An Efficient, Reliable and Observable Collective Communication Library in Large-scale GPU Training Clusters

Ziteng Chen[1, 2], Xiaohe Hu[1, 3], Menghao Zhang[4], Yanmin Jia[1], Yan Zhang[1], Mingjun Zhang[1], Da Liu[1], Fangzheng Jiao[4], Jun Chen[1], He Liu[1], Aohan Zeng[5, 6], Shuaixing Duan[6], Ruya Gu[1], Yang Jing[1], Bowen Han[7], Jiahao Cao[5], Wei Chen[1], Wenqi Xie[1], Jinlong Hou[3], Yuan Cheng[3], Bohua Xu[7], Mingwei Xu[5], Chunming Hu[4]

[1]*Infrawaves* [2]*Southeast University* [3]*Shanghai Innovation Institute* [4]*Beihang University*
[5]*Tsinghua University* [6]*Zhipu AI* [7]*China Unicom Research Institute*

Abstract

Large-scale LLM training requires collective communication libraries to exchange data among distributed GPUs. As a company dedicated to building and operating large-scale GPU training clusters, we encounter several challenges when using NCCL in production, including 1) limited efficiency with costly and cumbersome P2P communication, 2) poor tolerance to frequent RNIC port failures, and 3) insufficient observability of transient collective communication anomalies. To address these issues, we propose ICCL, an efficient, reliable, and observable collective communication library in large-scale GPU training clusters. ICCL offloads the P2P communication from GPU kernels to CPU threads for minimal SM consumption, and removes the redundant memory copies irrelevant to the actual communicating process. ICCL also introduces a primary-backup QP mechanism to tolerate frequent NIC port failures, and designs a window-based monitor to observe network anomalies at $O(\mu s)$ level. We open-source ICCL and deploy it in production training clusters for several months, with results showing that compared to NCCL, ICCL achieves a 23.4%/28.5% improvement in P2P throughput/latency as well as a 6.02% increase in training throughput. We also share ICCL's operating experience in large-scale clusters, hoping to give the communities more insights on production-level collective communication libraries in LLM training.

1 Introduction

Collective communication (CC) plays an essential role for data transmission among GPUs in distributed training of large language models (LLMs). CC libraries (CCLs) offer various collective primitives and utilize the intra-host and inter-host interconnections, whose communication efficiency is of great importance in large-scale LLM training systems. Given the dominant market share of NVIDIA GPUs, NVIDIA Collective Communication Library (NCCL) [10], specially tailored and optimized for NVIDIA GPUs, has become the *de-facto* CCL in mainstream large-scale GPU training clusters.

As a company dedicated to building and operating large-scale LLM training clusters, we strive to provide the most cost-efficient LLM infrastructure for our customers. However, when dealing with CC for large-scale GPU clusters, we encounter three main challenges with practical usage of NCCL.

Challenge 1: High SM consumption and cumbersome operations for P2P primitives. GPUs offer high parallelism for general matrix multiplication (GEMM) and value reduction, allowing programmers to write kernels and use GPU streaming multiprocessors (SMs) to accelerate computation. In NCCL, however, we observe the point-to-point (P2P) primitives (e.g., `send/recv` and `alltoall`) that do not involve reductions also consume non-negligible GPU SMs, leading to SM underutilization on computation tasks. Meanwhile, NCCL P2P involves some redundant steps irrelevant to the actual communicating process, thereby slowing down the P2P networking performance. Consequently, NCCL compromises customers' investment of GPU computing power to optimize LLM training efficiency.

Challenge 2: Poor tolerance to NIC port failures. In large-scale GPU clusters, accidental NIC port downs occur frequently, especially with network speed beyond 400Gbps, leading to stragglers and unexpected crashes of CC due to timeout error. Nevertheless, NCCL lacks a native fault tolerant mechanism and is therefore unable to migrate the transmission when failures happen. This can result in significant GPU waste to relaunch CC and LLM training frameworks.

Challenge 3: Limited observability for transient network anomalies. GPU training clusters are dominated by CC workloads, many of which usually finish within $O(ms)$ and even $O(\mu s)$. Nonetheless, when network anomalies occur, NCCL lacks fine-grained observability to capture the $O(\mu s)$-level network performance drops at runtime. It prevents operators from promptly localizing the anomalous links, which is essential to provide proofs for "network innocence".

Some existing studies from industrial communities [18, 24, 25, 29, 33, 44] have shared their experience in large-scale

Ziteng and Xiaohe contributed equally to this paper. This work was done while Ziteng was doing a joint research project at Infrawaves.

LLM training. However, none of them presents a comprehensive and systematic perspective on an ideal CC service with optimized efficiency, reliability, observability. In this paper, we propose ICCL (Infrawaves Collective Communication Library), an efficient, reliable and observable CCL for large-scale GPU training clusters. Based on real-world cases and shared requirements from our customers, we present our practices for designing and operating a production-level CCL in detail. To the best of our knowledge, this is the first work to thoroughly discuss the requirements, architectural design and operational considerations of a full-scale and production-ready CCL.

ICCL includes three key designs. Firstly, inspired by DPDK's philosophy of kernel bypasses and zero copies [1], ICCL adopts a DPDK-like P2P mechanism to 1) offload P2P primitives from GPU kernels to CPU threads for minimal SM consumption, and 2) remove the cumbersome operations for zero memory copies. DPDK-like P2P can not only spare more SM resources for GEMMs and improve the P2P bandwidth and latency performance, but also allow us to optimize the overlaps of pipeline parallelism during LLM training process. Secondly, ICCL introduces a primary-backup queue pair (QP) mechanism between each GPU pair. When a NIC port happens, ICCL switches to the backup QP, enabling runtime state migration between QPs and breakpoint retransmission to tolerate NIC port downs. Thirdly, ICCL exploits the narrow-waist abstraction of Remote Direct Memory Access (RDMA) to monitor throughput in $O(\mu s)$ granularity. It records the generation timestamps of Work Requests (WRs) and Work Completions (WCs), and utilizes a sliding window to smooth out fluctuations measured by the naive per-message scheme.

We implement and open-source ICCL at Github [4]. We evaluate the CC and LLM training performance in a production cluster. Experimental results show that, compared with NCCL, the DPDK-like design of ICCL can deliver 23.4% higher P2P throughput and 28.5% lower P2P latency, as well as improve the training throughput by 6.02%. When RDMA NIC (RNIC) port failures occur, ICCL's primary-backup QP mechanism can maintain 76.6% CC throughput and almost the same training performance as in normal conditions, which does not crash the entire training process. For RDMA network anomalies, ICCL's window-based monitor can capture transient throughput drops in $O(\mu s)$ granularity with low system overhead. We deploy ICCL in our customers' clusters over several months, and present several experiences and lessons when integrating ICCL with real-world large-scale GPU clusters. We hope this work can give communities more insights on the production-level CCL in LLM training.

The main contributions of this work include:

- We identify that current CCLs fall short of realizing efficient, reliable collective communication with sufficient observability in large-scale GPU clusters (§2).
- We design ICCL, comprising of a DPDK-like mechanism for P2P primitives, a primary-backup QP mechanism to tolerate NIC port downs, and a verb-based profiler for inter-host communication (§3).
- We implement and evaluate ICCL, with results validating its efficiency, reliability and observability (§4). We also share our experience of deploying ICCL in real-world production GPU clusters (§5).

This work does not raise any ethical issues.

2 Background and Motivation

2.1 Preliminary

Distributed LLM training. The size of LLMs and training dataset are experiencing a huge explosion in recent years. Current LLM training frameworks [3, 6, 43] support multiple parallelism strategies to enable efficient training among distributed GPUs, including: 1) data parallelism (DP) partitions the dataset into several subsets and dispatches each of them to a GPU; 2) tensor parallelism (TP) splits the computation of large tensors across multiple GPUs; 3) pipeline parallelism (PP) divides the model into stages, with each stage assigned to a different GPU; 4) Mixture-of-Experts (MoE) parallelism activates a subset of experts per input, with each placed on a different GPU. Each parallelism strategy involves massive data exchange among GPUs, necessitating a performant communication mechanism for efficient LLM training.

Collective communication library. CCL is a fundamental component for distributed LLM training, which offers a range of primitives to allow GPUs within the same group to perform continuous send and receive actions, for example, `allreduce`, `allgather`, `reducescatter`, `alltoall`, `send/recv`, `broadcast`. Generally, CC workloads are prevalent for various parallelism strategies in LLM training. For example, 1) DP uses `allreduce` to aggregate local gradients into a global gradient; 2) TP employs `allgather` and `reducescatter` to exchange activations, gradients and optimizers; 3) PP utilizes `send/recv` to synchronize activations and gradients, and 4) MoE parallelism leverages `alltoall` to dispatch tokens and outputs among experts. With increasing scale and complexity of LLM training, CCL has become more crucial in both academia and industry [2, 7, 10, 16, 40, 46, 50].

NCCL. NCCL is the most widely used CCL in the industry. It provides the interfaces and efficient implementation of various CC primitives optimized for NVIDIA GPUs. To enable CC among GPUs, firstly, NCCL creates a bootstrap network among decentralized ranks to gather their IP addresses and ports. Then, each rank detects the intra-host topology and constructs a graph, including hardware specifications (GPUs & NICs) and interconnections (PCIe & NVLink). Based on the graph, NCCL searches the optimal path between each GPU pair, and establishes logical channels (ring and tree) with appropriate transports (P2P, shared memory or network)

Table 1: NCCL SM utilization of P2P workloads.

Metric \ Workload	Intra-host P2P	Inter-host P2P	8×8 `alltoall`	16×16 `alltoall`
SM utilization (%)	23.1	3.2	24.7	4.1

to connect GPUs with maximized bandwidth. Once initialized, developers can invoke NCCL APIs to issue CC requests executed by GPUs.

GPU architecture and programming. We take NVIDIA GPUs as examples without loss of generality. GPU is comprised of: 1) multiple SMs, each of which includes many CUDA cores, registers and caches to execute kernel functions; 2) shared high bandwidth memory (HBM), a global DRAM with high memory bandwidth for SMs. Developers can write and submit kernels using CUDA toolkit [12], and CUDA runtime schedules them to multiple GPU SMs for parallel computation according to pre-defined resource constraints, e.g., numbers of grids and blocks. Due to the superiority in executing GEMM with high parallelism and generality, GPUs serve as the fundamental accelerator both in the present and foreseeable future of AI explosion era.

RDMA. RDMA is a kernel-bypass technology that allows clients to access the remote server's memory without CPU involvement. The basic transmission unit is QP. To begin with, the client application exchanges QP metadata with the server, such as QP number and *rkey*. To transmit a message, the client application uses unified verbs [15] to post a WR, and waits for the RNIC to send the message residing in the main memory. After processing a WR, the RNIC generates a WC to notify the application with completion status, e.g., success and error. Compared with traditional TCP/IP, RDMA delivers much higher throughput and lower latency, satisfying the stringent network requirement of large-scale LLM training.

2.2 Challenges in Large-Scale Communication

Although NCCL has become the *de-facto* CCL in production-level GPU clusters, we still encounter several challenges in real-world scenarios, falling short of several essential properties of efficiency, reliability and observability.

Non-negligible SM consumption. GPUs are designed to execute parallel GEMMs in deep learning, but we observe that CC primitives that do not involve reduction, such as `send/recv`, `alltoall`, `broadcast` and `allgather`, also consume SM resources in NCCL. For example, Figure 1 illustrates the basic procedure of inter-host `send/recv` primitives of NCCL. When the sender launches `send` kernels to transmit data to the remote receiver, NCCL involves three steps: 1) data preparation: sender APP prepares source data in the application buffer at GPU HBM; 2) buffer copy: sender GPU launches kernels to copy the application buffer to the chunk buffer, indicating the data is ready to be sent; 3) data transmission: network proxy posts WRs and notifies the sender RNIC, then GPU launches kernels to transmit packets to the remote RNIC by GDR. The workflow of `recv` is on the contrary to `send`.

We conduct a NCCL-Tests [8] experiment to evaluate the

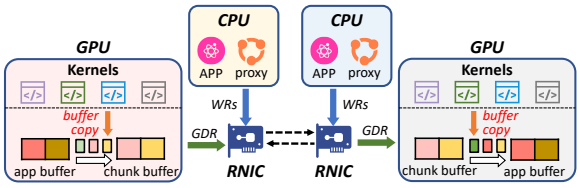


Figure 1: NCCL P2P w/ kernels

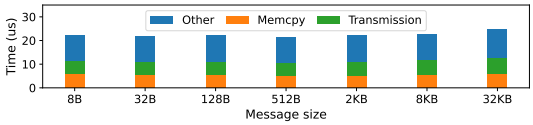


Figure 2: Overhead breakdown of NCCL P2P.

GPU SM utilization of `send/recv` and `alltoall` workloads on 2 servers, each of which is equipped with 8 NVIDIA Hopper GPUs and 9 NVIDIA ConnectX-7 RNICs. The inter-host topology follows a 1:1 oversubscribed two-tier CLOS topology with 400Gbps links. And we use Nsight [13] to monitor the GPU SM utilization. Table 1 presents the SM utilization of different P2P workloads. Although `send/recv` and `alltoall` are reduction-free operations, NCCL still consumes non-negligible SM resources. What's worse, SM occupation by kernels will not be freed until the entire P2P process finishes. With frequent P2P communication in LLM training, these kernels can continuously occupy the SM resources. For better training efficiency, our customers expect more GPU resources to be dedicated in GEMM and minimize SM wastes in communication tasks.

Cumbersome operations. In addition to SM consumption, we also notice there are redundant operations, while not directly involved in the communication process, still incur a considerable consumption of resources during a P2P process. We profiled the time consumption of each step in NCCL P2P demonstrated in Figure 1 with different message sizes using the above testbed. As shown in Figure 2, the duration of memory copies from the application buffer to the chunk buffer, which does not include the actual P2P communication, is comparable to data transmission, accounting for nearly 25% of the entire P2P process. These cumbersome memory copies can slow down the P2P communication in terms of bandwidth and latency. Given the frequent P2P communication in LLM training, our customers prefer a lightweight solution that eliminates the redundant operation in P2P implementation.

Frequent RNIC port downs. As inter-host communication is embracing much higher bandwidth than before, we encounter more frequent downs of RNIC ports within and beyond 400Gbps bandwidth. Figure 3 collects statistics on the failure types over 10 months in 2024 within a GPU cluster. It is obvious the RNIC port downs, caused by optical modules and RNIC hardware, contribute the most failures than GPUs and miscellaneous types. The reasons behind this phenomenon can be roughly summarized as: 1) device overheat-

Due to page limits, we only illustrate inter-host P2P process. For intra-host P2P, the data transmission is different where an inter-process `memcpy` is invoked from source GPU to destination GPU via NVLink or PCIe.

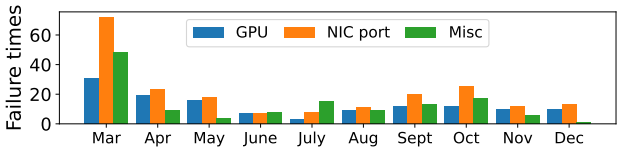


Figure 3: Failure statistics in 2024.

ing when handling the overwhelming packets with 400Gbps; 2) poor quality control due to rushed development and delivery by manufacturers.

In CC scenario, RNIC port downs can lead to severer performance drops than traditional workloads. This is because a CC completion depends on whether each participating node within the group performs the CC operation successfully. If a port down happens, the affected node fails to send data to its subsequent node via inter-host networks, which results in stragglers and suspension of the CC operation, and even crash of the entire LLM training by timeout exceptions. Nevertheless, NCCL lacks a built-in mechanism to tolerate the potential downs of RNIC ports, which introduces much time & financial costs and GPU wastes to restart the training. Therefore, our customers are seeking a lightweight fault tolerance mechanism to resist frequent RNIC port downs.

Transient network anomaly monitor. CC operations usually involves both computation and communication, and anomalies may occur in either phase. Nevertheless, regardless of where the anomaly originates, it ultimately manifests as a bandwidth drop. This not only complicates the difficulty to localize the root cause, but also makes the networking team usually blamed as the source of issues, despite the underlying cause potentially residing outside the networks. It becomes urgent and necessary to provide an effective monitor to localize potential network anomalies, and to assist the networking team in demonstrating their innocence.

However, it is non-trivial to pinpoint a network anomaly for a CC operation. Traditional wisdom usually uses binary search to isolate the problematic node, but this post-failure mechanism is unlikely to reproduce the exact network anomaly in detail due to the complexity to replicate the in-situ environment. We point out it is more feasible to monitor the network performance in an online manner. Current network monitor systems in production clusters are based on hardware counters in $O(s)$ granularity, but most CC operations usually finish within $O(ms)$ and even $O(\mu s)$. It prevents these network monitors from capturing the transient anomalies happened in the communicating phase of a problematic CC operation, which can greatly straggler the entire CC and training process. Although NCCL can profile completion results, it lacks visibility into the inner states during CC process. Therefore, our customers are demanding an observable CCL to monitor transient network anomalies for a CC operation.

Operators can reproduce the CC workloads within a subset of the cluster. If similar anomalies happen, it proves that the problematic node resides in this subset, otherwise it is located in the remaining subset. Operators can half the search space in an iterative manner until the fault node is found.

2.3 NCCL Analysis and Existing Works

The fundamental reasons why NCCL fails to satisfy the aforementioned demands of our customers are three-fold. Firstly, NCCL is a GPU-native CCL, with the design philosophy of assigning GPUs to take both control of the computation and communication to serve NVIDIA's business interests. It thus introduces inevitable and non-negligible burdens to GPUs for executing the communication, compromising SM utilization for general computation tasks. Secondly, NCCL is essentially an acceleration-oriented communication stack with more attention on efficiency than reliability and observability. However, as the GPU training clusters are becoming larger with higher-speed interconnections, network failures and performance degradations happen frequently, which make fault tolerance and fine-grained network monitor urgent than ever before.

Some works have been proposed to optimize CC in different aspects. For better GPU resource utilization, Centauri [22], CoCoNet [31] and Wang et al. [48] fuse kernels for fewer launches, and HFReduce [18] offloads the reduction of `allreduce` from GPUs to CPUs, but none of them consider GPU SM consumption for P2P primitives. For better reliability, industrial communities [18, 24, 25, 33, 44] tolerate device failures by longer NCCL timeout, traffic engineering, dual-plane switches and faster checkpoints, but they either lack timely analysis & tolerance on failures, or require hardware supports that hurt deployability. For higher observability, Microsoft [19, 27] and ByteDance [37, 38] visualize performance anomaly with end-host probing and logging, but need frequent tracks on QP runtime states, resulting in non-negligible overhead and a long time to complete a profile. Although sketches implemented in programmable switches [55] can monitor $O(\mu s)$ network dynamics, programmable switches are difficult to deploy in production-level clusters.

3 ICCL Design

3.1 Overview

Figure 4 demonstrates ICCL's overview. Generally, ICCL works as a middleware between upper-layer LLM training frameworks [3, 6, 43] and underlying heterogeneous hardware. ICCL uses the unified programming interfaces (CUDA and RDMA verbs) to enable CC among distributed GPUs during the training process. ICCL is developed based on NCCL, enabling the communication group with bootstrapping, topology search & graph construction and channel establishment. Additionally, ICCL introduces three extra modules: 1) DPDK-like P2P for resource-efficient and performant communication; 2) a primary-backup QP mechanism for fault tolerance; 3) a window-based monitor to capture transient network anomalies. These three modules are for efficient, reliable and observable CC, respectively.

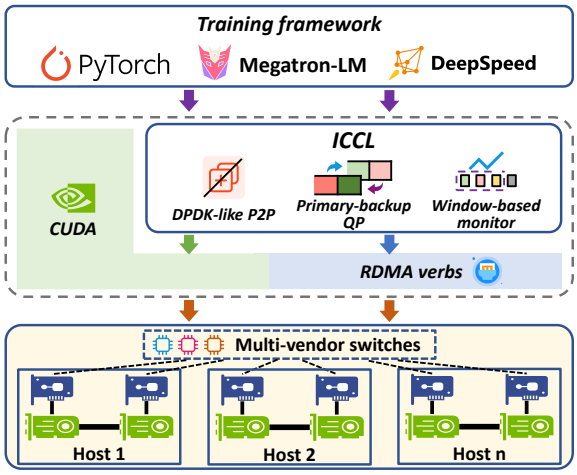


Figure 4: ICCL overview.

3.2 DPDK-like P2P

To achieve the objective of eliminating SM consumption and cumbersome operations in P2P communication, ICCL adopts the design philosophy of DPDK [1] and develops an efficient P2P mechanism. More specifically, inspired by DPDK's kernel-bypass design that migrates packet transportation to userspace, ICCL's P2P communication bypasses the CUDA programming, and migrates the P2P process from GPU kernels to CPU threads, removing any kernel launches at GPUs. Moreover, similar to DPDK's zero-copy design that allocates a dedicated buffer without memory copies between kernel space and userspace, ICCL allows RNICs to access the application buffer at GPU HBM without copies to the chunk buffer, thereby removing redundant operations.

SM-free P2P in ICCL. In a GPU training cluster, CPUs always exhibit much lower utilization and higher redundancy than GPUs [18, 29, 32]. Therefore, it is feasible to *migrate the entire P2P operations from GPUs to CPUs* with removal of kernel launches at GPUs, thereby sparing more GPU SM resources for general computation tasks. Figure 5 shows ICCL's SM-free design for P2P primitives. The basic procedure is similar to NCCL, with two additional modifications. Firstly, at the sender side, when the APP prepares the data at the application buffer, ICCL threads at CPU invoke `cudaMemcpy`, an interface provided by CUDA, to copy application buffer to the chunk buffer, instead of GPU kernels in NCCL. The invocation is asynchronized, with associated parameters of memory size & addresses, and transfer type `cudaMemcpyDeviceToDevice`. Secondly, ICCL invokes `cudaStreamQuery` to query the completion of buffer copy operation. If `cudaSuccess` is returned, ICCL allows the subsequent transmission to the remote peer.

During the training process, we need to guarantee a correct execution order, i.e., operations in the subsequent communication/computation stream cannot be executed until the prior computation/communication stream finishes. With kernel involvement, the training frameworks can control their

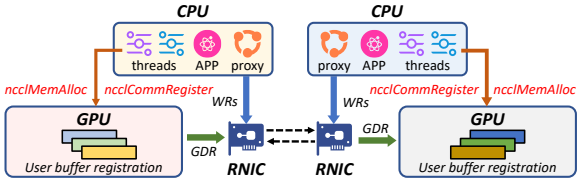


Figure 5: DPDK-like P2P in ICCL.

execution order by invoking a `wait` API to insert an event, pausing the subsequent stream until it receives the event notification in the prior stream. Nevertheless, ICCL's SM-free P2P prevents us from invoking `wait` to order streams. This is because ICCL offloads the entire process of P2P primitives to CPU, such that the SM-free P2P communication cannot wait for event notifications from GPU computation stream, lacking an execution dependency to order operations of P2P communication and computation. This requires us to develop another mechanism to build the stream dependency.

With deep investigation of CUDA programming, we find it is feasible for CPU application to control the stream order with host functions by calling `cudaLaunchHostFunc`, and a host function can be inserted to a stream to wait for event notification from the prior stream. Additionally, it can also work as a barrier to block the execution of subsequent operations in the same stream. Therefore, ICCL inserts two host functions in the communication stream for a P2P operation: the first one to wait for event notification from the prior computation stream, and the second one to barrier the subsequent operations in the same communication stream. When the first host function receives event notification from the computation stream, a P2P communication can be triggered, and simultaneously the communication stream proceeds to the second host function. It runs a `while` loop to block the subsequent operations in the same stream. When P2P communication finishes, the second host function exits the loop and allows subsequent operations in the communication stream.

Zero-copy P2P in ICCL. Although SM-free P2P relaxes the SM consumption at GPUs, we notice there are still redundant memory copies from the application buffer to chunk buffer. To further reduce cumbersome operations within a P2P process, ICCL leverages a zero-copy design. More specifically, ICCL utilizes User Buffer Registration [17] provided by CUDA to remove the memory copies. It allows us to directly send or receive the application buffer without additional memory copies to the chunk buffer. As depicted in Figure 5, when an upper-layer application initiates a P2P primitive, ICCL firstly intercepts the memory registration launched by training framework, then registers a user buffer via `ncclMemAlloc`, and returns its pointer and allows RNICs to directly access the memory region for P2P transmission and reception. With this mechanism, ICCL removes the overhead of copying application memory to the chunk buffer during a P2P process. Besides, the zero-copy mechanism further mitigates potential hangs that could occur when memory copies and host functions are invoked concurrently, ensuring the normal P2P

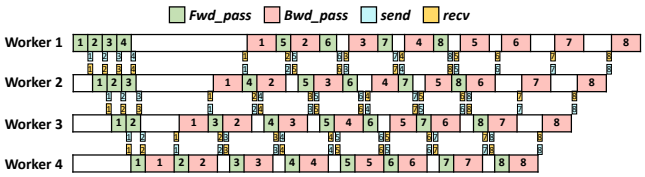


Figure 6: PP with ICCL.

communication during the LLM training.

Improve overlaps in PP. In addition to saving SM resources and reducing memory copy overhead, ICCL's DPDK-like P2P can also eliminate the SM competition between GEMM computation and P2P communication, allowing ICCL to overlap them for further training speedup. For example, pipeline parallelism uses P2P to synchronize the activations/gradients of forward/backward passes among GPU workers. However, P2P communication in NCCL have to compete for GPU SM resources with forward & backward passes, leading to compromised computation performance during overlap. In contrast, ICCL offloads P2P communication from GPUs to CPUs, and thus spares the SM resources for computation, empowering faster forward & backward passes and better overlaps to hide the P2P communication overhead.

Figure 6 shows ICCL's PP progress using 1F1B strategy [42] with 8 microbatches and 4 GPU workers. Unlike kernel-based P2P that conducts computation and P2P communication in sequence, ICCL overlaps the P2P communication with forward & backward passes during the PP process. Generally, after finishing a forward/backward pass of a microbatch, ICCL allows the worker to proceed to the backward/forward pass of the next microbatch, and simultaneously `send` the activation/gradient to the next/previous worker. As a receiver, the worker can also `recv` the results from a sender, while performing a forward or backward pass at the same time. Compared to naive PP by NCCL, ICCL overlaps the P2P communication and enables faster forward & backward passes with more SM resources, therefore accelerating the entire LLM training.

3.3 Fault Tolerance with Backup QPs

Why primary-backup QPs? Several approaches can realize fault tolerance in real-world clusters. For example, operators can use a dual-ToR design for switches [44] to tolerate single point of switch failure, or NIC bonding with primary-backup ports to resist single point of link failure. However, these solutions are neither deployable (e.g., dual-ToR requires switch modification) nor general (e.g., bonding requires dual-port NICs, which are not available as some customers prefer single-port NICs for easier configuration and better stability [44]).

Therefore, we resort to a flexible scheme with a primary-backup QP mechanism, which is more general and cost-efficient for our customers. During the bootstrap for the inter-host connection, in addition to a primary QP that uses the closest RNIC, ICCL creates a backup QP by using the second closest RNIC for each GPU. If a link fails due to port downs, ICCL migrates the traffic from the primary QP to the backup QP, and retransmits the data at the breakpoint. ICCL also monitors the failed link, and when it is fixed, ICCL turns back to the primary QP for faster transmission. Note that if the RNIC has dual ports, ICCL creates the backup QP on the other port of the same RNIC, enjoying the same hardware distance as the primary QP.

To make `send/recv` more noticeable, we draw the sizes of a `send/recv` operation comparable to GEMM process, but in fact they cost much less time than GEMM, resulting in much smaller bubbles.

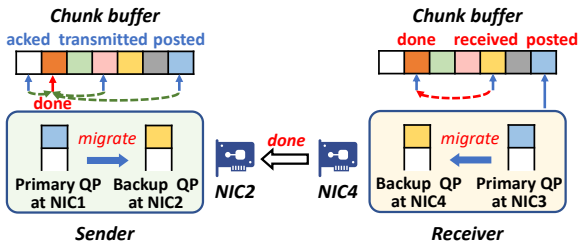
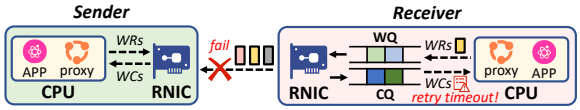


Figure 7: QP state migration.

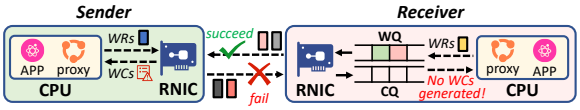
State synchronization and migration. To enable retransmission at the breakpoint (i.e., the last chunk failed to be sent), it is crucial to synchronize and migrate the states from the primary QP to the backup QP. ICCL maintains three pointers to represent the transmission and reception states at both sender and receiver, as shown in Figure 7. At the sender side, *posted* denotes the prepared chunk by GPU, and *transmitted* implies the CPU network proxy generates a `send` WR and starts transmitting the data by invoking `ibv_post_send`, and *acked* indicates the sender gets the WC acknowledged by the receiver. At the receiver, similarly, *posted* denotes the chunk ready for receiving, and *received* implies CPU network proxy generates a `recv` WR and starts receiving data by invoking `ibv_post_recv`, and *done* indicates data reception is finished and copied to the application buffer. The *done* location at receiver is synchronized to the *acked* location at sender with completion of every chunk transmission and reception.

If an RNIC port down occurs, since the sender is unaware of whether the inflight data is received by the receiver or not, we choose a receiver-driven strategy to let backup QP start retransmission from the last acknowledged chunk. Before the retransmission, the receiver actively retreats *received* to *done* to make backup QP start receiving data at the breakpoint. Then, receiver pushes *done* location to sender's *acked* to ensure the consistent breakpoint at both sides. With mutual agreement of breakpoint location, the sender also retreats *posted* and *transmitted* to *acked*, notifying the backup QP to restart transmission from *acked*. When the RNIC port downs are fixed, ICCL migrates the states from backup QP to the primary QP with the same aforementioned process.

Trigger QP switching. It is essential to ensure timely and secure triggers for QP switching. We present two trigger conditions of QP switching in case of different failure cases. Figure 8(a) illustrates the first case where a receiver cannot



(a) Case 1: triggered by local WC.



(b) Case 2: triggered by timeout threshold.

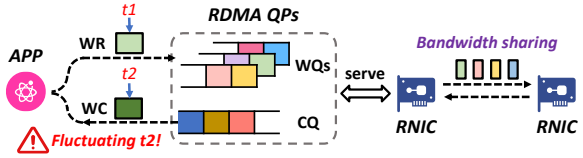
Figure 8: Trigger primary-backup QP switching.

notify the sender with buffer preparation due to failed communication from receiver to sender. In this case, after several retries and exceeding ICCL's retry timeout threshold (determined by `ICCL_IB_TIMEOUT` and `ICCL_IB_RETRY_CNT`), receiver RNIC creates a WC error report to notify CPU with failed communication. After receiving error report, CPU is aware of the RNIC port downs and thus triggers QP switching.

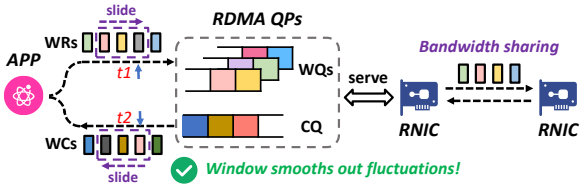
The second case shown in Figure 8(b) is more complicated, where RNIC port fails during the process when receiver notifies sender with buffer preparation, leading to successful communication from receiver to sender but failed reverse transmission. In this case, the sender can obtain WC's retry timeout report, but the receiver is unaware of the retry timeout error happened at the sender and does not generate WCs to notify the CPU. Consequently, the receiver CPU cannot trigger QP switching according to error reports from local WCs directly. Therefore, we trigger QP switching according to a fixed timeout threshold δ. When an WR is issued, the receiver records its timestamp and waits for its relevant WC. If no WCs are generated after timeout δ, the receiver resends a `CTS` (clear to send) message to check the link state, and if failed, an error WC is then generated at the receiver to trigger QP switching as in the first case. The reason why we develop a resend mechanism is that, even if the current link is normal, the relevant sender may not transmit data to the receiver when it is stalled waiting for data from the preceding node in the CC process, exhibiting the same anomaly as this link fails. Consequently, a resend mechanism double-checks the link status, preventing an innocent normal link from being misclassified as faulty due to data dependency on upstream failed link. Generally, δ can be slightly larger than ICCL's retry timeout, due to the inevitable queuing and propagation delay.

3.4 Transient Network Anomaly Monitor

Although it is non-trivial to realize fine-grained network observability in current monitoring mechanism (discussed in §2.2), fortunately, the narrow-waist abstraction of RDMA programming [34, 36] can help us capture $O(\mu s)$ inter-host network dynamics during a CC operation. RDMA uses *verbs*, an unified interface to operate RNIC hardware for data transmission and reception. As a result, all inter-host CC traffic



(a) Per-message scheme.



(b) Per-window scheme.

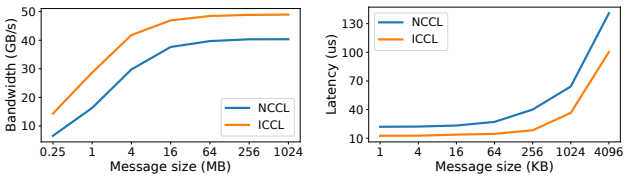
Figure 9: Throughput estimation.

via RDMA can be interpreted into verb combination, making it feasible to capture comprehensive and transient communication states inside a CC operation through verb invocation.

Naive per-message estimation. To monitor fine-grained CC performance, naively, we can record the timestamps of generating WRs and WCs to estimate the instant throughput for each message, as shown in Figure 9(a). When the sender application posts an WR to one of the WQs, ICCL records the timestamp t_1, and when application receives the relevant WC from CQ, ICCL records the timestamp t_2. Consequently, the instant throughput B for this message M can be computed by $B = \frac{\omega(M)}{t_2 - t_1}$, where $\omega(M)$ is the message size of this WR, and $t_2 - t_1$ denotes the total processing duration of M. However, the naive per-message throughput estimation is far from accuracy. This inaccuracy stems from inevitable competition of multiple messages sharing the link bandwidth, leading to unstable completion time to transmit M. For example, when packets from multiple messages are sharing the link bandwidth, the completion of M is much longer than the scenario where there is only M exclusively occupying the bandwidth. In this case, the relevant WC generation timestamp t_2 fluctuates greatly, leading to inaccurate results of instant throughput.

Per-window estimation. Although the per-message scheme is fluctuating, we can smooth out the throughput fluctuation with multiple messages. Therefore, ICCL uses a sliding window to estimate the average throughput inside the window, which is shown in Figure 9(b). When a new WR is posted to the target WQ, ICCL slides the window and records the timestamp t_1 of the first WR inside the window, and waits for RNIC to serve all WRs and transmit all messages within the window. With generation of a new WC from the CQ, ICCL slides the window and records the timestamp t_2 of the last WC inside the window. Consequently, we can estimate the window's average throughput by $\overline{B} = \frac{\sum_{i \in W} \omega(M_i)}{t_2 - t_1}$, where $i \in W$ indicates the i^{th} message inside the window W.

The success of window-based scheme stems from the fact that it smooths out the instant fluctuation by shadowing the stochastic completion timestamp of each message inside the



(a) P2P bandwidth. (b) P2P latency.

Figure 10: Throughput and latency performance.

Table 2: Kernel invocation and SM consumption.

	Operations (percentage)	SM utilization
NCCL	`ncclDevKernel_SendRecv` (68.8%)	3.2%
	`verifyPrepare` (27.0%)	
	`prepareInput2` (3.5%)	
	[CUDA memset] (0.4%)	
	[CUDA memcpy Host-to-Device] (0.4%)	
ICCL	`verifyPrepare` (86.1%)	2.8%
	`prepareInput2` (11.4%)	
	[CUDA memset] (1.3%)	
	[CUDA memcpy Host-to-Device] (1.2%)	

window, and uses a window-sized number of messages to reflect the average throughput rather than per-message instant throughput. In practical usage, a proper window size is crucial to measuring accuracy. With a small window size, ICCL cannot cover the fluctuation comprehensively, leading to measuring inaccuracy similar to the naive per-message scheme. However, if the window size is too large, ICCL can smooth out all subtle changes caused by real network anomalies of a CC, making ICCL always obtain an unchanged throughput and failing to capture transient CC dynamics. In practical usage, developers should optimize the size of sliding window according to their cluster environment and network dynamics.

4 Evaluation

Based on NCCL v2.21.5, we implement ICCL with 4,000 lines of C codes, which is publicly available at Github [4]. We evaluate ICCL in three dimensions: 1) how efficient is ICCL's DPDK-like P2P design to improve CC performance and overlaps in PP (§4.2)? 2) how reliable is ICCL's primary-backup QP mechanism to tolerate RNIC port downs (§4.3)? 3) how observable is ICCL's monitor to perceive transient CC's network anomalies (§4.4)?

4.1 Experimental Setups

Cluster environment. To avoid interfering with LLM training, we conduct experiments on the subset of servers in a production cluster, including: 1) 1024 NVIDIA Hopper GPUs; 2) NVIDIA ConnectX-7 RNICs [11] and BlueField-3 DPU [9]. Each server is equipped with 8 homogeneous GPUs and 9 RNICs, and they are connected with a 1:1 oversubscribed two-layer CLOS topology network with 400Gbps bandwidth.

Workloads and baseline. We use GPT-2 [45], an open-source dense LLM with different sizes of 6B, 32B, 70B, 177B and 314B as the training workloads. To perform large-scale distributed training, we integrate ICCL into Megatron-LM [6] with 3D parallelism. We also use NCCL-Tests [8] to generate some typical CC workloads, and Nsight [13] to visualize the runtime kernel invocation and SM utilization. As for baseline, we compare ICCL with NCCL v2.21.5 in terms of training throughput in Tera Floating Point Operations Per Second (TFLOPS), algorithm bandwidth in GB/s, and system overhead of GPUs and CPUs. The training hyperparameters and default settings are listed in Table 4 in Appendix.

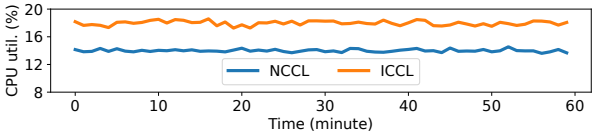


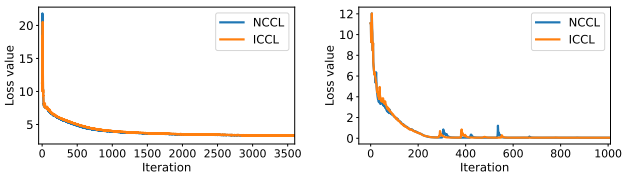
Figure 11: CPU utilization.

4.2 Efficiency

P2P performance. Figure 10 compares the network performance of `send/recv` workloads generated by NCCL-Tests. For bandwidth performance, as we increase the message size in Figure 10(a), ICCL's algorithm bandwidth outperforms NCCL by at least 20.12% when the message size is 1GB. This is because the zero-copy design of ICCL eliminates the frequent and consuming memory copies from the application buffer to the chunk buffer with User Buffer Registration. For latency performance, we also notice that ICCL achieves at least 28.5% lower P2P latency than NCCL when the message size is small. The reason is that in addition to the elimination of buffer copies, CPU offloading reduces the signal synchronization between GPUs and CPUs, whose overhead can dominate the entire process of P2P primitives especially with small messages. Therefore, the results prove ICCL's feasibility and efficiency to deliver better P2P performance with DPDK-like design for both large and small message sizes.

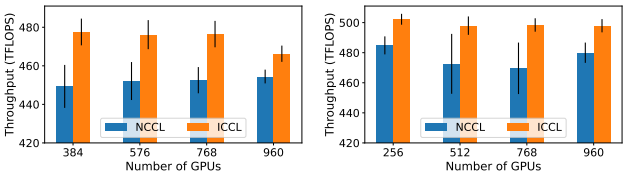
Resource consumption. Table 2 shows the kernel invocation and SM utilization of inter-host P2P communication using NCCL and ICCL. The second column in the table represents the invoked kernels and their respective percentage of executing duration, which are collected by Nsight. Obviously, NCCL invokes a `send/recv` kernel `ncclDevKernel_SendRecv` to perform the reduction-free P2P communication, which accounts for 68.3% execution time and 3.2% SM usage. Instead, except for two inevitable NCCL-Tests kernels (i.e., `prepareInput2` for data preparation and `verifyPrepare` for result verification), ICCL does not launch any GPU kernels to perform the P2P communication, thereby consuming lower SM utilization than NCCL. Note that in practical LLM training, `prepareInput2` and `verifyPrepare` will not be invoked, such that ICCL's SM-free P2P can achieve almost zero SM consumption, which can be scheduled in computation to accelerate the LLM training process.

Figure 11 compares the CPU utilization of a GPU server using NCCL and ICCL, respectively. Although ICCL's SM-free design offloads P2P from GPU kernels to CPU proxies, it



(a) 6B model with 40G dataset. (b) 32B model with 400M dataset.

Figure 12: Loss values.



(a) 177B model. (b) 314B model

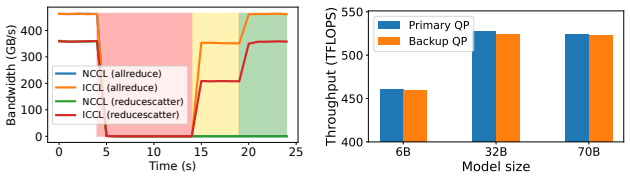
Figure 13: Training throughput.

only introduces about 4% more CPU utilization than NCCL, due to the zero-copy design that greatly reduces memory copy overhead at CPU proxies. The result validates ICCL's deployability and feasibility in real-world GPU cluster, where a low CPU utilization is essential to maintain the CC efficiency (discussed in §6).

Training performance. We present the training performance of different LLMs and GPU numbers, with ICCL integrated into Megatron-LM to enable overlap between forward & backward passes and `send/recv` operations during PP. For training accuracy, Figure 12(a) and (b) shows that ICCL exhibits the same loss value trend as NCCL for GPT-2 in different sizes, proving that ICCL's DPDK-like design using `hostFunc` guarantees the correct interaction and order between computation and communication as kernels do in NCCL. For training throughput, as Figure 13(a) and (b) show, ICCL achieves about 490 TFLOPS averagely in different model sizes and GPU numbers, which outperforms NCCL's training throughput by at most 6.02%. The reasons are twofold. Firstly, ICCL not only empowers overlap between computation and communication during PP, but also improves P2P bandwidth with zero memory copies for faster synchronization for activations and gradients among GPU workers. Secondly, the offloading mechanism with SM-free P2P ensures more SM resources can be allocated to computation tasks for faster forward & backward passes. The results validate that ICCL's zero-copy and SM-free P2P designs both contribute to the improvement of overall training performance.

4.3 Reliability

Runtime bandwidth. To evaluate fault tolerance, we manually bring down a specific RNIC port from 4s to 19s, whose failed duration exceeds the default retry timeout, and bring up the RNIC port at 19s again. Figure 14(a) compares the runtime bandwidth of NCCL and ICCL with `allreduce` and `reducescatter` workloads. The red area denotes the retry period, and yellow area denotes ICCL is transmitting using the



(a) CC bandwidth performance. (b) Training performance.

Figure 14: Primary-backup QP for fault tolerance.

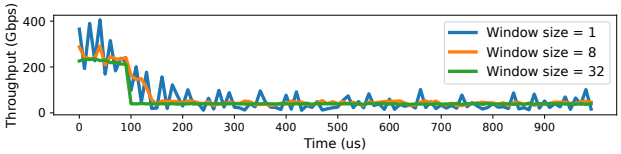


Figure 15: Monitor results with different window sizes.

backup QP, while green area denotes ICCL switches back to the primary QP when the RNIC port is up again. When a port down occurs, NCCL and ICCL spend about 10s to reconnect, both of which stay at 0GB/s during the retry period. After retry timeout, NCCL terminates the connection and the entire CC, failing to resume the communication due to the lack of fault tolerance. In contrast, after 10s retry, ICCL switches to the backup QP using another RNIC port in good condition, and for `allreduce` and `reducescatter` workloads, ICCL resumes 76.6% and 58.1% bandwidth of primary QP's performance, respectively. Additionally, when the RNIC port is brought up again, ICCL can switch back to the primary QP and deliver the ideal bandwidth.

Training throughput. Figure 14(b) presents the training throughput when fault tolerance is triggered to use the backup QP for transmission during training. When the RNIC port fails, ICCL's backup QP still delivers a comparable training throughput as the normal scenario using the primary QP, with TFLOPS only declines by 0.38% at most. It proves that our primary-backup QP mechanism can tolerate RNIC port failure and maintain a reliable training even with failure.

4.4 Observability

Runtime throughput. To evaluate the observability, we generate a background P2P workload between a pair of servers, and insert a disturbance traffic to compete for bandwidth, making the background workload converge to a lower and stable throughput. Figure 15 presents the runtime throughput in a 10μs granularity with different window sizes. Especially, when window size is 1, it equals to the naive per-message scheme. We notice ICCL can capture the transient $O(\mu s)$ throughput changes, but shows a very fluctuating measuring result with per-message scheme. With a large window size of 32, ICCL can smooth out the fluctuation and reflects more accurate results closer to the ground truth, but fails to perceive the instantaneous changes. For example, when disturbance traffic arrives at 100μs, per-message scheme shows an intense fluctuation, but a large window size directly drops to the new converged throughput. Therefore, we set the window size as 8 to achieve the tradeoff between measuring accuracy and

fluctuation sensitivity in our environment.

System overhead. As shown in Table 3, we evaluate the system overhead of our window-based monitor in terms of CPU and memory. Generally, it does not incur much consumption on CPU and memory, both of which are still far from reaching their capacity limits. This proves the efficiency of ICCL's monitor, which is a feasible approach not affecting training performance in real-world training systems (discussed in §5).

Table 3: System overhead of anomaly monitor.

Hardware \ Scheme	w/o monitor	w/ monitor
CPU utilization	9.32%	21.1%
Memory utilization	1.7%	2.1%

5 ICCL Operating Experience

We deploy ICCL in real large-scale GPU clusters of our customers, and it has been operating stably for several months. Here we share our operating experiences in running ICCL in real-world environment, and hope our experiences and lessons can help developers build a better CC service in production-level LLM training clusters.

ICCL is not always the culprit. During the LLM training process, our customers sometimes complain their CC performance is not ideal with throughput drops, and they suspect there are potential bugs with ICCL. However, after we deep dive into the clusters, we find that ICCL is not the culprit of performance anomaly in most cases. We present two typical cases of unexpected CC performance drops.

Case 1: insufficient CPU cores. One customer built a platform that allows developers to occupy hardware resources (including numbers of GPUs, CPU cores, DRAM) to train their LLMs. However, they found the CC throughput only achieved 30% of ideal bandwidth. They also used Nsight [13] to diagnose the training process, with results showing that GPU computation and other metrics except communication were normal as usual. They reported the issues and asked us to help them fix ICCL. Surprisingly, when we reproduced the training scenario in another GPU cluster, we found the CC performance was normal without any drops, proving that ICCL itself is not the culprit. We deep dived into their GPU clusters and finally identified the root cause. When allocating hardware resources for each developer, they unintentionally set an upper limit of at most 8 CPU cores per developer, in which case ICCL's network proxy did not have enough CPU resources to conduct communication.

Case 2: misconfiguration of GPU fan speed. One customer reported their CC performance encountered intermittent drops during LLM training, and Nsight [13] analysis showed that other metrics excluding communication worked well as usual. We inspected their cluster, and localized an anomalous GPU with higher temperature than the common cases. More specifically, they misconfigured the GPU by limiting its fan speed, such that it was overheated and slowed down the communication, resulting in compromised CC throughput.

According to our practical experience, CCLs themselves (including ICCL and NCCL) are more likely NOT to be the culprit of performance drops. To deliver an ideal CC performance, we encourage cluster operators to focus more on other software implementation and hardware health.

Efficiency of monitoring systems is essential. In large-scale GPU clusters, it is crucial to monitor the fine-grained runtime metrics (e.g., $O(\mu s)$-level throughput in §3.4) to help operators detect and pinpoint CC anomalies. However, we also point out the efficiency of monitoring system also matters, which can greatly impact the LLM training. We encountered a case where an inefficient monitoring system resulted in poor CC performance. One customer designed their own monitoring system by deploying the monitor agents at each host to output periodical logs of GPUs, RNICs and training procedure. Unfortunately, they found the CC performance presented intermittent throughput drops, while other metrics except communication were normal. After investigating their cluster settings, we found the average CPU utilization and memory footprint of monitoring hosts were higher than the common cases. Finally we identified the root cause: their monitor agents consumed significant CPU resources to report the overwhelming system logs. Particularly, when an agent tried to push new metrics to the centralized master, due to instability of front-end network, it outputted numerous timeout logs and retry events, occupying too much CPU resources that should be allocated for ICCL's network proxy. We suggest the operators of GPU clusters pay more attention to not only the granularity but also the efficiency of a monitoring system.

Use robuster networking configuration. We also point out networking configurations can influence ICCL's reliability. Our customer once reported their CC was terminated during training, and suspected the reason was ICCL's bugs. After in-depth investigation into their cluster, we localized the root cause was unexpected changes of global identifiers (GIDs). RDMA uses GIDs to mark each RNIC, each of which is mapped to a specific IP address in RoCEv2 scenario. In principle, the GID numbers are determined and remain unchanged once we initialize the communication group. However, we observed an exceptional scenario during training: when an RNIC port is down, the preallocated IP address for this port is temporarily missing; when the RNIC port is up again, even if we recover the same IP address, the GID number will increment by 1 and cannot be resumed as before. Consequently, the entire CC and LLM training crash due to GID changes.

We resolve this problem with two approaches. The first one is to modify the RNIC driver by canceling the GID incrementing logics even if IP address changes. Though it works in a bare-metal environment, this approach can result in unexpected errors in virtualized networks, which is still not robust and general in different clusters. Consequently, we explore the second approach by changing the configuration method for IP addresses in a lightweighted and reliable manner. We take

Linux Ubuntu operating system as an example. Ubuntu has two basic IP configuration methods, i.e., `NetworkManager` and `Networking` service. By `NetworkManager`, GIDs can change because of a temporary loss of IP addresses when RNIC port are down, illustrated in the aforementioned case. Instead, when we use `Networking` to configure the IP addresses, even if port down happens, its IP address still remains unchanged and avoids a temporary loss, leading to an unchanged GID number. Other Linux distributions can also use the similar approaches for IP address configuration.

Communication diagnosis via CPU offloading. Our customer once encountered a training anomaly during a PP process using NCCL. They complained their P2P communication time is larger than the computation time, which cannot be fully hided by the computation, leading to suboptimal training performance. However, due to the opaqueness of CUDA scheduler and complexity of kernel programming, it is difficult to diagnose the inner communication process and pinpoint the root cause of the straggler. After replacing NCCL with ICCL, due to the design of offloading kernels to CPUs, it allows us to easily instrument the P2P process running in CPU proxies, and to log and monitor the fine-grained execution of each step. Fortunately, we localize the straggler with ICCL. When setting up the model parameters, the computation of the first and last microbatches is larger than other microbatches in the PP process. Consequently, the send and receive communication of the middle microbatches have to wait for the completion of the first and last ones, and this dependency introduces substantial idle time and prevents effective overlap between communication and computation. Finally, by tuning the model parameter and training setting, the P2P communication is fully overlapped by computation. We believe ICCL's philosophy of offloading kernels to CPU threads is helpful to diagnose the fine-grained behaviors of the inner communication process.

LLM types. In the AI community, mainstream LLMs can generally be categorized into dense models (e.g., Llama 3 [26] and Qwen2.5 [52]) and MoE models (e.g., Llama 4 [5] and DeepSeek-R1 [28]). When serving customers elaborating on their own LLMs, we find different LLM types pose different requirements for CC performance, i.e., MoE models are much more communication-sensitive to CC performance than dense models. For example, when the runtime CC throughput drops from 360GB/s to 340GB/s, the training TFLOPS of dense models does not exhibit an obvious degradation, but MoE models can encounter a significant TFLOPS drop. The reason is that, the CC workload of dense models is mainly comprised of `allreduce`, `allgather` and `reducescatter` that introduce mild incast and congestion, such that the training throughput is bottlenecked by computation rather than communication. In contrast, MoE models involve significant `alltoall` traffic to dispatch and combine parameters among several experts. Our statistics show the ratio of computation and communication is 1:5 in an MoE model training, making

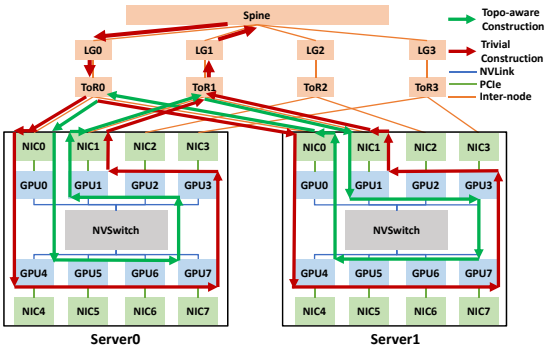


Figure 16: Channel construction in ICCL.

CC contribute much more to its training performance than dense models. For customers and companies training their MoE models, therefore, we advocate paying more attention to the CC performance.

Congestion control. As most CC primitives do not incur severe network congestion, (e.g., ring-based `allreduce`, `allgather`, `reducescatter`), we draw some lessons about how to cooperate ICCL with congestion control, for example, DCQCN [58]. On the one hand, if our customers are training dense models that incur mild incasts, tuning DCQCN parameters does not show an obvious improvement on training performance. As our network architecture already provides sufficient bidirectional bandwidth without oversubscription, we can disable the congestion control and only enable PFC [30] to ensure a lossless RoCEv2 environment. On the other hand, if our customers are training MoE models with significant `alltoall` workloads that generate severe incasts, it becomes necessary to tune performant DCQCN parameters for better CC performance. As a feasible solution, we can drive the DCQCN parameters in throughput-friendly or delay-friendly directions by flow distribution at runtime [23].

Communication scheduling. In practical usage of ICCL, the communication scheduling strategy also influences CC performance. For one thing, the numbers of channels and QPs can affect the bandwidth utilization. Multiple channels allow parallel and pipelined communication by dividing the entire data into several chunks. Multiple QPs improve routing entropy by splitting an elephant flow into several subflows to mitigate ECMP collision. However, multiple channels and QPs can introduce more hardware usage of GPUs and RNICs. For better CC performance, developers should optimize channel and QP settings by considering their traffic workload and hardware constraints.

For another, when establishing the ring, native NCCL selects GPUs with the same index at different servers as ingress/egress points deterministically. This results in significant inter-leaf traffic during inter-server communication, especially for DP communications. For example, in Figure 16, for all GPU servers, NCCL selects GPU0 and GPU1 as the ingress point and egress point respectively. As highlighted by the red lines, the traffic from GPU1 at Server0 to GPU0 at Server1 must go through leaf and spine switches, resulting in higher risks

of load imbalance and much longer latency. To address this, ICCL uses a topology-aware communication channel construction algorithm to reduce inter-leaf traffic. By rescheduling the egress point and ingress point on half of the GPU servers, ICCL specifies the traffic only going from one GPU to the GPU with the same index on any pre-built ring channel. As shown by the green lines in Figure 16, the traffic from GPU1 at Server0 to GPU1 at Server1 goes through only one hop (i.e., ToR1). According to our production statistics, topology-aware communication channel construction improves NCCL bus bandwidth by 5%∼15%.

Additionally, when pulling up a training task with MPI [14] for many GPU servers across a leaf switches, host order in MPI `hostfile` directly affects the degrees of cross-leaf traffic. For example, if host order is random in MPI `hostfile`, ring-based `allreduce` would induce substantial cross-leaf traffic. Therefore, we perform a topological sort to MPI `hostfile` to ensure that GPU servers under the same ToR switch are placed adjacently, making most servers are reachable within one hop and reducing ECMP collision risks.

6 Discussion

Computation anomalies of CC. ICCL enables $O(\mu s)$-level monitoring of RDMA to localize CC anomalies within the network. Actually, by analyzing a CC primitive's `opCount` (i.e., invocation counter of a CC API), computation anomalies of CC such as hang and slowdown at GPUs can be identified as well. Since the start of a communication is always followed by the end of a computation during training, when an anomalous GPU encounters a computational hang, the corresponding `opCount` on this rank will be smaller than that of other ranks within the communicating group, helping us pinpoint the anomalous GPU. Meanwhile, as different parallelism strategies are characterized with specific CC patterns, we can pinpoint the GPU experiencing computation slowdowns by analyzing invoking intervals of CC APIs [24]. Furthermore, according to the CC patterns of different parallelism, such as TP initiating an iteration with a `broadcast`, PP exclusively utilizing `send/recv`, DP starting `reducescatter` & `allgather` after the final backward of each rank, we can further localize the lagging training stages where GPUs experience slowdowns. We leave exploration of using ICCL to pinpoint computation issues of CC as our future work.

Extend ICCL to LLM inference. In large-scale LLM serving systems with parallelism, distributed GPUs also require CC to synchronize intermediate results, such as activations and KVCache [49, 56]. Although this paper mainly focuses on LLM training, we believe ICCL's philosophy can apply to LLM inference as well. For example, DPDK-like P2P can not only optimize throughput to fetch KVCache from a remote node, but also reduce latency to support efficient MoE model serving [57]. Meanwhile, the window-based monitor is essential to capture network anomalies in LLM serving, which is much more sensitive to latency jitters than training. Additionally, ICCL can tolerate the potential NIC port downs. We will explore how to further optimize ICCL in LLM inference in the future.

DPDK-like design for other reduction-free primitives. In this paper, we mainly focus on optimization of P2P primitives. Although there are other reduction-free CC primitives (e.g., `allgather` and `broadcast`), the reasons why we start with P2P primitives are two-fold. Firstly, P2P traffic usually resides in the critical path of inter-host communication with less bandwidth than intra-host, which is more likely to be the bottleneck of the LLM training, such that optimizing P2P implementation is expected to have higher improvement. Secondly, P2P primitives have easier implementation than other reduction-free primitives, due to less involvement of communication group and easier memory interaction at GPU HBM. We believe the DPDK-like philosophy of ICCL can also apply to other reduction-free CC primitives to reduce their SM consumption, which leaves as our future work.

7 Related Work

CC Optimization. Besides NCCL and various xCCLs from industrial community, many works from academia are proposed to optimize CC performance. Blink [47], SCCL [20], TACCL [46] and SyCCL [21] synthesize collective algorithms according to heterogeneous topology of intra-host and inter-host. TCCL [35], TECCL [40], MCCS [50], AutoCCL [51], ResCCL [39] and Wang et al. [54] optimize the CC scheduling and configuration by topology, traffic workload and user information. Despite promising, these works are not mature and comprehensive enough to address our challenges in production-level clusters.

Production-level LLM training. ByteDance [33], Meta [25, 26, 41], Alibaba [24, 44], DeepSeek [18, 28], Kuaishou [53] and Shanghai AI lab [29] share their practice on production-level LLM training, including optimizations on efficiency of computation & communication, reliability and observability for their large-scale GPU training clusters. However, they do not provide a systematic prospective on requirements, designs and deployment of collective communication. We expect ICCL together with existing works can contribute to a more thriving LLM training ecosystem.

8 Conclusion

In this paper, we design ICCL, an efficient, reliable and observable CCL with DPDK-like P2P, primary-backup QPs and window-based network monitor for production-level GPU clusters. We deploy and evaluate ICCL in large-scale production GPU clusters, with results showing that ICCL can 1) save GPU SM resources and deliver performant P2Ps for higher training speedup, 2) maintain training with RNIC port failures, 3) capture $O(\mu s)$ network anomaly. We also share ICCL's operating experience and lessons in large-scale environments, and hope this work can inspire LLM trainers to revisit CC and build better CCL service for their clusters.

References

[1] Data plane development kit, 2025. https://www.dpdk.org.

[2] Deepep, 2025. https://github.com/deepseek-ai/DeepEP.

[3] Deepspeed, 2025. https://github.com/deepspeedai/DeepSpeed.

[4] Iccl source code, 2025. https://github.com/sii-research/VCCL.

[5] The llama 4 herd: The beginning of a new era of natively multimodal ai innovation, 2025. https://ai.meta.com/blog/llama-4-multimodal-intelligence.

[6] Megatron-lm, 2025. https://github.com/NVIDIA/Megatron-LM.

[7] Microsoft collective communication library (msccl), 2025. https://github.com/microsoft/msccl.

[8] Nccl-tests, 2025. https://github.com/NVIDIA/nccl-tests.

[9] Nvidia bluefield-3 dpu, 2025.

[10] Nvidia collective communications library (nccl), 2025. https://github.com/NVIDIA/nccl.

[11] Nvidia connectx-7 datasheet, 2025. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf.

[12] Nvidia cuda toolkit, 2025. https://developer.nvidia.com/cuda-toolkit.

[13] Nvidia nsight systems, 2025. https://developer.nvidia.com/nsight-systems.

[14] Open mpi: Open source high performance computing, 2025. https://www.open-mpi.org.

[15] Rdma core userspace libraries and daemons, 2025. https://github.com/linux-rdma/rdma-core.

[16] Rocm communication collectives library (rccl), 2025. https://github.com/ROCm/rccl.

[17] User buffer registration, 2025. https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2273/user-guide/docs/usage/bufferreg.html.

[18] Wei An, Xiao Bi, Guanting Chen, Shanhuang Chen, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Wenjun Gao, Kang Guan, Jianzhong Guo, Yongqiang Guo, Zhe Fu, Ying He, Panpan Huang, Jiashi Li, Wenfeng Liang, Xiaodong Liu, Xin Liu, Yiyuan Liu, Yuxuan Liu, Shanghao Lu, Xuan Lu, Xiaotao Nie, Tian Pei, Junjie Qiu, Hui Qu, Zehui Ren, Zhangli Sha, Xuecheng Su, Xiaowen Sun, Yixuan Tan, Minghui Tang, Shiyu Wang, Yaohui Wang, Yongji Wang, Ziwei Xie, Yiliang Xiong, Yanhong Xu, Shengfeng Ye, Shuiping Yu, Yukun Zha, Liyue Zhang, Haowei Zhang, Mingchuan Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, and Yuheng Zou. Fire-flyer ai-hpc: A cost-effective software-hardware co-design for deep learning. In *SC*, 2024.

[19] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, et al. Empowering azure storage with rdma. In *NSDI*, 2023.

[20] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *PPoPP*, 2021.

[21] Jiamin Cao, Shangfeng Shi, Jiaqi Gao, Weisen Liu, Yifan Yang, Yichi Xu, Zhilong Zheng, Yu Guan, Kun Qian, Ying Liu, et al. Syccl: Exploiting symmetry for efficient collective communication scheduling. In *SIGCOMM*, 2025.

[22] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *ASPLOS*, 2024.

[23] Ziteng Chen, Menghao Zhang, Guanyu Li, Jiahao Cao, Yang Jing, Mingwei Xu, Renjie Xie, He Liu, Fangzheng Jiao, and Xiaohe Hu. Paraleon: Automatic and adaptive tuning for dcqcn parameters in rdma networks. In *ICNP*, 2024.

[24] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, et al. Evolution of aegis: Fault diagnosis for ai model training service in production. In *NSDI*, 2025.

[25] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *SIGCOMM*, 2024.

[26] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[27] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM*, 2016.

[28] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[29] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. Characterization of large language model development in the datacenter. In *NSDI*, 2024.

[30] IEEE. IEEE 802.1 Qbb - Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/, 2011.

[31] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *ASPLOS*, 2022.

[32] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *OSDI*, 2020.

[33] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. Megascale: Scaling large language model training to more than 10,000 gpus. In *NSDI*, 2024.

[34] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual rdma networking for containerized clouds. In *NSDI*, 2019.

[35] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. Tccl: Discovering better communication paths for pcie gpu clusters. In *ASPLOS*, 2024.

[36] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in rdma subsystems. In *NSDI*, 2022.

[37] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, et al. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In *SIGCOMM*, 2024.

[38] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in rdma servers. In *NSDI*, 2023.

[39] Tongrui Liu, Chenyang Hei, Fuliang Li, Chengxi Gao, Jiamin Cao, Tianshu Wang, Ennan Zhai, and Xingwei Wang. Resccl: Resource-efficient scheduling for collective communication. In *SIGCOMM*, 2025.

[40] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. Rethinking machine learning collective communication as a multi-commodity flow problem. In *SIGCOMM*, 2024.

[41] Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong, Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman. Quickupdate: a real-time personalization system for large-scale recommendation models. In *NSDI*, 2024.

[42] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *SOSP*, 2019.

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[44] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In *SIGCOMM*, 2024.

[45] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. 2019.

[46] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. Taccl: Guiding collective algorithm synthesis using communication sketches. In *NSDI*, 2023.

[47] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. *MLSys*, 2020.

[48] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *ASPLOS*, 2022.

[49] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In *SOSP*, 2024.

[50] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. Mccs: A service-based approach to collective communication for multi-tenant cloud. In *SIGCOMM*, 2024.

[51] Guanbin Xu, Zhihao Le, Yinhe Chen, Zhiqi Lin, Zewen Jin, Youshan Miao, and Cheng Li. Autoccl: Automated collective communication tuning for accelerating distributed and parallel dnn training. In *NSDI*, 2025.

[52] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[53] Tailing Yuan, Yuliang Liu, Xucheng Ye, Shenglong Zhang, Jianchao Tan, Bin Chen, Chengru Song, and Di Zhang. Accelerating the training of large language models using efficient activation rematerialization and optimal hybrid parallelism. In *ATC*, 2024.

[54] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Efficient direct-connect topologies for collective communications. In *NSDI*, 2025.

[55] Hao Zheng, Chengyuan Huang, Xiangyu Han, Jiaqi Zheng, Xiaoliang Wang, Chen Tian, Wanchun Dou, and Guihai Chen. μmon: Empowering microsecond-level network monitoring with wavelets. In *SIGCOMM*, 2024.

[56] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *OSDI*, 2024.

[57] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, et al. Megascale-infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In *SIGCOMM*, 2025.

[58] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *SIGCOMM*, 2015.

Appendix

A Experiment Settings

The training hyperparameters and ICCL settings of our evaluation (§4) are listed in Table 4.

Table 4: Training hyperparameters and ICCL settings.

Category	Setting	Value
Training	Global batch size	512
	Micro batch size	1
	DP size	8
	TP size	2
	PP size	4
	Attention head number	64
	Sequence length	2048
	layers_per_virtual_pipeline_stage	2
	Learning rate	1.5e-4
	Optimizer	Adam
	Precision	BF16
ICCL	IB_TIMEOUT	18
	IB_RETRY_CNT	7
	Window size	8
CC traffic generator	Message size	4G
	QP number	2
	Channel number	32