MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs

Ziheng Jiang and Haibin Lin, *ByteDance;* Yinmin Zhong, *Peking University;* 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, and Jianxi Ye, *ByteDance;* Xin Jin, *Peking University;* Xin Liu, *ByteDance*

https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs

Ziheng Jiang[1,*] Haibin Lin[1,*] Yinmin Zhong[2,*] Qi Huang[1] Yangrui Chen[1] Zhi Zhang[1]
Yanghua Peng[1] Xiang Li[1] Cong Xie[1] Shibiao Nong[1] Yulu Jia[1] Sun He[1] Hongmin Chen[1]
Zhihao Bai[1] Qi Hou[1] Shipeng Yan[1] Ding Zhou[1] Yiyao Sheng[1] Zhuo Jiang[1]
Haohan Xu[1] Haoran Wei[1] Zhang Zhang[1] Pengfei Nie[1] Leqi Zou[1] Sida Zhao[1]
Liang Xiang[1] Zherui Liu[1] Zhe Li[1] Xiaoying Jia[1] Jianxi Ye[1] Xin Jin[2,†] Xin Liu[1,†]

[1]*ByteDance* [2]*Peking University*

Abstract

We present the design, implementation and engineering experience in building and deploying MegaScale, a production system for training large language models (LLMs) at the scale of more than 10,000 GPUs. Training LLMs at this scale brings unprecedented challenges to training efficiency and stability. We take a full-stack approach that co-designs the algorithmic and system components across model block and optimizer design, computation and communication overlapping, operator optimization, data pipeline, and network performance tuning. Maintaining high efficiency throughout the training process (i.e., stability) is an important consideration in production given the long extent of LLM training jobs. Many hard stability issues only emerge at large scale, and in-depth observability is the key to address them. We develop a set of diagnosis tools to monitor system components and events deep in the stack, identify root causes, and derive effective techniques to achieve fault tolerance and mitigate stragglers. MegaScale achieves 55.2% Model FLOPs Utilization (MFU) when training a 175B LLM model on 12,288 GPUs, improving the MFU by 1.34× compared to Megatron-LM. We share our operational experience in identifying and fixing failures and stragglers. We hope by articulating the problems and sharing our experience from a systems perspective, this work can inspire future LLM systems research.

1 Introduction

Large language models (LLMs) [1] have emerged as a transformative technology in artificial intelligence (AI). Recent advancements in LLMs have significantly improved their capability. LLMs have demonstrated tremendous potential in a wide range of domains, such as machine translation, text summarization, and conversational agents [2]. As a company serving billions of users, we have been aggressively integrating AI into our products, and we are putting LLMs as a high priority to shape the future of our products.

Training LLMs is a daunting task that requires enormous computation resources. The scaling law [3] dictates that the model size and the training data size are critical factors that determine the model capability. To achieve state-of-the-art model capability, many efforts have been devoted to train large models with hundreds of billions or even trillions of parameters on hundreds of billions or even trillions of tokens. For example, GPT-3 [4] has 175 billion parameters and PaLM [5] has 540 billion parameters. Major players in this field build large-scale AI clusters with tens of thousands of GPUs to train LLMs.

Scaling LLM training to tens of thousands of GPUs brings unprecedented challenges. As AI has been at the core of many of our products, we have extensive experience in training deep neural networks (DNNs). Yet, training a model like ResNet [6] only takes tens or hundreds of GPUs. Compared to these models, the scale of training LLMs is unparallel. While we are not new to building and operating large-scale GPU clusters, these clusters are normally shared by many training jobs. Now, in the context of LLM training, a single job is occupying tens of thousands of GPUs and taking all the resources. The sheer scale of LLM training introduces two specific challenges from a systems perspective.

The first challenge is to achieve high training efficiency at scale. Model FLOPs utilization (MFU) is the ratio of the observed throughput to the theoretical maximum throughput assuming 100% of peak FLOPs [7]. It is a standard metric to evaluate training efficiency that directly translates to end-to-end training speed. LLM training is not embarrassingly parallel. To train an LLM, the model is split across GPUs and the GPUs heavily communicate with each other to make progress. Besides communication, other factors such as operator optimization, data preprocessing and GPU memory consumption also contribute significantly to MFU.

The second challenge is to achieve high training stability at scale, i.e., maintaining high training efficiency throughout

*Equal contribution.
†Correspondence to: Xin Liu <liuxin.ai@bytedance.com>, Xin Jin <xinjinpku@pku.edu.cn>.

the training process. Stability is particularly important from a production perspective, as LLMs take a long time to train. Training an LLM with one trillion tokens can take weeks. The scale and time are orders of magnitude larger than those of regular DNN training jobs. Failures and stragglers are the norm rather than the exception for LLM training. At such a scale, the consequences of failures and stragglers are devastating. Failures are very expensive, and it is critical to reduce the recovery time, given the large scale. A straggler not only affects its own work, but slows down the entire job involving tens of thousands of GPUs.

In this paper, we present the design, implementation and engineering experience of MegaScale, a production system for training LLMs at scale. MegaScale enables us to scale LLM training to more than 10,000 GPUs. We are able to harness the power of the massive number of GPUs to train LLMs with high training efficiency and stability. In building and operating MegaScale, we apply two systems principles: algorithm-system co-design and in-depth observability.

MegaScale is a specialized system tailored for LLM training. Algorithm-system co-design is a key principle to maximize performance for specialized systems, which has been applied widely in computer systems. We apply this principle to MegaScale in the context of LLM training with a full-stack approach that spans all important system components. We make several modifications and incorporate effective optimization techniques to the model architecture, including parallel transformer block [5], sliding window attention [8] and LAMB optimizer [9]. We leverage mixed parallelism strategies that combine data parallelism, pipeline parallelism, tensor parallelism, and sequence parallelism. Importantly, we design custom techniques based on the pattern of each parallelism strategy to maximize the overlapping between communication and computation. We apply prefetching and tree-based loading to optimize the data pipeline. We leverage non-blocking asynchronous operations and eliminate global barriers for large-scale collective communication group initialization. We design a custom network topology, reduce ECMP hash conflicts, customize congestion control, and tune retransmit timeout parameters for high network performance.

Stability problems including failures and stragglers in large-scale systems are notoriously hard to diagnose and fix. Many hard stability issues only emerge at large scale, which can stem from a wide range of software and hardware faults deep in the stack. Manually identifying and resolving every single issue is infeasible given the scale and complexity of the system. We apply the principle of in-depth observability to build a set of diagnosis tools. By 'in-depth observability', we mean a comprehensive monitoring and visualization strategy that penetrates beyond surface-level metrics to gather detailed, granular data across every component of the system stack, aiming to create a multidimensional view of system performance. The set of tools allows us to diagnose the system and identify root causes, by uncovering the intricate interactions and

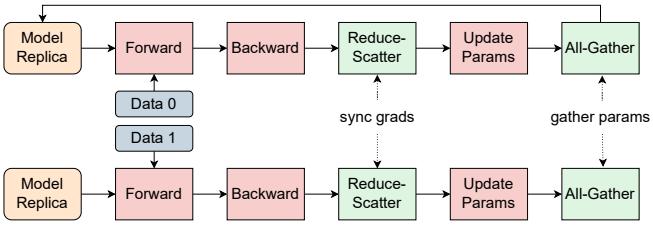


Figure 1: Data parallel training with Zero Redundancy Optimizer.

dependencies that contribute to stability issues. We develop a robust training framework to automate fault localization and recovery. We design heartbeat messages encapsulating various forms of information to facilitate real-time anomaly detection and provide early warnings. We implement a suite of diagnostic tests to identify nodes causing disruptions. We optimize the checkpointing and recovery procedure to reduce interruptions. To troubleshoot nuanced cases caused by stragglers, we develop a performance analysis tool to record fine-grained CUDA events and generate system-wide heat-map and timeline trace from a distributed view, and develop a 3D parallel training visualization tool to show data dependencies between ranks for diagnosis.

MegaScale is deployed in our datacenters to train LLMs for our products. Over the years, we have built several AI clusters with different size and hardware configurations. Our largest AI cluster has over 10,000 GPUs. In terms of training efficiency, MegaScale achieves 55.2% MFU when training a standard 175B transformer model on 12,288 GPUs, providing an improvement of 1.34× compared to the state-of-the-art open-source training framework Megatron-LM [10]. In terms of model converge and stability, we show a real production run of MegaScale that trains a proprietary model with hundreds of billions of parameters on multi-trillion tokens for several weeks. Over the weeks, the loss continues to converge, and MegaScale repairs and recovers the training process for over 100 times in presence of failures. We also share our experience in diagnosing and fixing some intriguing problems. We are working on open-sourcing components that can benefit the community on GitHub[3].

2 Background

The training of LLMs, characterized by their vast model architectures and massive datasets, is computationally intensive. Parallelism strategies distribute the training process across multiple devices.

Data parallelism. It replicates the model and optimizer states across multiple devices and the data is evenly divided among all devices. Each model replica executes the forward and backward propagation computation in parallel. Upon completion of each iteration, all model replicas synchronize to

[3]https://github.com/volcengine/veScale

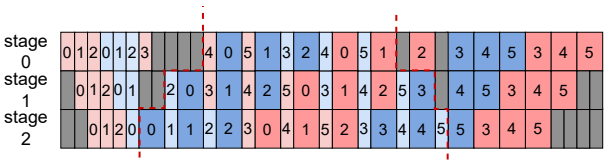


Figure 2: Interleaved 1F1B pipeline.

update the model. Instead of duplicating model states (like the optimizer states, gradients, and parameters), Zero Redundancy Optimizer (ZeRO) [11] shards these states across every data-parallel process. As a result, the traditional all-reduce operations that aggregate gradients are decomposed into separate reduce-scatter and all-gather operations. This is because every data-parallel process retains only a fraction of the total state. ZeRO is structured into three incremental stages of optimizations. Notably, the second stage is commonly adopted to shard both the optimizer states and gradients, while ensuring no additional communication overhead is introduced (Figure 1).

Pipeline parallelism. It distributes model layers among multiple devices and each device owns a portion of the model. Meanwhile, each training batch is subdivided into a number of micro-batches for pipelined execution. To reduce pipeline bubbles, various pipeline scheduling strategies are proposed, e.g., GPipe [12], PipeDream 1F1B [13], etc. Megatron-LM [7] employs the interleaved 1F1B scheduling. Each pipeline stage on every worker is subdivided into multiple virtual stages, which represents a subset of layers, referred to as a model chunk. Initially, workers enter a *warm-up* phase, executing the forward pass for a limited number of in-flight micro-batches. Following the warm-up, each worker progresses to the *steady* phase where workers perform one forward pass followed by one backward pass, often abbreviated as 1F1B. Upon concluding a batch, workers finalize the backward passes for any remaining in-flight micro-batches during this *cool-down* phase. Figure 2 shows an three-stage pipeline where each stage is further divided into two virtual stages.

Tensor parallelism. It distributes individual operators over multiple devices, with each device executing a portion of the computation in parallel. Depending on the specific partitioning strategy and its relationship to prior and subsequent operators in the model, partitioning can require communication among participating GPUs to split the input and then merge the output. For example, we can split GEMMs in the MLP and self-attention blocks among multiple GPUs to utilize more computational units. Some other operations like LayerNorm and Dropout are less computationally intensive but demand a considerable amount of activation memory. Another form of tensor parallelism called sequence parallelism is proposed to distribute these operators along the sequence dimension to effectively reduce the activation memory footprint.

Combination of parallelism strategies. These parallelism strategies can be combined into 3D parallelism to scale the training of LLMs across many GPUs [10]. Given the high communication overhead associated with tensor parallelism, it is preferable to confine such communication within a single cluster node. Conversely, data parallelism and pipeline parallelism are more amenable to inter-node communication. In this case, we choose to prioritize building the data parallelism groups over pipeline parallelism, which can mitigate cross-minipod communication for data parallelism.

3 Efficient Training at Scale

In the realm of LLMs, efficient training at scale becomes paramount. As we venture into deeper and more expansive models, the computational demands surge explosively. Handling such computation requirements without compromising on model accuracy necessitates the adoption of state-of-the-art algorithmic optimizations, communication strategies, data pipeline management, and network performance tuning techniques. This section delves deep into the methods employed to optimize the training of large models in order to achieve high training efficiency at scale.

3.1 Algorithmic Optimizations

We make a few modifications and incorporate recent optimizations at the algorithmic level to improve training efficiency, without compromising accuracy. We validate the impact of these techniques on model convergence in §6.2.

Parallel transformer block [14]. We adopt a parallel version of the transformer block in lieu of the standard serialized formulation. Specifically, the standard formula of the transformer block can be reformatted from

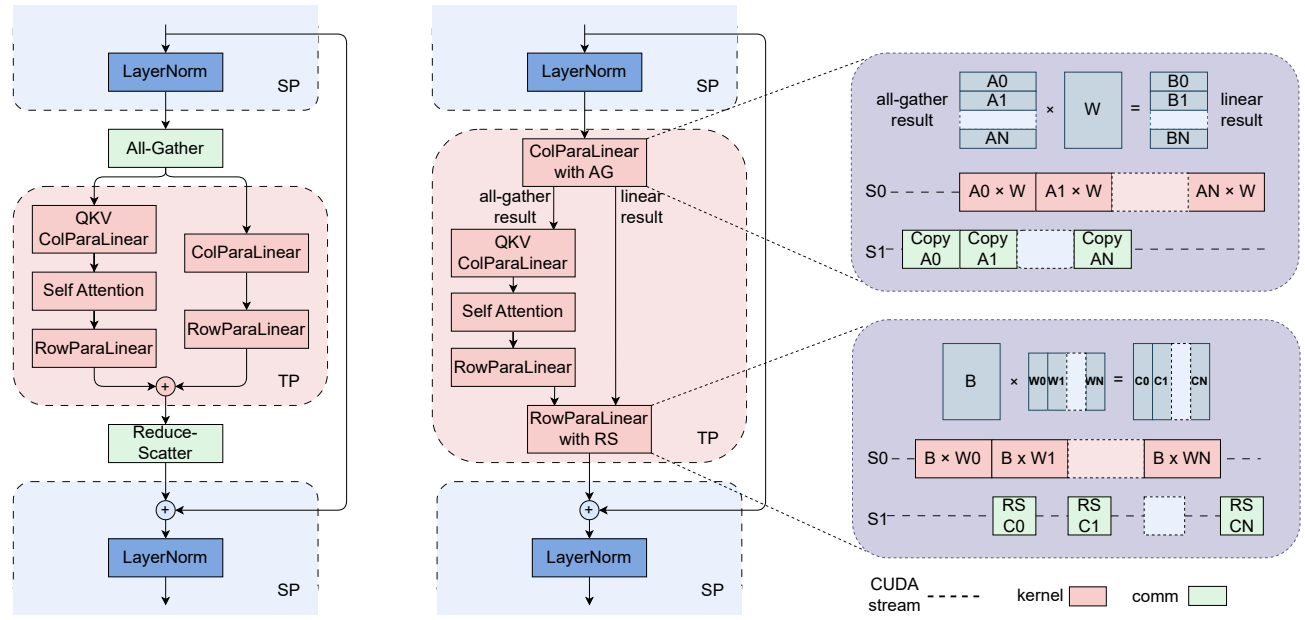
$$y = x + \text{MLP}(\text{LN}(x + \text{Attention}(\text{LN}(x)))) \quad (1)$$

into

$$y = x + \text{MLP}(\text{LN}(x)) + \text{Attention}(\text{LN}(x)) \quad (2)$$

With this approach, the computation of the attention block and the MLP block can be executed in parallel, thereby reducing the computation time. Prior work [5] shows that this modification does not degrade the quality of models with parameters in the hundreds of billions.

Sliding window attention (SWA). Sliding window attention [8] is a sparse attention mechanism that employs a fixed-size window surrounding each token in the input sequence. The computation complexity is $O(s \times w)$, where s is the input sequence length and w is the fixed window size. Sliding window attention is more efficient than the full self-attention, whose computation complexity is $O(s \times s)$, given that $w \ll s$. Past work [8] and our micro-benchmark (§6.2) have shown that the information across the entire input can be retained



(a) PTB with SP and TP (b) Fuse communication into Linears (c) Overlap communication with GEMM

Figure 3: Overlapping communication in tensor parallelism (TP) and sequence parallelism (SP) with parallel transformer block (PTB).

with a large receptive field created by stacking layers of such windowed attention. This enables faster training without compromising the accuracy.

LAMB optimizer. Efficient training at a large scale is often hindered by batch size constraints. Particularly, increasing the batch size may adversely affect model convergence. The LAMB optimizer [9] has been demonstrated to enable the scaling of BERT's training batch size to 64K without compromising accuracy. In the LLM setting, our experiments find that LAMB can scale the batch size to 4× without accuracy loss. With interleaved pipeline parallelism, the original schedule contains $\frac{4}{v}\frac{p-1}{m}$ pipeline bubbles when training four steps with 1× batch size [7], while the pipeline bubbles of training one step with 4× batch size are $\frac{1}{v}\frac{p-1}{4m}$. Hence, MegaScale reduces 87.5% of the pipeline bubbles via LAMB optimizer.

3.2 Communication Overlapping in 3D Parallelism

To reduce the iteration time, we systematically analyze the dependencies between computation and communication for all the operators in 3D parallelism, and design techniques to hide the overhead of all the off-the-critical-path operations.

Overlapping in data parallelism. As shown in Figure 1, for data parallelism, two main communication operations stand out. One is the *all-gather* operation, which fetches the most recent model parameters from workers in other data parallel ranks during the forward pass. The other is the *reduce-scatter* operation, which collect the gradients in the backward pass. In 3D parallelism, a single device may host multiple model chunks. Overlapping is implemented on a model chunk basis to maximize bandwidth utilization. The *all-gather* operation is triggered prior to the forward pass of a model chunk, and the *reduce-scatter* operation commences after its backward pass. This results in a challenge where the first *all-gather* operation and the last *reduce-scatter* operation cannot be hidden. Inspired by PyTorch FSDP [15], the initial *all-gather* operation is pre-fetched at the beginning of each iteration, allowing it to overlap with data loading operations, effectively reducing the communication time by a factor of $1/(2 * vpp\_size)$. We also launch the high priority communication first to maximize overlapping. The priorities of communication operators are determined by the order of the corresponding computation operators that depend on the communication result.

Overlapping in pipeline parallelism. Pipeline parallelism features point-to-point *send/receive* communication. MegaScale uses the interleaved 1F1B scheduling method mentioned in 2. We note that in the warm-up phase, the forward pass only depends on its previous *receive*. We thus decouple the *send* and *receive*, which are often implemented together and can be blocked by the slower one. By breaking this dependency, we enable the send operation to overlap with the computation as shown in the left part of Figure 4. The cool-down phase can be viewed as the inverse of the warm-up phase, allowing for the inverse application of the same technique. As for the steady phase, both the forward and backward computation are independent of adjacent communication operations. Taking the backward as an example, as shown in the right part of Figure 4, its previous *receive* is for the next forward compu-

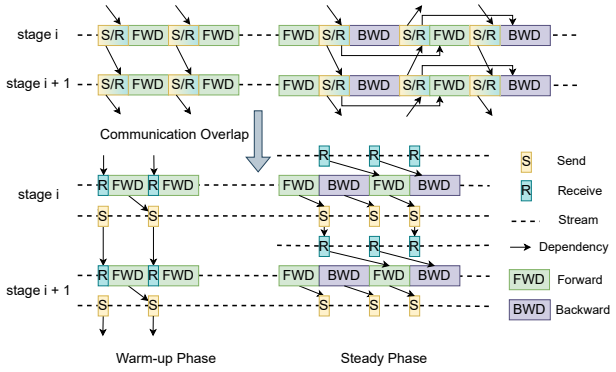


Figure 4: Overlapping communication in pipeline parallelism.

tation while the *send* is for the backward computation in the previous stage. So the *send* and *receive* operations can be launched asynchronously to overlap with the computation.

Overlapping in tensor/sequence parallelism. Tensor parallelism is commonly used to partition weights in computational-intensive operations, while operations like *LayerNorm* and *Dropout* are partitioned along the sequence dimension to save GPU memory. This necessitates *all-gather* and *reduce-scatter* operations for input collection and output redistribution across GPUs. Figure 3a shows this communication pattern in the parallel transformer block architecture. Here the two communication operators are in the critical path. To eliminate this overhead, we choose to fuse *all-gather* and *reduce-scatter* with the parallel *Linears* on the FFN path (Figure 3b). Since the GEMM kernels on the FFN path is larger, the communication can be hidden better. We break the GEMM kernel into small chunks, and pipeline the execution with the communication (Figure 3c). This strategy can be applied in the backward pass similarly.

3.3 Efficient Operators

Despite the optimization for GEMM operators in Megatron-LM, we identify opportunities for further enhancement in other operators. For the attention part, we adopt FlashAttention-2 [16], which improves work partitioning between different thread blocks and warps. For LayerNorm and GeLU, we observe that they are composed of fine-grained kernels in previous implementations. By fusing these kernels together, we reduce the overhead associated with launching multiple kernels and aid in optimizing memory access patterns, thereby achieving better performance.

3.4 Data Pipeline

Data preprocessing and loading are often overlooked. However, these operations create non-negligible GPU idle time at the beginning of each training step. Optimizing these operations are essential for efficiency of the training process.

Asynchronous data preprocessing. Data preprocessing is not on the critical path. As a result, while the GPU workers are synchronizing gradients at the end of each training step, the data preprocessing for the subsequent step can start, which hides the preprocessing overhead.

Redundant dataloader elimination. In a typical data loading phase of distributed training, each GPU worker is equipped with its own data loader, responsible for reading training data into the CPU memory before forwarding it to the GPU. This leads to competition among workers for disk read bandwidth, thereby creating a bottleneck. Notably, we observe that in the LLM training setting, GPU workers within the same machine are in the same tensor parallel group. Consequently, their inputs for each iteration are inherently identical. Based on this observation, we adopt a two-layer tree-based approach. We use a single, dedicated data loader on each machine to read the training data into a piece of shared memory. Subsequently, each GPU worker is responsible for copying the necessary data to its own GPU memory. This eliminates redundant reads and significantly enhances the efficiency of data transfer.

3.5 Collective Communication Group Initialization

In distributed training, the initialization phase involves the establishment of NVIDIA Collective Communications Library (NCCL) communication groups among GPU workers. Since this overhead is relatively negligible in small-scale scenarios, `torch.distributed` is used by default. As the number of GPUs scales to over ten thousand, the overhead introduced by naive implementations becomes intolerable. We conduct experiments on the same AI cluster in §6 and our empirical measurement indicates that the initialization time for Megatron-LM on 2,048 NVIDIA Ampere GPUs is approximately 1047 seconds. While this may appear relatively small compared to the training duration, it imposes a significant hurdle to routine testing and iterative development (e.g., minor code adjustments in hyperparameter tuning and debugging). It also hampers the implementation of fast restart-and-recovery mechanisms.

To address this issue, we perform a detailed profiling of `torch.distributed` [17] and identify two primary causes of excessive initialization time. The first issue resides in the synchronization step, where each process is involved in a barrier operation at the end of initialization a specific communication group. This barrier uses TCPStore, an inner distributed Key-Value Store implementation in Pytorch which operates in a single-threaded, blocking read-write manner. We replace TCPStore with Redis, which is non-blocking and asynchronous. This reduces the initialization time to 361 seconds on 2,048 GPUs. The second issue is related to the incautious usage of global barriers. Each process executes a global barrier after initializing its corresponding communication group. We care-

fully design the order in which communication groups are initialized to minimize the need for global barriers. This approach lowers the time complexity of the global barrier from $O(n^2)$ to $O(n)$. The initialization time is reduced to under 5 seconds on 2048 GPUs, and to under 30 seconds on more than 10,000 GPUs with those optimizations.

3.6 Network Performance Tuning

We analyze the traffic across machines in 3D parallelism and design techniques to improve network performance.

Network topology. Our datacenter network is built with high-performance switches based on Broadcom Tomahawk 4 chips. The total bandwidth of each Tomahawk chip is 25.6Tbps with 64×400Gbps ports. Three layers of switches are connected in a CLOS-like topology to connect more than 10,000 GPUs. For switches at each layer, the bandwidth percentage between downlink and uplink is 1:1. That is, 32 ports are used as downlink and 32 ports are used as uplink. The network provides high bandwidth with a small diameter. Every node can communicate with other nodes within a limited number of hops.

Reducing ECMP hashing conflicts. We carefully design the network topology and schedule network traffic to reduce ECMP hashing conflicts. First, at the top-of-rack (ToR) switch level, one 400G downlink port is split into two 200G downlink ports with specific AOC cables. The conflict probability is reduced as the bandwidth of each uplink is double of that of a downlink. Second, eight 200G NICs on the server is connected to eight different switches in a multi-rail way. The number of GPU servers connected by the same sets of ToR switches can reach 64. And we strategically schedule the data-intensive nodes from our training tasks to operate under the same Top of Rack (ToR) switch. This approach significantly reduces the number of switch hops required for communication and further reduce ECMP hashing conflicts probability.

Congestion control. In distributed training, all-to-all communication may lead to congestion and elevated levels of Priority Flow Control (PFC) [18] when employing the default DCQCN [19] protocol at scale. Excessive use of PFC can result in head-of-line (HoL) blocking [19], thereby diminishing network throughput. To mitigate these issues, we have developed an algorithm incorporating principles from both Swift [20] and DCQCN, which integrates the precise measurement of Round-Trip Time (RTT) with the rapid congestion response capabilities of Explicit Congestion Notification (ECN). This approach significantly enhances throughput and minimizes congestion related to PFC.

Retransmit timeout setting. Parameters in NCCL can be set to control retransmit timer and retry count. We tune these parameters for fast recovery under link flapping. To further reduce the recover time, we enable the adap_retrans feature on the NIC. This feature enables retransmission in a shorter interval and help recover the transmission more quickly when the link flapping period is short.

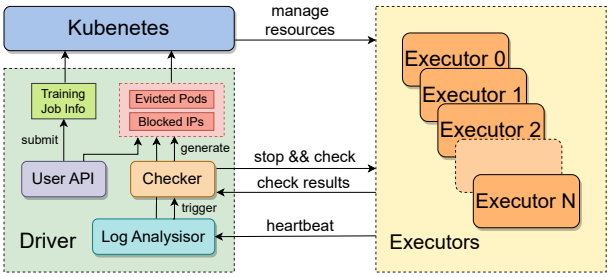


Figure 5: Robust training workflow.

4 Fault Tolerance

As the training cluster scales to over tens of thousands of GPUs, software and hardware faults become virtually inevitable. We introduce a robust training framework for LLM training that achieves automatic fault identification and fast recovery, enabling fault tolerance with minimal human intervention and negligible impact on ongoing training tasks.

4.1 Robust Training Workflow

As Figure 5 shows, upon receiving a submitted training task, the driver process interfaces with a custom Kubernetes to allocate computing resources and initiate the corresponding Pod for each executor. One executor manage one node. Once the executor has completed a series of initialization tasks, it creates the training process on each GPU and a robust training daemon which sends heartbeat to the driver periodically. These heartbeats encapsulate various forms of information to enable real-time anomaly detection and issue early warnings (§4.2). When the driver process detects an abnormal status in a particular training process, or fails to receive a heartbeat from an executor within a predefined time window, it triggers the fault recovery procedure. The driver will suspend the ongoing training task across all executors and command them to run a series of self-check diagnostics (§4.3). These diagnostic tests are carefully designed to be lightweight yet comprehensive, covering the majority of common hardware and software faults. Once the problematic nodes are identified, the driver submits the IP addresses of the nodes to be blocked, along with the information of the Pods running on them, to Kubernetes, which evicts the faulty nodes and replenishes the cluster with an equivalent amount of healthy ones which pass our diagnostic tests. Additionally, we provide a user interface that allows for manual eviction of nodes, particularly for those identified through manual analysis as in §5. After the recovery process is complete, the driver resumes training from the latest checkpoint. We optimize the checkpoint and resume process to minimize the loss of training progress (§4.4).

4.2 Data Collection and Analysis

The heartbeat messages includes the basic information of the executor, such as the IP address, the Pod name, and hardware information, etc. Additionally, the current status of the training processes is reported, enabling the driver to promptly detect any explicit anomalies. The *stdout/stderr* logs of training processes are also included. They will be aggregated, filtered and analyzed on the fly. If specific warning or error keywords are detected, the driver will report real-time diagnostic information. Moreover, RDMA traffic metrics are also included, serving as an indicator for network utilization and efficiency. Some anomalies in the training process may not manifest as explicit errors, giving the appearance that training is proceeding as expected. In such cases, RDMA traffic metrics serve as a critical indicator. Given the periodic nature of the training tasks, the network traffic characteristics for each step should exhibit similar patterns. Therefore, any significant decline or abnormal fluctuation in RDMA traffic is a signal of potential anomalies. Upon detecting such irregularities, the driver will issue alerts for manual investigation. If the traffic ceases entirely, the driver will automatically initiate the fault recovery procedure.

In order to enhance the monitoring of training stability and performance, we have developed a monitoring system with precision reaching the millisecond level. Different levels of monitoring are employed to track various indicators. Second-level monitoring is typically used to assess the overall health status and to rule out common configuration impacts on training. For instance, ECN/PFC/QoS configurations, link flapping, or any other issues of NICs. Millisecond-level monitoring, on the other hand, is used to determine if the network is congested and whether the data transfer speed of data parallelism and pipe parallelism has reached its physical limit.

4.3 Diagnostic Tests

There exists a trade-off between execution time and accuracy in self-check diagnostics. Extended diagnostic duration can adversely affect the effective training time, while high false positive rates can lead to unnecessary exclusion of machines that are actually functional. Through iterative experimentation and optimization, we have deployed a suite of lightweight diagnostic tests that effectively cover a broad spectrum of hardware and software faults encountered during actual training processes.

Intra-host network tests. To diagnose potential bottlenecks in intra-host network, we use our internally developed tool to test two things. The *Loopback* test measures the loopback bandwidth from all RDMA NICs (RNICs) to various intra-host endpoints, including memory nodes and GPUs. It conducts a full-mesh test within the host, covering all possible link combinations. This allows us to infer link-specific bandwidth degradation and irregularities in PCIe configurations based on end-to-end bandwidth results. The second *RNIC-to-RNIC* test examines the connectivity and bandwidth performance between different RNICs on the same host. These tests provide insights into whether the RNICs meet the hardware speed specifications and whether the underlying routing configurations are correctly configured.

NCCL tests. To identify potential faults in GPU communication, we run an *all-to-all* test among the GPUs within a single node to observe whether the bandwidth aligns with expected benchmarks. Once intra-host communication test is passed, each node also conducts an *all-reduce* test with neighboring machines under the same ToR switch to assess inter-node GPU communication.

4.4 Fast Checkpointing and Recovery

After identifying and evicting faulty machines, the driver needs to resume the training by loading model weights and optimizer states from the most recent checkpoint. It is critical to ensure that the latest checkpoint is as close as possible to the state of training progress when the faults happened, to minimize loss in computation and time. This require us to increase the frequency of checkpointing during training. However, we also want to reduce the latency introduced by the checkpointing process, especially the time on the critical path which blocks the training progress, thus impeding the overall system throughput.

To achieve fast checkpointing, we introduce an optimized, two-stage approach. In the first stage, each GPU worker writes its on-chip states to the host memory, and then continues the training process. After the optimization of Pytorch's serialization mechanism and the use of pinned memory, this process can be reduced to several seconds thanks to the high PCIe bandwidth, thereby minimally interrupting the ongoing training process. In the second stage, a background process takes over, asynchronously transferring the state from the host memory to a distributed file system (HDFS in our deployment) for centralized maintenance. This decoupling of operations into two stages allows the GPU workers to resume training almost immediately after dumping their state, while the more time-consuming process of writing to HDFS is offloaded to a separate, non-blocking process.

In the context of recovery from a checkpoint, it is on the critical path since training can not be started without the last checkpoint. The bottleneck is the bandwidth of HDFS, especially when each GPU worker needs to read its corresponding state partition. To alleviate this bottleneck, we propose an optimized data retrieval strategy. We recognize that multiple GPU workers often share the same state partition, e.g., the workers in the same data parallel group. Accordingly, we designate a single worker in the group to read the shared state partition from HDFS, thereby reducing the load linearly. This worker then broadcasts the state partition to all other GPU workers that share the same data. This approach effectively

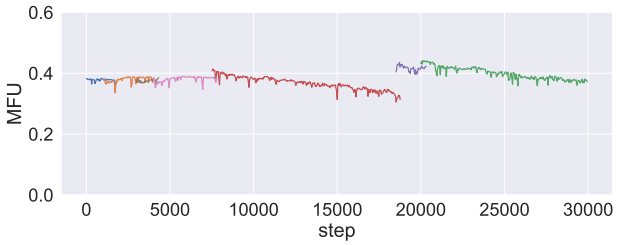


Figure 6: Inconsistent MFU observed in large-scale training. Different colors denote distinct executions of the same training job.

mitigates the bandwidth constraints of HDFS, leading to a substantial reduction in the recovery time.

5 Training Troubleshooting

Although our robust training framework automatically discovers, pinpoints, and resolves the majority of common faults, there remain certain hardware anomalies that manifest probabilistically and cannot be found by machine self-checks. Some anomalies may make the system appear to operate normally, yet significantly degrades the training efficiency. To address these nuanced cases, we have implemented several custom monitoring and analysis tools designed to support case-by-case anomaly detection.

5.1 Performance Diagnosis with CUDA Event Monitor

At the scale of tens of thousands of GPUs, we observe that, unlike in smaller-scale experiments, different runs exhibit varying computational efficiencies. Even with identical configurations , this inconsistency persist, as shown in Figure 6. We also observed that the performance of training tasks is not consistent at this scale. The MFU for various training tasks gradually declines over time. While this leads us to suspect variations between individual machines, no evident variations are detected under single GPU GEMM micro-benchmarks. To diagnose those performance issues, we develop a performance analysis tool that records the execution time of critical code segments on each machine rank during a run. In contrast to previous tools such as the torch profiler or the Megatron-LM timer, our tool times events based on the CUDA events method. This approach minimizes the need for CUDA synchronization, thus preventing performance degradation, allowing us to consistently run it in our production training jobs. This tool offers two visualization modes and can analyze the collected data from different perspectives.

The first mode uses a heat map to show time consumption differences between machines from various dimensions, depicted in Figure 7. We gather latency data of the computation phase (forward and backward) across devices and average the latency across steps. The aggregated data is visualized

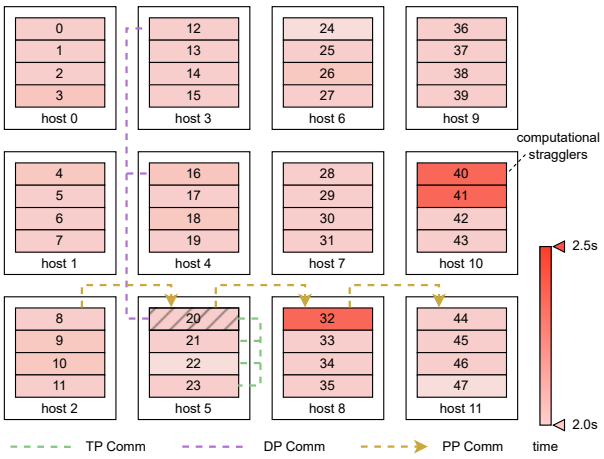


Figure 7: Performance heat-map. The color denotes the running time of the code segments on a rank. The figure also shows the 3D visualization feature, where rank 20 has been selected and the dependency across different parallelism dimensions become visible.

using a heat-map. The heat-map reveals that a minor fraction of machines (approximately 0.5%) exhibit substantially slower performance during training, thereby hindering overall training progress. The training efficiency is predominantly determined by the slowest machine's performance (i.e., stragglers), leading to inconsistencies in training efficiency across diverse runs, since machine scheduling within the cluster is stochastic. After excluding these outlier machines, the peak MFU across runs becomes consistent.

The other mode displays the event timeline on machines in a trace format from different distributed views (data parallelism, pipeline parallelism, tensor parallelism). Traditional profiler, such as PyTorch Profiler, is primarily designed for single-node activity analysis. This approach offers limited insight in distributed training scenarios where execution dependencies frequently span across multiple nodes. By aggregating the trace spans of various ranks onto a singular timeline, we gain a comprehensive perspective, revealing the overall execution order, pipeline bubbles, and synchronization characteristics among data parallel ranks. Figure 8 displays how our distributed tracer visualizes the actual execution of pipeline parallelism, detailing the data dependencies between different pipeline stages through the consolidation of event data across a pipeline parallelism group.

Every piece of data from the CUDA event timer is stored in a remote analytical database, allowing for easy retrieval of details from any step event. While the timer data is wrote to a local file in a line-by-line format, a separate streamer process then synchronizes this log file with a Kafka queue in real-time. The analytical database remains updated by consuming data from this Kafka queue, enabling on-the-fly analysis without interrupting the training job. All the monitoring features are turned on during real production training and the overhead is negligible compared to the training time.

Figure 8: The trace shows events collected in a pipeline group on a unified timeline. Dependencies become visible when an event is selected.

5.2 3D Parallel Training Visualization

With 3D parallelism and our optimization techniques (§3), the landscape of data flow and task sequencing is exceedingly intricate. Each GPU worker may be engaged in several synchronous or asynchronous operations at the given moment, leading to complex dependencies among them. This intricacy amplifies the challenges of fault diagnosis: when a single GPU worker experiences a fault, the entire cluster of nodes can stall in the NCCL communication operations, ultimately leading to a system-wide timeout. Externally, this situation manifests as a generic blockage, but the root cause of which is often buried under a deluge of timeout messages. To rapidly pinpoint the problematic nodes, we let each GPU worker log its own ongoing event upon communication timeout. These logs are then used to construct a visual representation of data dependencies based on the logical topology in the 3D parallel setting.

As Figure 7 shows, the cluster in 3D parallel training can logically be split into three dimensions: tensor parallelism, pipeline parallelism, and data parallelism. When we select a specific GPU worker, it displays its position within the logical topology, the direction of data flow and the different communication operations it involves. Importantly, in the event of an error, the tool provides direct access to the worker's error messages if any. This serves as a powerful tool for diagnosing training anomalies, enabling quicker identification and resolution of faults.

Consider the aforementioned case when defective GPUs probabilistically cause blocking when executing NCCL communication operations. Such blocking can hang the entire machine, leading to cascading timeouts across other dependent nodes and ultimately resulting in the paralysis of the entire training process. To swiftly identify these faulty nodes, we utilize the 3D parallel training visualization tool. Nodes that timeout due to waiting for the faulty ones will log their ongoing operations upon exiting. In contrast, the nodes with the faulty GPUs are hung and do not log any such information. Therefore, by examining the logs and the data flow within the visualization, these problematic nodes can be easily pinpointed. Once identified, these nodes can be manually isolated and flagged for maintenance through the robust training framework, as described in 4.1.

Model Size	Heads	Hidden Size	Layers	TP	PP
175B	128	12288	96	8	8
530B	160	20480	105	8	35

Table 1: Model configurations.

6 Experience

In this section, we describe our deployment and operational experience of MegaScale. We build dedicated AI clusters for LLM training. Over the years, we have iterated several versions of our specialized AI cluster architecture, and we are currently operating several AI clusters with varying size and hardware configurations. We use these AI clusters to train a wide range of models, from computer vision and recommendation models to LLMs. With the increasing importance of LLMs, we are building AI clusters with larger size to cater the need of LLM training. As of September 2023, the largest AI cluster in our production for LLM training contains more than 10,000 NVIDIA Ampere GPUs. We are also in the process of building large clusters based on the newest NVIDIA Hopper GPUs, as NVIDIA is ramping up production.

6.1 Training Performance

MegaScale is built on top of Megatron-LM [7], which is a state-of-the-art open-source LLM training framework that integrates 3D parallelism techniques and takes advantage of hardware resources. Our experiments use the Megatron-LM (commit hash: 285068c8) on Github [21], committed on January 11, 2023, chosen for its stability and feature set at the commencement of our experiments. We use the same batch size for Megatron-LM and MegaScale for fair comparison. We use two model sizes: 175B parameters and 530B parameters. We use interleaved pipeline-parallel schedule [22] with six and three interleaving stages for the 175B and 530B models, respectively. Sequence length is 2,048 and vocabulary size is 64,000 for all the cases. Table 1 shows the details of the model configuration.

Scalability. Figure 9 compares Megatron-LM and MegaScale when training the 530B model, where we set the batch size as the number of GPUs with adjusted learning rate to show the MFU results. We see that the MFU of MegaScale is higher than Megatron-LM by up to 6.1%. With increasing scales, the

Batch Size	Method	GPUs	Iteration Time (s)	Throughput (tokens/s)	Training Time (days)	MFU	Aggregate PFlops/s
768	Megatron-LM	256	40.0	39.3k	88.35	53.0%	43.3
		512	21.2	74.1k	46.86	49.9%	77.6
		768	15.2	103.8k	33.45	46.7%	111.9
		1024	11.9	132.7k	26.17	44.7%	131.9
	MegaScale	256	32.0	49.0k	70.86	65.3%(**1.23**×)	52.2
		512	16.5	95.1k	36.51	63.5%(**1.27**×)	101.4
		768	11.5	136.7k	25.40	61.3%(**1.31**×)	146.9
		1024	8.9	176.9k	19.62	59.0%(**1.32**×)	188.5
6144	Megatron-LM	3072	29.02	433.6k	8.01	48.7%	466.8
		6144	14.78	851.6k	4.08	47.8%	916.3
		8192	12.24	1027.9k	3.38	43.3%	1106.7
		12288	8.57	1466.8k	2.37	41.2%	1579.5
	MegaScale	3072	23.66	531.9k	6.53	59.1%(**1.21**×)	566.5
		6144	12.21	1030.9k	3.37	57.3%(**1.19**×)	1098.4
		8192	9.56	1315.6k	2.64	54.9%(**1.26**×)	1400.6
		12288	6.34	1984.0k	**1.75**	55.2%(**1.34**×)	2166.3

Table 2: Strong-scaling training performance for the 175B model. We set the batch size to 6144 when training with 3072 to 12288 GPUs. For 256 to 1024 GPUs, we decrease the batch size to 768 due to GPU memory limit. We report the training time required for training 300B tokens here. The number in parentheses in the MFU column represents the speedup of MegaScale compared to Megatron-LM.

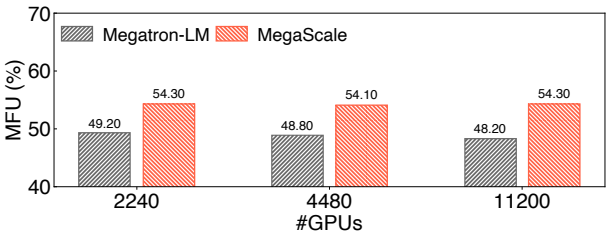


Figure 9: Weak-scaling training performance of Megatron-LM and MegaScale on the 530B model, where the batch size is scaled proportionally with the number of GPUs.

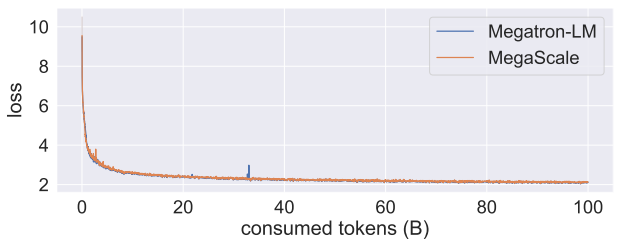
MFU of Megatron-LM decreases by 1.6% with more stragglers and communication, while MegaScale has near-linear scalability due to 3D-parallel communication overlapping.

In Table 2, we evaluate the strong-scaling training performance of Megatron-LM and MegaScale on the 175B model by increasing number of GPUs and maintaining a constant batch size. This experimental setting is more realistic, given that batch size is constrained by convergence effects and cannot be indefinitely scaled with the number of GPUs. MegaScale achieves up to 1.34× speedups over Megatron-LM across all settings. With increasing GPUs, we observe the MFU of MegaScale decreases from 59.1% to 55.2%. This is expected since the batch size is fixed and the computation-to-communication ratio decreases with more GPUs. Even in the largest scale with 12,288 GPUs, MegaScale still outperforms Megatron-LM by 14% MFU. For the smaller scale training, the speedup of MegaScale over the baseline ranges from 1.23× to 1.32×. Note that the difference in the maximum number of GPUs between this and the previous experiments (e.g., 12,288 vs. 11,200) is due to distinct 3D parallelism configurations for 175B and 530B models.

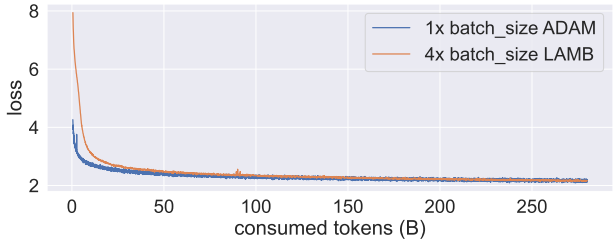
Ablation study. We evaluate the effectiveness of our optimization techniques of MegaScale. Table 3 shows the MFU improvement breakdown with different optimizations when training the 175B model on 256 GPUs. The baseline is the original Megatron-LM and has 47.7% MFU. It is worth noting that the networking optimizations are turned on for both Megatron-LM and MegaScale in this evaluation. We first apply two algorithmic techniques, parallel transformer block and sliding window attention, to Megatron-LM, achieving 5.6% MFU improvement. Communication is the major bottleneck of large-scale LLM training, and the 3D parallel communication overlapping of MegaScale hides the overhead and accelerates training by 6.2% MFU. We further adopt efficient operators and obtain 1.7% acceleration. Other optimizations such as data pipeline optimizations and the problematic code elimination mentioned in 6.3 further achieves 1.1% performance gain. Finally, we scale the batch size from 256 to 768 with LAMB optimizer, which significantly extends the steady phase in interleaved pipeline parallelism and achieves 3.0% MFU improvement. To sum up, MegaScale outperforms the baseline by 17.6% in the MFU number with all these optimizations.

6.2 Model Convergence and Stability

Model convergence microbenchmarks. We first conduct microbenchmark experiments to validate the algorithm tech-



(a) The training loss curve of MegaScale, which includes algorithm optimizations, in comparison with Megatron-LM.



(b) The training loss curve of ADAM optimizer and LAMB optimizer with four times of the batch size.

Figure 10: The training loss curves in microbenchmark experiments.

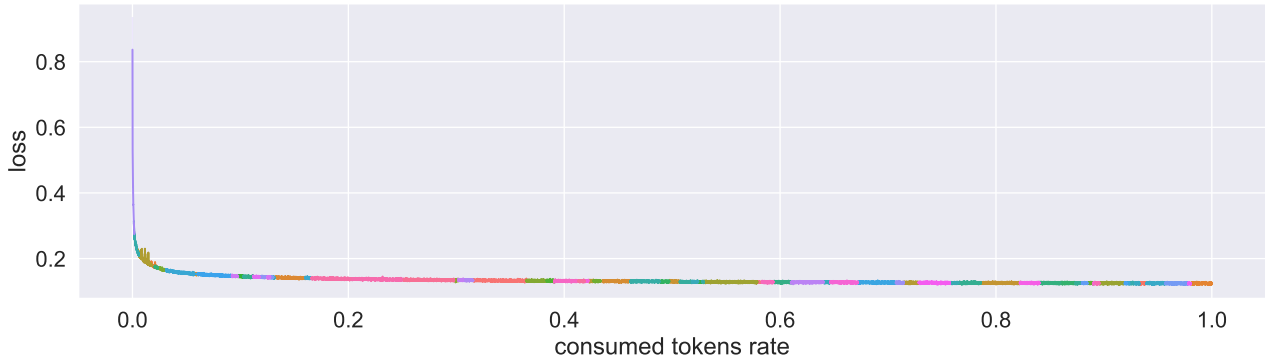


Figure 11: The normalized training loss curve of a real production run on more than 10,000 GPUs for several weeks. This run trains a model with hundreds of billions of parameters on multi-trillion tokens. Different colors indicate training restarts. MegaScale repairs and recovers the training process for over 100 times in presence of failures.

Idx	Method	MFU (Δ MFU)
1	baseline	47.7%
2	(1) with PTB	52.3% (4.6%)
3	(2) with SWA	53.3% (5.6%)
4	(3) with TP overlap	55.5% (7.8%)
5	(4) with PP overlap	58.0% (10.3%)
6	(5) with DP overlap	59.5% (11.8%)
7	(6) with efficient operators	61.2% (13.5%)
8	(7) with misc optimizations	62.3% (14.6%)
9	(8) with LAMB (BS×3)	65.3% (17.6%)

Table 3: MFU improvement breakdown when training the 175B model with 256 GPUs and batch size 256.

niques do not affect the model convergence. Due to the resource limit, the microbenchmarks are done on the 13B model. As shown in Figure 10a, while MegaScale adopts algorithm techniques, including parallel transformer block and sliding window attention, it achieves comparable loss results with the baseline when training with more than 100B tokens. We also evaluate the effect of LAMB optimizer as depicted in Figure 10b, which shows that LAMB optimizer with four times of batch size achieves the same loss as ADAM optimizer after around 250B tokens. Based on these observations, we turn on all the algorithmic optimizations in production training.

Model convergence and stability in real production LLM training. We show the model convergence and stability from a real production run. This run trains a proprietary model with hundreds of billions of parameters on multi-trillion tokens. This run uses more than 10,000 GPUs and lasts for several weeks. Figure 11 shows the loss continues to converge, with distinct colors indicating the training is restarted. Over the several weeks of this run, we experience training restarts over 100 times. With the robust training framework, over 90% of software and hardware faults are automatically identified and fixed by the techniques detailed in §4. The rest of the problems are handled with the help of the troubleshooting tools described in §5.

6.3 Problems Discovered and Fixed

We conduct an analysis of the fault records for the aforementioned production training job over several weeks. Our findings indicate that over 90% of the exceptions among them are automatically detected, located, and recovered using our robust training framework, such as CUDA error and segmentation fault. The average time required for detecting failure and

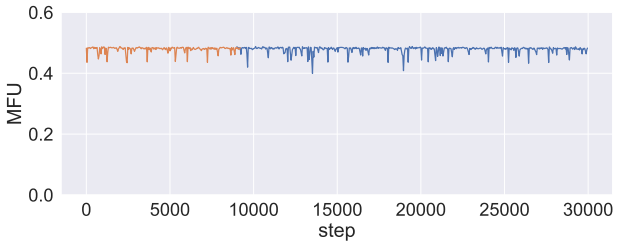


Figure 12: The MFU becomes stable after addressing the stragglers and problematic code segments. Different colors represent different training trials with the same setup.

executing diagnostic tests is less than 10 minutes. Moreover, the system can catch up to the training progress prior to the crash within 15 minutes from the latest checkpoints, maintaining over 90% effective training time rate, which is calculated as the number of iterations multiplied by the iteration training time, divided by the total training time. Below we show our experience in diagnosing and fixing some intriguing problems, which need to be analyzed using the troubleshooting tools in §5.

Computational stragglers. Building upon our utilization of CUDA event timers, we made another pertinent observation across multiple experimental setups. We noted that specific hosts took approximately 10% more time to execute the same forward computations compared to other ranks. This consistency across different experiments led us to conclude that the issue was not with the software but rather inherent to certain machines in the cluster. After isolating and removing these problematic hosts from the cluster, we observed an approximate 0.7% improvement in MFU.

MFU decreasing. In such large-scale training experiments, another phenomenon we observed is that training efficiency did not remain consistent over time. Instead, as the training progressed, the MFU of our training job gradually decreased. Through a step-by-step analysis based on CUDA event timer metrics, we noted several key findings. While the time consumed per training step was increasing, the time spent on forward, backward, and optimizer computations remained stable, irrespective of the increasing number of steps. This led us to infer that the time increase must be attributed to the collective communication overhead. Upon a reverse chronological examination, we identified the last collective communication step as the gradient reduce-scatter in data parallelism. If this step is delayed, the overall time per step elongates. Since we observed network bandwidth to be largely stable, we ruled out slowed communication speed as a factor for the increased time. According to the synchronization characteristics of collective communication, this leaves us with one conclusion: some ranks initiate the reduce-scatter operation later than others, forcing a wait for the slowest rank to catch up. In a scaled-down experiment involving only two ranks per data parallel group, we measured the launch times for reduce-scatter calls and found them to not be consistently staggered but rather fluctuating reciprocally. Furthermore, the size of this time stagger increased as more steps were executed. Specifically, Rank A may initially lag behind Rank B but might eventually surpass Rank B in speed and by a growing margin. Ultimately, all ranks waited for the slowest rank. To trace back the root cause of this time skew, we located the variance to occur during the forward computation stage. Digging deeper into the code, we attributed this irregularity to fluctuations caused by some code segments. For instance, irregular garbage collection can introduce disturbances into the training procedure, and certain PyTorch operations can lead to performance fluctuations. These operations are on the critical path but can be affected along the training procedure. After modifying or removing those problematic code segments, we no longer observed a significant decline in MFU, as shown in Figure 12.

Frequent network interface flapping problem. We occasionally encounter training stall or training speed drop problem due to frequent network interface flapping. When the network interface flapping phenomena happens, the network interface goes down at first then goes up again. The interval between down and up time usually lasts for several seconds. During the down process, all the packets in transmission will be dropped. The first lesson we learn is the timeout threshold should be set explicitly to a larger value , otherwise the default value will make NCCL timeout very quickly and return a completion error before the network card up again. The second lesson we learn is that the root cause of this problem is the bad link quality between network card, AOC cable and switch. The flapping frequency can be reduced to a satisfactory level by doing lower level quality control over network card signal strength, AOC cable quality and switch side signal strength.

7 Related Work

LLM training. A lot of efforts have been put to the training of pre-trained LLMs, including proprietary ones such as GPT-3 [1], GPT-4 [23], GShard [24], PaLM [5], and many others [25–29], as well as open-source alternatives like OPT [30], BLOOM [31], Llama [32], Llama-2 [33]. Existing technical reports in the field predominantly focus on model performance comparisons, leaving out the specific details of the system infrastructure that makes such training possible. This paper fills this gap by sharing our experience of end-to-end LLM pre-training at the scale of over 10,000 GPUs from a systems perspective.

After pre-training, pre-trained base models can be further fine-tuned to adapt to downstream tasks better. This has led to the emergence of a range of dialogue models [34–37] exemplified by ChatGPT. However, it is worth noting that the computational power and data requirements for fine-tuning are substantially lower than those for pre-training. With the application of optimization techniques such as quantization [38–41]

and low-rank adaptation [42], fine-tuning can be efficiently accomplished with limited resources.

LLM optimizations. In addition to the techniques mentioned previously in the paper, there exists a lot of other works targeted at improving the efficiency of LLMs. Sparse or linear attentions [43–45] are proposed to make the memory consumption scales approximately linearly. Several studies aim to design new architectures rather than conventional transformer architectures to address the efficiency issue, such as RWKV [46] and RetNet [47]. Many recent studies have been devoted to developing communication acceleration techniques for LLMs. Some works reduce communication traffic using gradient compression [48] or mixed-precision training [49], while others schedule communication to overlap it with computation. Many popular ML frameworks, such as TensorFlow [50] and PyTorch [51], enable overlapping communication with backward propagation by default. Recent works [52–55] further overlap gradient synchronization with forward computation via tensor partitioning, at the cost of extra overhead. Some works [56, 57] introduce fixed staleness to the training pipeline for full overlapping communication and communication. However, the staleness may degrade the model performance.

Diagnosis tools in datacenters. Many diagnosis tools have been developed to identify and pinpoint hardware and software problems in datacenters. Pingmesh [58] is an active probing system based on end hosts. Network wide RTT and packet loss and measured by sending probing ping packets and doing data analysis. Network-wide SLAs are provided and network problems including packet-blackhole and packet silent drop are detected. EverFlow [59], LossRadar [60], NetBouncer [61] exploits the capability of switches to diagnose detailed network problems like network path failures or specific network port failures. NetBouncer leverages IP-in-IP tunnel techniques to do path probing. EverFlow requires mirroring network packets to a centralized server to do debugging. Hostping [62] is a diagnosis system based on end hosts that focuses on intra-host bottlenecks. It actively senses complex GPU server PCIe/NVLINK interconnects and does loopback bandwidth and latency tests.

Fault tolerance in large-scale distributed systems. Fault tolerance has been a major concern in large-scale distributed systems, where a wide range of hardware and software failures can occur. Many fault tolerance techniques have been proposed in the past that cater the needs of different systems and deployment scenarios. Reactive fault tolerance techniques are used to reduce the impact of failures on a system when the failures occur. There are many techniques in this category such as Retry [63], Replication [63], Checkpointing [64] and Message Logging [65]. These techniques incur some system overhead to recover from failures. Proactive fault tolerance techniques keep healthy components in place as backups of the faulty components, obviating the need of recovery from faults and errors, e.g., preemptive migration [66–68] and load balancing [69]. However, these approaches often assume that failures are predictable, while it is challenging for real large-scale distributed systems to predict the failures due to the complexity of the systems.

8 Conclusion

In this paper, we offer an in-depth look at the design, implementation and deployment of MegaScale, a production-grade system built to train LLMs at the scale of over 10,000 GPUs. MegaScale exploits algorithm-system co-design to optimize training efficiency. MegaScale achieves 55.2% MFU when training a 175B LLM model on 12,288 GPUs, a 1.34× improvement over Megatron-LM. We emphasize the need for fault tolerance throughout the training process and implement a tailored robust training framework to locate and fix faults automatically. We provide a comprehensive set of monitoring tools for deep observability into system components and events, facilitating root cause identification for intricate anomalies. We believe that our work not only offers practical insights for those working on LLM training, but also paves the way for future research in this rapidly evolving field.

Acknowledgments

We thank our shepherd Deepak Narayanan and the anonymous NSDI reviewers for their insightful and constructive feedback. We thank Zhenyuan Yang and Wenjia Zhu for their thorough feedback on earlier drafts of this manuscript. We thank Zuquan Song, Gaohong Liu and the broader ByteDance Applied Machine Learning team for their support throughout this project.

References

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[2] "Introducing chatgpt." https://openai.com/blog/chatgpt, 2022.

[3] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[4] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[5] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[7] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," 2021.

[8] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," 2020.

[9] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large batch optimization for deep learning: Training bert in 76 minutes," in *International Conference on Learning Representations*, 2020.

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

[11] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models." ArXiv, May 2020.

[12] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism ," in *NeurIPS*, 2019.

[13] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *ACM SOSP*, 2019.

[14] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model." https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[15] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, A. Desmaison, C. Balioglu, P. Damania, B. Nguyen, G. Chauhan, Y. Hao, A. Mathews, and S. Li, "Pytorch fsdp: Experiences on scaling fully sharded data parallel," 2023.

[16] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[17] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," 2020.

[18] I. Group, "Ieee 802.1 qbb - priority-based flow control." https://1.ieee802.org/dcb/802-1qbb/, 2009.

[19] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion Control for Large-scale RDMA Deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 523–536, 2015.

[20] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, *et al.*, "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter," in *SIGCOMM*, pp. 514–528, 2020.

[21] "Megatron-LM." https://github.com/NVIDIA/Megatron-LM/tree/main, 2021.

[22] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[23] OpenAI, "Gpt-4 technical report," 2023.

[24] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," 2020.

[25] A. Askell, Y. Bai, A. Chen, D. Drain, D. Ganguli, T. Henighan, A. Jones, N. Joseph, B. Mann, N. DasSarma, N. Elhage, Z. Hatfield-Dodds, D. Hernandez, J. Kernion, K. Ndousse, C. Olsson, D. Amodei, T. Brown, J. Clark, S. McCandlish, C. Olah, and J. Kaplan, "A general language assistant as a laboratory for alignment," 2021.

[26] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2022.

[27] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," 2022.

[28] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training compute-optimal large language models," 2022.

[29] H. Su, X. Zhou, H. Yu, X. Shen, Y. Chen, Z. Zhu, Y. Yu, and J. Zhou, "Welm: A well-read pre-trained language model for chinese," 2023.

[30] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. ChenG, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022.

[31] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," *arXiv preprint arXiv:2211.05100*, 2022.

[32] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[33] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[34] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford alpaca: An instruction-following llama model." https://github.com/tatsu-lab/stanford_alpaca, 2023.

[35] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, "Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality," 2023.

[36] X. Geng, A. Gudibande, H. Liu, E. Wallace, P. Abbeel, S. Levine, and D. Song, "Koala: A dialogue model for academic research." Blog post, April 2023.

[37] Y. Ji, Y. Deng, Y. Gong, Y. Peng, Q. Niu, B. Ma, and X. Li, "Belle: Be everyone's large language model engine." https://github.com/LianjiaTech/BELLE, 2023.

[38] Z. Li, E. Wallace, S. Shen, K. Lin, K. Keutzer, D. Klein, and J. Gonzalez, "Train big, then compress: Rethinking model size for efficient training and inference of transformers," in *International Conference on Machine Learning (ICML)*, 2020.

[39] G. Xiao, J. Lin, M. Seznec, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," *arXiv*, 2022.

[40] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv*, 2022.

[41] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm. int8 (): 8-bit matrix multiplication for transformers at scale," *arXiv*, 2022.

[42] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021.

[43] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," 2019.

[44] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," 2020.

[45] C. Zhu, W. Ping, C. Xiao, M. Shoeybi, T. Goldstein, A. Anandkumar, and B. Catanzaro, "Long-short transformer: Efficient transformers for language and vision," 2021.

[46] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho, H. Cao, X. Cheng, M. Chung, M. Grella, K. K. GV, X. He, H. Hou, P. Kazienko, J. Kocon, J. Kong, B. Koptyra, H. Lau, K. S. I. Mantri, F. Mom, A. Saito, X. Tang, B. Wang, J. S. Wind, S. Wozniak, R. Zhang, Z. Zhang, Q. Zhao, P. Zhou, J. Zhu, and R.-J. Zhu, "Rwkv: Reinventing rnns for the transformer era," 2023.

[47] Y. Sun, L. Dong, S. Huang, S. Ma, Y. Xia, J. Xue, J. Wang, and F. Wei, "Retentive network: A successor to transformer for large language models," 2023.

[48] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding," in *NeurIPS*, 2017.

[49] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed Precision Training," in *ICLR*, 2018.

[50] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[51] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library ," in *NeurIPS*, 2019.

[52] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based Parameter Propagation for Distributed DNN Training ," in *MLSys*, 2019.

[53] S. H. Hashemi, S. Abdu Jyothi, and R. Campbell, "TicTac: Accelerating Distributed Deep Learning with Communication Scheduling," in *MLSys*, 2019.

[54] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *SOSP*, 2019.

[55] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive All-reduce Scheduling for Expediting Distributed DNN Training," in *INFOCOM*, 2020.

[56] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing, "Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training," in *NeurIPS*, 2018.

[57] Y. Chen, C. Xie, M. Ma, J. Gu, Y. Peng, H. Lin, C. Wu, and Y. Zhu, "Sapipe: Staleness-aware pipeline for data parallel dnn training," *Advances in Neural Information Processing Systems*, vol. 35, pp. 17981–17993, 2022.

[58] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," *SIGCOMM Comput. Commun. Rev.*, vol. 45, p. 139–152, aug 2015.

[59] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," *SIGCOMM Comput. Commun. Rev.*, vol. 45, p. 479–491, aug 2015.

[60] Y. Li, R. Miao, C. Kim, and M. Yu, "Lossradar: Fast detection of lost packets in data center networks," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, (New York, NY, USA), p. 481–495, Association for Computing Machinery, 2016.

[61] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, "Netbouncer: Active device and link failure localization in data center networks," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, (USA), p. 599–613, USENIX Association, 2019.

[62] K. Liu, Z. Jiang, J. Zhang, H. Wei, X. Zhong, L. Tan, T. Pan, and T. Huang, "Hostping: Diagnosing intra-host network bottlenecks in RDMA servers," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, (Boston, MA), pp. 15–29, USENIX Association, April 2023.

[63] S. Haider, N. R. Ansari, M. Akbar, and M. R. Perwez, "Fault tolerance in distributed paradigms," 2011.

[64] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–14, 2018.

[65] A. S. Tanenbaum, *Distributed systems principles and paradigms*. 2007.

[66] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive fault tolerance in mpi applications via task migration," in *International Conference on High-Performance Computing*, pp. 485–496, Springer, 2006.

[67] S. Chakravorty, C. Mendes, and L. V. Kale, "Proactive fault tolerance in large systems," in *HPCRI Workshop in conjunction with HPCA*, vol. 2005, pp. 1–7, Citeseer, 2005.

[68] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo, "Elastic parameter server load distribution in deep learning clusters," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 507–521, 2020.

[69] I. Behera and C. R. Tripathy, "Performance modelling and analysis of mobile grid computing systems," *International Journal of Grid and Utility Computing*, vol. 5, no. 1, pp. 11–20, 2014.