Scalable, Topology- and Multi-HCA-Aware Hierarchical GPU Allgather Using Parallel Rings

Amirreza Barati Sedeh, Ryan Grant, Ahmad Afsahi
Queen's University, Canada
{amirreza.baratisedeh, ryan.grant, ahmad.afsahi}@queensu.ca

***Abstract*—The ring algorithm is a key communication pattern for optimizing collective operations, particularly for large-message collectives such as Allreduce and Allgather, which are critical for efficient AI model training. While the traditional ring algorithm exchanges data sequentially, it does not fully exploit all available HCAs in modern high-performance clusters, leaving potential bandwidth underutilized. In this work, we introduce a GPU-based, topology-aware, hierarchical, and highly scalable communication approach called Parallel Rings, which fully leverages inter-node bandwidth and significantly enhances large-message collective performance. Our design (implemented in Open MPI) also supports effective overlap between intra- and inter-node communication phases. Furthermore, we highlight the importance of selecting an efficient intra-node algorithm and propose a congestion-aware algorithm optimized for nodes with inter-socket GPU topologies. We extensively evaluate our design on different cluster architectures and benchmarking levels, providing detailed performance analysis. We compare our design with the state-of-the-art communication libraries, and the experimental results show that our approach reduces Allgather latency up to 74%, 80%, and 31% compared to Open MPI, MVAPICH-Plus, and NCCL, respectively.**

***Index Terms*—GPU-based Allgather, Parallel Rings, Multi-HCA, Topology-aware, Fully Sharded Data Parallelism (FSDP)**

I. Introduction

The rapid advancement of Artificial Intelligence (AI) has driven the development of increasingly large and complex models, such as Large Language Models (LLMs) and transformers. Training these expansive models necessitates distributing the computational workload across numerous GPUs and nodes, which is known as distributed training. Central to the efficiency of distributed training are collective communication operations—Allreduce, Allgather, ReduceScatter, and Alltoall—which facilitate the synchronization of model parameters and the aggregation of gradients across multiple devices. These collective communications can take a significant fraction of training time in AI models and be constrained by the GPU communication bandwidth [1–3].

Requirements such as large messages exchanged during these collectives, bridging the bandwidth gap between intra- and inter-node communication channels, and LLM communication patterns [4, 5] have significantly influenced server design. Intra-node communication channels, such as NVIDIA's NVLink and AMD's Infinity Fabric, offer high bandwidth between GPUs within a single server, supporting efficient data transfer for multi-GPU configurations. For instance, servers with 4 GPUs often employ a full-mesh topology, providing bi-directional links between any pair of GPUs (we will use NVIDIA terms for the rest of the paper; however, our design applies to any vendor), enabling direct communication between all GPUs. However, as distributed training scales to hundreds or thousands of GPUs, the need for high-bandwidth inter-node communication becomes critical. To bridge the bandwidth gap between intra- and inter-node communications, modern servers are equipped with multiple Host Channel Adapters (HCA)s, enabling the *capability of having higher bandwidths* across nodes. This architectural evolution has emerged among supercomputers, according to the TOP500 list, which utilizes multiple HCAs per server to meet the demanding requirements of large-scale AI model training [6].

GPU-aware communication libraries, such as Open MPI [7] (with UCX [8] as the underlying communication framework), commonly employ the ring algorithm to exchange large messages across processes distributed over multiple nodes. In this approach, a virtual ring is formed among the processes, where each process communicates only with its immediate neighbours—receiving data from the left and sending to the right. While conceptually simple and effective for modest-scale systems, this design exhibits two significant limitations. First, as the system scales out, the linear nature of the ring algorithm constrains performance: for P processes, the naïve ring requires P–1 sequential steps, which increases latency and reduces throughput. Second, at each communication step, only a single HCA per direction is utilized; for instance, in a server with four HCAs and four GPU ranks, only rank 0 receives from the network and only rank 3 sends, leaving the remaining HCAs underutilized. Consequently, existing libraries fail to exploit the full inter-node bandwidth available on multi-HCA servers. To address this inefficiency, we propose a GPU-based, multi-HCA-aware, topology-aware, hierarchical, and highly scalable communication algorithm that employs Parallel Rings (an algorithm in which multiple independent rings run in parallel), enabling full utilization of inter-node bandwidth and significantly improving large-message collective communication performance.

Allgather is a fundamental collective operation used across a wide range of numerical workloads—including radix sort, differential equation solvers, and especially matrix–matrix multiplication [9], a core kernel in modern AI training [10]. Its importance has grown further with the emergence of Fully Sharded Data Parallelism (FSDP), which improves memory efficiency by sharding model parameters, gradients, and op-

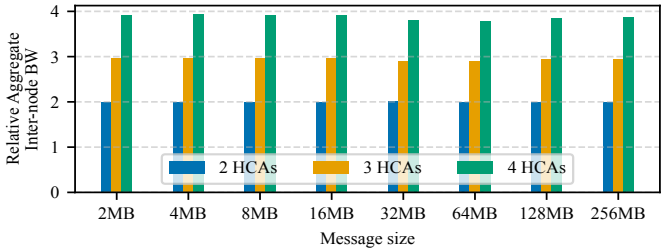
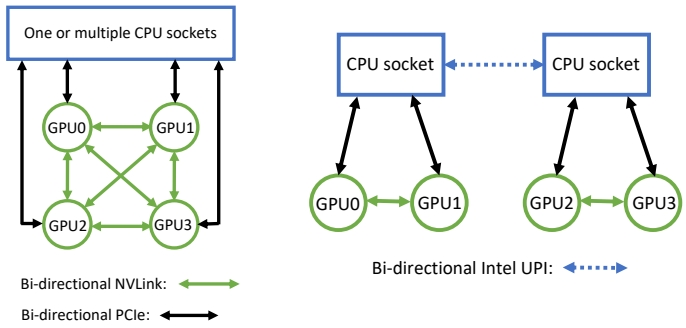


Fig. 1: Relative aggregate bandwidth of multi-pair point-to-point test with 2, 3, and 4 processes per node compared to one. Each process is mapped to a distinct HCA, and its data resides in the GPU memory.



(a) NVLink-only connection between all GPUs (b) NVLink/UPI connection between GPUs

Fig. 2: Intra-node GPU topologies

timizer states and reconstructing them via Allgather during both forward and backward passes [11]. Because these phases involve exchanging large messages, large-scale systems typically implement Allgather using the ring algorithm, making it a prime target for optimization in our proposed multi-HCA-aware communication design. Given the centrality of Allgather in distributed matrix multiplication and FSDP-based AI training, we include matrix multiplication as a representative benchmark and use FSDP to evaluate the performance of our optimized Parallel Rings at the application level. Although our approach can be applied to any collective operation built on a ring structure, this work focuses specifically on accelerating Allgather and makes the following main contributions:

1) ***Hierarchical and Multi-HCA-Aware Algorithm****:* We develop a communication algorithm using Parallel Rings that leverages all the available HCAs in a server and has a pipelined hierarchical structure to overlap intra- and inter-node communication.

2) ***Scalable and Topology-Aware Design****:* By employing Parallel Rings, our design improves algorithmic scalability proportionally to the number of HCAs per server. Additionally, the algorithm is topology-aware, adapting to intra-node GPU configurations to maximize intra-node communication efficiency.

3) ***Experiments and Analysis****:* We evaluated Parallel Rings on two different systems with various interconnect and GPU topologies at three benchmarking levels. We have achieved up to 74%, 80%, and 31% speedup for Allgather over Open MPI, MVAPICH-Plus [12] and NCCL [13]. We also provide detailed analyses for observations at the application layer.

II. Motivation and Background

A. Motivation

Modern clusters commonly include multiple HCAs per server, offering the potential to narrow the gap between intra- and inter-node communication by increasing inter-node bandwidth, if all HCAs are effectively utilized. In the traditional ring algorithm, however, only one rank per node participates in inter-node communication in each direction, limiting bandwidth to a single HCA and leaving the remaining HCAs idle. To motivate the usage of multiple HCAs, Figure 1 shows relative aggregate bandwidth from the *osu_mbw_mr* test in OSU Micro-Benchmarks (OMB) [14] when 2, 3, and 4 processes per node are used, each bound to a distinct HCA with data residing in GPU memory. The results scale nearly linearly with the number of HCAs, showing the need for collective algorithms to adapt to modern networking capabilities. As a result, the traditional ring algorithm is no longer sufficient and must be redesigned to scale with today's networking architectures.

B. Existing Allgather designs

Since this work focuses on large message sizes (in the order of megabytes), we specifically target the ring algorithm, which is best suited for this message range and widely adopted in modern GPU communication frameworks. GPU-aware communication libraries such as Open MPI (with UCX) and NCCL predominantly rely on the ring algorithm for exchanging large inter-node messages in Allgather collective on large-scale systems. In Open MPI, processes are organized in a virtual flat ring where each process communicates only with its immediate neighbours. While this method is conceptually simple and effective for large message transfers, its linear communication pattern and limited utilization of multiple available HCAs lead to inefficient performance. NCCL also employs a ring-based approach for the Allgather collective, but its implementation is more sophisticated [15]. Rather than using a single ring, NCCL constructs multiple logical communication paths, known as channels, which operate concurrently to complete the collective. In NCCL, each channel corresponds to a ring that processes a portion of the message (known as a chunk), whose performance depends heavily on how well NCCL adapts to the system's software stack, hardware topology, and available HCAs. Our design differs from NCCL in several key aspects. First, while NCCL executes multiple flat rings without distinguishing between intra- and inter-node bandwidth, it utilizes only a subset of available intra-node NVLink connections [16]. In contrast, our topology-aware approach fully exploits all NVLink links, resulting in higher overall link utilization. Second, every NCCL ring requires $P-1$ steps for P processes, constraining scalability at large

GPU counts, whereas our approach relaxes this limitation by a factor equal to the number of HCAs, thereby improving scalability. Third, our design assigns ring neighbours so that GPUs with the same local rank participate in the same ring, aligning naturally with rail-optimized network designs commonly used in LLM training, an alignment that NCCL rings do not necessarily provide.

C. Intra-node GPU Topology

In modern GPU-accelerated servers, intra-node connectivity typically follows one of two high-level architectural topologies. (1) All GPUs are interconnected via NVLink, often arranged in a full-mesh configuration that provides each GPU with dedicated high-bandwidth links to every other GPU (Figure 2a). This design enables uniform communication performance across all the GPUs in a node. (2), the topology in systems where GPUs are distributed across two CPU sockets and only GPUs within the same socket share NVLink connectivity. Communication between GPUs on different sockets must traverse inter-socket technologies such as Intel UPI, resulting in significantly lower effective bandwidth compared to NVLink (Figure 2b). This asymmetric connectivity introduces performance heterogeneity that must be carefully considered when designing communication algorithms.

III. Design and Implementation

In this section, we first discuss the advantages of Parallel Rings, then present the general stage-based Allgather design, and subsequently describe the Parallel Rings approach in detail for each GPU topology.

1) Multi-HCA-aware: Unlike the conventional ring algorithm, which employs only a single HCA (or at most two HCAs but in a unidirectional manner), the Parallel Rings algorithm leverages all available HCAs bi-directionally. This design enhances network parallelism and significantly increases inter-node bandwidth, thereby reducing the performance gap between intra-node and inter-node communication.

2) Hierarchical: The Parallel Rings employs a pipelined hierarchical structure that enables overlap between intra- and inter-node communications. While the message corresponding to step i is exchanged across the nodes, the message for step $i-1$ is concurrently exchanged at the node level.

3) Scalable: At large scales, the linear progression of the conventional ring algorithm limits performance. Specifically, for P processes, a linear ring (such as Open MPI and NCCL) requires P–1 sequential communication steps, resulting in increased latency. In contrast, the proposed Parallel Rings design enhances scalability by exploiting all of HCAs available on a server. The number of required steps is effectively reduced by a factor equal to the number of HCAs. This represents one of our key advantages over NCCL, making Parallel Rings more suitable for scaling to larger GPU counts. For example, with $P = 64$ processes, the conventional ring requires 63 steps, whereas the Parallel Rings require only (P/HCA)–1 steps. Assuming four HCAs, this results in just 15 steps.

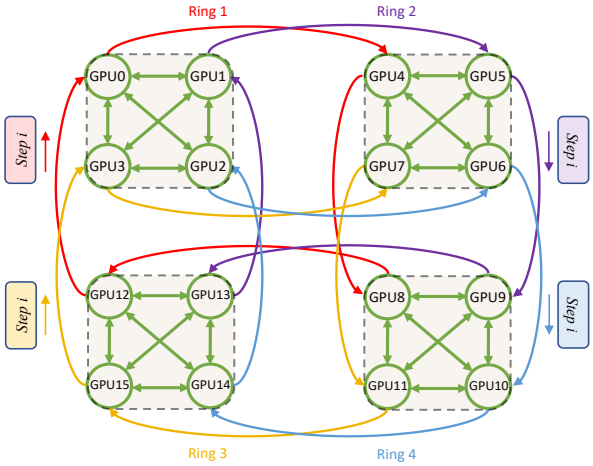


Fig. 3: Design of Parallel Rings for 16 GPUs across 4 nodes. The inter-node Parallel Rings design is independent of the intra-node topology (e.g., full-mesh NVLink shown in this figure)

4) Topology-aware: The Parallel Rings algorithm proposed in this work leverages simultaneous parallel rings for inter-node communication, making it applicable across diverse system architectures. However, intra-node communication performance is highly dependent on the underlying GPU topology. We demonstrate that the GPU topology significantly influences intra-node data exchange efficiency, highlighting the importance of selecting an appropriate intra-node algorithm. For instance, while a linear Allgather algorithm achieves the best performance on a full-mesh topology, it performs inefficiently on a UPI-based system. The algorithms presented in this work are therefore optimized for both topologies to ensure efficient communication across different system configurations.

A. General Design

We decompose the Allgather operation into a three-stage hierarchical algorithm:

- *Stage 1 - Local Message Exchange (LME)*: This is the initial one-time intra-node communication happening between the GPUs on the same node to share their messages. So, messages belonging to the local GPUs within a node are exchanged.
- *Stage 2 - Parallel Rings*: This is the recurring inter-node stage, which is consistent across all topologies, employing multiple parallel rings rather than a single ring. These rings operate simultaneously, with each ring mapped to a distinct HCA. A ring consists of GPUs that share the same local rank across all nodes; in other words, GPUs with identical local ranks participate in the same ring. Figure 3 illustrates this design. For example, GPUs with local rank 0 (ranks 0, 4, 8, 12, ...) form one ring, GPUs with local rank 1 (ranks 1, 5, 9, 13, ...) form another, and so forth.
- *Stage 3 - Remote Message Exchange (RME)*: This is the recurring intra-node communication stage in which the

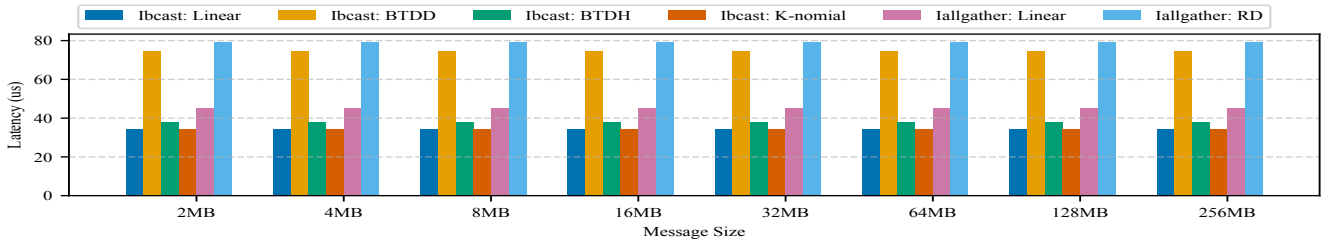


Fig. 4: Latency of different algorithms for intra-node communication (Stage 1 and Stage 3), Full-mesh topology. Measurements are done on a single node of cluster B described in Section IV-A.

received messages from remote GPUs in the second stage are exchanged between GPUs on the same node. The specific algorithm and ordering of the exchange depends on the intra-node GPU topology. Stages 1 and 3 are intra-node communication, so we use the same collective for both, depending on the intra-node topology. Our design aims to maximize the overlap between inter-node and intra-node communication to improve overall efficiency.

B. Cluster-wide Parallel Rings Allgather with Full-mesh Intra-node Topology

In architectures with a full-mesh topology, all four GPUs receive a message from the network at each step—originating from other ranks within their ring—that must be redistributed to other GPUs on the same node. Several strategies can be employed for this redistribution. Broadly, these approaches fall into two categories: (1) an intra-node non-blocking broadcast (Ibcast) from each rank, implemented using algorithms such as Linear, Binomial Tree with Distance Doubling (BTDD), Binomial Tree with Distance Halving (BTDH), or K-nomial tree (K equals 4 for all K-nomial-based collectives throughout this work); or (2) an intra-node non-blocking Allgather (Iallgather) among local ranks, using either a Linear algorithm or Recursive Doubling (RD). Figure 4 presents the latency of these intra-node algorithms on a full-mesh topology.

As shown in Figure 4, Linear and K-nomial algorithms show the lowest latency for intra-node communication, which is due to the existence of bi-directional NVLink connections between every pair of GPUs. Consequently, we adopt the Linear algorithm for the intra-node communication stages of the Allgather on full-mesh topologies. Algorithm 1 presents the cluster-wide Allgather procedure for such systems. At initialization, each GPU rank copies its own data into its designated region of the final *rbuf* (Lines 4-5). The variable $localsize$ denotes the number of GPU ranks on a node, and $localrank$ refers to a rank's index within the node, ranging from 0 to $localsize-1$. These values are cached to avoid runtime overhead. During Stage 1, i.e., *LME*, (Lines 7–16), GPUs on the same node exchange their messages using the Linear algorithm so that each local rank obtains the message from all other ranks on that node. Next, each rank determines its incoming and outgoing neighbours within its own ring (Lines 17-18). Unlike the conventional ring algorithm—where neighbours are one rank apart—in the Parallel Rings design, the rank distance between a GPU and its neighbours equals the number of local ranks per node ($localsize$). The recurring communication loop (Lines 20–34) then begins. In each iteration, a GPU rank computes the indices of the message segments to send/receive to/from remote GPUs, i.e., Stage 2 (Lines 21–23). While this inter-node transfer is taking place, the intra-node exchange

Algorithm 1: Cluster-wide Parallel Rings Allgather with Full-mesh Intra-node Topology

Input : $sbuf, scount, sdtype, rbuf, rcount, rdtype, comm$
Output: All data gathered in $rbuf$ across all ranks

```
1  ompi_coll_base_parallel_rings_Allgather()
2  size ← comm.size() ;            // Total number of ranks
3  rank ← comm.rank() ;             // Rank of this process
   // Copy local send buffer into receive buffer
4  if sbuf ≠ IN_PLACE then
5      Copy sbuf to rbuf[rank];
   // Initialize local rank and size (cached)
6  localrank, localsize ← precomputed values;
7  for q = 0 to localsize - 1 do
8      if q = localrank then
9          continue
10     target ← q + (rank - localrank);
11     Irecv rbuf[target] from target ;   // Receive local data
12 for q = 0 to localsize - 1 do
13     if q = localrank then
14         continue
15     target ← q + (rank - localrank);
16     Isend rbuf[rank] to target ;          // Send local data
17 outgoing_neighbour ← (rank + localsize) mod size;
18 incoming_neighbour ← (rank − localsize + size) mod size;
19 new_comm_size ← size/localsize;
20 for i = 0 to new_comm_size - 2 do
21     recv_src ← (rank − (localsize ∗ (i + 1)) + size) mod size;
22     send_src ← (rank − (i ∗ localsize) + size) mod size;
23     Sendrecv: (1) send rbuf[send_src] to outgoing_neighbour and
         (2) receive rbuf[recv_src] from incoming_neighbour;
24     for q = 0 to localsize - 1 do
25         if q = localrank then
26             continue
27         target ← q + (rank − localrank);
28         offset ← (target − localsize ∗ (i + 1) + size) mod size;
29         Irecv rbuf[offset] from target ;    // Receive remote
             data
30     for q = 0 to localsize - 1 do
31         if q = localrank then
32             continue
33         target ← q + (rank − localrank);
34         Isend rbuf[recv_src] to target ; // Send remote data
```

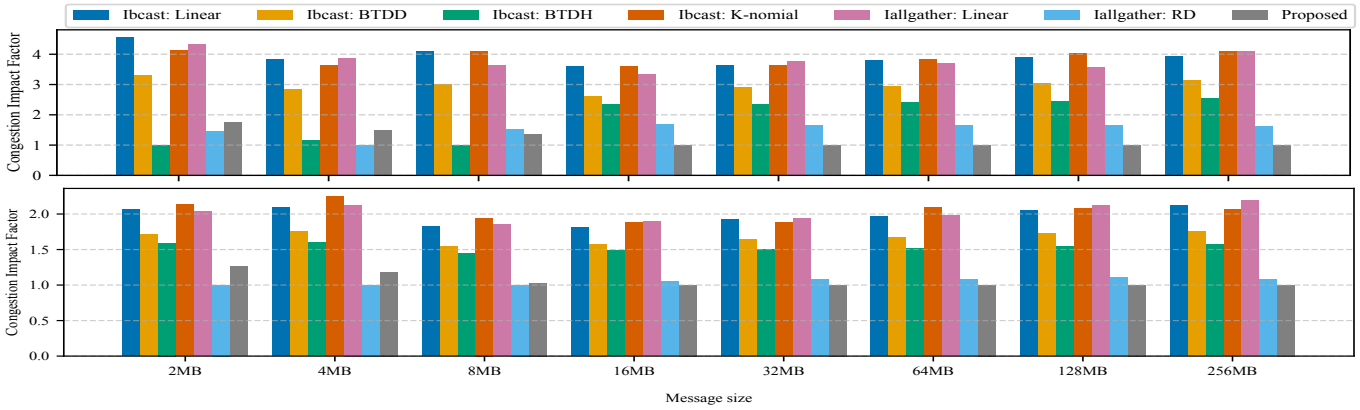


Fig. 5: Comparison of the congestion impact factor (relative performance) on inter-socket topology (Figure 2b) for **intra-node** (top) and **multi-node** (bottom) communication across various algorithms and message sizes. At each message size, algorithm latencies are normalized relative to the lowest latency achieved at that size among all algorithms. Measurements are done on cluster A described in Section IV-A.

from the previous iteration proceeds concurrently, i.e., Stage 3 or *RME*, (Lines 24–34), again using the Linear algorithm, which, as shown earlier, provides the lowest latency for full-mesh intra-node communication.

C. Cluster-wide Parallel Rings Allgather with Inter-socket Intra-node Topology

The intra-node communication stages on systems where GPUs must traverse an inter-socket path (e.g., UPI) experience substantial performance degradation compared to NVLink-connected configurations. The non-blocking collectives used for intra-node stages—namely Ibcast and Iallgather—employ a range of algorithms whose communication patterns, when mapped onto an inter-socket topology, may introduce unintended congestion or redundant data transfers. Table I summarizes these algorithms and analyzes the factors contributing to their performance degradation on inter-socket systems, highlighting the necessity of developing topology-aware communication patterns. Based on this, we present our enhanced congestion-aware algorithm specifically designed to improve performance on inter-socket GPU topologies.

In the Ibcast collective using the Linear algorithm, each GPU rank receives data from the other three ranks. Communication with the NVLink-connected peer proceeds efficiently; however, communication across the socket is suboptimal. Receiving data from inter-socket GPUs leads to incast congestion, while sending to them incurs serialization, as the copy engine obtains exclusive access to the transfer bus (PCIe or NVLink). The BTDD algorithm initially causes incast congestion at rank 1 and subsequently suffers from inter-socket serialization between ranks (0,2) and (1,3). The BTDH algorithm mitigates this behaviour by beginning with a congestion-free step, but it still encounters serialization in the second step. The effect is less severe because some UPI-based serialization is replaced by the faster NVLink-based serialization. The Ibcast K-nomial algorithm and the Iallgather Linear algorithm exhibit the same communication pattern and associated inefficiencies as the Ibcast Linear algorithm. For the Iallgather RD algorithm, the first

TABLE I: Congestion Analysis of Intra-node GPU Communication on Inter-socket Topology (Figure 2b). (I: Incast congestion, S: Serialization, D: Duplicate data transfer, F: Congestion-free)

Algorithm	**Step 1**	**Step 2**	**Step 3**
Ibcast: Linear	0 → 1 F 1 → 0 F 2 → 0 I 3 → 0 I	0 → 2 I 1 → 2 I 2 → 1 I 3 → 1 I	0 → 3 I 1 → 3 I 2 → 3 F 3 → 2 F
Ibcast: BTDD (Binomial Tree Distance Doubling)	0 → 1 F 1 → 0 F 2 → 1 I 3 → 1 I	0 → 2 S 1 → 3 S 0 → 2 S 1 → 3 S 2 → 0 F 1 → 3 S 3 → 2 F 1 → 0 F	N/A
Ibcast: BTDH (Binomial Tree Distance Halving)	0 → 2 F 1 → 3 F 2 → 0 F 3 → 1 F	0 → 1 S 2 → 3 S 1 → 2 S 3 → 0 S 2 → 3 S 0 → 1 S 3 → 0 S 1 → 2 S	N/A
Iallgather: RD (Recursive Doubling)	0 → 1 F 1 → 0 F 2 → 3 F 3 → 2 F	0 → 2 D 1 → 3 D 2 → 0 D 3 → 1 D	N/A
Proposed *(congestion-aware)*	0 → 1 F 1 → 0 F 2 → 3 F 3 → 2 F	0 → 2 F 1 → 3 F 2 → 1 F 3 → 0 F	0 → 1 F 1 → 0 F 2 → 3 F 3 → 2 F

step is entirely congestion-free because each rank exchanges data solely with its NVLink-connected peer. However, in the subsequent step, each GPU transfers its own data plus the accumulated data from the previous step to a rank across the socket. While this strategy reduces the number of steps and is effective for smaller message sizes, it becomes increasingly inefficient as message sizes grow. As a result, to address these issues, we enhance intra-node communication by reordering selected operations. In our congestion-aware algorithm, each rank first shares its data with its NVLink-connected peer. We then construct a virtual ring ($0 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 0$) in which each rank forwards only its own data, reducing the inter-socket transfer volume by half. In the final step, NVLink-connected peers exchange the data received from across the socket. This method fully eliminates congestion at the cost of one additional step compared with the RD algorithm.

Figure 5 presents a relative comparison of the intra-node communication latency for the proposed congestion-aware algorithm. We quantify this comparison using the Congestion Impact Factor (CIF), which reflects how much slower each algorithm is relative to the minimum latency observed at each message size (i.e., the best-performing algorithm at that size). Our congestion-aware algorithm delivers latency improvements of 74.6%, 68.2%, 60.5%, 75.7%, 75.5%, and 40.5% over Ibcast with Linear, BTDD, BTDH, K-nomial, and Iallgather with Linear and RD algorithms, respectively. The maximum CIF observed across all cases is 4.57. Figure 5 also shows a similar comparison at the multi-node scale, where Parallel Rings are used for inter-node communication, and one of the evaluated algorithms is used for intra-node communication. In this setting, the maximum CIF decreases to 2.24, demonstrating the positive impact of overlapping inter-node and intra-node communication. However, the results also indicate that intra-node communication is not fully overlapped, underscoring the need for a topology- and congestion-aware intra-node algorithm. For our final evaluations in Section IV, we adopt the proposed congestion-aware algorithm for intra-node communication on inter-socket topologies. Note that, regardless of the intra-node topology, the Parallel Rings algorithm is used for inter-node communication.

Figure 6 shows how the receive buffer for each rank on a node is populated at a specific iteration (considering 16 GPUs across 4 nodes). Algorithm 2 presents the cluster-wide procedure for inter-socket systems. The calculation of outgoing and incoming neighbours for inter-node communication follows the same approach as in Algorithm 1 (Lines 8–10). Next, each rank determines the local and global ranks of its NVLink-connected peer (Lines 11–16). Then, each rank computes the local and global ranks for UPI-based communication for both sending and receiving. The send and receive peers differ because we form a virtual ring (bottom-right in Figure 6) (Lines 17–26). We further decompose the congestion-aware intra-node algorithm to account for dependencies between phases. Specifically, since the second and third phases are dependent, for each iteration of intra-node communication, the first and second phases are executed in the current iter-

Algorithm 2: Cluster-wide Parallel Rings Allgather with Inter-socket Intra-node Topology

Input : $sbuf, scount, sdtype, rbuf, rcount, rdtype, comm$
Output: All data gathered in $rbuf$ across all ranks

```
1  ompi_coll_base_parallel_rings_Allgather()
2  bufLDE[ localsize ];
3  bufLDEprev[ localsize ];
4  size ← comm.size() rank ← comm.rank()
5  if sbuf ≠ IN_PLACE then
6      Copy sbuf to rbuf[rank];
7  localrank, localsize ← precomputed values;
8  outgoing_neighbour ← (rank + localsize) mod size;
9  incoming_neighbour ← (rank − localsize + size) mod size;
10 new_comm_size ← size/localsize;
11 if rank mod 2 = 0 then
12     GlobalPeerNVLink ← rank + 1;
13     LocalPeerNVLink ← localrank + 1;
14 else
15     GlobalPeerNVLink ← rank - 1;
16     LocalPeerNVLink ← localrank - 1;
17 pos ← −1;
18 for k = 0 to localsize - 1 do
19     if L[k] = localrank then
20         pos ← k;
21         break;
22 LocalPeerUpiSend ← L[(pos + 1) mod localsize];
23 LocalPeerUpiRecv ← L[(pos − 1 + localsize) mod localsize];
24 nodeBase ← localsize * (rank / localsize);
25 GlobalPeerUpiSend ← LocalPeerUpiSend + nodeBase;
26 GlobalPeerUpiRecv ← LocalPeerUpiRecv + nodeBase;
27 tmprecv ← rbuf + (GlobalPeerNVLink * rcount);
28 tmpsend ← rbuf + (rank * rcount);
29 Irecv tmprecv from GlobalPeerNVLink;
30 Isend tmpsend to GlobalPeerNVLink;
31 tmprecv ← rbuf + (GlobalPeerUpiRecv * rcount);
32 tmpsend ← rbuf + (rank * rcount);
33 Irecv tmprecv from GlobalPeerUpiRecv;
34 Isend tmpsend to GlobalPeerUpiSend;
35 for i = 0 to new_comm_size - 2 do
36     recvdatafrom
       ← ( rank − (localsize ∗ (i + 1)) + size ) mod size;
37     senddatafrom ← ( rank − (i ∗ localsize) + size ) mod size;
38     for j = 0 to localsize - 1 do
39         bufLDE[j] ← ( (rank − localrank + j) − ((i + 1) ∗
             localsize) + size ) mod size;
40         bufLDEprev[j] ← ( (rank − localrank + j) − (i ∗
             localsize) + size ) mod size;
41     tmprecv ← rbuf + (recvdatafrom * rcount);
42     tmpsend ← rbuf + (senddatafrom * rcount);
43     Sendrecv tmpsend → outgoing_neighbour and receive tmprecv
         ← incoming_neighbour;
44     tmprecv ← rbuf + (bufLDEprev[LocalPeerUpiSend] * rcount);
45     tmpsend ← rbuf + (bufLDEprev[LocalPeerUpiRecv] * rcount);
46     Irecv tmprecv from GlobalPeerNVLink;
47     Isend tmpsend to GlobalPeerNVLink;
48     tmprecv ← rbuf + (bufLDE[LocalPeerNVLink] * rcount);
49     tmpsend ← rbuf + (bufLDE[localrank] * rcount);
50     Irecv tmprecv from GlobalPeerNVLink;
51     Isend tmpsend to GlobalPeerNVLink;
52     tmprecv ← rbuf + (bufLDE[LocalPeerUpiRecv] * rcount);
53     tmpsend ← rbuf + (bufLDE[localrank] * rcount);
54     Irecv tmprecv from GlobalPeerUpiRecv;
55     Isend tmpsend to GlobalPeerUpiSend;
```

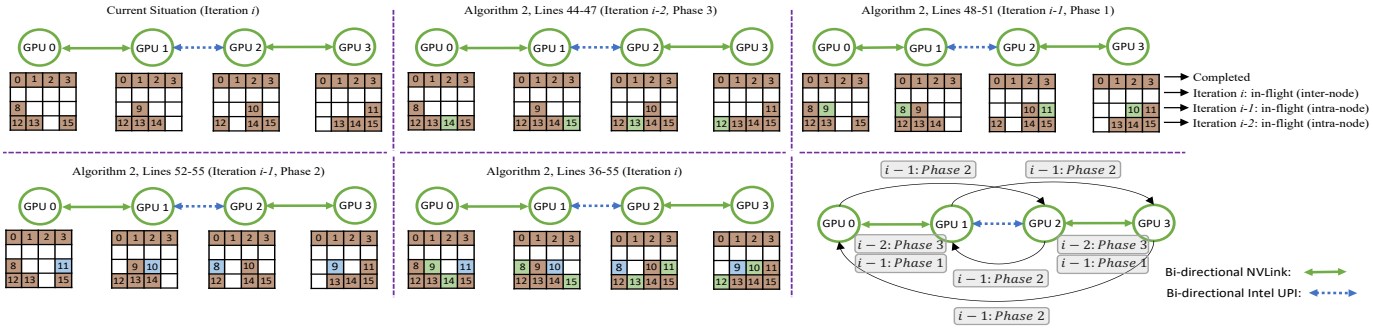


Fig. 6: Visual illustration of communication patterns in the proposed congestion-aware algorithm for inter-socket topologies. The correspondence between the lines of Algorithm 2 and the manner in which the *rbuf* is populated is shown.

ation, while the third step is deferred to the next iteration. This ensures that the required data for the third phase is available when needed. Consequently, Lines 27–34 implement the first and second phases of Stage 1, i.e., *LME*. Lines 36-40 calculate the indices of message segments for Stage 2 and Stage 3. Lines 41–43 implement the recurring Parallel Rings (Stage 2) for inter-node communication, overlapped with intra-node communication. During each iteration of inter-node communication, e.g., iteration i (top-left situation in Figure 6), the algorithm simultaneously executes the third phase of intra-node communication from iteration i–2 (Lines 44-47 in Algorithm 2 and top-middle in Figure 6), the first phase of iteration $i-1$ (Lines 48-51 in Algorithm 2 and top-right in Figure 6), and the second phase of iteration i–1 (Lines 52-55 in Algorithm 2 and bottom-left in Figure 6). In contrast to Algorithm 1, which overlaps two consecutive iterations to combine intra- and inter-node communication, Algorithm 2 overlaps decomposed communication phases from three consecutive iterations. This design maximizes overlap and improves overall performance on inter-socket topologies.

IV. Performance Evaluation and Analysis

A. Experimental Setup

We have used two clusters in this study: A and B. Cluster A features four A30 (24GB) GPUs per node, with two GPUs attached to each socket. GPUs within the same socket are interconnected via NVLink (NV4), while communication between GPUs across sockets traverses UPI. Cluster B is equipped with four NVIDIA H100 (80GB) GPUs per node, configured in a full-mesh NVLink topology (NV6). Both clusters use four high-speed InfiniBand ports per node, with cluster A providing 100Gb/s per port (two per socket) and cluster B providing 200Gb/s per port. In both configurations, each GPU has a dedicated inter-node communication channel. Cluster A uses two Intel Xeon Gold 6338 CPUs, whereas cluster B uses a single AMD EPYC 9454 processor. It is worth noting that our goal is not to compare clusters but to demonstrate the applicability of our design across a variety of topologies.

We compare our work against three widely used communication libraries: Open MPI, MVAPICH-Plus, and NCCL. Open MPI is an open-source, high-performance implementation of the Message Passing Interface (MPI) standard. In this study, we use Open MPI v4.1.8 with UCX 1.19.0, as it yielded more stable and performant baseline results compared to version 5. MVAPICH-Plus v4.1 is an advanced MPI library that unifies the capabilities of MVAPICH2-GDR and MVAPICH2-X. NCCL (v2.27.3) is a high-performance communication library optimized for NVIDIA GPUs. All three libraries evaluated in this work are GPU-aware and were tuned for maximum performance using their respective environment variables.

We evaluate our work at three different levels. At the microbenchmark level, we use Allgather from OMB. Because this study focuses on large message sizes, we select message ranges between 2MB and 256MB. On cluster A, experiments use 16 GPUs (maximum count available), while on cluster B, we extend the evaluation to 32 and 64 GPUs. We then assess our design using a distributed matrix multiplication application. The program operates on three matrices: A of size $M \times K$, B of size $K \times N$, and C of size $M \times N$, where $C = A \times B$. Matrices A and B are partitioned across all ranks. The program first performs an Allgather on matrix B so that every GPU obtains the full matrix. Each rank then computes the multiplication of its local portion of matrix A with matrix B to produce a partial result of the final output. A second Allgather is subsequently performed to assemble the complete result matrix across all ranks. All buffers in this application are allocated directly on the GPU. At each GPU count, the dimensions M, K, and N are chosen to ensure that both Allgather operations involve large message exchanges, consistent with the objectives of this work. The correctness of all microbenchmark experiments was verified using the `--validation` flag. We use 50 warm-up iterations followed by 200 measurement iterations.

At the application level, we evaluate our work using FSDP to train a neural network (using PyTorch v2.8.0 [11]). FSDP reduces GPU memory usage by sharding model parameters, gradients, and optimizer states across GPUs. The model used in our experiments follows a Transformer-style architecture composed of multiple identical blocks, each containing linear layers with residual connections and ReLU activations. The entire model is wrapped with FSDP, and a final linear projection layer processes the output of the stacked blocks. To study communication behaviour, we vary the hidden layer size across 4096, 8192, 16384, and 32768. The model is

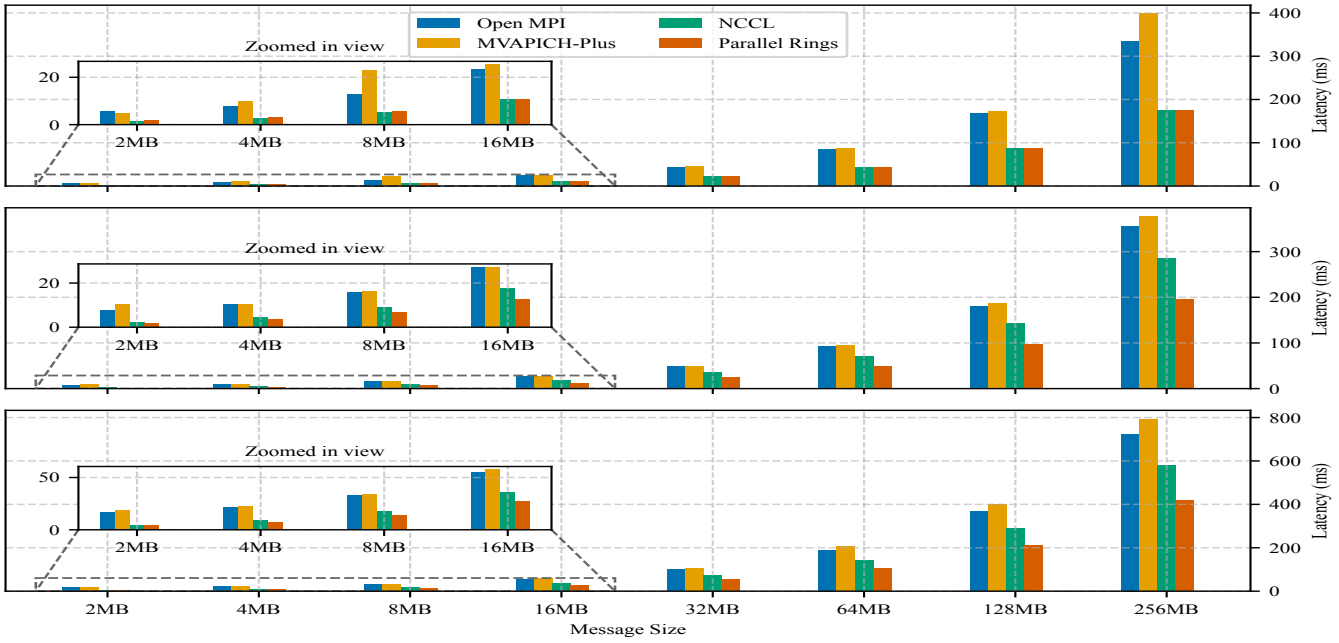


Fig. 7: Comparison of OMB Allgather latency between Open MPI, MVAPICH-Plus, NCCL, and the proposed Parallel Rings from top to bottom: cluster A (16 GPUs), cluster B (32 and 64 GPUs)

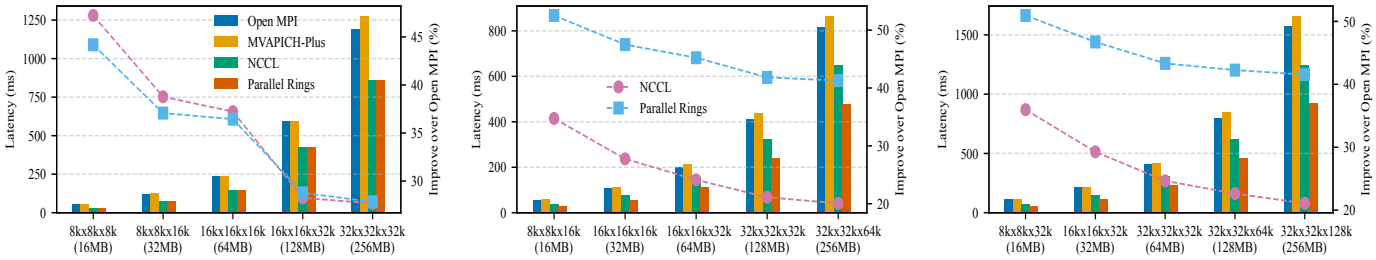


Fig. 8: Comparison of GEMM latency between Open MPI, MVAPICH-Plus, NCCL, and proposed Parallel Rings. Left to right: cluster A (16 GPUs), B (32 and 64 GPUs). X axis shows matrix size ($M \times K \times N$) and Allgather message size in parentheses.

trained for 100 iterations (batch size = 64). FSDP training involves two collectives: ReduceScatter (RS) and Allgather. For a fair comparison across communication libraries, we optimize the Open MPI implementation (not algorithm) of RS so that the impact of our work on Allgather performance can be clearly observed. In the baseline implementation, Open MPI cannot perform reduction directly on GPU-resident buffers. As a result, GPU buffers are first copied to host memory, reduced on the CPU, and the result is then transferred back to GPU memory. In addition, Open MPI allocates intermediate buffers along the critical execution path, which introduces non-negligible overhead. In our optimized RS implementation, all required intermediate buffers are pre-allocated, thereby removing dynamic memory allocation from the critical path. Moreover, the reduction is performed entirely on the GPU, which is significantly faster than the CPU and eliminates intermediate GPU↔CPU data transfers. These RS implementation optimizations are independent of, and orthogonal to, the proposed Allgather design; therefore, their performance benefits compound in the overall application-level results.

B. Microbenchmark and Kernel-Level Studies

Figure 7 presents the OMB Allgather latency comparison between our Parallel Rings design and the evaluated communication libraries. On cluster B with 32 GPUs, our approach achieves latency improvements of up to 74%, 80%, and 31% over Open MPI, MVAPICH-Plus, and NCCL, respectively. At 64 GPUs, the corresponding improvements are 73%, 76%, and 27%. On cluster A with 16 GPUs, our design outperforms Open MPI and MVAPICH-Plus by 68% and 75%, respectively. For NCCL on cluster A, our performance is comparable, achieving similar latency to NCCL. Figure 8 shows the latency comparison of Generalized Matrix Multiplication (GEMM) across Parallel Rings and other communication libraries. Each bar reports the measured program latency. The pink dashed line indicates the improvement of NCCL over Open MPI, while the blue dashed line represents the improvement achieved by Parallel Rings over Open MPI. As shown, the

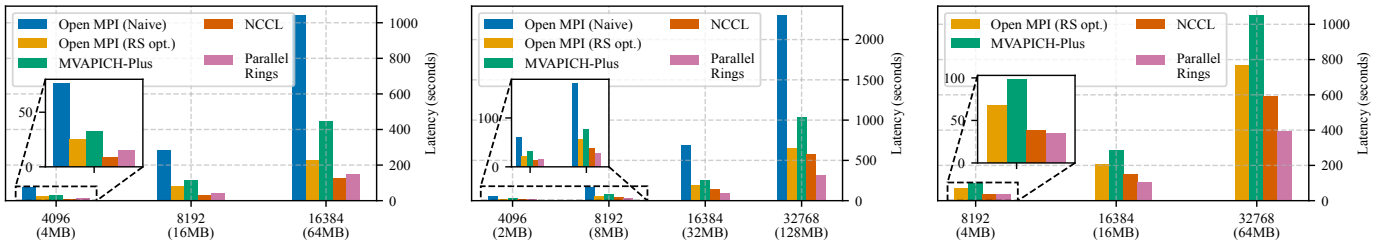


Fig. 9: Comparison of FSDP latency between Open MPI, MVAPICH-Plus, NCCL, and proposed Parallel Rings. Left to right: cluster A (16 GPUs), cluster B (32 and 64 GPUs). X axis shows the hidden layer size and Allgather message size in parentheses.

trend closely follows that of the Allgather microbenchmark, as communication remains the dominant cost in the GEMM workflow. The Allgather message size used in the program scales with the input matrix dimension, which is reported for each input case.

C. Application-Level Studies

Figure 9 reports the application-level results for FSDP. As noted at the end of Section IV-A, we also optimized the ReduceScatter collective to ensure a fair comparison. To illustrate the impact of this optimization, we include the naïve Open MPI (without RS optimization) on cluster A, and cluster B with 32 GPUs; the behaviour on cluster B with 64 GPUs follows the same trend. Parallel Rings in these figures also incorporate the RS optimization. The results show that RS optimization significantly reduces the overall training time, yet Parallel Rings still achieves lower latency, indicating that the two optimizations are orthogonal. On cluster A, Parallel Rings outperform Open MPI and MVAPICH-Plus by up to 45% and 66%, respectively, although it remains 16% slower than NCCL. On cluster B with 32 GPUs, the improvements reach 49%, 69%, and 46% compared to Open MPI, MVAPICH-Plus, and NCCL, respectively. For cluster B with 64 GPUs, the corresponding gains are 49%, 64%, and 33%.

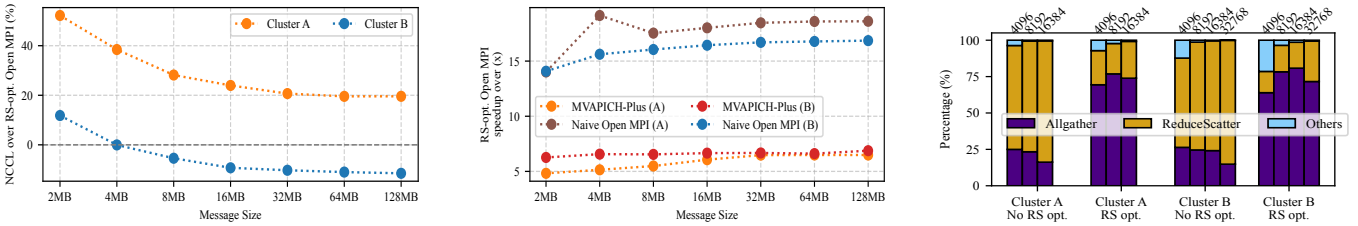
D. Results Analysis

In the following, we address several observations at the application layer. ❶ Performance improvements at the application layer are typically smaller than those observed in microbenchmarks. However, comparing the OMB and FSDP results for RS-optimized Open MPI shows that the improvements at the application layer remain high and closely track the microbenchmark trends. To understand this, we profiled the latency breakdown of FSDP on both clusters, with and without RS optimization, as shown in Figure 10c. Before RS optimization, ReduceScatter dominates FSDP's runtime; therefore, improvements in Allgather alone would have had a limited impact. After RS optimization, however, Allgather becomes the primary contributor to latency, which explains why the gains from Parallel Rings remain substantial and comparable to those observed in the microbenchmarks when evaluated against RS-optimized Open MPI. ❷ The improvement achieved by Parallel Rings over MVAPICH-Plus at the application layer is larger than the improvement observed in the microbenchmarks. This outcome arises because the benefits at the application layer are driven by both the Parallel Rings design and the RS optimization. As Figure 10b illustrates, our optimized ReduceScatter significantly outperforms MVAPICH-Plus. Consequently, the combination of the RS optimization and the improved Allgather performance amplifies the overall application-level improvement beyond what is captured in microbenchmarks, which isolate only the Allgather optimization. ❸ Why does NCCL achieve lower latency for FSDP on cluster A? FSDP's performance is determined primarily by ReduceScatter and Allgather. While our Parallel Rings implementation for Allgather closely matches NCCL on cluster A (because of inter-socket optimization and congestion-aware algorithm), and our general RS optimization performs well, ReduceScatter is not optimized in our design for inter-socket communication across UPI. As confirmed in Figure 10a, NCCL's ReduceScatter outperforms our current RS optimization in such cross-socket scenarios, explaining why NCCL achieves lower FSDP latency.

V. Discussion

A. Applicability to other Collectives

For large messages, Allreduce is implemented using Ring, Segmented Ring, and RSA (ReduceScatter followed by Allgather) schemes [17]. In both the Ring and Segmented Ring approaches, the communication pattern is identical to that of Allgather: each process with rank P sends data to rank $P+1$ and receives data from rank $P-1$. Consequently, these schemes exhibit the same HCA underutilization as Allgather and can therefore benefit from the proposed Parallel Rings technique. RSA consists of two phases. The ReduceScatter phase is implemented using recursive vector halving and recursive distance doubling (which takes $\log_2 P$ steps), followed by an Allgather phase based on recursive vector doubling and recursive distance halving (taking $\log_2 P$ steps). With this communication pattern, and considering, for example, a configuration with four GPUs per node, only two of the total $2 * \log_2 P$ communication steps are intra-node. For large GPU counts, where $2 * \log_2 P \gg 2$, the majority of the communication steps are inter-node. As a result, all GPUs within a node actively utilize their corresponding HCAs, and no HCA remains underutilized. When both the ReduceScatter and Allgather phases of RSA are implemented using a ring-based algorithm, as is the case in NCCL, the proposed Parallel Rings optimization is directly applicable. In Open MPI, Re-



(a) Comparison of ReduceScatter benchmark (NCCL vs. RS-opt. Open MPI). Cluster A with 16 and cluster B with 32 GPUs

(b) Comparison of ReduceScatter benchmark (RS-opt. Open MPI vs MVAPICH-Plus and naïve Open MPI)

(c) FSDP latency breakdown before and after RS optimization

Fig. 10: Detailed application-level analyses

duceScatter for large messages is implemented using either the Ring or Butterfly algorithms. The Ring-based implementation follows the same communication pattern as Allgather. This leads to HCA underutilization and is therefore well-suited to the proposed Parallel Rings optimization. The Butterfly algorithm uses recursive distance doubling and, after the first two steps for large GPU counts, fully utilizes all HCAs.

B. Overlapping Effect

Overall performance is governed by the extent to which intra-node communication can be overlapped with inter-node communication, which is influenced by the underlying intra-node topology and the selected intra-node algorithm. On the full-mesh topology, inter-node and intra-node communications progress concurrently, resulting in full overlap of intra-node transfers, whereas on the inter-socket topology, the achievable overlap depends on the specific intra-node algorithm, with some algorithms enabling substantially higher overlap and therefore incurring lower bottlenecks. Consequently, the observed performance differences between clusters A and B stem from the combined effects of HCA bandwidth and intra-node topology. As long as the full-mesh topology is used, the bandwidth ratio between intra-node and inter-node communication channels has only a limited impact on the overall performance of hierarchical collectives. Although the balance between intra-node and inter-node bandwidth can influence performance, this effect is rarely observed on full-mesh platforms, since modern intra-node interconnects such as NVLink provide at least four times higher bandwidth than the inter-node network. This advantage can be substantially diminished in architectures that rely on inter-socket communication links, such as UPI, for intra-node data movement, which significantly reduces effective intra-node performance.

VI. Related Works

A. Multi-HCA Allgather

A recent study [18] has proposed hierarchical, Multi-HCA-Aware (MHA) designs for the Allgather collective in MPI, targeting performance optimization in CPU-only systems with multi-rail networks. Motivated by the imbalance between inter-node bandwidth and limited intra-node shared-memory bandwidth, these works address inefficiencies in traditional collectives that underutilize available HCAs. Their approach consists of two components: the MHA-intra design, which leverages idle HCAs to accelerate intra-node data movement, and the hierarchical MHA-inter design, which overlaps inter-node communication with intra-node transfers to alleviate bandwidth disparities. Although evaluated on systems like Argonne's ThetaGPU, the focus remains on addressing CPU-side communication bottlenecks rather than GPU acceleration. Suresh et al. [19] presented basic and optimized multi-rail-aware algorithms for MPI Allgather and Alltoall on modern multi-HCA GPU clusters, aiming to improve network utilization at the collective level. The evaluation is limited to micro-benchmarks and message sizes below 1 MB, and the designs do not take GPU topology into account.

B. Other Multi-HCA Collectives

Yu et al. proposed Nezha [20]. The Nezha system addresses scalability limitations in distributed systems by providing full-stack support for allreduce on multi-rail networks, integrating protocols such as TCP and the in-network computing protocol SHARP. Nezha facilitates the scheduling of multi-rail networks at the software stack level, incorporating a load-balancing data allocation scheme based on cost feedback to maximize data transfer rates. Experimental results demonstrated 58% to 87% improvement in homogeneous dual-rail configurations. Chen et al. [21] focused on optimizing specific collective operations within heterogeneous environments, proposing unified designs for multi-rail-aware MPI Allreduce and Alltoall operations across diverse modern HPC systems. These designs utilize a multi-leader two-level approach, incorporating a persistent GPU buffer for device-side reduction and leveraging inter-process communication techniques to form a shared region on the GPU for Alltoall.

C. RDMA-Level Multi-Path Network Communication

Qian and Afsahi [22] designed efficient RDMA-based multi-port collective communication schemes on multi-rail Quadrics QsNet^{II}, addressing the lack of native multi-rail collective support. Their early work introduced message striping at the Elan library level for operations such as Scatter, Gather, and Alltoall, achieving up to 6.35× improvement for large messages. They later extended this design to Allgather, evaluated various algorithms and improved short-message performance on SMP

clusters by proposing SMP-aware Allgather variants that combined shared memory for intra-node and multi-port RDMA for inter-node communication. Unlike collective-level multi-HCA approaches, other works [23–25] focus on exploiting multiple network paths for a single RDMA connection in data center networks. These user-level, software-based solutions—such as Virtuoso [24] and UL-MPRDMA [25] address the limitations of single-path RDMA by enabling multi-path load balancing and reliability without requiring specialized hardware. Virtuoso splits large RDMA messages across multiple virtual interfaces, while UL-MPRDMA partitions flows over multiple Queue Pairs mapped to different physical paths, with flow-size–aware scheduling and active NIC resource management to optimize performance. These approaches demonstrate effective utilization of multi-path topologies and reduce flow completion times compared to traditional single-path RDMA.

VII. Conclusion

The traditional ring algorithm, widely employed in collective communications such as Allgather, underutilizes the full bandwidth potential of modern multi-HCA clusters. In this work, we redesigned the ring algorithm and introduced Parallel Rings, a topology-aware, hierarchical communication design that simultaneously leverages all available HCAs, improves scalability proportionally to the number of HCAs, and enables effective overlap between intra- and inter-node communication phases. We further demonstrated the importance of selecting an efficient intra-node communication and proposed a congestion-aware method tailored for inter-socket topologies. We evaluated our design on two distinct clusters, across three benchmarking levels, and against three state-of-the-art communication libraries. Our results show that Parallel Rings reduce Allgather latency by up to 74%, 80%, and 31% relative to Open MPI, MVAPICH-Plus, and NCCL, respectively.

Acknowledgment

We acknowledge the support of Digital Research Alliance of Canada and Natural Sciences and Engineering Research Council of Canada (NSERC), [ALLRP 578539-22].

References

[1] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[2] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik, "Scaling Distributed Machine Learning with In-Network Aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 785–808.

[3] D. De Sensi, E. C. Molero, S. Di Girolamo, L. Vanbever, and T. Hoefler, "Canary: Congestion-Aware In-Network Allreduce using Dynamic Trees," *Future Generation Computer Systems*, vol. 152, pp. 70–82, 2024.

[4] W. Wang, M. Ghobadi, K. Shakeri, Y. Zhang, and N. Hasani, "Rail-only: A Low-Cost High-Performance Network for Training LLMs with Trillion Parameters," *arXiv preprint cs.NI/2307.12169*, 2024.

[5] Z. Li, L. Xu, Z. Huang, S. Qian, H. Bu, M. Yang, M. Luan, W. Chen, and X. Wen, "CTCCL: Cost-Efficient Joint Device-Network Load Balancing for LLM Training in RoCE-based Intelligent Computing Network," in *Proceedings of the 39th ACM International Conference on Supercomputing*, 2025, pp. 355–367.

[6] List of Top 500 supercomputers - June 2025. [Online]. Available: https://www.top500.org/lists/top500/2025/06/

[7] "Open MPI: A High Performance Message Passing Library." [Online]. Available: https://www.open-mpi.org/

[8] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, "UCX: An Open Source Framework for HPC Network APIs and Beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.

[9] H. Zhou, J. Gracia, and R. Schneider, "MPI Collectives for Multi-core Clusters: Optimized Performance of the Hybrid MPI+MPI Parallel Codes," in *Workshop Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[10] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang *et al.*, "RDMA over Ethernet for Distributed Training at Meta Scale," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 57–70.

[11] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer *et al.*, "Pytorch FSDP: Experiences on Scaling Fully Sharded Data Parallel," *arXiv preprint arXiv:2304.11277*, 2023.

[12] "MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, RoCE, and Slingshot." [Online]. Available: https://mvapich.cse.ohio-state.edu/

[13] "NVIDIA Collective Communication Library." [Online]. Available: https://developer.nvidia.com/nccl

[14] "OSU Micro-Benchmarks 7.5.1." [Online]. Available: https://mvapich.cse.ohio-state.edu/benchmarks/

[15] Z. Hu, S. Shen, T. Bonato, S. Jeaugey, C. Alexander, E. Spada, J. Dinan, J. Hammond, and T. Hoefler, "Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms," in *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2025, pp. 48–59.

[16] A. Shah, V. Chidambaram, M. Cowan, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, O. Saarikivi, and R. Singh, "TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 593–612.

[17] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[18] T. Tran, B. Ramesh, B. Michalowicz, M. Abduljabbar, H. Subramoni, A. Shafi, and D. K. Panda, "Accelerating communication with multi-HCA aware collectives in MPI," *Concurrency and Computation: Practice and Experience*, vol. 36, no. 1, p. e7879, 2024.

[19] K. K. Suresh, A. P. Guptha, B. Michalowicz, B. Ramesh, M. Abduljabbar, A. Shafi, H. Subramoni, and D. Panda, "Efficient Personalized and Non-Personalized Alltoall Communication for Modern Multi-HCA GPU-based Clusters," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2022, pp. 100–104.

[20] E. Yu, D. Dong, and X. Liao, "Full-Stack Allreduce on Multi-Rail Networks," *arXiv preprint arXiv:2405.17870*, 2024.

[21] C.-C. Chen, J. Yao, L. Xu, H. Subramoni, and D. K. Panda, "Unified Designs of Multi-Rail-Aware MPI Allreduce and Alltoall Operations Across Diverse GPU and Interconnect Systems," in *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025, pp. 938–949.

[22] Y. Qian and A Afsahi, "RDMA-based and SMP-aware Multi-port All-Gather on Multi-rail QsNet II SMP Clusters," in *2007 International Conference on Parallel Processing (ICPP 2007)*. IEEE, 2007, pp. 48–57.

[23] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda, "Multi-Path Transport for RDMA in Datacenters," in *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, 2018, pp. 357–371.

[24] F. Tian, W. Feng, Y. Zhang, and Z.-L. Zhang, "A Novel Software-based Multi-path RDMA Solution for Data Center Networks," *arXiv preprint arXiv:2009.00243*, 2020.

[25] S. Lee, Y. Kim, H. Woo, and I. Yeom, "Efficient User-Level Multi-Path Utilization in RDMA Networks," *IEEE Access*, vol. 9, pp. 127 619–127 629, 2021.