes.

Artifact Setup (incl. Inputs)

Hardware. See the hardware requirements in A_1.

Software.

- Miniconda: https://www.anaconda.com/docs/getting-started/miniconda/install#linux
- mpich-4.2.3-cxl: https://github.com/skhynix/cMPI.git
- Extended osu-micro-benchmarks-7.4 (OMB): https://github.com/skhynix/cMPI/tree/main/mpi_run_bench/osu-micro-benchmarks-ext

Installation and Deployment. The MPI library and benchmarks need to be installed on each of the two servers.

- Download cMPI:

```
cd $HOME
git clone --recursive https://github.com
    /skhynix/cMPI.git
```

- Create CONDA environment:

```
conda create -n mpich_cxl
conda activate mpich_cxl
conda install -c conda-forge compilers
```

- Install mpich-4.2.3-cxl:

```
conda activate mpich_cxl
cd $HOME/cMPI/mpich
./autogen.sh
```

```
./configure --prefix=$CONDA_PREFIX --
    with-shared-memory=cxl
    MPICHLIB_CFLAGS="-march=native -
    mclflushopt"
make -j32 && make install
```

- Install OMB:

```
conda activate mpich_cxl
cd $HOME/cMPI
cp -R mpi_run_bench/osu-micro-benchmarks
    -ext osu-micro-benchmarks-7.4-
    mpich_cxl
cd osu-micro-benchmarks-7.4-mpich_cxl
./configure  --prefix=$CONDA_PREFIX/osu-
    micro-benchmarks CC=$CONDA_PREFIX/
    bin/mpicc CXX=$CONDA_PREFIX/bin/
    mpicxx
make -j32 && make install
```

Artifact Execution

- T_1 Configure the IPs of the two servers in a hostfile:

```
vim ~/hostfile
```

- T_2 Test one-sided based on CXL SHM:

```
conda activate mpich_cxl
mpirun -np 2 -hostfile ~/hostfile $HOME/
    cMPI/osu-micro-benchmarks-7.4-
    mpich_cxl/c/mpi/one-sided/
    osu_get_latency
```

- T_3 Test two-sided based on CXL SHM:

```
conda activate mpich_cxl
mpirun -np 2 -hostfile ~/hostfile $HOME/
    cMPI/osu-micro-benchmarks-7.4-
    mpich_cxl/c/mpi/pt2pt/standard/
    osu_latency
```

B.3 Computational Artifact A_3

Relation To Contributions

Artifact A_3 provides script generation scaffold for the latency and bandwidth evaluation of MPI inter-node communication over different interconnects, along with extended OMB benchmarks that support testing one-sided bandwidth and latency with multiple processes, supporting contribution C_3.

Expected Results

The results will show that cMPI outperforms TCP over a standard Ethernet NIC by up to 49× in latency and 72× in bandwidth for all messages, and outperforms TCP over a high-end SmartNIC by up to 48× in latency and 3.7× in bandwidth for small messages (as shown in Figures 5–9).

Expected Reproduction Time (in Minutes)

Running tests for each sub-figure in Figures 5–9 takes around 60 minutes.

Artifact Setup (incl. Inputs)

Hardware. See the hardware requirements in A_1.

Software.

- Miniconda: https://www.anaconda.com/docs/getting-started/miniconda/install#linux
- mpi_run_bench: https://github.com/skhynix/cMPI/tree/main/mpi_run_bench
- mpich-4.2.3: https://github.com/pmodels/mpich/tree/v4.2.3
- mpich-4.2.3-cxl: https://github.com/skhynix/cMPI.git
- Extended osu-micro-benchmarks-7.4 (OMB): https://github.com/skhynix/cMPI/tree/main/mpi_run_bench/osu-micro-benchmarks-ext

Installation and Deployment. The MPI library and benchmarks need to be installed in each of two servers.

- Download cMPI:

```
cd $HOME
git clone --recursive https://github.com
    /skhynix/cMPI.git
```

- Install mpich-4.2.3:

```
# create CONDA env
conda create -n mpich
conda activate mpich
conda install -c conda-forge compilers
# build mpich-4.2.3
cd $HOME/cMPI
git clone https://github.com/pmodels/
    mpich.git mpich-4.2.3
cd mpich-4.2.3 && git checkout v4.2.3
./autogen.sh
./configure --prefix=$CONDA_PREFIX
make -j32 && make install
```

- Install extended OMB with mpich-4.2.3:

```
conda activate mpich
cp -R $HOME/cMPI/mpi_run_bench/osu-micro
    -benchmarks-ext $HOME/cMPI/osu-micro
    -benchmarks-7.4-mpich
cd $HOME/cMPI/osu-micro-benchmarks-7.4-
    mpich
./configure  --prefix=$CONDA_PREFIX/osu-
    micro-benchmarks CC=$CONDA_PREFIX/
    bin/mpicc CXX=$CONDA_PREFIX/bin/
    mpicxx
make -j32 && make install
```

- Install mpich-4.2.3-cxl: See steps in A_2.
- Install extended OMB with mpich-4.2.3-cxl: See steps in A_2.

Artifact Execution

- T_1 Generate scripts for MPI evaluation:

```
cd $HOME/cMPI/mpi_run_bench
# Generate scripts for OMB one-sided
    latency test
bash gen_one_sided_lat.sh
```

```
# Generate scripts for OMB one-sided
    bandwidth test
bash gen_one_sided_mbw.sh
# Generate scripts for OMB two-sided
    latency test
bash gen_two_sided_lat.sh
# Generate scripts for OMB two-sided
    bandwidth test
bash gen_two_sided_mbw.sh
# Generate scripts for two-sided
    bandwidth test on CXL SHM with
    different cell size
bash gen_cxl_diff_cell.sh
```

- T_2 Run MPI evaluation:

```
cd ~
# Run OMB one-sided latency test
source $HOME/cMPI/mpi_run_bench/
    run_one_sided_lat.sh
# Run OMB one-sided bandwidth test
source $HOME/cMPI/mpi_run_bench/
    run_one_sided_mbw.sh
# Run OMB two-sided latency test
source $HOME/cMPI/mpi_run_bench/
    run_two_sided_lat.sh
# Run OMB two-sided bandwidth test
source $HOME/cMPI/mpi_run_bench/
    run_two_sided_mbw.sh
# Run two-sided bandwidth test on CXL
    SHM with different cell size
source $HOME/cMPI/mpi_run_bench/
    run_cxl_diff_cell.sh
```

Artifact Analysis (incl. Outputs)

- Install dependencies:
  `pip3 install matplotlib numpy`
- Draw figures:
  `cd $HOME/cMPI/mpi_run_bench/plot`
  `bash plot.sh`
- See Figure 5 in osu_get_mbw.png.
- See Figure 6 in osu_get_multi_lat.png.
- See Figure 7 in osu_pt2pt_mbw.png.
- See Figure 8 in osu_pt2pt_multi_lat.png.
- See Figure 9 in osu_pt2pt_mbw_cell.png.
- See Figure 11 in memset_latency.png.

B.4 Computational Artifact A_4

Relation To Contributions

Artifact A_4 provides scaling tests comparing CXL SHM with TCP over Mellanox (CX-6 Dx) and TCP over Ethernet using the SimGrid simulator. It evaluates the scalability of one-sided and two-sided MPI communication, supporting C_4.

Expected Results

For CG, CXL SHM reduces communication time by 25.3% compared to TCP over Mellanox (CX-6 Dx) and by 37.6% compared to TCP over Ethernet. For miniAMR, it reduces communication time by 4.9% and 9.8%, respectively (see Figure 10).

Expected Reproduction Time (in Minutes)

For CG, each test using one interconnect under a specific node count configuration takes around 60 minutes. For miniAMR, each test using one interconnect under a specific node count configuration takes around 15 minutes.

Artifact Setup (incl. Inputs)

Hardware. One server with two sockets, each equipped with an AMD EPYC 7742 64-core processor.

Software.

- Miniconda: https://www.anaconda.com/docs/getting-started/miniconda/install#linux
- SimGrid: https://github.com/simgrid/simgrid.git
- CG: https://www.nas.nasa.gov/assets/npb/NPB3.4.3.tar.gz
- miniAMR: https://github.com/Mantevo/miniAMR.git

Installation and Deployment.

- Install SimGrid simulator:

```
cd $HOME/cMPI
git clone https://framagit.org/simgrid/
    simgrid.git
cd simgrid && cmake -
    DCMAKE_INSTALL_PREFIX=$HOME/cMPI/
    simgrid .
make && make install
```

- Install CG:

```
cd $HOME/cMPI/simulation/NPB3.4-MPI
# Set MPIFC in config/make.def to
    smpif90
make cg CLASS=D
```

- Install miniAMR:

```
cd $HOME/cMPI/simulation/miniAMR/ref &&
    make
```

Artifact Execution

- T_1 Run the simulation of CG:

```
cd $HOME/cMPI/simulation/NPB3.4-MPI
conda activate mpich_cxl
sh cg_Mellanox.sh
sh cg_Ethernet.sh
sh cg_cxl.sh
```

- T_2 Run the simulation of miniAMR:

```
cd $HOME/cMPI/simulation/miniAMR
conda activate mpich_cxl
sh amr_Mellanox.sh
sh amr_Ethernet.sh
sh amr_cxl.sh
```