The Big Send-off: Scalable and Performant Collectives for Deep Learning

Siddharth Singh
Department of Computer Science
University of Maryland
College Park, USA
ssingh37@umd.edu

Keshav Pradeep
Department of Computer Science
University of Maryland
College Park, USA
keshprad@umd.edu

Mahua Singh
Dept. of Comp. Sci. and Engg.
Indian Institute of Technology
Guwahati, India
s.mahua@iitg.ac.in

Cunyang Wei
Department of Computer Science
University of Maryland
College Park, USA
cunyang@umd.edu

Abhinav Bhatele
Department of Computer Science
University of Maryland
College Park, USA
bhatele@cs.umd.edu

***Abstract*—Collective communication is becoming increasingly important in data center and supercomputer workloads with an increase in distributed AI related jobs. However, existing libraries that provide collective support such as NCCL, RCCL, and Cray-MPICH exhibit several performance and scalability limitations on modern GPU supercomputers. To address these challenges, we introduce the Performant Collective Communication Library (PCCL), specifically targeted for distributed deep learning (DL) workloads. PCCL provides highly optimized implementations of key collectives used in distributed DL: all-gather, reduce-scatter, and all-reduce. PCCL uses a hierarchical design with learning-based adaptive selection of the best performing algorithms to scale efficiently to thousands of GPUs. It achieves substantial performance speedups over RCCL on 2048 GCDs of Frontier – up to 168× for reduce-scatter, 33× for all-gather and 10× for all-reduce. More modest but still significant gains up to 5.7× over NCCL are observed on Perlmutter. These gains translate directly to performance improvement of production DL workloads: up to 4.9× speedup over RCCL in DeepSpeed ZeRO-3 training, and up to 2.4× speedup in DDP training.**

***Index Terms*—collective communication, distributed deep learning, hierarchical collectives, GPUs**

I. Motivation

Communication overheads in parallel applications become an increasing fraction of the overall execution time as these applications are scaled to more and more nodes and GPUs. In the case of deep learning (DL) applications [1]–[8], communication is typically composed of collective operations such as all-gather, all-reduce, and reduce-scatter. MPI and vendor-specific libraries such as NCCL from NVIDIA and RCCL from AMD provide implementations of such operations for use in DL frameworks. As opposed to traditional high performance computing (HPC) applications, the message sizes in DL applications are significantly larger, from tens to hundreds of MBs, and current implementations of collective libraries struggle with maintaining high performance in this regime.

In this work, we analyze the efficiency and scaling limitations of existing communication libraries such as Cray-MPICH, NCCL and RCCL on multiple platforms. We examine three collectives that predominantly make up the communication in DL workloads: all-gather, reduce-scatter, and all-reduce. We discover that there are unique shortcomings, inefficiencies, and scalability issues in each of the messaging libraries. Primarily, we observe that all libraries struggle with efficiency and scalability in regimes of moderate message sizes (tens of MBs) and large GPU counts.

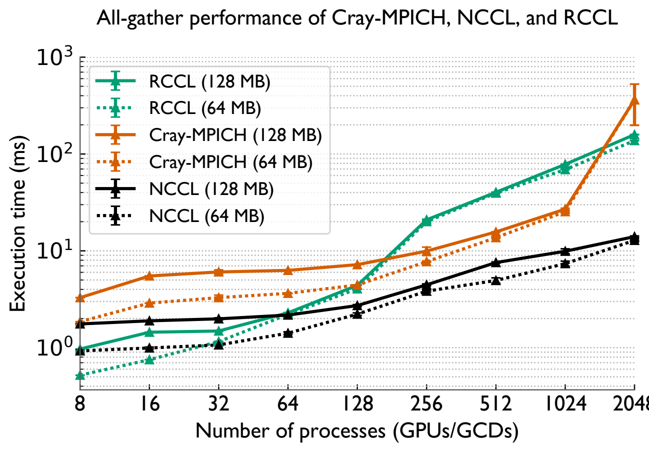


Fig. 1. Performance comparison of all-gather using RCCL (Frontier), Cray-MPICH (Frontier), NCCL (Perlmutter) for 64 and 128 MB output buffer sizes. The ideal scaling behavior (flat horizontal line) is not achieved by either library, highlighting their limited scalability at increasing GPU counts.

Figure 1 presents the results of benchmarking the all-gather operation on Frontier (AMD MI250X GPUs) and Perlmutter (NVIDIA A100 GPUs) with two output buffer (message) sizes: 64 and 128 MB using Cray-MPICH, NCCL, and RCCL. We observed that Cray-MPICH is the slowest on lower GPU/GCD counts but RCCL scales the most poorly beyond hundreds of GCDs. The ideal scaling curve is a horizontal line and none of the libraries are able to achieve that. This suggests that there is a significant performance gap that can be closed, to improve the performance of large-scale DL workloads.

In this paper, we introduce the Performant Collective Communication Library (PCCL), designed to accelerate collective operations – specifically all-gather, reduce-scatter, and all-reduce – for large buffer sizes (>10 MB) found in parallel deep learning workloads. Our design exploits hierarchical algorithms, use of CUDA kernels for reductions, and adaptive selection between different algorithmic choices to provide the best performance possible.

We evaluate PCCL on Frontier and Perlmutter, dragonfly-based supercomputers with AMD MI250X and NVIDIA A100 GPUs respectively, demonstrating its efficiency and scalability on both systems. Our implementations of all-gather, reduce-scatter, and all-reduce achieve significant speedups over all three libraries – Cray-MPICH, NCCL, and RCCL. The optimized collectives in PCCL thus pave the way for scalable performance of large-scale deep learning workloads on next-generation GPU supercomputers.

The paper makes the following key contributions:

- We analyze the limitations of existing communication libraries – Cray-MPICH, NCCL, and RCCL on Perlmutter and Frontier for all-gather and reduce-scatter operations in parallel deep learning workloads.
- We develop optimized implementations of all-gather, reduce-scatter, and all-reduce in PCCL, with a focus on effectively utilizing system resources and ensuring scalability for large messages and GPU counts.
- We demonstrate substantial performance speedups over RCCL on 2048 GCDs of Frontier – up to 168× for reduce-scatter, 33× for all-gather and 10× for all-reduce. More modest but still significant gains up to 5.7× over NCCL are observed on Perlmutter.
- We benchmark the performance of multi-billion-parameter LLM training workloads to validate the practical benefits of our optimizations, demonstrating significant speedups in training throughput: up to 4.9× speedup over RCCL in DeepSpeed ZeRO-3 training, and up to 2.4× speedup over RCCL in DDP training.

II. Background on Collective Operations

In this section, we provide relevant background on the role of collective communication in distributed DL workloads.

A. Collective Communication in Distributed Deep Learning

Sharded data parallelism (SDP) and distributed data parallelism (DDP) are widely used for large scale training [1], [2], [9]. They are briefly described below.

Collective Communication in Sharded Data Parallelism: In SDP, Model parameters and gradients are partitioned (or "sharded") across GPUs, which significantly reduces memory requirements and allows for the training of extremely large models. Two critical collective communication operations – *all-gather* and *reduce-scatter* – play a central role. These operations aggregate distributed data across GPUs: the all-gather operation collects model parameters from all shards to form a complete copy, while the reduce-scatter operation performs a reduction and distributes gradients across participating processes. In Figure 2, we plot the all-gather and reduce-scatter message sizes for three frameworks that support sharded data parallelism – FSDP [2], DeepSpeed ZeRO-3 [1], and AxoNN [10]. Unlike FSDP and ZeRO-3, AxoNN performs all-gathers and reduce-scatters for each linear layer separately, which results in a wide range of buffer sizes. Notice how the message sizes across these three frameworks are in the tens to hundreds of megabytes, even becoming more than a gigabyte for larger models.

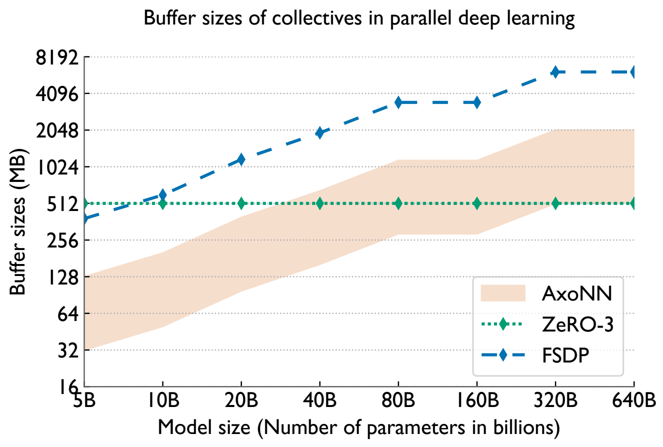


Fig. 2. Distribution of all-gather and reduce-scatter message sizes for several deep learning frameworks for a range of transformer [11] model sizes. The y-axis represents input buffer sizes for all-gathers but output buffer sizes for reduce-scatters.

Collective Communication in Distributed Data Parallelism: In DDP, model parameters are replicated across GPUs, and the *all-reduce* collective is critical for synchronizing parameter replicas. During training, each GPU performs forward and backward passes on a distinct local mini-batch of the input dataset, producing gradients specific to each GPU's mini-batch. After the backward pass, DDP invokes all-reduce to average the gradients across GPUs, and each GPU performs a local parameter update to keep all replicas in sync. Note that the all-reduce message sizes are directly proportional to the number of model parameters. For example, training a 1B-parameter model requires an all-reduce on 4 GB of gradients per iteration (one FP-32 gradient scalar per parameter). To reduce latency and improve overlap with computation, PyTorch's DDP framework splits this large all-reduce into several smaller all-reduces with sizes ranging from 48–80 MB [4], and overlaps them with the backward pass compute.

B. Algorithms for Dissolving Different Collectives

Efficient implementations of all-gather, reduce-scatter, and all-reduce operations are critical for distributed deep learning.

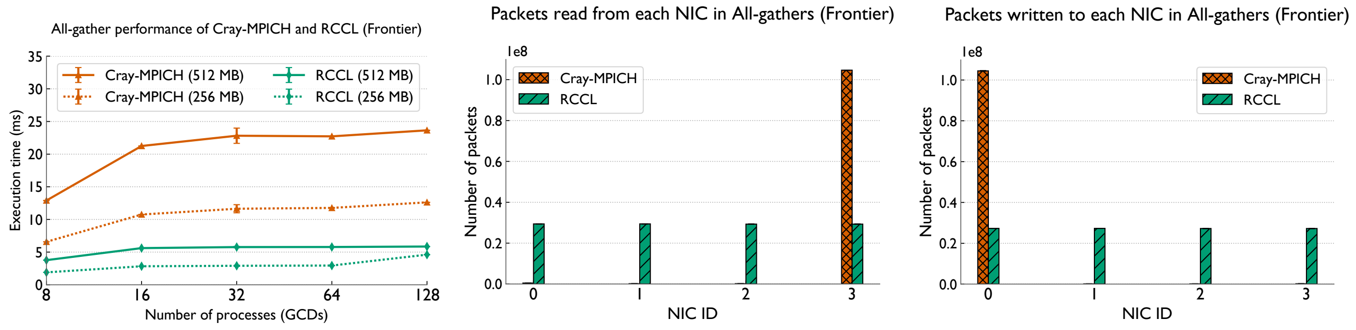


Fig. 3. All-gather performance of Cray-MPICH and RCCL on Frontier for a bandwidth-bound scenario with large message sizes (256 and 512 MB) and small GPU counts (left). The middle and right plots demonstrate the number of packets read from (middle) and written to (right) each of the four NICs on a Frontier compute node during all-gather operations.

In this work, we build on popular algorithms and introduce enhancements to them to improve performance and scalability.

Ring: The ring algorithm is a popular method for implementing collective communications due to its simplicity and efficiency in certain network topologies. In a ring-based collective operation, each process communicates with its immediate neighbors in a circular fashion. While effective at moderate scales and large message sizes, the ring algorithm can suffer from inefficiencies at larger scales due to its latency term being linearly proportional to the number of processes. The communication time of a ring all-gather can be modeled as,

$$T_{\text{ring}} = \overset{\text{startup latency}}{\alpha} \times (\overset{\text{number of processes}}{p} - 1) + \underset{\text{inverse of bandwidth}}{\beta} \times \frac{p-1}{p} \underset{\text{buffer size}}{m} \tag{1}$$

where p is the number of processes, α represents the startup latency per message, m is the size of the output buffer on each GPU, and β is the inverse of the peer-to-peer bandwidth.

Recursive Halving/Doubling: A popular way of minimizing latency costs involves utilizing the recursive halving or doubling algorithms for all-gathers and reduce-scatters respectively. These algorithms divide the communication task into logarithmically many steps, and hence their performance scales better than the ring algorithm. The communication time for a recursive-doubling all-gather can be modeled as,

$$T_{\text{rec}} = \overset{\text{startup latency}}{\alpha} \times \log_2(\overset{\text{number of processes}}{p}) + \underset{\text{inverse of bandwidth}}{\beta} \times \frac{p-1}{p} \underset{\text{buffer size}}{m} \tag{2}$$

For small message sizes or very large process counts, the logarithmic growth in the latency term often leads to lower overall communication costs. Building on these primitives, an all-reduce can be implemented as a reduce-scatter followed by an all-gather operation. More details about these and other algorithms can be found in Thakur et al. [12].

III. Current State of Communication Libraries

We begin with investigating the current state of popular communication libraries – Cray-MPICH, NCCL, and RCCL, on Perlmutter and Frontier. We find unique issues that limit the performance of each library, and highlight these below.

A. Benchmarking Setup

To ensure a fair comparison, we follow best practices for benchmarking all three libraries – optimal process placement, NUMA-aware NIC binding, disabling eager messaging, and enabling GPU Direct RDMA [13].

Message Sizes: In line with Figure 2, our evaluation in this section focuses on two message sizes – 256 MB and 512 MB. Note that for all-gathers and reduce-scatters, these values refer to the output and input message size per GPU respectively.

Software Stack: On Frontier, our software stack comprises of ROCm 6.2.4, RCCL 2.20.5, Cray-MPICH 8.1.31, libfabric 1.15.2 and the aws-ofi-rccl plugin version v1.4. On Perlmutter, we use CUDA 12.4, NCCL 2.24.3, Cray-MPICH 8.1.30, and libfabric 1.22.0.

For each combination of library, collective, message size, and GPU count, we perform ten independent trials. We measure the total time spent in the collective during each run using AMD's `hipeventtimers` instrumentation on Frontier, and NVIDIA's `cudaEventElapsedTime` on Perlmutter. All reported timings are end-to-end measurements that include any necessary data movement and transformation costs, including intra-GPU transposes required by hierarchical algorithms. We compute the mean and standard deviation over the ten trials to ensure statistical robustness.

B. Resource Under-utilization in Cray-MPICH

Figure 3 (left) presents a comparative analysis of all-gather performance between Cray-MPICH and RCCL on Frontier for message sizes of 256 MB and 512 MB. We observe that RCCL achieves approximately a $4\times$ performance advantage in this bandwidth-bound scenario. To explain this disparity, we examine two hardware performance counters provided by the Cassini Slingshot-11 Network Interface Controllers

(NICs) [14] - `parbs_tarb_pi_posted_pkts` and `parbs_tarb_pi_non_posted_pkts`. These counters represent the count of packets read from and written to each NIC within a node during job execution, respectively. As shown in Figure 3 (middle, right), we observe a consistent trend across all runs – Cray-MPICH exclusively utilizes NIC-0 for all write operations, and NIC-3 for all read operations. Conversely, RCCL distributes its read and write operations evenly across all four NICs on the node. This underutilization of available network resources by Cray-MPICH explains the $4\times$ performance gap we observe in all-gather operations.

Moreover, for collectives that require computation, such as reduce-scatters and all-reduces, Cray-MPICH suffers from another performance bottleneck – it performs the reduction operations on the CPU instead of offloading them to the GPU. This introduces significant computational overheads, as shown in Figure 4. Cray-MPICH (orange) performs significantly worse than RCCL (green). Notably, this performance gap is far greater than the $4\times$ difference observed earlier for all-gather in Figure 3.

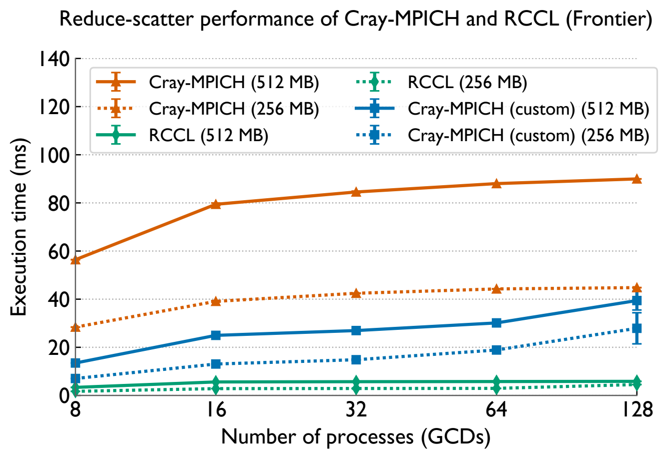


Fig. 4. Performance comparison of reduce-scatter using Cray-MPICH, RCCL, and a custom implementation of reduce-scatter that uses MPI point-to-point primitives and GPU compute kernels.

Observation 1

Cray-MPICH severely underutilizes available network (NIC) and computational (GPU) resources. It routes all network traffic through a single NIC, and performs reduction operations on the CPU, instead of offloading them to the GPU.

While the NIC underutilization issue outlined previously still persists for Cray-MPICH reduce-scatter operations, it alone cannot explain this performance disparity. We hypothesize that this disparity stems from Cray-MPICH's choice to perform reductions on the CPU instead of the more powerful GPUs. To validate this, we manually implemented the reduce-scatter operation using MPI point-to-point primitives and a GPU vector reduction kernel. As shown in Figure 4, our implementation (blue line) achieves performance that is several times faster than Cray-MPICH's native reduce-scatter, further supporting our hypothesis.

C. Poor Scaling of NCCL and RCCL at Large GPU Counts

Performance limitations of Cray-MPICH have made NCCL and RCCL popular for distributed deep learning, but as we demonstrate next, these libraries have algorithmic limitations that prevent them from scaling efficiently to large GPU counts. Let us again look at all-gather performance of RCCL (Frontier) and NCCL (Perlmutter) in Figure 1. We observe that both of these libraries scale extremely poorly. On investigating deeper, we found that NCCL and RCCL only support the ring algorithm for all-gather and reduce-scatter (refer to Section II-B). While effective for bandwidth-bound workloads, the ring algorithm performs poorly in latency-sensitive scenarios because each process must send and receive $(p-1)$ messages sequentially, causing the total communication time to grow linearly with the number of processes.

Notably, while NCCL and RCCL implement a logarithmic latency scaling algorithm for all-reduce based on the double-binary tree structure [15], for all-gather and reduce-scatter, they lack log latency scaling algorithms such as recursive doubling or halving (refer to Section II-B). These are known to reduce the number of communication steps to $\log_2 p$ and are generally preferred for small message sizes or high process counts. NCCL has recently introduced a logarithmic scaling algorithm for all-gather and reduce-scatter called PAT. However, at the time of writing, it only supports one GPU per node [16]. This lack of algorithmic diversity directly contributes to the sub-optimal scaling we observe at large GPU counts.

Observation 2

NCCL and RCCL rely solely on the ring algorithm for all-gather and reduce-scatter, leading to poor scaling in latency-bound scenarios. More efficient algorithms such as recursive doubling and halving are not supported.

IV. Design of Scalable Collectives in PCCL

In Section III, we identified several challenges affecting Cray-MPICH, NCCL, and RCCL, in the context of collective communication for deep learning workloads. These challenges create significant barriers to efficiently scaling DL workloads across thousands of GPUs. We now present the Performant Collective Communication Library (PCCL), which addresses these challenges through a three-pronged approach.

First, PCCL provides the option to directly call existing collective libraries (Cray-MPICH, NCCL, or RCCL) when they perform well for a given message size and GPU count. Second, PCCL provides new latency-optimized implementations of collectives for scenarios where existing libraries are sub-par – specifically, training at large-scale, as identified in Section III. These new implementations employ a hierarchical

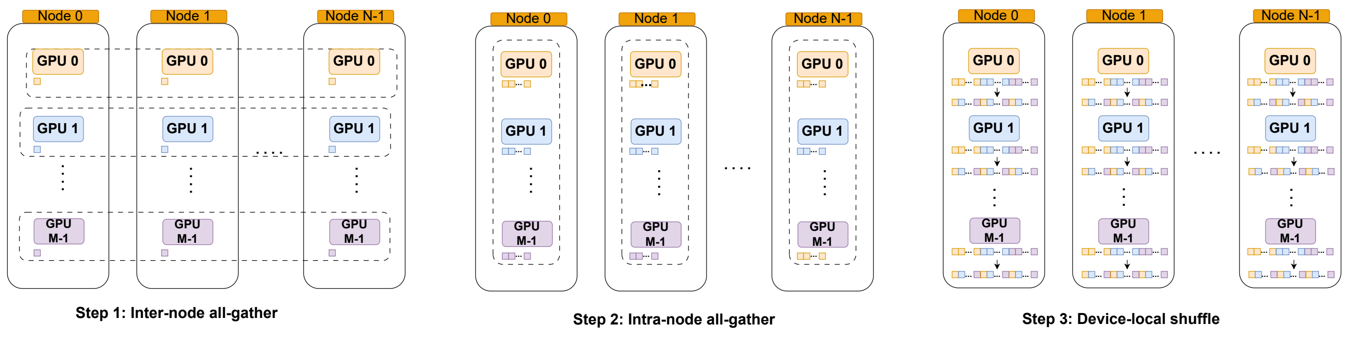

Node 0
Node 1
Node N-1
GPU 0
GPU 1
GPU M-1
Step 1: Inter-node all-gather
Step 2: Intra-node all-gather
Step 3: Device-local shuffle

Fig. 5. Diagram showing our hierarchical (two-level) implementation to dissolve an all-gather operation on a GPU-based cluster with N nodes and M GPUs per node. In step 1, we perform inter-node all-gathers, in step 2, we perform intra-node all-gathers, and in step 3, each GPU performs a local shuffle of the received data.

intra-node and inter-node design with two inter-node algorithm backends: `PCCL_ring` (ring-based) and `PCCL_rec` (recursive doubling/halving). Third, PCCL includes a learning-based adaptive dispatcher that automatically selects the most performant option at runtime based on the specific workload characteristics. The dispatcher can choose from any of the available backends: the existing libraries (Cray-MPICH, NCCL, RCCL) or the new proposed hierarchical collectives – (`PCCL_ring` and `PCCL_rec`). This design allows PCCL to achieve strong performance across the full spectrum of configurations—from small-scale runs with large messages to large-scale runs with small messages. Below, we discuss the design of our new latency-optimized collective implementations, followed by a detailed description of the learning-based adaptive dispatcher.

A. *Hierarchical Collective Algorithms for Scalability*

Our optimized implementations of all-gather, reduce-scatter, and all-reduce are based on a two-level hierarchical design. While our primary motivation is to address the underutilization of NICs (identified in Section III-B), this design also reduces latency and improves scalability [17], [18].

We illustrate our design in Figure 5 for an all-gather operation on a hypothetical system with N nodes and M GPUs per node. The global collective operation to be performed across all GPUs is divided into two distinct phases (inter-node and intra-node) using sub-communicators. Inter-node sub-communicators are formed by grouping together corresponding GPUs (with the same local ID) across nodes. For example, in Figure 5, all GPUs with the same within-node ID are grouped together to create a total of M inter-node sub-communicators. Similarly, intra-node sub-communicators are formed by grouping together all GPUs within a node to create N intra-node sub-communicators.

The hierarchical communication for dissolving the collective unfolds in three steps. First, we schedule concurrent all-gather operations within all inter-node sub-communicators (Step 1 of Figure 5). After the completion of this phase, each GPU has received data from its counterparts in every other node, yielding a partial result distributed across GPUs within each node. Second, we perform an intra-node all-gather (Step 2 of Figure 5), after which each GPU holds the complete output in memory, albeit in an incorrect order. Third, a device-local shuffle (Step 3 of Figure 5) rearranges each GPU's data into the correct order; in practice, this is implemented as a transpose kernel. We implement reduce-scatter similarly, starting with the intra-node phase followed by the inter-node phase. All-reduce is implemented by composing a two-level reduce-scatter with a two-level all-gather.

This design addresses NIC under-utilization by scheduling all inter-node all-gathers in Step 1 concurrently. On Frontier, each node has four NICs connected to two GCDs each. We ensure each GCD exclusively uses its corresponding NIC (e.g., GCDs 0 and 1 use NIC 0, GCDs 2 and 3 use NIC 1, etc.), evenly distributing inter-node traffic across all NICs. While our hierarchical design explicitly exploits the intra-node topology by mapping GPUs to their respective NICs, we do not explicitly map our communication patterns to the inter-node network topology. On Dragonfly-based systems such as Frontier and Perlmutter, UGAL routing [19] distributes inter-node traffic across the network, reducing the need for topology-aware collective mapping.

Choice of Communication Libraries: Our choice of libraries for each level of the hierarchy is driven by performance and reliability. For inter-node communication, we use MPI due to RCCL failures at scale, as reported in prior work [20] and noted by HPE[1] and the OLCF User Guide[2]. Moreover, we find that Cray-MPICH exhibits lower performance variability than RCCL on Slingshot interconnects, further motivating its use at the inter-node level.

For intra-node communication, we use GPU-vendor libraries (NCCL or RCCL) as they are highly optimized for intra-node communication, efficiently utilizing shared memory, PCIe, and Infinity Fabric or NVLink connections [13]. NCCL and RCCL only support the ring algorithm for intra-node all-gather and reduce-scatter. Since the ring algorithm is

[1]https://www.olcf.ornl.gov/wp-content/uploads/OLCF_AI_Training_0417_2024.pdf

[2]https://docs.olcf.ornl.gov/software/analytics/pytorch_frontier.html#environment-variables

well-suited when the number of GCDs/GPUs per node is small (eight on Frontier, and four on Perlmutter), we adopt this for all intra-node collectives. This ensures that the GPU-GPU link bandwidth is saturated effectively.

Messaging Protocol Selection: For intra-node communication, we rely on the vendor libraries' heuristics for internal protocol selection, such as NCCL and RCCL's ability to switch to low-latency (LL/LL128) protocols for small messages on NVLink and Infinity Fabric. While Cray-MPICH does not support these low-latency protocols for inter-node communication, this is not a concern as the scope of our work is focused on large message sizes (16 MB to 1 GB).

B. Custom Implementations for Inter-node Operations

The use of MPI for collective operations in the inter-node phase presents some challenges. With potentially thousands of GPUs participating in the collective, the choice of the communication algorithm becomes critical for performance. However, Cray-MPICH only offers the ring algorithm. As described in Section II-B, the ring algorithm's linear scaling in latency with respect to the number of processes makes it sub-optimal at large-scale. Additionally, as discussed in Section III-B, CPU-based compute in Cray-MPICH limits performance. Hence, we have developed custom implementations of each collective in PCCL for the inter-node phase that use MPI point-to-point send and receive messages. We implement two different algorithms as described below.

Ring Algorithm: We implement an inter-node version of the ring algorithm described in Section II-B. This can be more performant in bandwidth-bound scenarios. We refer to our implementation of the ring algorithm as `PCCL_ring`. For reduce-scatter and all-reduce, we schedule the local reduction computation on the GPU to improve performance, as opposed to performing it on the CPU as in Cray-MPICH.

Recursive Doubling/Halving Algorithm: We utilize the recursive doubling algorithm for inter-node all-gather and recursive halving for inter-node reduce-scatter operations [12], and implement them using MPI point-to-point messages. These algorithms offer logarithmic latency terms (refer to Section II-B), enabling significantly better scaling performance as the number of GPUs increases. Similar to our ring implementation for reduce-scatter operations, we also ensure that our local vector reduction computation is efficient by scheduling it on GPU cores. We refer to our implementation using recursive doubling/halving as `PCCL_rec`. Further, our all-reduce in `PCCL_rec` uses recursive halving followed by recursive doubling for inter-node traffic.

Figure 6 shows the speedup of recursive halving over ring for inter-node reduce-scatter. In bandwidth-bound scenarios (fewer processes and/or larger messages), ring performs better. In latency-bound scenarios (more processes and/or smaller messages), recursive halving achieves significant improvements, confirming our algorithmic complexity analysis. Our goal is to be able to choose the ideal algorithm in different scenarios, and we describe our efforts in that direction below.

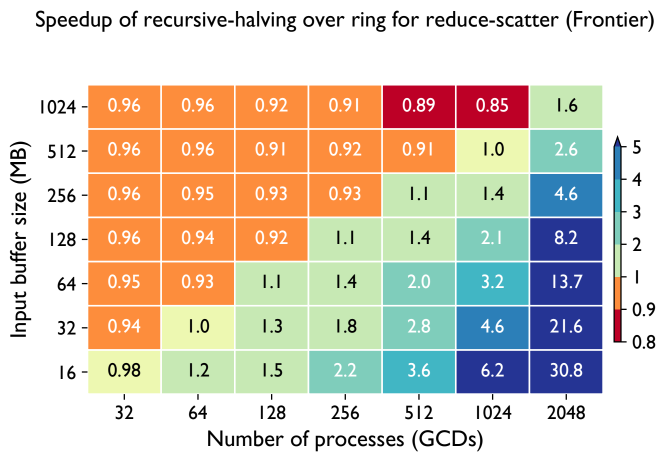


Fig. 6. Heatmap showing speedups from using recursive halving over the ring algorithm in the inter-node phase of the reduce-scatter implementation in PCCL.

We have developed these implementations in C++ as part of PCCL and expose Pybind11 bindings to enable seamless integration with Python-based deep learning frameworks such as ZeRO-3 [1]. Implementing these algorithms in C++ proves to be critical for achieving high performance.

C. Learning-based Adaptive Dispatching

We have observed, based on our empirical analysis, that no single communication library or algorithm – Cray-MPICH, NCCL, RCCL, `PCCL_ring`, or `PCCL_rec` – is universally fastest. Performance depends on the specific configuration, with message size and GPU count being the dominant factors. For example, RCCL often outperforms PCCL's ring/recursive algorithms at lower GPU counts and larger message sizes. This motivated us to develop an adaptive dispatcher capable of selecting the most suitable library at runtime. This modified design of the PCCL library is shown in Figure 7.

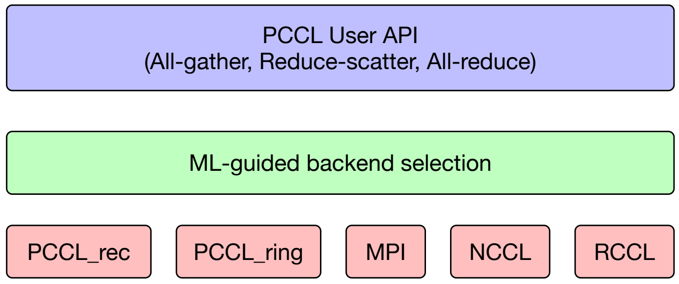

PCCL User API
(All-gather, Reduce-scatter, All-reduce)
ML-guided backend selection
PCCL_rec
PCCL_ring
MPI
NCCL
RCCL

Fig. 7. The ML-guided selection mechanism in PCCL can enable choosing the best performing backend from several available options.

In order to enable optimal backend selection, we implemented a lightweight dispatcher based on Support Vector Machines (SVMs) [21], [22] – a supervised learning algorithm that classifies data by finding the optimal decision boundary that separates distinct categories. SVMs work by identifying the hyperplane that maximizes the margin between different classes in the feature space, allowing for effective classification even in complex scenarios. For each machine and collective pair, we train a dedicated SVM classifier using empirical data spanning message sizes from 1 MB to 1024 MB and GPU counts from 4 to 2048. For each configuration (library, message size, GPU count), we include ten independent runs in our dataset. This dataset is partitioned using a stratified 80/20 train-test split to maintain class balance. Hyperparameter selection for each SVM is performed via five-fold cross-validation on the training set, ensuring robust model selection and mitigating overfitting. At runtime, the dispatcher queries the appropriate trained SVM with the GPU count and message size as input features to predict the optimal backend. We evaluate the performance of the trained model on 20% unseen test data. As Table I shows, the high prediction accuracy and low misclassification rates indicate that the dispatcher generalizes well to previously unseen configurations.

TABLE I
SVM DISPATCHER PERFORMANCE ON THE UNSEEN TEST SET (20% OF DATA).

Machine	Collective	Test Size	Correctly Classified	Accuracy (%)
Frontier	All-gather	20	17	85.0
	Reduce-scatter	20	18	90.0
	All-reduce	20	16	80.0
Perlmutter	All-gather	22	20	90.9
	Reduce-scatter	22	21	95.4
	All-reduce	20	15	75.0

V. Experimental Setup

Next, we provide details of the experimental setup for our comparison of PCCL collectives with other communication libraries, and in the context of production DL workloads.

A. Benchmarking Collective Operations

First, we compare the standalone performance of all-gather, reduce-scatter, and all-reduce operations, which are the primary focus of this paper. Our experiments cover a range of message sizes from 16 MB to 1 GB. For all-gather, this range denotes each GPU's output buffer size; for reduce-scatter, the input buffer size; for all-reduce, both input and output buffer sizes. For each message size, we measure performance across 32 to 2048 GCDs (4 to 256 nodes) on Frontier, and 32 to 2048 GPUs (8 to 512 nodes) on Perlmutter.

We conduct ten independent trials for each combination of library, collective, message size, and GPU count. We consider the following libraries on each system: Cray-MPICH (the default MPI implementation on HPE Cray systems with Slingshot interconnects), NCCL or RCCL, and PCCL. For both systems, our setup for process placement, measurement protocol, communication tuning, and software stack is consistent with the configuration described in Section III-A. For all-reduce, at the time of our runs, a more recent software stack was available: on Frontier we use ROCm 6.4.1, RCCL 2.22.3, Cray MPICH 8.1.32, libfabric 1.22.0 and the aws-ofi-rccl plugin version v1.4. On Perlmutter, we use CUDA 12.9, NCCL 2.27.3, Cray-MPICH 8.1.30, and libfabric 1.22.0.

B. Comparative Evaluation of Production DL Workloads

We use two deep learning workloads to study the impact of using improved implementations in PCCL on overall application performance.

DeepSpeed ZeRO-3: To evaluate the practical benefits of PCCL's all-gather and reduce-scatter, we measure the end-to-end training performance of large language models using DeepSpeed ZeRO-3 [1], a widely adopted parallel deep learning framework. We perform strong scaling experiments on seven billion and 13 billion parameter GPT-style transformer models [23] using model hyperparameters from Zhang et al. [24]. We list these hyperparameters in Table II. We use a global batch size of four million tokens and a sequence length of 2048 and use the OpenWebText [25] corpus to create our training data.

TABLE II
ARCHITECTURAL DETAILS OF THE GPT-STYLE TRANSFORMER MODELS [23] USED IN THE EXPERIMENTS. WE BORROW THESE HYPERPARAMETERS FROM ZHANG ET AL. [24].

Model	Framework	Params	Layers	Hidden Size	Heads
GPT-7B	ZeRO-3	7B	32	4096	32
GPT-13B	ZeRO-3	13B	40	5120	40
GPT-1.3B	DDP	1.3B	24	2048	32

PyTorch DDP: PyTorch DDP is a distributed data parallel framework which relies on all-reduce (unlike ZeRO-3). To evaluate the practical benefits of PCCL's all-reduce, we use PyTorch DDP [4] to train a 1.3 billion parameter GPT-style transformer model [23], with hyperparameters summarized in Table II. We use a global batch size of one million tokens and use the OpenWebText [25] corpus as training data.

For both ZeRO-3 and DDP experiments, on Frontier, we scale from 128 to 1024 GCDs, and on Perlmutter, we scale from 256 to 2048 GPUs. We run experiments with RCCL (Frontier) and NCCL (Perlmutter) as baselines, then swap in `PCCL_rec` to handle all-gather and reduce-scatter in ZeRO-3, and to handle all-reduce in DDP. For each configuration, we run 10 training batches across three trials and compute the average time per iteration over the last eight batches in each run to minimize warm-up effects. We observed significant variability in RCCL all-reduce performance; to account for this, we ran five trials for DDP experiments and report results from the three trials with the smallest mean batch time.

VI. Performance Results

We now present performance comparisons of PCCL with other collective libraries using benchmarks and in the context of DL applications.

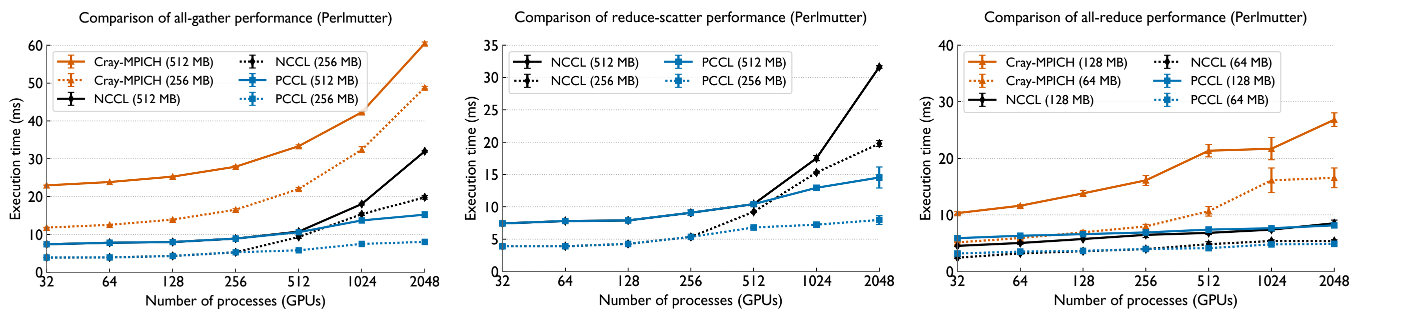


Fig. 8. Performance comparison of all-gather (left), reduce-scatter (middle), and all-reduce (right) using Cray-MPICH, NCCL, and PCCL with adaptive dispatching, for different per-process buffer sizes (256 and 512 MB for all-gather and reduce-scatter; 64 and 128 MB for all-reduce) and varying process counts on Perlmutter.

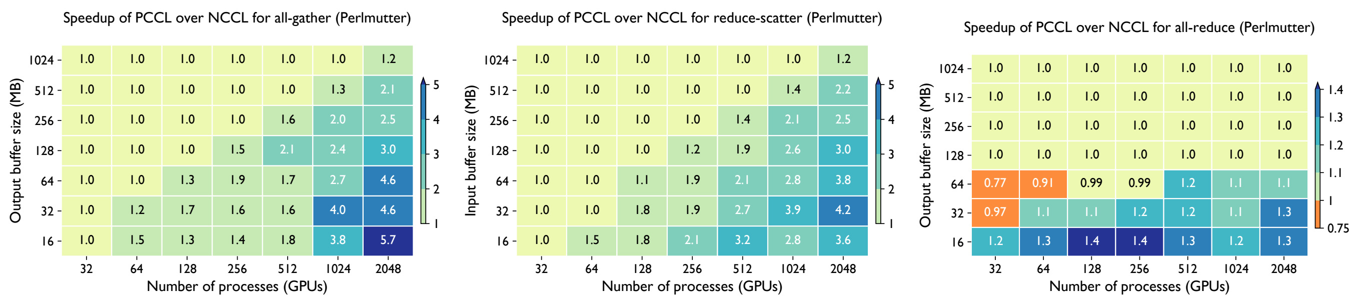


Fig. 9. Heatmaps showing speedups from using PCCL with adaptive dispatching over NCCL for all-gather (left), reduce-scatter (middle), and all-reduce (right) on Perlmutter. The speedup is shown as a function of per-process output/input buffer size (in MB) and process count.

A. Comparisons with Cray-MPICH and NCCL on Perlmutter

Figure 8 presents results for all-gather (left) and reduce-scatter (middle) collectives on Perlmutter for representative per-process message sizes of 256 and 512 MB. For all-gather and reduce-scatter, we observed that both Cray-MPICH (orange lines) and NCCL (black lines) fell short of ideal scaling, which should be a flat horizontal line. NCCL's performance begins to degrade noticeably beyond 512 processes. In contrast, PCCL scales nearly perfectly across both collectives, maintaining desirable performance curves, and achieving speedups in the range of $1.3 - 4.6\times$ over NCCL and 8.8–15× over Cray-MPICH on 1024 and 2048 GPUs. Figure 8 (right) presents results for all-reduce operations with message sizes of 64 MB and 128 MB. Both NCCL and PCCL exhibit strong scalability with node count. NCCL employs double-binary trees to achieve log-latency scaling, which explains why the performance of NCCL and PCCL is nearly identical for the all-reduce operation.

Figure 9 presents heatmaps that present the relative performance improvement of PCCL over NCCL for a larger range of message sizes and GPU counts. For all-gather (left) and reduce-scatter (middle), PCCL performs similar to NCCL in the top-left regions of the heatmaps, representing bandwidth-bound scenarios. As expected, our adaptive dispatching protocol selects NCCL for these cells. However, as we transition to latency-sensitive regions in the bottom-right corners of the heatmap, PCCL's advantages become evident. Around 1024–2048 processes and 16–32MB message sizes, PCCL achieves significant speedups over NCCL, ranging from 3–5×. While speedups for larger message sizes are smaller, they remain notable. For example, at 2048 processes and 128–512 MB message sizes, PCCL is approximately 2–3× faster than NCCL, which is still a significant improvement. For all-reduce (right), we observed similar performance between PCCL and NCCL as both libraries use log-latency scaling algorithms.

B. Comparisons with Cray-MPICH and RCCL on Frontier

Next, we evaluate PCCL's effectiveness on Frontier, which features AMD MI250X GPUs. Figure 10 (left) shows the performance of all-gather operations on Frontier using PCCL and other communication libraries for output buffer sizes 256 and 512 MB. For each configuration, we scale the number of GCDs from 32 to 2048. Since the output buffer size per GPU remains fixed, the ideal performance curve for each buffer size is a flat horizontal line, indicating perfect scaling.

We observe that RCCL and Cray-MPICH fell short of the ideal scaling line. Beyond 128 GCDs, RCCL's (green) execution time scales almost linearly with the number of processes. Cray-MPICH (orange lines) demonstrates a similar pattern, with performance dropping sharply as we scale to higher process counts. We attribute the poor scaling of RCCL and Cray-MPICH to their reliance on the ring algorithm, the latency of which grows linearly with the number of processes.

In contrast, PCCL (blue) maintains nearly flat scaling trends across all message sizes, demonstrating significantly better scalability and efficiency. We attribute PCCL's better

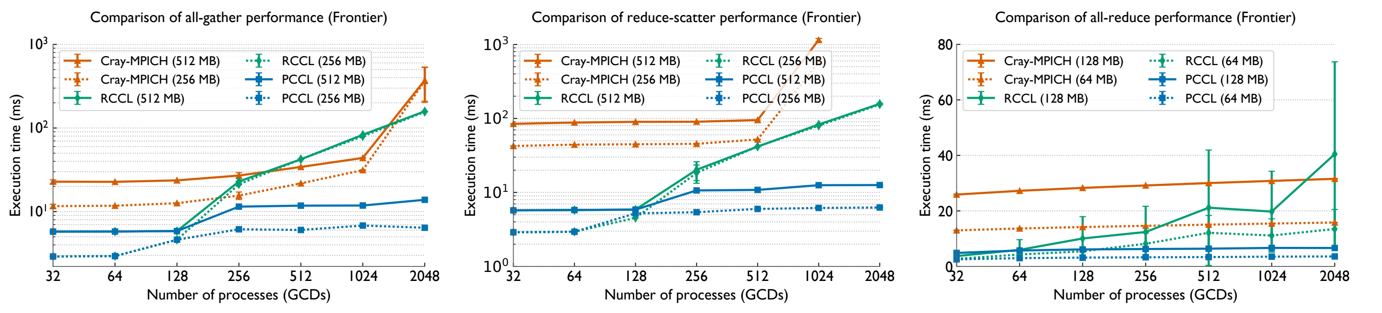


Fig. 10. Performance comparison of all-gather (left), reduce-scatter (middle), and all-reduce (right) using Cray-MPICH, RCCL, and PCCL with adaptive dispatching, for different per-process buffer sizes (256 and 512 MB for all-gather and reduce-scatter; 64 and 128 MB for all-reduce) and varying process counts on Frontier.

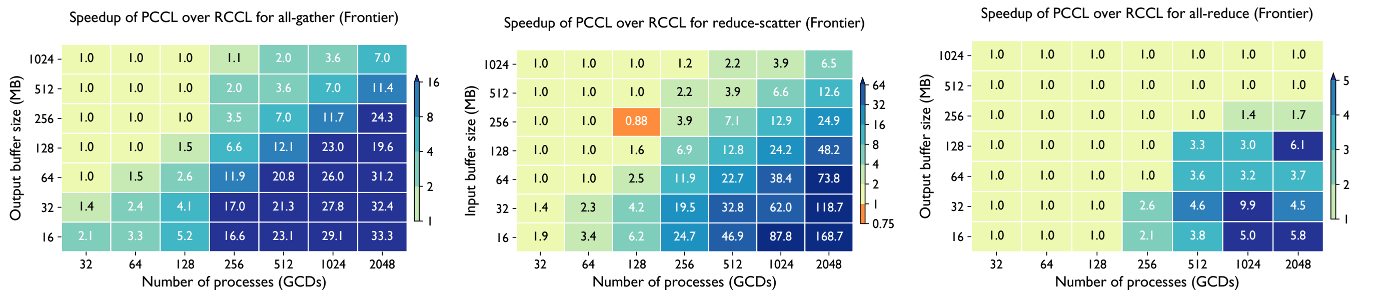


Fig. 11. Heatmaps showing speedups from using PCCL with adaptive dispatching over RCCL for all-gather (left), reduce-scatter (middle), and all-reduce (right) on Frontier. The speedup is shown as a function of per-process output/input buffer size (in MB) and process count.

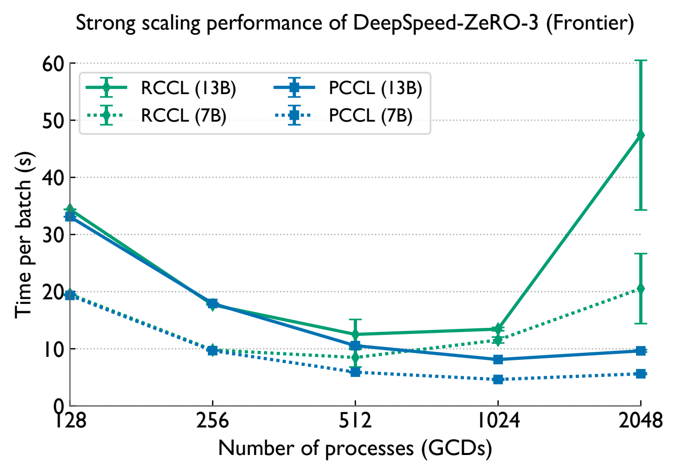
performance to its hierarchical algorithms, described in Section IV. By using the ring algorithm within nodes (limited to eight processes) and recursive doubling across nodes, PCCL bounds the latency overhead that otherwise grows linearly in traditional ring-based implementations. This design enables better scalability across large GPU counts. The performance improvements of PCCL over RCCL and Cray-MPICH become increasingly pronounced as we increase the number of processes. At 2048 GCDs, PCCL all-gather achieves speedups ranging from 7–24× over RCCL, and an even larger 27–82× over Cray-MPICH, depending on the message size. These results highlight PCCL's ability to deliver highly efficient communication at scale.

The performance trends for reduce-scatter on Frontier follow the same pattern established by all-gather, as depicted in Figure 10 (middle). Both RCCL and Cray-MPICH fall short of the ideal performance curve, and PCCL achieves significant speedups over both libraries with increasing scale. However, the speedups for all-reduce operations are comparatively smaller, as shown in Figure 10 (right). Here, RCCL employs double-binary trees to achieve log-latency scaling, which explains its improved scalability over its own reduce-scatter or all-gather algorithms. Despite this, PCCL's two-phase strategy demonstrates near-flat scaling, ultimately achieving significant speedups over RCCL at higher GCD counts.

Figure 11 shows the speedups of PCCL over RCCL for all-gather (left), reduce-scatter (middle), and all-reduce (right) operations on Frontier, respectively, across a range of output buffer sizes and process counts. In the top-left regions of all three heatmaps, where we have large messages and small GPU counts, representing bandwidth-bound scenarios, PCCL's adaptive dispatching picks RCCL as the backend. This is expected, as RCCL's flat ring algorithm can achieve higher bandwidth than PCCL's hierarchical two-phase strategy [17].

In contrast, in the bottom-right corners of the heatmap, representing the latency-sensitive regime with small messages and large GPU counts, PCCL delivers substantial gains. On 2048 GCDs, for 16, 32, and 64 MB buffer sizes, PCCL achieves speedups of more than 30× and 50–150× over RCCL for all-gather and reduce-scatter, respectively. To uncover the source of this performance improvement, we analyze the values of the Cassini NIC hardware counters for these runs. Specifically, RCCL exhibits 200× higher value for the `lpe_net_match_overflow_0` counter, compared to PCCL. According to the official Cassini documentation[3], this counter tracks the "number of messages where payload data was delivered to a buffer on the overflow list because there was no match on the priority list", noting that these messages "incur higher cost because data must be copied from the overflow buffer". While RCCL frequently triggers these expensive software-side copies, PCCL (by virtue of dissolving inter-node collective communication into MPI point-to-point operations) maintains its communication on the hardware-accelerated "priority list," ensuring zero-copy data movement.

[3]https://cpe.ext.hpe.com/docs/latest/getting_started/HPE-Cassini-Performance-Counters.html



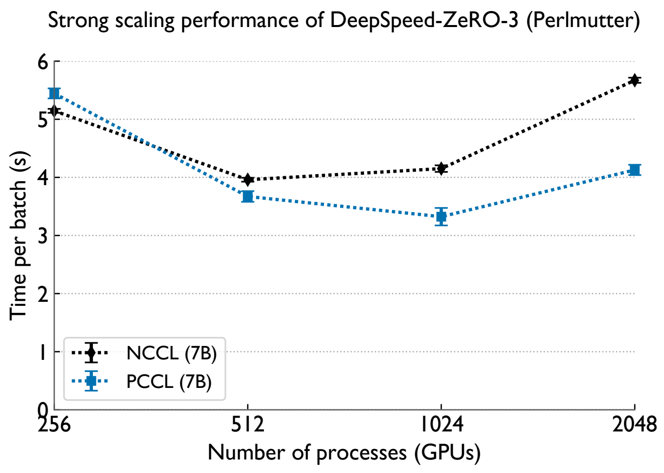


Fig. 12. Strong scaling performance of DeepSpeed ZeRO-3 using RCCL or NCCL, and PCCL, on Frontier (left) and Perlmutter (right) for two model sizes: GPT-3 7B and GPT-3 13B.

For larger messages at 2048 GCDs, the speedups are comparatively smaller but still significant– 24.3 and 7.0× for all-gather, 24.9 and 6.5× for reduce-scatter on 256 and 1024 MB, respectively. Similarly, for all-reduce, PCCL achieves up to a 9.9× speedup over RCCL in regions where latency dominates. These results underscore PCCL's strength in latency-bound scenarios and highlight its ability to scale efficiently to thousands of GPUs.

C. *Impact on DL Training Performance*

Finally, we examine how these communication gains translate into improvements in DL training performance at scale. Figure 12 (left) presents the batch times for strong scaling GPT-3-style transformer training on Frontier using the DeepSpeed ZeRO-3 framework [1]. Green lines represent ZeRO-3 runs with RCCL, the default communication library and blue lines represent runs with all-gather and reduce-scatter collectives in ZeRO-3 issued with PCCL. At smaller scales (128 and 256 GCDs), both libraries perform comparably. However, as we scale further, PCCL begins to outperform RCCL. When scaling to 1024 GCDs, RCCL fails to maintain strong scaling and even exhibits increased batch times compared to 512 GCDs. In contrast, PCCL continues to scale efficiently, delivering a 2.5× speedup for the 7B model and a 1.6× speedup for the 13B model. Finally, at 2048 GCDs, although both libraries exhibit diminishing returns in strong scaling efficiency, PCCL still achieves substantial speedups 3.3–4.9× relative to RCCL.

We observed similar trends on Perlmutter, as shown in Figure 12 (right). For a 7B parameter model, at 256 GPUs, PCCL underperforms NCCL with slowdown of 0.94×. However, as we scale to larger GPU counts, PCCL begins to outperform NCCL–achieving a 1.07× speedup at 512 GPUs and a significantly higher 1.37× speedup at 2048 GPUs.

Figure 13 presents batch times for strong scaling GPT3-1.3B on Frontier using PyTorch DDP. At smaller scales, RCCL outperforms PCCL, with PCCL showing a 0.55× and 0.80× slowdown at 128 and 256 GCDs, respectively. However, at higher GCD counts, PCCL rapidly closes this performance gap and ultimately surpasses RCCL, achieving a 1.8× and 2.4× speedup over RCCL at 1024 and 2048 GCDs, respectively. These results highlight PCCL's ability to deliver performance improvements for collective communication across multiple GPU architectures, and more importantly, translate those gains into significant speedups for production DL applications.

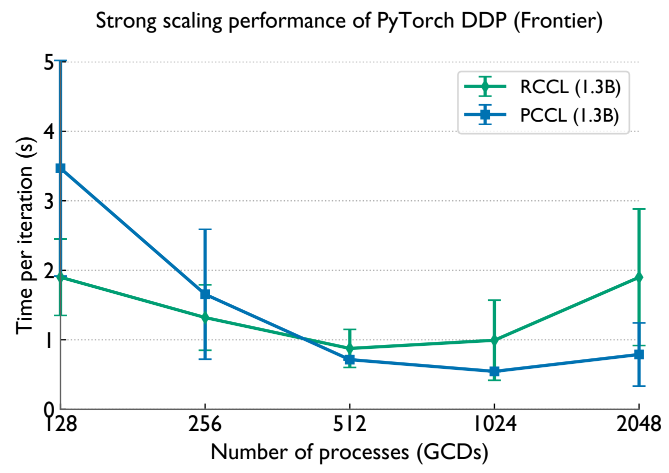


Fig. 13. Strong scaling performance of PyTorch DDP using RCCL and PCCL on Frontier for GPT-3 1.3B

VII. Related Work

Optimizing collective communication has been a long-standing challenge in high performance computing (HPC) and parallel computing and has been the focus on extensive research. Thakur et al.'s seminal work focuses on optimizing a plethora of collective operations including all-gathers and reduce-scatters in MPICH [12]. The authors explore the design space of several algorithms for each collective, and provide guidelines for selecting the most appropriate algorithm for different scenarios. In contrast, our work focuses on large-scale deep learning workloads and with messages sizes in the tens to hundreds of megabytes.

Patarsuk et al. propose the bandwidth-optimized ring algorithm for all-reduce operations [26]. Graham et al. develop optimize MPI collective to effectively exploit shared memory on multi-core systems. Chan et al. [27] develop highly optimized collectives for the IBM Blue Gene/L, exploiting unique properties of the system [28]. Kandalla et al. develop a scalable multi-leader hierarchical algorithm for all-gather [29]. Note that the focus of these works is on optimizing the performance of collective communication in traditional HPC workloads with small message sizes, and not on the large message sizes inherent to deep learning.

De Sensi et al. study the performance of NCCL, RCCL and Cray-MPICH across various state-of-the-art supercomputers across a variety of latency and bandwidth bound scenarios [13]. Cho et al. propose a multi-level hierarchical ring algorithm for all-reduce and study the tradeoff of bandwidth and latency between the flat and hierarchical ring algorithms [17]. In this work, we build on this and exploit more latency optimal algorithms such as recursive doubling/halving in the inter-node levels of the hierarchy and also demonstrate how this design can be utilized to load-balance network traffic across NICs. Note that similar hierarchical designs have been explored in other works as well [18], [30].

Hidayetoglu et al. present HiCCL, a hierarchical collective communication library targeting the same systems as our work – Perlmutter and Frontier [31]. While HiCCL demonstrates performance advantages over NCCL and RCCL, these gains are primarily observed at smaller node counts. As shown in their evaluation, HiCCL's performance matches or falls below vendor libraries at larger node counts. In contrast, PCCL is specifically designed and optimized for the large-scale regime, consistently outperforming vendor libraries at thousands of GPUs. Thus, HiCCL and PCCL address complementary performance regimes.

Cai et al. develop a systematic theoretical approach to synthesize novel communication algorithms for optimizing collective communication on a particular topology [32]. Cho et al. develop a strategy to maximize the overlap of a tree-based all-reduce with the computation in neural network training [33]. There is also a body of work focused on exploiting data compression to minimize communication overheads in distributed deep learning. For example, Feng et al. optimize all-to-all communication in recommendation model training via a novel error-bounded compression algorithm [34]. Huang et al. develop hZCCL, a communication library that enables collective operations on compressed data [35]. Zhou et al. develop a GPU-based compression scheme for all-gathers and reduce-scatters [36] and optimize FSDP [2] training at scale.

In the context of algorithm selection and optimization, Liu et al. develop parameterized ring and recursive algorithms for GPU-to-GPU collectives, and introduce a simulation technique for parameter auto-tuning [37]. Wilkins et al. develop ACCLAiM, which uses machine learning for auto-tuning MPI collective communication [38], an approach conceptually related to our SVM-based adaptive dispatching mechanism. Hunold et al. present an auto-tuning framework that uses regression models to predict the fastest algorithm for different collective routines [39]. Venkata et al. develop UCC, a unified collective communication library with pluggable transport layers supporting CPUs, GPUs, and DPUs, targeting broad applicability across programming models and hardware [40].

VIII. Conclusion

We investigated the current state of performance of collective communication routines in several state-of-the-art communication libraries. Specifically, we focused on the all-gather, reduce-scatter, and all-reduce collectives, which are commonly used in distributed deep learning workloads – both during training and inference. We evaluated the performance of Cray-MPICH, RCCL, and NCCL on leadership-class GPU supercomputers, and identified critical performance bottlenecks that limit their scaling to large GPU counts.

In order to address these limitations, we introduced PCCL, a scalable, performant and portable communication library with highly optimized implementations of all-gather, reduce-scatter, and all-reduce operations. PCCL combines three key ideas: (1) hierarchical two-level collective designs that better exploit the available hardware, (2) latency-optimal recursive doubling/halving implementations for inter-node communication, and (3) an SVM-based adaptive dispatcher that dynamically selects the most performant backend based on message size and GPU count. We demonstrated significant performance improvements over all three libraries, both in collective communication benchmarks as well as production DL training. In future work, we plan to benchmark PCCL on clusters with InfiniBand interconnects to evaluate its performance beyond Slingshot-based systems. As model sizes and system scales continue to grow, we believe that intelligent, architecture-aware collective communication will be a central enabler of efficient large-scale AI workloads.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120. We thank Joy Kitson of the Parallel Software and Systems Group for providing feedback on the paper and for improving the heatmap figures in the paper.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy (DOE) under Contract No. DE-AC05-00OR22725. This research also used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. DOE Office of Science User Facility, operated under Contract No. DE-AC02-05CH11231 using NERSC award DDR-ERCAP0034262.

References

[1] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[2] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, A. Desmaison, C. Balioglu, P. Damania, B. Nguyen, G. Chauhan, Y. Hao, A. Mathews, and S. Li, "Pytorch fsdp: Experiences on scaling fully sharded data parallel," *Proc. VLDB Endow.*, vol. 16, no. 12, p. 3848–3860, aug 2023.

[3] S. Singh and A. Bhatele, "AxoNN: An asynchronous, message-driven parallel framework for extreme-scale deep learning," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '22. IEEE Computer Society, May 2022.

[4] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3005–3018, Aug. 2020. [Online]. Available: https://doi.org/10.14778/3415478.3415530

[5] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," Tech. Rep., 2020.

[6] S. Singh, O. Ruwase, A. A. Awan, S. Rajbhandari, Y. He, and A. Bhatele, "A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 203–214. [Online]. Available: https://doi.org/10.1145/3577193.3593704

[7] S. Singh and A. Bhatele, "Exploiting sparsity in pruned neural networks to optimize large model training," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 245–255. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS54959.2023.00033

[8] A. K. Ranjan, S. Singh, C. Wei, and A. Bhatele, "Plexus: Taming billion-edge graphs with 3D parallel full-graph GNN training," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. ACM, Nov. 2025. [Online]. Available: https://doi.acm.org/10.1145/3712285.3759890

[9] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017. [Online]. Available: http://arxiv.org/abs/1706.02677

[10] S. Singh, P. Singhania, A. Ranjan, J. Kirchenbauer, J. Geiping, Y. Wen, N. Jain, A. Hans, M. Shu, A. Tomar, T. Goldstein, and A. Bhatele, "Democratizing AI: Open-source scalable LLM training on GPU-based supercomputers," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '24, Nov. 2024.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[12] R. Thakur and W. D. Gropp, "Improving the performance of collective operations in mpich," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, D. Laforenza, and S. Orlando, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 257–267.

[13] D. De Sensi, L. Pichetti, F. Vella, T. De Matteis, Z. Ren, L. Fusco, M. Turisini, D. Cesarini, K. Lust, A. Trivedi, D. Roweth, F. Spiga, S. Di Girolamo, and T. Hoefler, "Exploring gpu-to-gpu communication: Insights into supercomputer interconnects," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '24. IEEE Press, 2024. [Online]. Available: https://doi.org/10.1109/SC41406.2024.00039

[14] "Hpe cassini performance counters," 2024. [Online]. Available: https://cpe.ext.hpe.com/docs/latest/getting_started/HPE-Cassini-Performance-Counters.html

[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," 2025. [Online]. Available: https://arxiv.org/abs/2507.04786

[16] S. Jeaugey, "Pat: a new algorithm for all-gather and reduce-scatter operations at scale," 2025. [Online]. Available: https://arxiv.org/abs/2506.20252

[17] M. Cho, U. Finkler, D. Kung, and H. Hunter, "Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," in *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia, Eds., vol. 1, 2019, pp. 241–251. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2019/file/0c8abcf158ed12d0dd94480681186fda-Paper.pdf

[18] T. Thao Nguyen, M. Wahib, and R. Takano, " Hierarchical Distributed-Memory Multi-Leader MPI-Allreduce for Deep Learning Workloads ," in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2018, pp. 216–222. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CANDARW.2018.00048

[19] A. Singh, "Load-balanced routing in interconnection networks," Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, 2005, http://cva.stanford.edu/publications/2005/thesis_arjuns.pdf.

[20] J. Geiping, S. McLeish, N. Jain, J. Kirchenbauer, S. Singh, B. R. Bartoldson, B. Kailkhura, A. Bhatele, and T. Goldstein, "Scaling up test-time compute with latent reasoning: A recurrent depth approach," 2025. [Online]. Available: https://arxiv.org/abs/2502.05171

[21] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, September 1995.

[22] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, August 2004.

[23] T. B. Brown *et al.*, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[24] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[25] A. Gokaslan and V. Cohen, "Openwebtext corpus," http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[26] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *J. Parallel Distrib. Comput.*, vol. 69, no. 2, p. 117–124, feb 2009. [Online]. Available: https://doi.org/10.1016/j.jpdc.2008.09.002

[27] E. Chan, R. van de Geijn, W. Gropp, and R. Thakur, "Collective communication on architectures that support simultaneous communication over multiple links," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 2–11. [Online]. Available: https://doi.org/10.1145/1122971.1122975

[28] R. L. Graham and G. Shipman, "Mpi support for multi-core architectures: Optimized shared memory collectives," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 130–140.

[29] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, and D. K. Panda, "Designing multi-leader-based allgather algorithms for multi-core clusters," in *2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009, pp. 1–8.

[30] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "Designing efficient shared address space reduction collectives for multi-/many-cores," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 1020–1029.

[31] M. Hidayetoglu, S. G. de Gonzalo, E. Slaughter, P. Surana, W. mei Hwu, W. Gropp, and A. Aiken, "Hiccl: A hierarchical collective communication library," 2024. [Online]. Available: https://arxiv.org/abs/2408.05962

[32] Z. Cai, Z. Liu, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, and O. Saarikivi, "Synthesizing optimal collective algorithms," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 62–75. [Online]. Available: https://doi.org/10.1145/3437801.3441620

[33] S. Cho, H. Son, and J. Kim, "Logical/physical topology-aware collective communication in deep learning training," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 56–68.

[34] H. Feng, B. Zhang, F. Ye, M. Si, C.-H. Chu, J. Tian, C. Yin, S. Deng, Y. Hao, P. Balaji, T. Geng, and D. Tao, "Accelerating

communication in deep learning recommendation model training with dual-level adaptive lossy compression," 2024. [Online]. Available: https://arxiv.org/abs/2407.04272

[35] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, Z. Jian, X. Liang, K. Zhao, X. Lu, Z. Chen, F. Cappello, Y. Guo, and R. Thakur, "hzccl: Accelerating collective communication with co-designed homomorphic compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '24. IEEE Press, 2024. [Online]. Available: https://doi.org/10.1109/SC41406.2024.00110

[36] Q. Zhou, Q. Anthony, L. Xu, A. Shafi, M. Abduljabbar, H. Subramoni, and D. K. D. Panda, "Accelerating distributed deep learning training with compression assisted allgather and reduce-scatter communication," pp. 134–144, 2023.

[37] P. Liu, S. Rhee, M. Wilkins, and P. Dinda, "Parameterized algorithms and parameter selection for fast gpu-gpu collective communication," in *2025 33rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2025, pp. 1–9.

[38] M. Wilkins, Y. Guo, R. Thakur, P. A. Dinda, and N. Hardavellas, "Acclaim: Advancing the practicality of mpi collective communication autotuning using machine learning," *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 161–171, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:251793247

[39] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting MPI collective communication performance using machine learning," in *Proceedings of the IEEE Cluster Conference*, ser. Cluster '20, Sep. 2020.

[40] M. G. Venkata, V. Petrov, S. Lebedev, D. Bureddy, F. Aderholdt, J. Ladd, G. Bloch, M. Dubman, and G. Shainer, "Unified collective communication (UCC): an unified library for cpu, gpu, and DPU collectives," in *IEEE Symposium on High-Performance Interconnects, HOTI 2024, Albuquerque, NM, USA, August 21-23, 2024*. IEEE, 2024, pp. 37–46. [Online]. Available: https://doi.org/10.1109/HOTI63208.2024.00018

Artifact Description (AD)

Appendix A
Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 Analyzing the limitations of existing communication libraries, Cray-MPICH and RCCL, for all-gather, reduce-scatter, and all-reduce collectives in parallel deep learning workloads.

C_2 Developing optimized implementations of these collectives in PCCL (the communication library proposed in this work), with a focus on effectively utilizing system resources and ensuring scalability in latency bound scenarios. PCCL also includes a machine learning algorithm that selects the best communication backend at runtime based on GPU count and message size.

C_3 Conducting benchmarking of large-scale LLM training workloads using both DeepSpeed ZeRO-3 and PyTorch DDP to validate the practical benefits of our optimizations, demonstrating significant speedups in training throughput.

B. Computational Artifacts

A_1 (PCCL) is our implementation of the accelerated all-gather, reduce-scatter, and all-reduce collectives proposed in this work.

A_2 (nanoGPT, DeepSpeed ZeRO-3 branch) is a codebase for training large language models (LLMs) using the DeepSpeed parallel training framework. It supports both NCCL/RCCL and PCCL as communication backends.

A_3 (nanoGPT, PyTorch DDP branch) is a codebase for training LLMs using PyTorch DDP, used for benchmarking of PCCL's all-reduce against RCCL on Frontier.

Appendix B
Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

Artifact A_1 is PCCL, the communication library proposed in this work, featuring optimized implementations of all-gather, reduce-scatter, and all-reduce operations. This artifact includes scripts for benchmarking these collectives in isolation and for comparing their performance against NCCL, RCCL, and Cray-MPICH. It also provides scripts for building PCCL and its dependencies, as well as for running the comparison benchmarks on Perlmutter and Frontier.

Expected Results

Reproducing A_1 allows us to reproduce all plots in the paper except Figures 12 and 13.

Expected Reproduction Time (in Minutes)

Artifact Setup: Building the artifact takes less than five minutes on both Perlmutter and Frontier.

Artifact Execution: For a given process count, a communication library (RCCL/NCCL, Cray-MPICH, or PCCL), and a collective (all-gather/reduce-scatter/all-reduce), profiling times across all message sizes (1 MB to 1024 MB) takes around 2 minutes. To complete the full sweep, one must launch jobs for all three libraries, the three collectives, as well as all node counts (4–256 nodes on Frontier, and 8–512 on Perlmutter), and multiple trials (atleast 10) to account for performance variability. While these jobs can execute in parallel, the exact execution time depends on the job scheduler. In practice, 24 hours are sufficient to gather all data points.

Artifact Analysis: Analyzing the outputs of this artifact involves running grep commands on the outputs, which overall take less than 10 seconds.

Artifact Setup (incl. Inputs)

Hardware: This artifact has been tested for correctness and performance on Perlmutter, which has NVIDIA A100 GPUs, and Frontier, which has MI250X GPUs. We expect it to run on any cluster that supports PyTorch (and its distributed communication backend) and a performant distribution of MPI (like Cray-MPICH on Cray machines).

Software:

Software Name and URL	Version	Comments
PCCL	0.0.1	N/A
Cray-MPICH	8.1.31 (Frontier) 8.1.30 (Perlmutter)	Available as modules
PyTorch	2.6.0	Ships with RCCL/NCCL
AWS Plugin	1.4	Only for Frontier
ROCm	6.4.1	Available as a module on Frontier
CUDA	12.4	Available as a module on Perlmutter

Datasets / Inputs: Input data for the communication benchmarks is generated on the fly (randomly initialized GPU tensors) in the scripts.

Installation and Deployment: Bash scripts in the `scripts/` folder install this artifact and its dependencies on both machines.

Artifact Execution

This is the task workflow, assuming the artifact has been built successfully. One can complete all of these tasks using variations of the sample bash script `run_raw_collectives_benchmark.sh` provided for each machine in the `scripts` folder. The scripts can be modified to change the collective to be benchmarked (between all-gather, reduce-scatter, and all-reduce) and the communication library (NCCL/RCCL, Cray-MPICH, PCCL).

- T_1 Benchmark all-gather times for NCCL/RCCL on all node counts (4–256 nodes on Frontier and 8–512 nodes on Perlmutter). Repeat ten times.
- T_2 Benchmark all-gather times for Cray-MPICH on all node counts. Repeat ten times.
- T_3 Benchmark all-gather times for PCCL on all node counts. Next, change the algorithm to recursive and run the benchmarks again. Repeat ten times.
- T_4 Repeat T_1–T_3 for reduce-scatter.
- T_5 Repeat T_1–T_3 for all-reduce.

Artifact Analysis (incl. Outputs)

The outputs of these tasks are execution times for different message sizes and node counts.

B. Computational Artifact A_2

Relation To Contributions

Artifact A_2 is a codebase for training an LLM on multiple GPUs using the DeepSpeed parallel training framework. It supports PCCL, NCCL, and RCCL as communication libraries for all-gathers and reduce-scatters. We use this artifact to generate the results shown in Figure 12.

Note: One must use the `pccl-zero-3` branch of the GitHub repo for this artifact.

Expected Results

In Figure 12 we compare the batch iteration times for training 7B and 13B parameter LLMs on Frontier and Perlmutter respectively. For each GPU count we first run training with RCCL/NCCL (the default communication library in DeepSpeed) and then repeat with PCCL. Given that PCCL's speedups over RCCL/NCCL increase with process count, we expect the same trend in end-to-end training, i.e., PCCL should provide increasing speedups with more GPUs.

Expected Reproduction Time (in Minutes)

Each run (a given model size, process count, and communication library) requires approximately 10 minutes.

Artifact Setup (incl. Inputs)

Hardware: Same as A_1.

Software: All dependencies of A_1 are required. Additionally, the following Python packages are needed: `transformers`, `datasets`, `tiktoken`, and `tqdm`. All can be installed with `pip install <package-name>`.

Datasets / Inputs: This artifact uses the OpenWebText dataset. Scripts to download and prepare the dataset are provided in `data/openwebtext/`. Note that we use a small subset of the dataset for benchmarking (5%), so the preparation steps are fast (less than 30 minutes). The scripts also support using the full dataset if desired.

Installation and Deployment: Bash scripts in the `scripts/` folder install this artifact and its dependencies on both machines.

Artifact Execution

This is the task workflow, assuming the artifact has been built successfully. One can complete all tasks using variations of the sample bash script `run_deepspeed_benchmark.sh` provided for each machine in the `scripts` folder. The scripts can be modified to change the model size (7B or 13B) and the communication library (NCCL/RCCL or PCCL).

- T_1 Benchmark the 7B model on all node counts (16–256 nodes on Frontier and 64–512 on Perlmutter) using RCCL/NCCL. Repeat 5 times.
- T_2 Benchmark the 7B model on all node counts using PCCL. Repeat 3 times.
- T_3 Repeat T_1–T_2 for the 13B model.

Note: the 13B model runs out of memory at almost all GPU counts on Perlmutter.

Artifact Analysis (incl. Outputs)

The outputs are batch iteration times for each run. These are used as data points for Figure 12.

C. Computational Artifact A_3

Relation To Contributions

Same nanoGPT codebase as A_2 (`pccl-ddp` branch), but trains with PyTorch DDP and all-reduce to produce Figure 13.

Expected Results

Figure 13 shows batch iteration times for training a GPT-3 1.3B model on Frontier using PyTorch DDP with RCCL vs. PCCL. Because DDP relies on all-reduce (rather than all-gather/reduce-scatter), this experiment validates PCCL's all-reduce performance at scale. The crossover point where PCCL surpasses RCCL is expected to occur at large GPU counts ($\geq$1024 GCDs).

Expected Reproduction Time (in Minutes)

$\sim$10 min per run (process count $\times$ communication library).

Artifact Setup (incl. Inputs)

Hardware: Same as A_1.
Software: Same as A_2.
Datasets / Inputs: Same as A_2.
Installation and Deployment: Same as A_2; use the `pccl-ddp` branch.

Artifact Execution

Use `run_frontier.sh` (in the `scripts/` folder) and modify the communication library flag.

T_1 Benchmark GPT-3 1.3B on Frontier (128–2048 GCDs) using RCCL. Repeat 5 times.
T_2 Repeat T_1 with PCCL. Repeat 5 times.

Artifact Analysis (incl. Outputs)

The outputs are batch iteration times per run, used to produce Figure 13.