Connecting 100K+ GPUs: Building the Communication Stack for Large-Scale LLM Training

Hongyi Zeng, Min Si, Pavan Balaji, Yongzhou Chen*, Ching-Hsiang Chu*, Adithya Gangidi, Prashanth Kannan, Bingzhe Liu*, Saif Hasan, Dong He*, Deep Shah*, Ashmitha Jeevaraj Shetty, Gregory R Steinbrecher*, Srikanth Sundaresan*, Yulun Wang, Yexin Wu*, Mingran Yang, Kenny Yu*, Minlan Yu, Cen Zhao, Shengbao Zheng*, Wesley Bland, Denis Boyda, Suman Gumudavelli, Subodh Iyengar, Daniel Johnson, Cristian Lumezanu, Kai Luo*, Rui Miao, Zhe Qu, Venkatraghavan Ramesh, Jingliang Ren, Maxim Samoylov*, Jan Seidel, Qiye Tan, Feng Tian, Xinfeng Xie, Jingyi Yang*, Yimeng Zhao, Shuqiang Zhang*, Tiane Art Zhu

Meta

Abstract

The arrival of 100K+ GPU clusters marks a new frontier in AI infrastructure. Standard communication stack meets new challenges as physical topologies span multiple datacenter buildings, introducing high bandwidth-delay product links where latency increases by up to 30× compared to intra-rack traffic. Furthermore, the transition toward Mixture-of-Experts architectures generating bursty all-to-all patterns that create transient congestion hotspots. These constraints, combined with an operational environment where hardware failures shift from anomalies to frequent occurrences, renders traditionally lightweight operations like initialization and resource management challenging.

We present Meta's network architecture and software stack designed to support one of the world's largest RoCE fabrics, currently connecting over 100,000 GPUs across multiple datacenter buildings. To overcome scaling barriers, we introduce a scalable initialization strategy that reduces startup times by 11× via eager process group creation and $O(N)$ topology discovery, alongside a resource management system that cuts GPU memory usage by 2× through on-demand allocation. We further detail a custom transport layer utilizing Dynamic Queue Pair Load Balancing to saturate links, and a set of operation toolings. These innovations have been deployed in production, providing the foundational communication fabric for training state-of-the-art Large Language Models.

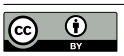
CCS Concepts

• **Computing methodologies** → **Machine learning**; • **Networks** → **Data center networks**.

Keywords

GPU Communication, Collective Communication, RDMA, LLM Training

*Work done while at Meta



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGCOMM '26, Denver, CO, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829152

ACM Reference Format:
Hongyi Zeng, Min Si, Pavan Balaji, Yongzhou Chen*, Ching-Hsiang Chu*, Adithya Gangidi, Prashanth Kannan, Bingzhe Liu*, Saif Hasan, Dong He*, Deep Shah*, Ashmitha Jeevaraj Shetty, Gregory R Steinbrecher*, Srikanth Sundaresan*, Yulun Wang, Yexin Wu*, Mingran Yang, Kenny Yu*, Minlan Yu, Cen Zhao, Shengbao Zheng*, Wesley Bland, Denis Boyda, Suman Gumudavelli, Subodh Iyengar, Daniel Johnson, Cristian Lumezanu, Kai Luo*, Rui Miao, Zhe Qu, Venkatraghavan Ramesh, Jingliang Ren, Maxim Samoylov*, Jan Seidel, Qiye Tan, Feng Tian, Xinfeng Xie, Jingyi Yang*, Yimeng Zhao, Shuqiang Zhang*, Tiane Art Zhu . 2026. Connecting 100K+ GPUs: Building the Communication Stack for Large-Scale LLM Training. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3789240.3829152

1 Introduction

The rapid evolution of Large Language Models (LLMs) has transitioned from scaling single-node performance to architecting massive-scale distributed systems. Training state-of-the-art models now requires orchestrating over 100K GPUs in a single job, a scale that pushes existing datacenter networking and communication stack to its limit. At this magnitude, the conventional approaches to collective communication encounter scaling challenges:

Infrastructure Constraints: The physical topology required to connect GPUs spans multiple datacenter buildings. This introduces high bandwidth-delay product (BDP) links and variant round-trip times (RTT) where latency increases by up to 30× compared to intra-rack traffic. Such disparities cause standard transport protocols to struggle with maintaining throughput consistency across the fabric.

New Workloads: The widespread adoption of Mixture-of-Experts (MoE) architectures fundamentally alters traffic characteristics. Unlike dense models with predictable communication, MoE models rely on dynamic token routing that generates highly bursty, irregular all-to-all patterns. This creates massive transient congestion hotspots that challenge traditional network load balancing.

Overheads of "Lightweight" Operations: Operations traditionally considered lightweight begin to impose prohibitive overheads. Collective initialization relying on global barriers or $O(N^2)$ topology discovery can degrade job restart times to several minutes or even hours—a delay that is unsustainable given the cluster's Mean Time Between Failures (MTBF). Furthermore, standard libraries can consume roughly 10GB of HBM (12.5% of an 80GB H100), directly competing with model parameters for limited memory.

Operational Reliability: As the cluster scales, hardware failures shift from rare anomalies to daily occurrences. Ensuring high training "goodput" demands a suite of fault localization, performance monitoring, and large-scale emulation tools capable of functioning under extreme concurrency.

To overcome these, we innovate at three categories:

At the *communication library layer*, we present CCLX, designed to address the key overhead of initialization and memory consumption. We introduce scalable initialization via three key mechanisms: eager initialization, asynchronous I/O bootstrapping, and lazy initialization resource allocation. Our solution can reduce the initialization time by 11×. To tackle the memory constraints of CCLX to leave more memory to model training, we introduce lazy memory management which implements lazy channel provisioning, on-demand buffer allocation, and a slab allocator for metadata, which collectively over-provisioning and reduces communication memory overhead by 2×.

At the *transport layer*, while zero-copy collectives minimize control plane latency, they often cause excessive switch buffer build-up due to bursty, unsegmented transfers—a problem exacerbated by MoE workloads. We introduce Dynamic Queue Pair Load Balancing (DQPLB), a transport mechanism that combines the low latency of zero-copy with the flow control of segmentation. By dynamically tuning outstanding data limits and queue pair counts to match the BDP of specific paths (e.g., cross-datacenter vs. intra-zone), DQPLB reduces switch buffer occupancy by an order of magnitude while maintaining near-peak throughput.

For the *operational tooling*, we have developed a comprehensive toolset comprising three key components in Section 5: an automated fault analyzer that distinguishes root-cause failures from cascading stalls using inter-collective dependency graphs, drastically reducing diagnosis time in environments where failures occur frequently; PerfProfiler, a lightweight instrumentation framework that correlates algorithmic and transport-level metrics to pinpoint bottlenecks; and a distributed CPU emulation environment that utilizes library interposition to validate control-plane scalability on CPU hardware, avoiding the cost of testing on GPUs.

Overall, our experience is that reaching 100K+ GPUs requires *co-designing* the collective library, transport mechanisms, and operational observability: at this scale, naive reuse of existing stacks fails under multi-building RTT/BDP regimes, MoE-induced burstiness, and frequent hardware faults. By rethinking initialization and HBM management in CCLX, introducing DQPLB for BDP-aware flow control, and building white-box tooling for fault localization and profiling, we make large-scale LLM training both performant and operationally robust.

2 Background

2.1 The 100K+ GPU Network

Meta's RoCE-based network has been evolving to meet the throughput and latency requirements of large-scale generative AI training. In its latest iteration, the network supports 100K+ GPUs; throughout this paper we refer to this deployment as *the Cluster*. Figure 1 summarizes the physical layout and the main hierarchy of the fabric. Table 2 illustrates the scale and latency of different components. We highlight three characteristics that shape our design:

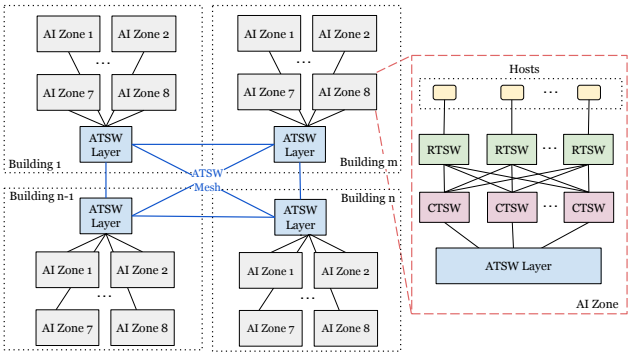


Figure 1: Multi-building network architecture.

Multi-building: The Cluster spans multiple datacenter (DC) buildings (Figure 1). Each building contains up to eight AI Zones [8]. Each AI Zone can support up to 288 racks with 16 GPUs per rack (~4K GPUs per zone), yielding up to ~32K GPUs per building. Scaling beyond a single building is therefore necessary to reach 100K+ GPUs, and it introduces longer links and greater variability in latency.

Multi-hop fabric: The Cluster uses a multi-stage Clos topology that connects all GPUs via a single high-performance RoCE fabric. Within this fabric, the Rack Training Switch (RTSW) connects GPUs within a rack, Cluster Training Switches (CTSW) connect racks within an AI Zone, and Aggregator Training Switches (ATSW) connect CTSWs across a datacenter building; the ATSW layer is a fully connected mesh. This hierarchy provides incremental scale-out—additional buildings can be added over time—and, compared to the previous generation [8], reduces the cross-AI-Zone over-subscription ratio from 1:7 to 1:2.8 to better support multi-dimensional parallelism at larger scale.

Latency: GPU-to-GPU latency grows quickly with hop count. GPUs within a rack have the lowest latency, while communication across racks within the same AI zone, across AI zones, and across datacenter buildings can experience 7×, 15×, and 30× higher latency, respectively. This increase comes from cumulative switching delays and longer cable runs. Note that the RTT increase is a direct result of larger cluster size, even within the same building, instead of a unique problem to our multi-building cluster. To achieve good collective performance in this hierarchy, the collective must inject enough data to fill the path bandwidth-delay product (BDP), but not so much that it creates persistent queueing and congestion.

2.2 Large-scale LLM Workloads

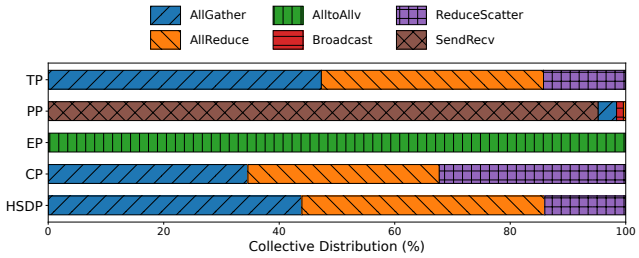
Recent developments in LLMs have significantly changed training workloads and introduced new networking requirements, particularly in the Cluster. We highlight three drivers and their impact on GPU communication: (1) *Architecture*: Recent LLMs increasingly adopt Mixture-of-Experts (MoE)[7], introducing irregular *All-to-All* collectives for token routing. (2) *Context length*: Context windows have expanded from thousands to millions of tokens, motivating Context Parallelism (CP) to support long-sequence training. (3) *Multimodality*: Modern LLMs are increasingly multimodal, introducing additional synchronization and data-handling complexity due to fused text and vision tokens.

Category	Unique problems and requirements	Features
Communication Library	Initialization among training hosts before starting training	Scalable initialization (Section 3.2)
	Keep high communication performance with minimal HBM, GPU compute, and network resources	Resource management (Section 3.3)
Transport	Managing high BDP and bursty all-to-all traffic across multi-building, high-latency links	DQPLB (Section 4)
Operational Tooling	Fast faulty hardware localization; fast user fault debugging in multi-dimensional parallelism programming	Fault localization (Section 5.1)
	Detailed performance insights to quickly pinpoint the bottleneck	Performance Observability (Section 5.2)
	Prohibitively expensive testing at large scale	Distributed CPU emulation (Section 5.3)

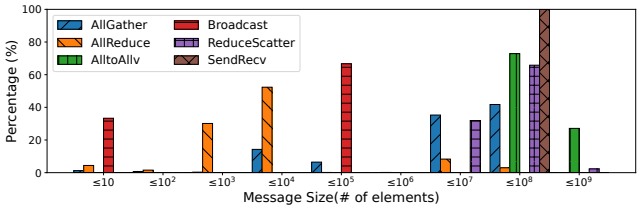
Table 1: Key problems and features described in this paper

	Host	Rack	AI Zone	Building	Cluster
GPUs	8	16	4K	32K	100K+
Latency	small	1×	7×	15×	30×

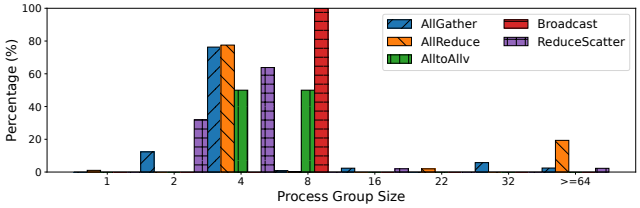
Table 2: Scale and latency hierarchy across the Cluster's multi-building fabric.



(a) Fraction of collectives in each parallelism scheme



(b) Distribution of message sizes across collective types



(c) Distribution of process group sizes across collective types

Figure 2: Detailed analysis of collective communication patterns in a large scale LLM pre-training

The immense scale of LLM training requires the use of various parallelization strategies to distribute the model and data across the Cluster. Parallelisms are fundamental techniques in large model training that break down a complex training job into smaller, concurrently executable tasks to manage memory, reduce computation time, and improve hardware utilization. To achieve this, we have employed a sophisticated combination of these methods, all of which rely on collective communication operations.

Hybrid Sharded Data Parallel (HSDP): Fully Sharded Data Parallel (FSDP)[37] is used together with Distributed Data Parallel (DDP) for sharding memory usage (e.g., optimizer states and gradients). FSDP uses *AllGather* for parameters and *ReduceScatter* for gradients, and DDP uses *AllReduce*.

Pipeline Parallelism (PP): Used to divide the model layers across GPU ranks. Point-to-point (P2P) communication is critical in PP. The implementation features zero-copy send/receive to remove interference between communication and computation, which in turn necessitates a Tensor Pool for P2P buffer management to improve buffer reusability.

Expert Parallelism (EP): Essential for training MoE models, EP shards the experts across a group of ranks to reduce memory pressure and mitigate the job from becoming limited by FSDP communication. Its primary communication primitive is the *All-to-All* collective, used to send tokens to the ranks where their assigned experts reside.

Context Parallelism (CP): A parallelism paradigm along the sequence length dimension, primarily supporting long sequence training by reducing memory consumption. CP Attention is implemented using an *AllGather* collective to exchange Key and Value tensors for the attention mechanism across the CP group ranks, while Query remains local.

Tensor Parallelism & Sequence Parallelism: Used to shard individual layer weights and distribute computation. Collectives include *AllReduce*, *AllGather* and *ReduceScatter*.

Figure 2 shows the breakdown of collectives used in a LLM training job, their message sizes and process group sizes.

2.3 Discussion

While our overall infrastructure encompasses over 100K GPUs, the actual operational workloads shard this massive footprint into multi-dimensional parallelization domains. To correctly design and optimize the communication stack, it is critical to distinguish between the scaling requirements of different operations.

The control plane operations, such as initialization, require true full-cluster participation. While the frequency of these operations is low, the scale makes them highly susceptible to serialization and network contention. As discussed in Section 3.2, such overhead becomes a major bottleneck at this scale impacting job restart times.

On the data plane front, hybrid parallelisms partition the cluster into much smaller, localized Process Groups. While the outermost

parallelization schemes like HSDP can span thousands of GPUs, they primarily handle lower-frequency weight and gradient synchronization. The most communication-intensive parallelisms operate at an even tighter scale. For example, EP relies on demanding *All-to-Allv* operations for token routing; however, as shown in Figure 2c, this bursty traffic is structurally confined to small process groups typically consisting of less than 8 GPUs.

Next, we describe how CCLX optimizes library-level operations (Section 3), how DQPLB manages transport-layer congestion in high-BDP environments (Section 4), and how our operational tooling ensures system reliability (Section 5).

3 GPU Communication Library

3.1 CCLX

The GPU communication library serves as the abstraction between training workloads and the NIC, interfacing through the verbs API layer. It translates collectives (e.g., *AllReduce*) into logical topology implementations (e.g., ring or tree) and further breaks those down into scheduling point-to-point data transactions between GPUs with verbs. These transactions require GPU-to-RDMA NIC support for optimal performance. The collective library schedules verbs calls over Queue Pairs (QP) created on NICs.

While NCCL is used in the vast majority of the NVIDIA based training systems, we have found several challenges when leveraging it directly in our environment. To overcome these, we have developed CCLX[1]. The key goal for CCLX is to provide a high-performance, scalable, and customizable communication framework for diverse application model needs and networking backends, while making it easier for communication developers to program. Figure 3 gives an overview of the CCLX stack, which provides users a rich selection of communication semantics in three execution modes: Host-initiated APIs, Host-initiated APIs with GPU-resident metadata, and Device-initiated APIs. Each mode supports collective, point-to-point and remote memory access (RMA) semantics. To support the communication semantics, we have developed a custom transport framework CTran [31] that follows a host-driven communication framework with the design principle to promote zero-copy and stream multiprocessor (SM) free communication.

When the model calls into the collective communication layer, CCLX dispatches the communication into either the baseline NCCL or the CTran code path. For custom communication operations (e.g., RMA, GPU-resident collectives) that do not have an implementation in baseline NCCL, they are directly dispatched to CTran. We allow users to explicitly choose the underlying baseline or CTran algorithms via environment variables.

3.2 Scalable Initialization

Communication library initialization is an important overhead for large-scale training due to its high fault rates and frequent job restarts. For example, to ensure 90% effective training time on average, we can only afford 144 minutes of downtime per day. As we run jobs at 100K GPU scale, the expected failure count is around 49.15 times per day (assuming 4 restarts every 1000 servers per day), which means we can only afford 175 seconds per restart. Although

[1]The CCLX version that runs on NVIDIA platform is also known as NCCLX, while the version on AMD platform is known as RCCLX.

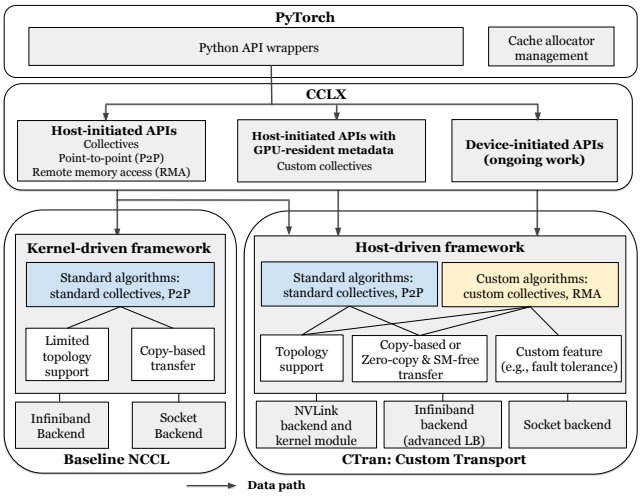


Figure 3: CCLX overview

existing technologies such as `torchft` [25] can reduce the blast radius of a single host failure, initialization overhead continues to worsen rapidly as training scales.

When a training job restarts, communication is initialized by creating PyTorch process groups (PGs), where NCCL bootstraps the communication group (`ncclCommInitRankConfig`) that includes multiple sub-operations: each rank initializes local resources (CUDA runtime, topology detection, proxy servers), establishes control channels for peer discovery, and exchanges local state information with all ranks.

Communication initialization time scales non-linearly due to serialized operations, network contention, and computational complexity. At 100K scale, initialization using NCCL requires more than 4 min, which goes beyond the tolerable restart time for reacting to failures. There are three key ideas to enable scalable initialization:
Eager initialization: NCCL initialization (*"Lazy Mode"*) requires each rank in a PG to independently create its communicator upon the first collective operation. This process involves the root rank broadcasting a unique ID (`ncclUniqueId`) via a centralized key-value store, followed by each rank calling `ncclCommInitRankConfig` to initialize its communicator. While this approach works for small-scale deployments, it suffers from two major drawbacks at scale: (1) excessive load on the key-value store due to repeated, independent creation for numerous PGs (e.g., 10+ PGs in a large model), and (2) inefficiency in creating sub-PGs in existing PGs because of the lack of state reuse and the need for redundant state exchange for every new group.

To resolve these scalability issues, CCLX introduces the *Eager mode*. It begins with each rank establishes a single global communicator that encompasses all global ranks (the default-PG). Subsequently, any sub-PG is derived or split from this pre-existing global communicator. By reusing the global communicator's established state, the need for further `ncclUniqueId` exchanges is eliminated. This fundamental change significantly reduces the cost of sub-PG creation and lowers the overall initialization overhead.

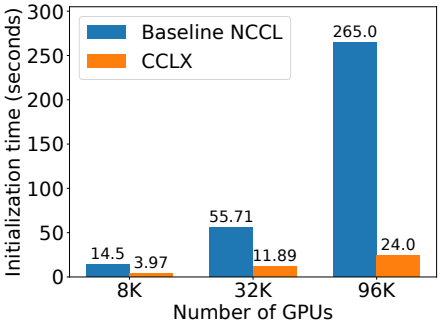


Figure 4: Comparison between NCCL 2.21.5 and CCLX on default process group initialization

Asynchronous I/O bootstrapping: NCCL bootstraps communicator by initializing a bootstrap server on rank 0. Each rank then communicates with this server to obtain the IP address and port of its subsequent rank, thereby forming a ring. To mitigate the risk of overloading the bootstrap server at larger scales (exceeding 128 ranks), each rank is required to stagger its connection request by waiting for a duration proportional to its rank ID, measured in milliseconds. At 100K scale, the final rank must wait for around 100 seconds to establish a connection with the bootstrap server. In practice, this will allow us to finish the bootstrap within the budget.
Lazy initialization resource allocation: The NCCL baseline allocates all resources during `ncclCommInitRank`, which can block the applications. CCLX defers resource allocation from initialization time to the first collective call, performing minimal necessary setup. When the first collective is invoked, algorithm specific resource allocation happens asynchronously in the background, overlapping with ongoing application work such as model loading, data fetching, and kernel computations. This approach hides allocation overhead within existing computation, further reducing time-to-first-collective in real-world training scenarios.

In addition to the architecture changes, we did a few optimizations that further improve the performance. For example, *AllGather* is used in bootstrap phase to exchange control data among participating ranks within a communicator. We optimized *AllGather* with bi-directional *AllGather* [12] (reducing steps from $N-1$ to $\frac{N}{2}$) and also evaluated tree-based algorithms ($O(logN)$ vs $O(N)$ scaling). We combined seven *AllGather* operations of gathering several data types to four *AllGather* operations to further reduce the overhead during initialization. We improved topology computation from $O(N^2)$ to $O(N)$ complexity [11], consuming 10s at 48K ranks and around 100s at 100K scale. Additional improvements include eliminating unique ID broadcasts and optimizing P2P communicator creation for pipeline parallelism.

Figure 4 shows the initialization performance results, comparing the baseline NCCL with CCLX. This represents an up to 11× improvement at 96K GPUs scale.

3.3 HBM Management

Resource utilization, including SMs, memory, and network resources, is an important factor in steady-state training. Overconsumption of these resources degrades training throughput and efficiency and, in extreme cases, can cause training job failures. In LLM training, this risk is amplified because each parallelism domain may instantiate its own communicators, causing cumulative resource consumption to grow with the number of communicators.

Among these constrained resources, GPU high-bandwidth memory (HBM) is often the dominant bottleneck for LLM pre-training. When more HBM is available to the training job, it improves both throughput and eventual model quality by enabling larger batch sizes and supporting exploration and tuning over a wider range of hyperparameter configurations. However, in practice, pre-training runs frequently fail with out-of-memory (OOM) errors because the usable HBM is significantly below the device's specification. For example, on H100 GPUs, only about 60 GB of the advertised 80 GB HBM may be available after accounting for system overheads and framework reservations.

Communication libraries are typically optimized for peak collective performance rather than resource efficiency. NCCL allocates substantial internal GPU buffers that are opaque to, and not shared with, the application in order to enable highly tuned pipelines across diverse hardware topologies and collective patterns. In today's multi-dimensional LLM training, which instantiates multiple parallelism groups and thus multiple NCCL communicators, these internal buffers are allocated separately for each communicator, leading to substantial HBM overhead. In LLM pre-training experiments, we observed NCCL consuming roughly 10 GB of HBM (≈ 12.5% of an 80 GB H100) across more than 10 parallelism groups, where each group is associated with a distinct NCCL communicator and maintains its own dedicated buffer allocations.

We have found three main design choices in NCCL that lead to inefficient resource utilization.
Eager communicator allocation: NCCL sustains high performance for copy-based communication by eagerly allocating substantial memory and peer connections at communicator initialization. Specifically, for each communicator, it allocates internal buffers and QPs per send/receive peer and per protocol (LL/LL128/SIMPLE). As the number of communicators grows, this strategy inflates memory usage—especially in workloads dominated by *All-to-All* communication, a pattern commonly observed in MoE training. NCCL further allocates algorithm-specific resources (e.g., ring) at runtime, even though training typically exercises only a subset of these paths, leaving the remainder underutilized. Additionally, communicators whose peers reside within an NVLink domain tend to receive even larger allocations, since NCCL can exploit additional buffering and concurrency to saturate NVLink's high bandwidth. However, because NCCL lacks knowledge of the actual communication pattern at initialization, this eager provisioning often results in substantial resource waste. Parallelism groups execute a fixed set of collectives over well-defined message-size regimes determined by the model configuration and scale. For example, a FSDP group typically performs *AllGather* and *ReduceScatter* using only the ring algorithm, making pre-allocation for tree-specific resources unnecessary.
Channel over-provisioning: NCCL employs a multi-channel design to increase concurrency in data movement and improve network utilization. To maximize performance, each communicator provisions the maximum number of channels at initialization, and each channel maintains its own dedicated resources. This strategy becomes inefficient when a communicator does not require many channels to achieve near-peak throughput—for example, when it

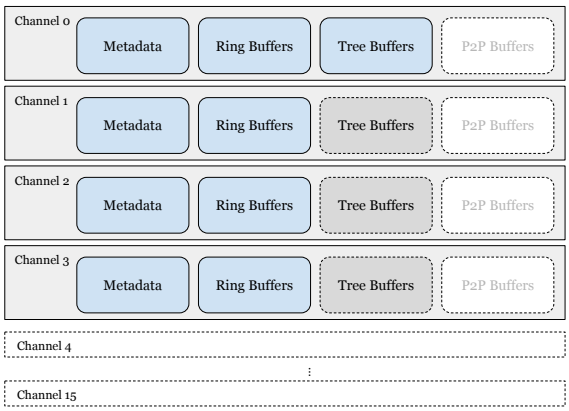


Figure 5: Lazy resource allocation and lazy channel provisioning example. CCLX saves memory by allocating only necessary channels and algorithm buffers.

Feature	Saved HBM per GPU
Lazy algorithm connect	1.44 GB
CTran lazy connect	0.79 GB
Slab allocator	0.39 GB
Lazy channel allocation	1.19 GB

Table 3: Example of memory saving by the proposed features for LLM pre-training on 64K GPUs. For each memory-saving feature, we conducted separate experiments using comparable configurations.

primarily executes small-message collectives—since a substantial fraction of per-channel resources remains rarely used.

Channel metadata alignment: NCCL's "channel" state includes metadata such as FIFO status and peer-connection information. NCCL places this metadata in HBM primarily to enable low-latency access by its kernels. Although the channel state is lightweight, it consumes around 2 MiB per channel due to alignment constraints imposed by CUDA low-level virtual memory management interfaces (*e.g.,* `cuMem` APIs). This alignment and per-channel footprint increase HBM pressure and exacerbate memory fragmentation and waste. This effect is amplified at large scale–the total metadata footprint scales with the number of ranks in a communicator, accumulating to over 1GB at 100k scale.

To address these limitations, we have improved resource efficiency by introducing three corresponding mechanisms.

Lazy communicator resource allocation: To mitigate the resource waste caused by NCCL's eager provisioning, we adopt a lazy strategy: it allocates algorithm-specific resources and CTran peer connection resources only when an algorithm is actually exercised at runtime. As the example illustrated in Figure 5, if a communicator only performs Tree algorithm for small *AllReduce* at runtime, we can only allocate Tree buffers on 1 channel, compared to NCCL would allocate resources in all channels and only use it partially. In an empirical study of LLM pre-training runs, enabling lazy connect alone reduced memory consumption by approximately 2.2GB as shown in Table 3.

Lazy channel provisioning: To reduce resource waste due to NCCL channel over-provisioning, we allocate only the minimum number of channels initially and expands the channel count at runtime as needed. The key observation is that some communicators execute only small collectives over the lifetime of a training job and therefore do not require multi-channel concurrency to approach peak bandwidth. This mechanism complements lazy resource allocation, which defers algorithm-specific resource allocation across all channels. Lazy channel provisioning further avoids pre-allocation resources for unnecessary channels altogether. As illustrated in Figure 5, if a collective requires only 4 channels for a ring-based algorithm, CCLX refrains from allocating resources for the remaining 12 channels (assuming a 16-channel default). As shown in Table 3, this feature yields an additional ~1.2 GB memory reduction.

Slab allocator for channel metadata: At large scale, per-channel metadata allocations becomes highly fragmented, leading to substantial memory waste. To mitigate this overhead, we implement a slab allocator that packs metadata from multiple communicators into contiguous GPU pages. By co-locating many channels' metadata within the same page, the allocator reduces both internal fragmentation and the number of pages that must be reserved. As shown in the example of Figure 6, a 2 MB GPU page can store metadata for roughly 3000 peers; without slab allocation, fragmentation forces the same metadata to be spread across multiple pages. All slab slots are released when a communicator is destroyed at the end of the training job, preventing resource leaks. As shown in Table 3, this optimization reduces memory usage by approximately 400 MB.

With these optimizations, we have reduced GPU memory usage by nearly 2× across more than 10 communicators at large scale, both in our evaluation (Table 3) and in production runs. These features reduce the number of RDMA QPs to below 2000 per NIC, thanks to the lazy QP allocation, which is within the capacity of NICs without performance degradation.

4 Transport Layer

4.1 Challenges

When the number of network hops increases, the GPU-to-GPU communication latency increases significantly. Specifically, GPUs within the same rack have the lowest latency, while communication across different racks within the same AI zone, across AI zones, and across separate datacenter buildings experience 7×, 15×, and 30× higher latency, respectively [31]. This increased latency is primarily due to cumulative switching delays and increased cable lengths.

As we move to MoE models, the prevalent communication patterns can have an on-off pattern particularly with small messages. This traffic is naturally bursty [9], which results in higher buffer utilization, as we describe later in this section, and can result in increased latency, and fairness issues when mixed RTT flows compete for bandwidth and buffer.

Managing latency is a tradeoff. The two-stage copy mechanism within NCCL requires clear-to-send control messages from the receiver to the sender on the critical path. To mitigate the high control-message latency of the two-stage copy solution, we implemented zero-copy collectives, enabling the NIC to perform RDMA directly. As a result, GPU-to-GPU exchanges require only a single control message from receiver to sender, significantly reducing

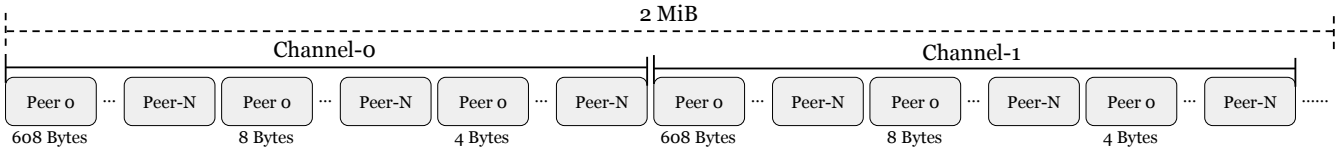
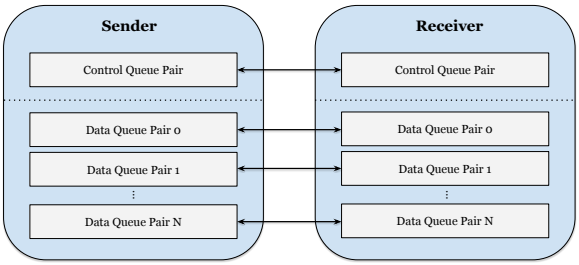


Figure 6: Slab allocator illustration: multiple metadata can fit into 2MB slabs to avoid waste

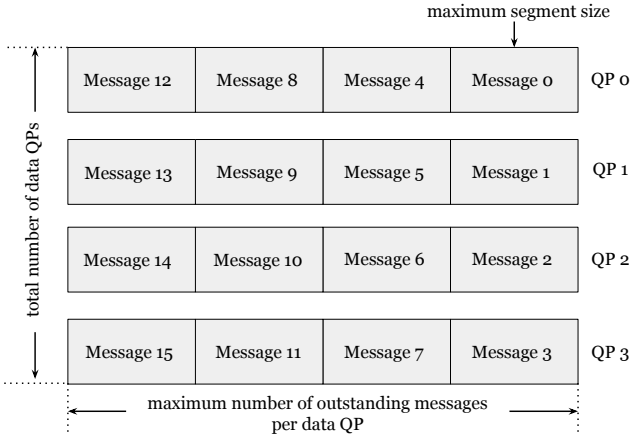
network latency. However, zero-copy results in the entire message being handed off to the NIC, which results in bursty sends.

The increased burstiness impacts our congestion control strategy. We do not employ traditional congestion control mechanisms such as DCQCN to limit switch buffer occupancy due to the difficulty of tuning it correctly in a diverse environment [8]. Instead, we rely on a combination of receiver-driven flow control in the collective library, and coarse NIC-based flow control to limit outstanding data, and deep-buffer spine switches to absorb transient bursts. This is not wholly effective: zero-copy communication reduces opportunities for receiver feedback, increasing the likelihood of posting larger messages at once. We therefore see excessive buffer build-up. Our evaluation confirms this analysis: using zero-copy communication alone resulted in suboptimal performance. In contrast, copy-based communication inherently segments data into smaller chunks—due to temporary buffer size constraints—thereby providing implicit flow control.

To combine the advantages of both approaches, we employ zero-copy communication while internally partitioning data into smaller message segments and rate-limiting the number of segments in flight. To this end, we develop Dynamic Queue Pair Load Balancing (DQPLB) to configure the amount of outstanding data on a per-connection and per-topology basis, allowing higher limits for cross-DC or cross-AI-Zone links' bandwidth-delay product (BDP). This fine-grained control enables more effective control of buffer build-up, and therefore better sharing and fairness. Together with network load balancing improvements described in [8], we reduce switch buffer build-up by an order of magnitude.



(a) DQPLB uses one Control QP and multiple Data QPs



(b) Example of load balancing across four QPs.

Figure 7: DQPLB design overview.

4.2 DQPLB Design

DQPLB controls the amount of data in the network using a small number of configurable parameters: the number of QPs per destination, the segment size, and the number of outstanding unacknowledged messages. The product of these three parameters should ideally match the BDP of the path between the sender and the receiver. Using smaller segment sizes also helps spread out the data in the network instead of bursting all at once. This design allows us to configure settings according to both the RTT of the path, and also the message sizes. We define four categories of connection types: within the same rack, cross-rack within the same zone, cross-zone within the same DC, and cross-DC. Based on the relative proximity of GPUs, we adjust the outstanding message limits for each connection. For connections with closer proximity, we use more conservative settings to accommodate the lower BDP. In contrast, for distant connections, we employ more aggressive configurations—such as higher number of data QPs and higher number of maximum number of outstanding messages—to better utilize the higher network BDP. Similarly, with smaller messages, we can be more aggressive in outstanding data to better utilize the network bandwidth.

As shown in Figure 7a, DQPLB uses one control QP and one or more data QPs. The control QP is responsible for exchanging memory addresses at the start of a collective operation, while the data QPs handle data transmission over the scale-out network. DQPLB achieves ordered message delivery across multiple data QPs by leveraging sequence numbering, immediate data encoding, and out-of-order message tracking. The algorithm distributes messages to data QPs in a round-robin fashion (Figure 7b). When a data QP's work queue reaches its configured limit, message transmission is paused until a completion queue element (CQE) is received on the corresponding completion queue (CQ). Upon receiving a CQE, the system resumes posting work queue elements (WQEs) for that QP. This dynamic allocation enables QPs with lower network contention to transmit data more efficiently, allowing them to handle a greater load.

To ensure ordered message delivery, we maintain two sequence counters: the sender tracks the next sequence number to transmit, while the receiver tracks the next expected sequence number.

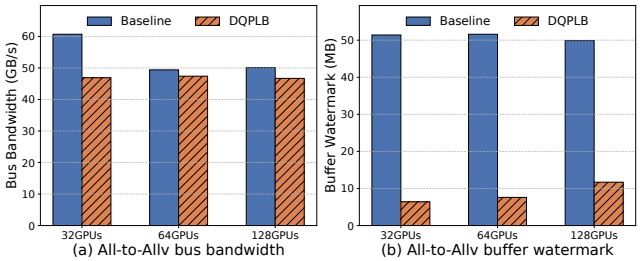


Figure 8: DQPLB performance with All-to-Allv

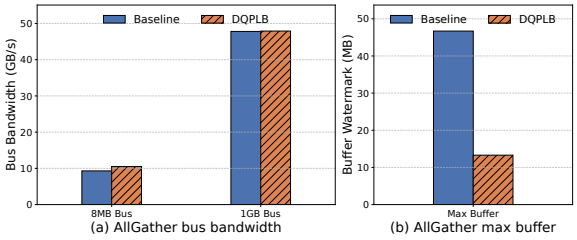


Figure 9: 256 GPU AllGather bus bandwidth and buffer before and after tuning

When transmitting messages in DQPLB mode, the sender utilizes the `IBV_WR_RDMA_WRITE_WITH_IMM` opcode for post send operations, instead of the standard `IBV_WR_RDMA_WRITE`. This enables the embedding of control information within the 32-bit immediate data field: bits 0–23 encode the sequential message number, bit 30 indicates fast path usage (explained in detail later), and bit 31 serves as a notification flag for the final write of a multi-packet message. When a message exceeds the maximum segment size, it is divided into multiple WQEs, with each partition assigned a consecutive sequence number; only the final fragment is marked with the notification bit in the immediate data field.

On the receiver side, the algorithm supports out-of-order delivery by examining the immediate data of incoming messages to extract sequence numbers and notification flags. Out-of-order packets with sequence numbers beyond the next expected value are temporarily stored in a hash map indexed by sequence number, with boolean values indicating whether each packet carries a notification flag. The algorithm then applies a sliding window protocol that continuously checks for the arrival of the next expected sequence number; when it is received, the algorithm processes it along with any subsequent consecutive packets stored in the hash map, increments the notification counter for those marked with the notification bit, and removes the processed entries from the hash map. This approach guarantees that notifications are triggered only after all preceding messages in the sequence have been received, thereby preserving strict ordering semantics even when packets arrive out of order due to distribution across multiple data QPs.

The algorithm also incorporates a fast path optimization for high-frequency operations, enabling messages to bypass the multi-QP distribution and be sent directly on a dedicated data QP. On the receiver side, these messages are processed by directly incrementing the receive-next sequence counter and updating notification counts, eliminating the need for out-of-order tracking. The fast path minimizes the CPU overhead of each RDMA operation, especially useful for algorithms with multiple small to medium RDMA operations such as the inference use case.

4.3 DQPLB performance

We first look at an *All-to-Allv* (*All-to-All* with varying message sizes) experiment with 32, 64, and 128 GPUs – this is one of the most network intensive traffic profiles. With our baseline, we use NCCL with NVLink support, whereas with DQPLB, we do not use NVLink. The reason being that at the time of running the experiments, NVLink support in our library was not mature. Furthermore, DQPLB is targeted towards network congestion issues, which are not relevant to host interconnects, which are significantly faster. Running DQPLB without NVLink therefore allows us to see how it performs in the network without potential bottlenecks being hidden or compensated by the faster host interconnects.. We ran our experiments with 400 Gbps NICs. We run all our experiments emulating perfect spray underneath. We configure DQPLB with 128KB in flight with 1 QP – this gives us fine-grained control over the data in flight. DQPLB is able to maintain close to maximum throughput. Note that NVLink improves the bus bandwidth beyond 50GBps because of the faster interconnect within hosts. Figure 8 shows that DQPLB achieves close to 50GBps at a fraction of peak buffer utilization – with almost a 90% reduction at 32 GPUs to about 75% reduction at 128 GPUs.

Next, we test with a 256 GPU *AllGather* that uses a ring implementation with rail-based setting (no NVLink). This is a representative experiment size as actual training workloads shard the cluster into much smaller, localized Process Groups (typically 32–256 GPUs, especially for MoE All-to-Allv operations). Although ring-based collectives do not have incast traffic patterns, the mix of RTTs will still result in momentary bursts at the receiver. First we run this with default settings, and then we tune the segment size and the number of QPs. The default is 512KB with 16 QPs. We tune this to 64KB segments, but only 2 QPs for in-rack scope, 4 QPs for cross-rack but within the same datacenter, and 64 QPs across datacenters. QP counts are chosen based on BDP of the path, which in this context is entirely a function of RTT. More BDP requires more proportionately more QPs. Figure 9 shows a 72% decrease in peak buffering, at no performance penalty at 1GB message sizes, and a 13% improvement in bus bandwidth for 8MB message sizes. First we note that the buffer wins are slightly smaller than with a more intensive *All-to-Allv*; this is because of the impact of cross-DC traffic. Larger BDP paths and the inability to arbitrarily reduce segment size (as mentioned previously) means that we need to burst somewhat, reducing the wins. However, smaller segments does offer some level of pacing benefit. Secondly, we see a performance improvement for the smaller, 8MB collective size; this is likely due to the latency of lower buffering.

5 Operational Tooling

5.1 Fault Localization

Due to the synchronized execution of LLM training, a single machine crash can stall or terminate the entire training job. In production environments, engineers often spend hours or even days

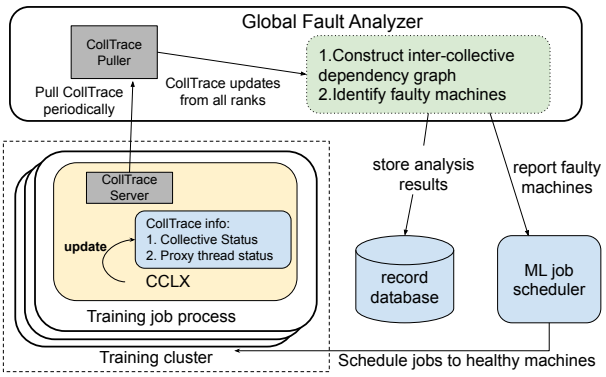


Figure 10: Overall architecture for the Fault Analyzer

identifying and isolating the faulty machine, during which additional failures may occur. To address this operational challenge, we developed *Fault Analyzer* (Figure 10), an automated system that analyzes training job failures and efficiently localizes faulty machines in real time.

In an LLM training job, many collective operations can execute simultaneously. The primary challenge is to identify the root-cause collective from this large set of operations when the job fails. Multi-dimensional model parallelism introduces complicated dependency between collectives, forming intricate execution graphs. When a single collective hangs, it can trigger cascading effects that impact all dependent collectives. It is impractical to exhaustively analyze all collective operations due to the massive volume of logs, the large number of participating machines, and the high algorithm complexity. Therefore, the design goal of Fault Analyzer is to identify the culprit collective operations in a fast and scalable manner.

During job execution, it incrementally constructs an inter-collective dependency graph in real time. The graph dynamically removes completed collectives and adds newly scheduled ones. The system continuously monitors unfinished collectives that have no outstanding dependencies and flags them as culprit candidates if their execution time exceeds a predefined timeout threshold. Fault Analyzer localizes the specific failing node associated with each culprit collective.

When multiple collectives hang simultaneously, only a subset of them are true root causes while the remaining ones are blocked while waiting for their dependencies to complete. To construct the inter-collective dependency graph, we rely on two assumptions: (1) When a job has been stalled for a long duration, all normal collectives should have finished. (2) If a collective has been scheduled but has not yet started, it must be waiting, either directly or indirectly, on other started but unfinished collectives on the same node. In other words, an unstarted collective depends on all such unfinished collectives. Although the second assumption is empirical, we find it highly effective in practice. An unstarted collective may be stalled because it shares the same CUDA stream as an unfinished collective, has CUDA event or data dependencies on it, or because the unfinished collective has exhausted critical hardware resources.

Based on these assumptions, we iterate over every node with stalled collectives, analyze the dependency relationships among

Algorithm 1 Constructing the Dependency DAG Among Hanging Collectives

Require: $\mathcal{M}_{\text{timeout}}$: machines that have timeout collectives
Ensure: G_{dep}: dependency DAG between collectives

procedure BUILDDAG($\mathcal{M}_{\text{timeout}}$)
  $G_{\text{dep}} \leftarrow \emptyset$
  for all node $M \in \mathcal{M}_{\text{timeout}}$ **do**
    $C_M \leftarrow$ getTimeoutCollectives(M)
    $C_A \leftarrow \emptyset$ ▷ Started yet incomplete collectives
    for all collective $C \in C_M$ **do**
      if kernelStart(C) **and** ¬KernelEnd(C) **then**
        $C_A \leftarrow C_A \cup \{C\}$
      end if
    end for
    for all collective $C \in C_M$ **do**
      if ¬KernelStarted(C) **then**
        $G_{\text{dep}}[C] \leftarrow C_A$ ▷ C depends on C_A
      end if
    end for
  end for
  return G_{dep}
end procedure

collectives on the node, and construct a global dependency directed acyclic graph (DAG). Collectives without outgoing dependencies in the DAG are identified as culprit collectives. Algorithm 1 presents the detailed procedure.

We further illustrate the procedure with a simple example. On Node 1, Collective A has started but remains unfinished, while Collective B is scheduled but unstarted. On Node 2, Collective B is unfinished and Collective C is unstarted. Following the algorithm, we add a dependency from B to A on Node 1 and from C to B on Node 2. Since A has no outgoing dependencies, it is identified as the culprit collective.

To track whether a collective is scheduled, started and completed, we instrumented CCLX with tracing instructions at per-collective granularity, called *CollTrace*. CollTrace forms the foundation of our dependency analysis and provides fine-grained visibility into collective execution states.

For each culprit collective, we identify the responsible rank and mark the node hosting that rank as faulty using the following procedure. First, we check per-rank arguments (*e.g.,* operation type, element count, etc) in the collective. If a rank provides inconsistent or invalid arguments relative to its peers, we flag it as faulty. Next, we check whether all ranks have launched the kernel and entered the execution phase. Any node that fails to start the kernel is identified as faulty. Subsequently, we detect network-level failures by examining errors thrown between communicating rank pairs. If multiple errors are observed, we select the rank involved in the largest number of failed pairwise communications as the most likely faulty ranks.

If the above checks reveal no issues and no errors are thrown, we construct a wait-for dependency graph based on proxy operation states. In a collective operation, before the sender write data chunks into the receiver's HBM, the receiver uses the proxy thread to send control messages describing the destination buffer metadata to the sender. The sender then initiates data transmission from its proxy

thread. We model two types of dependencies: (1) *control-message wait dependencies*: The sender is blocked waiting for the receiver's control message containing destination buffer metadata. (2) *data wait dependencies*: The receiver is blocked waiting for the sender's data transfer. The Fault Analyzer builds the wait-for dependency graph based on these two dependencies by checking the proxy operation states. Nodes without outgoing dependency are considered candidate faulty nodes. We present two brief case studies to go through the fault localization procedure.

Model code bug: Bugs introduced during model development may eventually manifest as NCCL timeouts in PyTorch, which are often difficult and time-consuming to debug. In a 4K-GPU training job, approximately 500 collectives experienced timeouts. Fault Analyzer automatically constructed the inter-collective dependency graph and identified the root-cause collective as the final collective in a tensor-parallel (TP) process group. By checking the kernel execution status across hosts, we observed that one rank had not launched the corresponding collective kernel, while all other ranks had started execution. This rank was therefore flagged as the likely culprit. With this insight, the developer traced the issue to a software bug that caused the specific rank to hang before scheduling the final collective.

Hardware failure: In another training job spanning 8K GPUs, the job experienced repeated restarts accompanied by hundreds of collectives timeouts. Fault Analyzer first aggregated the collective and network operation traces, and built the inter-collective dependency graph. After analysis, it found the last *AllReduce* in a data-parallel (DP) process group as the culprit collective. Network-level traces did not report explicit communication errors. We therefore constructed a wait-for dependency graph based on proxy operation states. The analysis revealed that one host had stopped transmitting data to its peer, and further investigation confirmed that a malfunctioning NIC on that host was the root cause.

5.2 Performance Observability

Unexpected network latency or congestion can severely compromise overall job performance. Conventional profiling tools often lack the comprehensive context spanning the workload and communication layers needed to isolate network issues and to assess the impact based on whether the affected communication are exposed.

We developed a lightweight *PerfProfiler* to provide real-time, fine-grained, and context-aware performance monitoring. PerfProfiler is designed for minimal CPU overhead and flexible sampling, ensuring that the profiling overhead is negligible and allows it to run continuously throughout the entire training process. It has two primary modules:

AlgoProfiler: It gathers essential timestamps at the collective algorithm layer. It then dissects and measures the latency of various collective stages, including buffer registration, control message synchronization, and data transfer. These measured durations help pinpoint the bottleneck stage in a slow collective; for instance, determining if the delay is due to memory registration (a specific overhead for zero-copy collectives), network congestion (extended data transfer), or a slow rank (prolonged message synchronization).

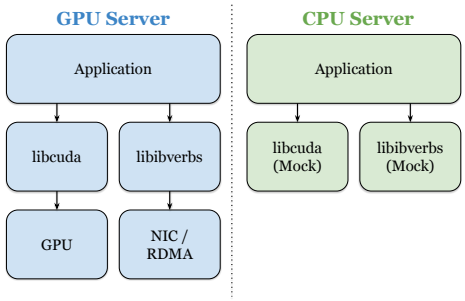


Figure 11: Comparison between a GPU communication stack and its CPU emulation.

SlowRankDetector: Monitors per-rank network efficiency at the CTran transport layer and identifies slow ranks. It captures the latency of individual RDMA packets by recording timestamps for the Work Queue Entry (WQE) issuing and completion events. Subsequently, it calculates the bus bandwidth for each rank over a rolling window and reports any slow peer connection below a threshold.

Case Study: The model engineer noticed a significant training performance drop during certain iterations. The symptom was slow pipeline parallelism point-to-point communication in some per-rank traces. The initial step was to ascertain whether the cause was slow communication/network or a different type of non-communication delay. By examining the synchronization duration sampled from the affected iteration, we observed that some ranks reported a 100x longer delay, while the data transfer duration was normal. Focusing on the peers with this extended synchronization led us to the root cause: the ranks were blocked by a slow I/O operation, classifying them as a "slow rank".

5.3 Distributed CPU emulation

Testing changes to the collective communication stack at full 100K+ scale is prohibitively expensive. Control-plane regressions can idle tens of thousands of GPUs, yet many of the most costly failures manifest only under extreme concurrency and topology scale. To shift these risks earlier in the development cycle, we built a distributed CPU emulation framework that executes *unmodified* communication and PyTorch Distributed binaries on an existing CPU cluster. The framework provides early, high-signal validation of control-plane correctness and scale behavior.

GPU communication libraries cannot run on CPUs out of the box because it depends on GPU- and RDMA-specific user-space stacks even during initialization. Our approach in Figure 11 preserves the production binaries and swaps only the dependency surfaces. We provide drop-in replacements for `libcudart`, `libcuda`, `libnvidia-ml`, and `libibverbs`; redirect resolution with `LD_LIBRARY_PATH`; and share common emulator state via a small library loaded with `LD_PRELOAD`. This design keeps the application and PyTorch, and the communication libraries *unmodified*.

We target *control-plane fidelity*, not dataplane performance. The emulator reproduces API-level semantics exercised during CCLX initialization and bring-up: device enumeration; stable GPU-memory identities and handle lifetimes; topology and P2P capability queries; verbs object creation and state transitions; and

correct error propagation and teardown behavior. CUDA allocations are backed by pinned host memory. We maintain stable handle/pointer translation so the communication library observes consistent identities and lifetimes across allocation, IPC-handle use, and teardown. We have emulated the initialization-relevant subset of `libibverbs.so` sufficient for CCLX bring-up: device/context discovery, protection domains, memory-region metadata, QP objects, and event handling. We exercise CCLX connection-graph construction, resource scaling behavior, and error handling under realistic rank counts. We configure each CPU host to present eight virtual GPUs and run one rank per virtual device, enabling 96K-rank experiments on 12K CPU servers.

This framework enabled early, high-signal validation and performance modeling of our scalable initialization techniques (Section 3.2), allowing us to iterate quickly. It also reliably exposed scale-triggered control-plane issues that are effectively invisible at smaller scales. For example, beyond 64K ranks we saw bootstrap fan-in hit OS-level limits (*e.g.,* ephemeral port pressure and accept-queue overflow under connection storms), and used the framework to validate burst-tolerant retry/backoff behavior. We also identified initialization paths whose state unintentionally grew with the number of peers (*e.g.,* eager QP and metadata provisioning), leading to prohibitive memory growth during global communicator creation and directly motivating on-demand provisioning. Finally, we observed quadratic amplification in initialization-time telemetry when each rank emitted per-peer metadata, creating $O(N^2)$ bursts that overwhelmed downstream systems and motivating aggregation and rate-limiting during bring-up.

6 Deployment

6.1 Implementation

The Cluster is in a single DC region, comprising a total of 5 DC buildings. Built in phases, the Cluster has about 129,000 H100 GPUs [32] at the completion. Grand Teton [17] is the host system of these H100s. Figure 12 depicts each Grand Teton's internals. The node is divided into 3 trays - CPU Tray housing 2 CPUs and frontend NICs, Switch Tray housing 4 PCIe Gen5 switches, NVMe storage, as well as 8 RDMA NICs, and GPU Tray housing 8 GPUs. GPUs are fully-connected using NVSwitch[20]. There is a 1:1 mapping between GPUs and NICs. CCLX and its associated tooling have been supporting all the state-of-the-art LLM training running in the Cluster since late 2024.

6.2 Limitations

Initialization: While CCLX significantly mitigates initialization overhead at our current scale, this global state exchange strategy faces fundamental scaling limits as clusters transition toward next-generation regional deployments. Future efficiency gains will likely require moving toward hierarchical initialization or decentralized, peer-to-peer discovery mechanisms where ranks only maintain localized connectivity metadata. Due to development complexity and production deployment timelines, we did not explore these decentralized architectures in the current fabric. Furthermore, while fault-tolerance frameworks like `torchft` [25] successfully reduce the blast radius of individual host failures , integrating decentralized

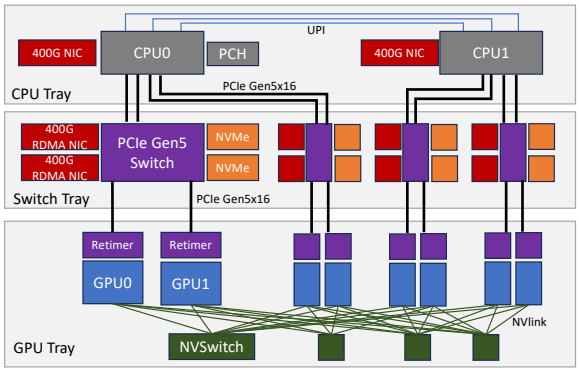


Figure 12: Grand Teton platform

initialization with robust replica-based recovery models remains an open necessity for scaling beyond the 100K-GPU threshold.

Emulation: We rely on our distributed CPU emulation framework to expose scale-triggered control-plane regressions early in the development cycle. However, as designed, this framework prioritizes control-plane fidelity rather than data-plane performance. It successfully validates initialization mechanics, connection-graph construction, and buffer allocation behaviors under realistic rank counts, but it does not model raw data-plane execution. Achieving faithful, end-to-end data-plane emulation at full production scale (100K+) remains an open question for the community.

Fault Localization: Emerging system optimization paradigms are beginning to introduce visibility challenges for our analytical fault localization and profiling infrastructure. In pursuit of peak hardware utilization, the industry is rapidly transitioning toward highly optimized, fused GPU kernels[22] that interweave computation directly with communication operations. This architectural blending inherently obscures the precise, deterministic boundaries of individual collective operations. Consequently, such fusion poses a significant challenge to tracing tools like CollTrace, which structurally rely on distinct per-collective execution states to construct accurate inter-collective dependency graphs. As static analytical models face diminishing visibility due to opaque, fused runtime traces, our reliance on explicit wait-for state tracking must adapt. Future diagnostic architectures will likely need to explore how AI-assisted debugging can complement or replace static graph-based models to aggregated performance data.

Production Ablations: Most of our CCLX improvements are validated solely in communication-focused benchmarks. Conducting live, full-cluster network ablations in a production environment connecting over 100K GPUs is rendered impossible by prohibitive resource, scheduling, and financial constraints. Typically, a variety of infrastructure features - including architectural changes, compute kernels, communication enhancement, and configuration changes - are bundled into a single package deployed on a pre-defined date. Each of these features is tested at a smaller scale and gated by a configuration flag. Once pre-training begins, the job usually runs for several weeks, which may pause and resume from checkpoints many times. A feature can be disabled quickly if it actively causes issues. For features that run smoothly, detailed telemetry is often turned off to save overhead. Although it is challenging to validate

their isolated effectiveness in production jobs, we can infer their impact indirectly. For example, the fast initialization feature was kept active to ensure training could restart within its time budget after any interruption, and all collectives were successfully hidden behind computation when DQPLB was turned on.

7 Related Works

There is an extensive body of recent research on collective communication libraries [15, 16, 28, 29, 38] as well as higher-layer communication frameworks [3, 21, 27, 30]. Our work on CCLX *does not* claim inherent superiority over these specialized approaches; rather, it focuses on the specific operational and architectural learnings required to sustain training at the 100K scale. Our contributions are complementary to existing GPU communication research, providing a bridge between collective optimizations and the practical requirements of massive-scale production fabrics.

Initialization: Initialization has historically received limited attention in the GPU-based distributed training literature, primarily because its overhead only becomes prohibitive at the large scale. Conversely, the HPC community has long addressed similar scaling bottlenecks in CPU clusters through frameworks such as PMIx [2]. As GPU clusters reach unprecedented scales, this issue has resurfaced as a primary concern, evidenced by MegaScale [13], which operates at 10K scale, and NVIDIA's introduction of the scalable initialization API [23]. Our experience suggests that this domain requires continuous optimization; as clusters scale, the compounding effects of increased interruption frequency and initialization time can significantly degrade training efficiency.

Fault localization: There have been many works on fault localization for AI infrastructure. For example, Minder [4] and PerfTracker [10] focus on identifying stragglers and gray failures by analyzing machine health and applying differential observability by leveraging temporal and spatial similarity across workers. To debug the complex interactions between workers, Mycroft [5] and Aegis [6] intercept and trace dependencies within collective communication primitives to pinpoint faults. BiAn [33] leverages LLMs to automate the reasoning over monitoring logs for rapid root cause analysis. From our experiences, we find simple and scalable algorithms that construct inter-collective dependency graphs are more effective for fault localization at this scale.

Transport: There are two lines of work in AI transport: load-balancing, and congestion management. They are complementary approaches to solving the overall problem of congestion, buffering, and latency in the network. The default solution for load-balancing of ECMP assumes large flow entropy. However, AI training traffic tends not to have sufficient entropy [8]. Dynamic Load Balancing offers more responsive load balancing at a flow or even packet level [14, 26, 36]. However, load balancing does not by itself solve the problem of overall congestion in the network fabric; it merely distributes it evenly. We need better congestion management schemes. AI traffic can also be bursty which also limits the above solutions which for all practical purposes, rely on network feedback, which is suboptimal [1]. Our approach aims for maximum flexibility, which points us to the Communications layer software stack. Knowledge about the traffic patterns, and the ability to do effective flow control drives DQPLB design, which seamlessly complements

Emulation: Simulators (e.g., ASTRA-Sim [35]) and training-stack substitutions (e.g., SimAI [34]) are valuable for what-if analysis, but they typically do not execute the *unmodified* production collective stack under a real OS/network environment. Our CPU-based emulation provides a low-cost, high-signal way to surface these scale-dominant control-plane failures early, before full-scale GPU rollouts. Our experience shows that investing in unmodified-stack emulation and repeatable fault injection is a necessary way for control-plane correctness and operational hardening, complementing performance-oriented simulation tools.

8 Conclusion

The physical infrastructure of AI clusters is transitioning toward multi-gigawatt scale deployments, such as Colossus 2 [18], Rainier [19], Stargate [24], and Hyperion [32]. These next-generation facilities will house more than one million GPUs, shifting from multi-building to regional-scale interconnects. As we look toward these massive deployments, the communication stack must evolve to address the unique engineering hurdles that accompany this unprecedented scale.

At 100K+ scale, physical topology constraints and the bursty nature of workloads renders existing techniques insufficient. By co-designing a communication library with the transport layer and a suite of operational tools, we built a robust foundation that sustains high-goodput training of frontier LLMs despite scale, latency variance, and frequent faults. This experience provides a blueprint for the industry as LLM training moves toward larger deployments.

This work does not raise any ethical issues.

Acknowledgements

Many current and former people at Meta have contributed to productionizing collective communication libraries for AI training and inference over the years.

This work is a close collaboration with our partners in Meta's AI teams. We would like to express our gratitude to the following individuals for their invaluable help and contributions: Kunal Bhalla, Sai Jayesh Bondu, Will Constable, Zachary DeVito, Mikel Jimenez Fernandez, Wenyin Fu, Naman Goyal, Andrew Gu, Yuchen Hao, Eliot Hedeman, Ajay Hotchandani, Jianyu Huang, Jenya Lee, Shikai Li, Keyu Man, Mustafa Ozdal, Jason Park, Jongsoo Park, Shangfu Peng, Sarunya Pumma, Tristan Rice, Ahmed Sharif, Omkar Salpekar, Zoey Sun, Michael Suo, Rohan Varma, Shawn Xu, Vedanuj Goswami, Weiwei Chu, Wei Sun, Jie Wang, Xiaodong Wang, Yifu Wang, Ke Wen, Lukasz Wesolowski, Ji Yang, Jiecao Yu, and Haoci Zhang.

We are also grateful for the project support and guidance from Balaji Balasubramanian, Omar Baldonado, Harish Kumar Chandrappa, Shashi Gandham, Guna Lakshminarayanan, Teng Li, Maxim Naumov, Mathew Oldham, Chunqiang Tang, Srinivas Vaidyanathan, and Chuanhao Zhuge.

We extend our sincere thanks to our partners in the NVIDIA NCCL team for their insightful discussions and technical support.

We are also indebted to our shepherd Dina Papagiannaki, and the anonymous SIGCOMM reviewers for their comments and suggestions on earlier drafts.

References

[1] Christopher Canel, Balasubramanian Madhavan, Srikanth Sundaresan, Neil Spring, Prashanth Kannan, Ying Zhang, Kevin Lin, and Srinivasan Seshan. 2024. Understanding Incast Bursts in Modern Datacenters. In *Proceedings of the 2024 ACM on Internet Measurement Conference* (Madrid, Spain) *(IMC '24)*. Association for Computing Machinery, New York, NY, USA, 674–680. doi:10.1145/3646547.3689028

[2] Ralph H. Castain, David Solt, Joshua Hursey, and Aurelien Bouteiller. 2018. PMIx: Process management for exascale environments. *Parallel Comput.* 79 (Nov 2018), 9–29. doi:10.1016/j.parco.2018.08.002

[3] DeepSeek. 2025. DeepEP Communication Library. https://github.com/deepseek-ai/DeepEP.

[4] Yangtao Deng, Xiang Shi, Zhuo Jiang, Xingjian Zhang, Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, Gaohong Liu, Fuliang Li, Shuguang Wang, Haibin Lin, Jianxi Ye, and Minlan Yu. 2025. Minder: faulty machine detection for large-scale distributed model training. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation* (Philadelphia, PA, USA) *(NSDI '25)*. USENIX Association, USA, Article 27, 17 pages.

[5] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaohong Liu, Yang Bai, Shuguang Wang, Wencong Xiao, Jianxi Ye, Minlan Yu, and Hong Xu. 2025. Mycroft: Tracing Dependencies in Collective Communication Towards Reliable LLM Training. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 254–269. doi:10.1145/3731569.3764848

[6] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, Zhihui Ren, Zhicheng Wang, Bin Luo, Peng Zhang, Yang Liu, Yanqing Chen, Yu Guan, Weicheng Wang, Chaojie Yang, Yang Zhang, Man Yuan, Hanyu Zhao, Yong Li, Zihan Zhao, Shan Li, Xianlong Zeng, Zhiping Yao, Binzhang Fu, Ennan Zhai, Wei Lin, Chao Wang, and Dennis Cai. 2025. Evolution of Aegis: fault diagnosis for AI model training service in production. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation* (Philadelphia, PA, USA) *(NSDI '25)*. USENIX Association, USA, Article 46, 17 pages.

[7] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *J. Mach. Learn. Res.* 23 (2022), 120:1–120:39. https://jmlr.org/papers/v23/21-0998.html

[8] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 57–70. doi:10.1145/3651890.3672233

[9] Soudeh Ghorbani, Yimeng Zhao, Srikanth Sundaresan, Ying Zhang, Yijing Zeng, Abhigyan Sharma, Prashanth Kannan, and Cristian Lumezanu. 2025. Congestion Patterns in a Large-scale RDMA Datacenter. In *Proceedings of the 2025 ACM Internet Measurement Conference* (USA) *(IMC '25)*. Association for Computing Machinery, New York, NY, USA, 944–951. doi:10.1145/3730567.3764494

[10] Yu Guan, Zhiyu Yin, Haoyu Chen, Sheng Cheng, Chaojie Yang, Kun Qian, Tianyin Xu, Yang Zhang, Hanyu Zhao, Yong Li, Wei Lin, Dennis Cai, and Ennan Zhai. 2025. PerfTracker: Online Performance Troubleshooting for Large-scale Model Training in Production. arXiv:2506.08528 [cs.DC] https://arxiv.org/abs/2506.08528

[11] Saif Hasan. 2025. NCCL Fast Init - CPU Optimizations for NCCL Initialization Large Scale. https://github.com/NVIDIA/nccl/pull/1789.

[12] Saif Hasan. 2025. NCCL Fast Init - Improve Bootstrap AllGather by 2x at large scale. https://github.com/NVIDIA/nccl/pull/1791.

[13] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: scaling large language model training to more than 10,000 GPUs. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 41, 16 pages.

[14] Mohan Kalkunte, Niranjan Vaidya, and Pete Del Vecchio. 2023. Cognitive routing in the Tomahawk 5 data center switch. https://www.broadcom.com/blog/cognitive-routing-in-the-tomahawk-5-data-center-switch.

[15] Tongrui Liu, Chenyang Hei, Fuliang Li, Chengxi Gao, Jiamin Cao, Tianshu Wang, Ennan Zhai, and Xingwei Wang. 2025. ResCCL: Resource-Efficient Scheduling for Collective Communication. In *Proceedings of the ACM SIGCOMM 2025 Conference* (São Francisco Convent, Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 55–70. doi:10.1145/3718958.3750514

[16] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. 2024. Rethinking Machine Learning Collective Communication as a Multi-Commodity Flow Problem. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 16–37. doi:10.1145/3651890.3672249

[17] Meta. 2024. Grand Teton. https://www.opencompute.org/documents/grand-teton-intel-based-cpu-tray-specification-v1-0-pdf.

[18] Elon Musk. 2026. Colossus 2 supercomputer. https://x.com/elonmusk/status/2012500968571637891.

[19] Amazon News. 2025. AWS activates Project Rainier: One of the world's largest AI compute clusters comes online. https://www.aboutamazon.com/news/aws/aws-project-rainier-ai-trainium-chips-compute-cluster/.

[20] NVIDIA. 2024. NVLink. https://www.nvidia.com/en-us/data-center/nvlink/.

[21] Nvidia. 2025. NVIDIA Inference Xfer Library (NIXL). hhttps://github.com/ai-dynamo/nixl.

[22] NVIDIA Corporation. 2025. Fusing Communication and Compute with New Device API and Copy Engine Collectives in NVIDIA NCCL 2.28. https://developer.nvidia.com/blog/fusing-communication-and-compute-with-new-device-api-and-copy-engine-collectives-in-nvidia-nccl-2-28/.

[23] NVIDIA Corporation. 2025. New Scaling Algorithm and Initialization with NVIDIA Collective Communications Library 2.23. https://developer.nvidia.com/blog/new-scaling-algorithm-and-initialization-with-nvidia-collective-communications-library-2-23/.

[24] OpenAI. 2025. OpenAI, Oracle, and SoftBank expand Stargate with five new AI data center sites. https://openai.com/index/five-new-stargate-sites/.

[25] PyTorch. 2025. Torchft Documentation. https://github.com/meta-pytorch/torchft.

[26] Chenchen Qi, Wenfei Wu, Yongcan Wang, Keqiang He, Yu-Hsiang (Sean) Kao, Zongying He, Chen-Yu Yen, Zhuo Jiang, Feng Luo, Surendra Anubolu, Yanjin Gao, Bingfeng Lin, Wenda Ni, Yiming Yang, Donglin Wei, Boyang Zhou, Jian Wang, and Shan Ding. 2025. SGLB: Scalable and Robust Global Load Balancing in Commodity AI Clusters. In *Proceedings of the ACM SIGCOMM 2025 Conference* (São Francisco Convent, Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 626–644. doi:10.1145/3718958.3750527

[27] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. arXiv:2407.00079 [cs.DC] https://arxiv.org/abs/2407.00079

[28] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2023. MSCCL: Microsoft Collective Communication Library. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*. USENIX Association, Boston, MA, 833–846.

[29] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2023. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*. USENIX Association, Boston, MA, 817–831.

[30] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[31] Min Si, Pavan Balaji, Yongzhou Chen, Ching-Hsiang Chu, Adi Gangidi, Saif Hasan, Subodh Iyengar, Dan Johnson, Bingzhe Liu, Regina Ren, Ashmitha Jeevaraj Shetty, Greg Steinbrecher, Yulun Wang, Bruce Wu, Xinfeng Xie, Jingyi Yang, Mingran Yang, Kenny Yu, Minlan Yu, Cen Zhao, Wes Bland, Denis Boyda, Suman Gumudavelli, Prashanth Kannan, Cristian Lumezanu, Rui Miao, Zhe Qu, Venkat Ramesh, Maxim Samoylov, Jan Seidel, Srikanth Sundaresan, Feng Tian, Qiye Tan, Shuqiang Zhang, Yimeng Zhao, Shengbao Zheng, Art Zhu, and Hongyi Zeng. 2025. Collective Communication for 100k+ GPUs. arXiv:2510.20171 [cs.DC] https://arxiv.org/abs/2510.20171

[32] Yee Jiun Song and Kaushik Veeraraghavan. 2025. Meta's Infrastructure Evolution and the Advent of AI. https://engineering.fb.com/2025/09/29/data-infrastructure/metas-infrastructure-evolution-and-the-advent-of-ai/.

[33] Chenxu Wang, Xumiao Zhang, Runwei Lu, Xianshang Lin, Xuan Zeng, Xinlei Zhang, Zhe An, Gongwei Wu, Jiaqi Gao, Chen Tian, Guihai Chen, Guyue Liu, Yuhong Liao, Tao Lin, Dennis Cai, and Ennan Zhai. 2025. Towards LLM-Based Failure Localization in Production-Scale Networks. In *Proceedings of the ACM SIGCOMM 2025 Conference* (São Francisco Convent, Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 496–511. doi:10.1145/3718958.3750505

[34] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, Yang Liu, Pengcheng Zhang, Kun Qian, Kunling He, Jiaqi Gao, Ennan Zhai, Dennis Cai, and Binzhang Fu. 2025. SimAI: Unifying Architecture Design and Performance Tuning for Large-Scale Large

Language Model Training with Scalability and Precision. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 541–558. https://www.usenix.org/conference/nsdi25/presentation/wang-xizheng-simai

[35] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. ASTRA-sim2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-model Training at Scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, New York, NY, USA, 283–294. doi:10.1109/ISPASS57527.2023.00035

[36] Xiguang Wu. 2022. Reducing job completion time in AI/ML clusters. https://www.broadcom.com/blog/reducing-job-completion-time-in-ai-ml-clusters.

[37] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (aug 2023), 3848–3860. doi:10.14778/3611540.3611569

[38] Yang Zhou, Zhongjie Chen, Ziming Mao, ChonLam Lao, Shuo Yang, Pravein Govindan Kannan, Jiaqi Gao, Yilong Zhao, Yongji Wu, Kaichao You, Fengyuan Ren, Zhiying Xu, Costin Raiciu, and Ion Stoica. 2025. An Extensible Software Transport Layer for GPU Networking. arXiv:2504.17307 [cs.NI] https://arxiv.org/abs/2504.17307