MEGATRACE: Troubleshooting Hang and Slowdown in Large-scale LLM Training Clusters

Fangzheng Jiao[1], Menghao Zhang[1], Bolin Chen[4], Jiaxun Huang[1], Yanmin Jia[2], Xiaohe Hu[2], Bohua Xu[3], Bowen Han[3], Chunming Hu[1]
[1]School of Software, Beihang University [2]Infrawaves
[3]China Unicom Research Institute [4]Nanyang Technological University

***Abstract*—With the rapid scaling of large language model (LLMs) training clusters, troubleshooting performance issues presents significant challenges. Silent failures, e.g., training *hangs* and *slowdowns*, are particularly difficult to detect and locate in a timely and accurate manner. Existing solutions primarily focus on server-level localization but fail to pinpoint the specific training stage, limiting effective fault diagnosis. In this paper, we present MEGATRACE, a lightweight profiling framework tailored for diagnosing performance anomalies in production-scale LLM clusters. MEGATRACE includes an online profiling tool that instruments the narrow-waist NCCL layer to collect lightweight telemetry, capturing essential aspects of the training process with negligible overhead. MEGATRACE features an asynchronous analyzer that constructs a runtime graph based on the online collection results and training configurations, and subsequently generates a trace graph for the entire training job. By analyzing the critical path in the trace graph, MEGATRACE can accurately pinpoint the root causes of hangs and slowdowns, precisely identifying when and where the issue happens. We open source MEGATRACE, and evaluate it through extensive testbed experiments and real-world deployments. Evaluation results show that MEGATRACE achieves 100% precision in detecting hang issues, and improves the accuracy of identifying slowdown ranks by 29.44% compared to state-of-the-art solutions, with only a 0.16% performance overhead, which significantly accelerates the troubleshooting process for large-scale LLM training clusters.**

***Index Terms*—large language models, distributed training, fault localization, performance analysis**

I. INTRODUCTION

In recent years, large language models (LLMs) such as GPT-5 [1] and Grok-4 [2] have achieved remarkable success across a wide range of tasks, including text generation, machine translation, and autonomous driving. These models are typically trained over several months on large-scale computing clusters comprising tens to hundreds of thousands of GPUs. Such clusters typically employ NVIDIA GPUs interconnected via high-performance networking technologies like InfiniBand (IB) [3] or RoCEv2 [4] to support Remote Direct Memory Access (RDMA). In these large-scale distributed training systems, the failure of any individual component can degrade performance, halt an ongoing task, or even disrupt the entire training process. However, the failure rate scales with the system size, as evidenced by fault statistics reported during the training of models such as Llama 3.1 [5] and OPT [6].

When failures occur, the primary remediation step involves identifying the faulty node and removing it from the resource pool. However, driven by the imperative to maximize resource utilization, recovery strategies must extend beyond merely replacing faulty nodes and restarting tasks. Effective recovery requires enhanced observability to pinpoint the exact training stage where the failure occurred, thereby enabling differentiated and more efficient recovery strategies. Therefore, comprehensive monitoring and fine-grained tracing of the cluster infrastructure have become essential.

As an end-to-end LLM infrastructure provider, Infrawaves has sustained large-scale deployments across multiple data centers, with a cumulative delivery of tens of thousands of GPUs over the past three years. Based on our investigations, cluster issues can be broadly categorized into three types: *hang*, *slowdown*, and *fail-stop*. Fail-stop issues, caused by distinct software or hardware faults (e.g., GPU memory errors [7], ECC errors, or driver bugs), are generally easier to isolate. These anomalies typically trigger specific framework-level or system-level error logs, facilitating the quick identification of the affected host or GPU. In contrast, relying solely on system logs is ineffective for diagnosing non-fail-stop anomalies (i.e., hangs and slowdowns), as these silent failures often leave no explicit signatures in logs. When such issues arise, practitioners typically resort to manual analysis tools or raw trace inspection. Unfortunately, this approach is time-consuming even with automated scripts and necessitates stopping the training task. Furthermore, conventional profiling tools like PyTorch Profiler [8] and Nsight [9] incur high runtime overhead, rendering them impractical for continuous, always-on monitoring in production environments.

Recent studies have explored failure localization in depth. JIT [10] proposes a watchdog-based approach to identify hanging, and Minder [11] detects anomalies by analyzing system metrics and training feature models. Other works [12]–[14] exploit the mathematical characteristics of NCCL calls to detect and localize anomalies. MegaScale [15] and its follow-up study [16] use intrusive statistics from CUDA events and simulation to analyze expected execution times for slowdown detection. While these approaches actively collect signals to exploit training regularity, they primarily answer *when* a failure occurs and *which* server experiences the anomaly. They often overlook the *training stage* (i.e., the specific phase of computation or communication) at which the task fails, a context that is critical for optimizing subsequent recovery.

In this paper, we present MEGATRACE, a system designed to accurately troubleshoot performance anomalies in LLM

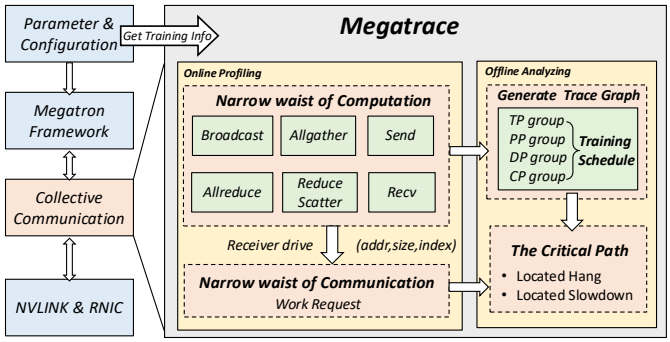


Fig. 1: Overview of MEGATRACE system.

training by analyzing computation and communication characteristics. MEGATRACE targets three key design goals. First, MEGATRACE must accurately and promptly identify the faulty rank and the specific execution stage where the issue arises. Second, to address runtime anomalies, MEGATRACE must operate online, capturing critical context the moment an anomaly occurs. Finally, MEGATRACE must be lightweight, introducing negligible overhead during training process.

As shown in Figure 1, MEGATRACE monitors the training process by dividing it into computation and communication components. Collective communication serves as the *narrow waist* of parallel training, directly reflecting the status of both components. In particular, MEGATRACE intercepts the NCCL communication APIs (e.g., `Broadcast`, `AllReduce`, `AllGather`, etc.) to capture the computation status, and monitors Work Request (WR) during data transmission to evaluate communication rate and status. Leveraging time-based tracing principles, MEGATRACE constructs a dependency graph for the training flow, identifies critical paths [17], and detects performance degradation along these paths to achieve precise localization.

We have implemented a prototype system, deployed it at the Shanghai Innovation Institute, and open-sourced the code [18]. Extensive experiments on real testbeds demonstrate that MEGATRACE achieves 100% precision in detecting hang issues with sub-second granularity. For slowdown identification, MEGATRACE improves accuracy by 29.44% compared to state-of-the-art solutions, covering 100% of the detection scenarios. Moreover, MEGATRACE provides interpretable output conclusions, effectively answering the critical *when*, *where* and *which stage* questions. The profiling overhead is merely 0.16%, ensuring negligible impact on the training tasks. We have also deployed MEGATRACE across two production clusters from Infrawaves, where it has processed tens of thousands of runs, enabling real-time detection and aiding in the diagnosis of complex performance issues.

II. BACKGROUND AND MOTIVATION

A. Distributed LLM Training

LLMs are characterized by their massive model parameter scales and vast datasets, typically requiring thousands of GPUs working in concert to complete pretraining or fine-tuning tasks. To support efficient LLM training, mainstream frameworks such as Megatron-LM [19] and DeepSpeed [20] commonly employ various hybrid parallel strategies to coordinate GPU resources.

Data Parallelism (DP) distributes training data across GPUs, each maintaining a complete model replica and synchronizes gradients via `AllReduce`. While DP is simple to implement, it incurs high communication overhead and is constrained by GPU memory for model replicas. *Tensor Parallelism (TP)* splits tensors across GPUs, reducing memory usage and improving operation efficiency. However, it involves significant communication overhead exchanging partial results, and is usually set within a GPU server to fully utilize the high-speed NVLink bandwidth. *Pipeline Parallelism (PP)* divides models into sequential stages assigned to different GPUs, enabling training of larger models through carefully balanced workload partitioning across stages. However, pipeline bubbles can degrade efficiency, requiring careful computation-communication overlap to maximize hardware utilization.

By combining these parallelism strategies, training workloads are efficiently distributed across GPU clusters. Tokens are grouped into batches and processed in parallel, allowing LLMs to effectively learn from massive datasets. This hybrid approach is essential for achieving scalability and computational efficiency in modern LLM training.

B. Types of Failures in LLM Training

Production-scale LLM training is susceptible to diverse failures stemming from infrastructure and software stacks [21]. Since static misconfigurations are typically resolved during initialization, prolonged training sessions primarily suffer from runtime anomalies. We categorize these anomalies into three modes: *fail-stop*, *slowdown*, and *hang*.

Fail-stop errors (e.g., GPU ECC errors, XID issues [22]) are relatively straightforward to handle, as they trigger explicit exceptions that terminate the process, allowing for immediate identification of the faulty node via stack traces. In contrast, *slowdowns* and *hangs* are insidious: they degrade performance or freeze progress without generating explicit framework logs, making fault localization notoriously difficult.

To maximize cluster efficiency, the operational goal is to minimize Mean Time To Recovery (MTTR) by rapidly isolating the faulty node for replacement, deferring deep-dive analysis to asynchronous diagnosis. An incident involving 4,096 GPUs from our production cluster illustrates this challenge: training throughput dropped from 404 to 330 TFLOPs due to a transient *software slowdown* on a single GPU. Lacking precise localization tools, operators have to spend several hours on manual bisection to isolate the culprit.

C. Existing Work Falls Short

To identify such anomalies, existing solutions have incorporated certain relevant designs, yet they continue to suffer from limitations across several critical dimensions.

Inability to run continuously over long periods. General traditional tools, such as Nsight [9] or Torch Profiling [8], have relatively high overhead and are unsuitable for prolonged, continuous use during training. These tools often involve

tracking and logging many low-level details about GPU operations, memory usage, kernel launches, data transfers, and other system-level information, which introduces high overhead to gather and store performance data. Besides, non-negligible resources like memory, processing power, and bandwidth are used to capture performance metrics, which results in less available memory and bandwidth for the model's operations, increasing the overall training time. In industrial settings, these tools are typically employed during debugging or only in the first few iterations of training to monitor whether the initial steps of the process are proceeding correctly. They are useful for assessing the start-up conditions of a training session but are not designed for long-term, ongoing monitoring.

Timeliness and accuracy in determining *when*, *which server* and *which stage*. For hang detection, JIT [10] employs a watchdog to monitor the completion status of the `cudaStreamWaitEvent` event list, and reports a hang if an event on a node fails to complete within the expected time. However, this method cannot directly identify the exact location or phase of the hang, providing only coarse-grained detection results. MegaScale [15], [16] introduces additional program instrumentation into the training framework to track execution time and visualize performance bottlenecks using a heatmap, thereby identifying slower ranks. Although this method can reveal performance disparities, it cannot accurately pinpoint the problematic rank. The follow-up work, What-if [16], demonstrates that this approach can detect general performance differences and, through manual in-depth analysis of the training framework, uncover some common critical issues. However, as an automated tool for problem detection and localization, the What-if analysis and simulation-based adjustment methods cannot timely identify the faulty node or the specific training stage at which the issue occurs. Greyhound [14], Aegis [12], and Holmes [13] recognize the "boundary" role of collective communication libraries in training log collection, and make simple judgments of training failures and slowdowns by counting the number and frequency of API calls. However, Greyhound merely uses these features for anomaly detection, with verification still relying on unit tests, which limits its timeliness. Aegis and Holmes employ mathematical characteristics and search-based methods to identify abnormal nodes, but remain weak in analyzing failure information at the training stage.

In summary, there is an urgent need for a tool that can continuously run alongside large-scale training without causing any performance degradation, while accurately and timely detecting and pinpointing the location and phase of hangs and slowdowns. This tool should be able to operate seamlessly over the entire training process, providing comprehensive monitoring and diagnosis for the training system.

D. Challenges and Observations

While achieving lightweight, effective, and precise problem localization presents certain difficulties, a thorough analysis of LLM training process reveals numerous characteristics that provide valuable insights to resolve these challenges.

Challenge 1: Low overhead and high effectiveness. Existing hardware logs and training logs are insufficient for monitoring and analyzing training performance. To effectively identify the root causes of slowdowns and hangs, it is necessary to insert program instrumentation at appropriate locations within the training framework, thereby creating an effective profiling tool to collect key performance metrics during training. Because hangs and slowdowns often occur randomly during training, the profiling method must run online alongside training to capture abnormal states while remaining lightweight and imposing minimal overhead.

Key Insight 1. In each LLM training iteration, each rank follows the predefined schedule and repeats the execution accordingly, making the timing for each batch's forward or backward pass deterministic and periodic. Therefore, based on the training configuration, we can derive a computational graph for the entire training process. This schedule, combined with key points from each iteration, enables rapid identification of the precise location where issues arise.

Furthermore, LLM training typically employs collective communication libraries like NCCL as the backend for communication. Collective communication represents the *narrow waist* of computation, as each successful invocation of a collective communication operation indicates the completion of a computational phase. By monitoring the calls to NCCL APIs, we can obtain both the finished time and the status of the computation. Similarly, communication transmission also exhibits a *narrow waist*: in the transmission, each data dispatch triggers an increment in the associated RDMA verbs counter, and the counter decreases once the transmission task is complete. By analyzing the changes in these counters, we can assess the health of the communication process. Through the identification of the two *narrow waist* points mentioned above, we can minimize overhead while effectively pinpointing issues in both computation and communication.

This strategy optimally balances visibility and overhead. Existing framework-level methods (e.g., cudaEvent) often miss physical network behaviors, whereas kernel-level profiling (e.g., CUPTI [23]) suffers from prohibitive data volume. In contrast, the collective communication layer yields the highest information-to-overhead ratio. By profiling this logical *narrow waist*, we capture both computation boundaries and physical transmission status, enabling precise, low-cost diagnosis.

Challenge 2: Rapid and accurate node and stage identification under complex dependencies. Low overhead and effectiveness ensure that the tool can collect relevant information online, but accurately determining the affected node and the execution stage under complex inter-dependencies between distributed computation and communication remains a major challenge. For instance, a slowdown in communication may cause the computation of one rank to lag behind other ranks within the same group, subsequently delaying the entire group's progress. This delay can propagate to dependent nodes, creating a cascading effect across the cluster. Consequently, rapid and precise identification of the root cause in such scenarios is extremely difficult.

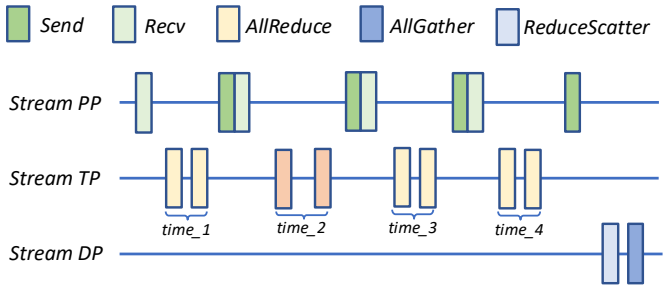


Fig. 2: NCCL API calls with Megatron-LM [19] in one iteration. Different APIs are called at different times on different streams.

Key Insight 2. Precise localization of faulty nodes and stages requires decoupling the intricate dependencies between distributed ranks. For communication, analyzing the transmission rate allows us to determine whether the issue originates from data transfer and to pinpoint the specific communication primitive where the problem occurs. For computation, we observe that different parallel groups perform communication API calls on different streams as shown in Figure 2, each with its own unique characteristics and timing. For example, the TP group typically employs `AllReduce`, `AllGather` and `ReduceScatter` communication, while the PP group uses paired `Send` and `Recv` calls. DP communication occurs at the end of each iteration, using `AllGather`&`ReduceScatter` (with distributed optimizer). By leveraging these characteristics, we can map each stage of the training process to the constructed trace tree, enabling fine-grained localization of the problematic stage.

In summary, by utilizing the inherent regularities and characteristics of training, we can decouple computation and communication. This allows for more efficient problem analysis and timely localization of issues within the training process.

III. MEGATRACE DESIGN

A. MEGATRACE *Overview*

MEGATRACE is a lightweight fault detection and precise localization system for LLM training, which operates during routine LLM model training, enabling real-time monitoring and analysis of training performance. It is designed to promptly detect performance anomalies, such as slowdowns or training hangs, and to identify the nodes and execution stages responsible for these issues. MEGATRACE requires no modifications to the training framework and introduces only negligible analysis and monitoring overhead.

Figure 3 illustrates the framework of MEGATRACE. It consists of two primary components: *online profiling* and *asynchronous analysis*. The online profiling module captures performance telemetry in real-time during training. It is further divided into two submodules: API interception and WR measurement. The API interception module captures and logs API calls, while the WR measurement module tracks the issuing and completion counts of WR operations. The asynchronous analysis module uses a critical path [17] analysis approach to construct a trace tree from the collected training data. This enables a detailed performance analysis of each iteration.

At initialization, MEGATRACE reads the training parameters for the entire model, including model architecture, partitioning details, and optimization settings relevant to training performance. These parameters are used to build the training pattern. During execution, the profiling modules continuously gather information on API calls and data transmission. Concurrently, the asynchronous analysis module consumes these telemetry logs on-the-fly to trace the performance of computation and communication, identifying critical paths and anomalies. This design enables both lightweight operation and precise fault localization, offering a viable solution to troubleshoot large-scale LLM training clusters.

B. *Online profiling*

1) ***API interception****:* LLM training deterministically repeats the computation and communication processes in the same pattern. Each rank executes the forward and backward computations for all batches according to the 1-Forward-1-Backward (1F1B) rule. Once the matrix computation of a rank is completed, communication operations are required at different stages. Based on the characteristics of the communication operations, we classify them into two types.

Synchronous communication. Figure 2 shows the TP stream where `AllReduce` is the main communication method. After the ranks within the same TP group complete their tensor matrix computation, they invoke the `AllReduce` operation for data reduction. If one rank computes slowly, the other ranks in the same TP group must wait for it to complete the `AllReduce` communication. Therefore, `AllReduce` not only serves as a collective communication primitive to synchronize data across TP groups, but also acts as a barrier. By comparing the start times of the `AllReduce` calls, we can observe the completion status of the computations across different ranks. Similarly, `AllGather` and `ReduceScatter` exhibit similar characteristics.

Pipeline communication. Figure 2 illustrates the `Send` and `Recv` operations within the PP stream. Except for the first and last PPs, each batch's computation requires a `recv` operation to receive data from the previous rank before execution, followed by a `send` operation to transmit the results to the next rank after computation. If any rank fails to invoke the `send` or `recv` operation due to an anomaly, it will inevitably affect the execution of subsequent processes.

Consequently, the timing of communication calls serves as a robust indicator for the computational state. In Megatron-LM [19], there are only six types of collective communication calls, as shown in Figure 1, making collective communication the narrow waist of LLM training. By intercepting and analyzing the timing of these collective communication calls, we can log the type and time of each call and infer the execution state of the computations. By processing the log output with an additional CPU thread, we can achieve near-zero overhead for API interception.

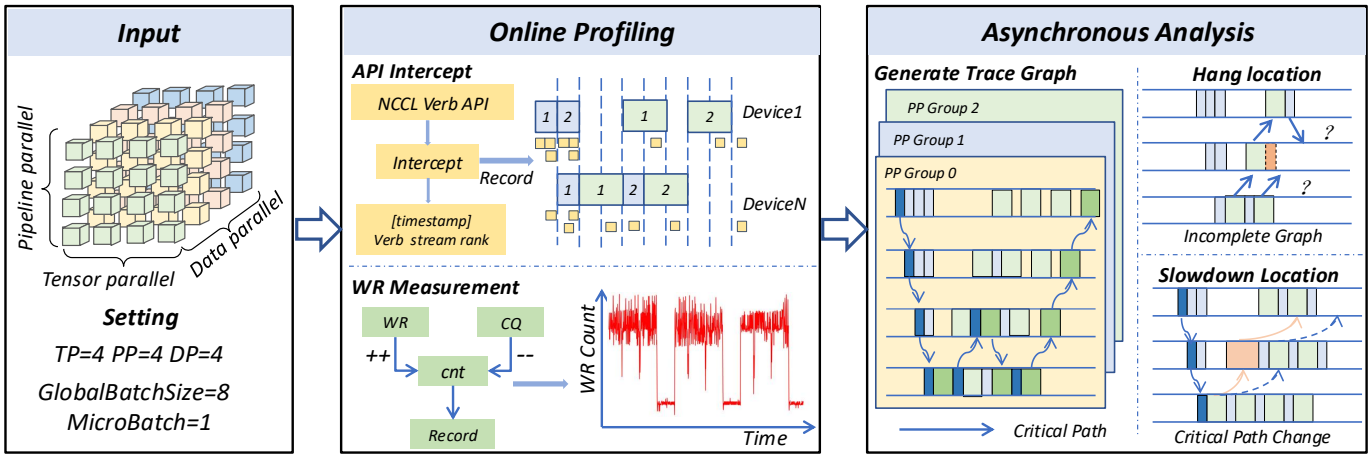


Fig. 3: MEGATRACE Framework.

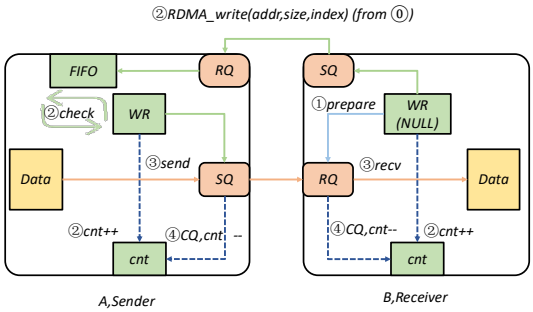


Fig. 4: A typical data transfer process in NCCL.

*2) **WR measurement**:* RDMA serves as the backbone of collective communication. Regardless of the upper-level topology (e.g., Ring, Tree) or the API (e.g., `AllReduce`), all data transmissions converge to RDMA Work Requests (WRs). Thus, WRs constitute the *narrow waist* of communication, offering a unified vantage point for profiling. To this end, we implement lightweight telemetry directly within the RDMA verbs interface. By tracking the WR lifecycle, we estimate network saturation and throughput with microsecond granularity via the following two mechanisms.

In-flight WR tracking. To monitor RNIC load, we maintain an atomic counter, $C_{inflight}$. We hook `ibv_post_send` to increment $C_{inflight}$ upon WR issuance, and `ibv_poll_cq` to decrement it upon completion, as shown in Figure 4. A sustained high $C_{inflight}$ with zero throughput signals a network hang, while heavy fluctuations suggest congestion.

Sliding window throughput estimation. Since posted WRs may queue in the RNIC, simple event counting fails to capture effective bandwidth. We employ a sliding window (default $w = 64$) to record cumulative transmitted bytes S and timestamps t of completed WRs. The instantaneous throughput BW_{est} is derived as the gradient over the window:

$$BW_{est} = \frac{S_i - S_{i-w}}{t_i - t_{i-w}} \tag{1}$$

This approach smoothes out jitter inherent in the data transmission process, yielding high-precision measurement with negligible CPU overhead.

C. Asynchronous Analysis

Trace model. Given the telemetry from online profiling, our goal is to precisely pinpoint root causes. For communication, WR measurements directly reveal bandwidth saturation, allowing straightforward verification against performance targets. In contrast, diagnosing computation is more intricate due to complex inter-rank dependencies.

PP schedule orchestrates the continuous execution flow of the iteration, whereas DP acts as a synchronization barrier at the iteration boundary. Consequently, we construct the global execution skeleton primarily based on the PP schedule. Crucially, pipeline bubbles introduce *slack time* that masks non-critical delays, as shown in Figure 6. Thus, relying solely on execution time yields false positives.

MEGATRACE deterministically reconstructs the expected Directed Acyclic Graph (DAG) of dependencies for each iteration, which represents the forward and backward computation dependencies for each batch. We formulate anomaly detection as a *critical path problem* on this DAG. In $G(V, E)$, vertices V represent Forward/Backward computation blocks, and edges E denote dependencies. The duration T_i of each batch can be approximately determined by the interval between the first call to `AllReduce` (except the first layer's computation time in this method), and the last call to `AllReduce`. We apply the critical path algorithm to identify the sequence of tasks dictating end-to-end latency, thereby filtering out noise from non-critical bubbles.

Execution times of nodes on the critical path are verified against historical expectations. Specifically, a running average is maintained, excluding initial iterations to filter out warm-up jitter. The average accumulates updates when deviations are within α (e.g., 5%), while deviations exceeding β (e.g., 50%) flag the node as anomalous.

Algorithm scalability optimization. To enable rapid identification of problematic graph nodes and allow timely analysis in large-scale training clusters, we design the following measures to reduce overhead.

First, each node performs preprocessing on the collected data by marking its own rank's API calls according to the

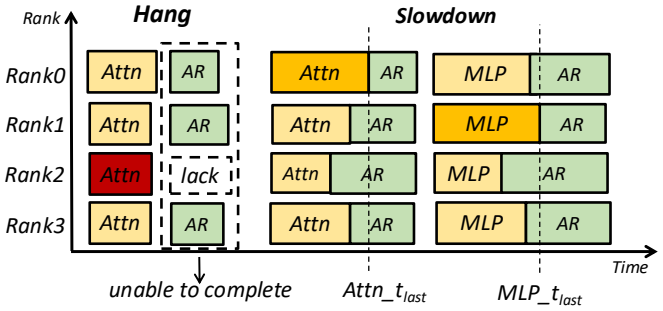


Fig. 5: Hang and slowdown detection in TP group.

Algorithm 1: Hang Detection Algorithm

```
// Worker Thread
Retrieve data: D_i from online profiling;
Reset timer T_i;
if T_i > n · T_batch then
    Notify management thread;
end
// Management Thread
if WR Measurement reports anomaly: A_net then
    Generate graph: G(V, E), where V is the set of nodes,
      E is the set of edges;
end
for each group g ∈ G do
    r_min = arg min_r (completed_batches(r));
    Determine dependencies: D_r ← dependency(r);
    Identify hang: H = arg min_{r∈g} (time_diff(r));
end
```

training schedule and categorizing them based on the corresponding communication calls of TP, PP, and DP groups. Each machine independently monitors its network bandwidth and connectivity, reporting any anomalies to the master process on the CPU server for further analysis. This distributed preprocessing significantly reduces the log processing overhead on the master process.

Second, we leverage the synchronization characteristics of DP to prune the analysis scope. Since DP requires global gradient synchronization, the end-to-end iteration time is dictated by the slowest model replica (i.e., the straggler PP group). Therefore, we only need to construct the trace graph for the specific PP group that exhibits the longest computation time, rather than for the entire cluster. In large-scale training, scaling mainly relies on expanding DP size. For example, in the LLaMA 405B model training, while the PP dimension might only involve 16 ranks, there can be 128 DP groups. By focusing on the single straggler PP group that exceeds the slowdown threshold, we reduce the graph generation complexity by orders of magnitude.

Locate hang issue. The diagnosis process consists of three steps: detection, validation, and localization (Algorithm 1).

1) Detection via Timeout. The asynchronous analyzer monitors the *heartbeat* of the training process. It maintains an expected execution time ($T_{expected}$) for each batch. If no telemetry data is received for a duration $t > n \times T_{expected}$ (e.g., $n = 2$), the system flags a potential hang and triggers the diagnosis workflow.

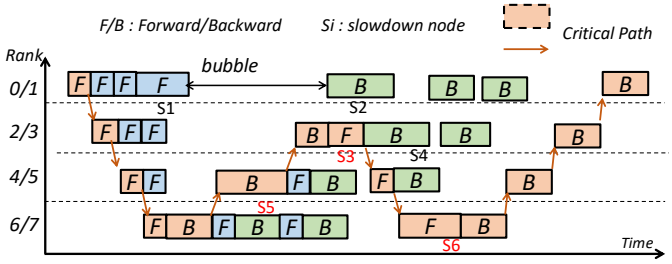


Fig. 6: A typical critical path in TP=2 PP=4.

2) Validation via Active Probing. To distinguish network faults from other issues, we trigger *active probing* by issuing zero-payload WRs. In a healthy network, these probes complete quickly. If the software stack hangs, no new WRs are posted. Conversely, if the physical network is disconnected, posted WRs accumulate in the NIC buffer but fail to complete. Therefore, if the WR count accumulates monotonically without decreasing, it confirms a physical network fault. Any faulty paths identified are reported to the manager.

3) Localization of Rank and Stage. Upon a hang, the manager constructs a *partial graph* based on the available logs.We traverse the graph to identify the *halt point*, the first node with missing completion signals. This node directly reveals the *faulty stage*:

- **Network Stage:** If the halt point aligns with a reported network fault, the issue is pinpointed to the specific communication primitive (e.g., `AllReduce`) and the physical link.
- **Computation Stage:** If no network fault is reported, missing API calls indicate that the preceding computation kernel (e.g., `MatMul` in the backward pass) failed to complete, identifying both the faulty rank and the specific execution stage.

Locate slowdown issue. Slowdown detection (Algorithm 2) is triggered upon log consolidation for each iteration. Concurrently, WR telemetry acts as a pre-check: a bandwidth drop exceeding threshold m (e.g., 20%) immediately flags a potential network bottleneck, prioritizing network diagnosis.

In the absence of network saturation, the analyzer targets the *straggler* PP group (i.e., the one determining iteration latency). By overlaying the critical path on the trace graph, we identify anomalous nodes whose deviations exceed historical expectations, effectively filtering out delays masked by pipeline bubbles. Since each graph node maps to a specific training stage (e.g., a computation block or communication collective), this mapping enables precise stage localization.

We perform a *drill-down* analysis on the identified anomalous node:

- **Rank Localization:** By comparing timestamped synchronization calls within the collective group (TP or DP), the specific rank causing the barrier wait is pinpointed.
- **Stage Identification:** The metadata of the straggler node reveals the exact faulty stage (e.g., a degraded

Algorithm 2: Slow Node Detection Algorithm

```
// Worker Thread
Retrieve data from online profiling: D_i for iteration i;
Update expected batch time:;
if i = 2 then
    Set T_expect = D_2 // Second iteration time as
        init expected
end
else if i ≥ 3 then
    if |D_i − T_expect| < a · T_expect then
        Update T_expect = D_i;
    end
end
Notify manager thread after each iteration;
// Manager Thread
Wait until all workers notify;
if WR measurement reports low link performance:
  P_link < T_threshold then
    Network issue detected: Report slow link and end;
end
else
    for each pp group g ∈ G do
        g: T_g = Σ_{r∈g} T_r;
        if T_g < T_mean · (1 − m) then
            continue
        end
        g:G_trace(V, E);
        find critical path;
        for each node v ∈ Critical Path do
            if T_v > T_expect · (1 + m) then
                Identify slowest rank in TP_v:
                    r_max = arg max_{r∈TP_v} (AllReduce_time(r))
                Report slow node v and slow rank r_max;
            end
        end
    end
end
```

`AllReduce` kernel or a slow `FlashAttention` computation).

We visualize this filtering mechanism in Figure 6. While nodes S1 through S6 exhibit local degradation, critical path analysis reveals that delays in S1, S2, and S4 are *absorbed* by pipeline bubbles. Consequently, only S3, S5, and S6 contribute to the end-to-end slowdown. Further drilling into node S5 (a TP group), Figure 5 shows that identifying the straggler in the `AllReduce` operation isolates the root cause to a specific rank and confirms the slowdown occurred during the gradient synchronization stage.

IV. Implementation

We implemented the core system using C++. Our instrumentation methodology is designed to support mainstream NCCL versions currently in production, ensuring broad applicability across different cluster environments. The implementation comprises approximately 1,000 lines of code for API interception and WR measurement, and an additional 1,500 lines for the Trace Analyzer, which is responsible for log analysis, trace tree generation, and anomaly monitoring. All source code is publicly available [18].

TABLE I: Training Parameter for a typical LLM training task.

Description	Symbol	Parameter Value
Transformer Layers	L	48
Attention Heads	H	48
Sequence Length	S	2048
Hidden Dimension size	h	21504
Global Batch size per DP	B	8
micro-batch size	b	1
Tensor Parallel size	T	2
Pipeline Parallel size	P	4
Data Parallel size	D	4

Online profiling. By instrumenting standard NCCL communication primitives, we capture critical metadata upon invocation, including API type, timestamp, payload size, and stream ID. We minimize intrusiveness by embedding a lightweight asynchronous logging mechanism directly into the relevant interfaces. To monitor network throughput, we track invocations of `ib_send` and `ib_test` during WR submission and completion. Given the dynamic nature of these values, we employ a sliding window of size 50 to accurately estimate bandwidth usage. To accommodate the temporal regularity of LLM training while filtering out transient noise, we employ adaptive thresholds. Specifically, we configure a timeout guardband (defaulting to 500ms) and a relative degradation threshold (defaulting to 20%) to identify WR anomalies, triggering alerts only when these specific thresholds are breached.

Asynchronous analysis. To strictly limit overhead on the critical training path, we design the trace analysis module to operate asynchronously. We first load the Training framework configurations (see Table I) to initialize the trace tree.

A dedicated worker thread is assigned to each rank to process log data concurrently with the training. These workers aggregate TP, PP, and DP metrics for each rank. Upon completing data aggregation for an iteration or detecting a potential anomaly, the worker thread signals the Manager thread. The Manager thread performs a rapid backup of the data buffer to offload graph generation and logic verification to a separate processing thread. Regarding the detection logic, we instantiate the timeout threshold as $n = 100\%$ and the slowdown sensitivity as $m = 50\%$ relative to the expected iteration time. Since these thresholds are ratio-based rather than absolute values, the system naturally generalizes across different model sizes and parameter settings without requiring manual tuning for each new task.

V. Evaluation

In this section, we evaluate MegaTrace to answer the following questions:

- How effective is MegaTrace in troubleshooting hang and slowdown issues (§V-B)?
- What advantages does the design of MegaTrace have compared to other approaches (§V-C)?
- How does MegaTrace perform in real-world environments (§V-D)?

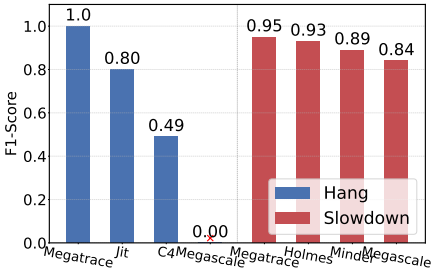


Fig. 7: F1-Score of different methods.

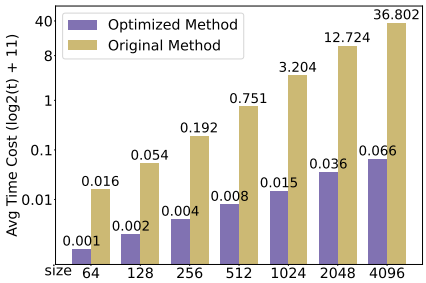


Fig. 8: Alg. overhead at scale.

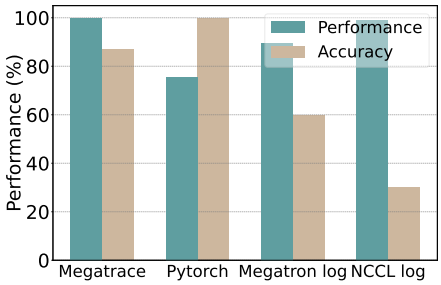


Fig. 9: Instrumentation overhead.

A. Experimental Setup

Our hardware testbed consists of 256 Supermicro GPU servers. Each server is equipped with 8 NVIDIA Hopper GPUs, 8 NVIDIA Mellanox ConnectX-7 SmartNICs (400GbE), 2 AMD EPYC 9575F 64-Core CPUs, and 2048GB of RAM, running Ubuntu 22.04 and GPU Driver 560.35.05. The servers are connected to a leaf-spine network using a multi-path setup. One H3C UniServer R4900 G5 CPU server is used to deploy MEGATRACE asynchronous analyzer. For the workload, we use Megatron-LM mcorev0.7 as the framework.

We emulate faults occurring at randomly chosen positions during training, including the emulation of hangs with infinite delays and slowdowns by adding additional computation burden. The emulated positions include three specific execution stages: *forward computation* (e.g., `Attention`, `MLP`), *backward computation*, and *data transmission* (e.g., `AllReduce`, `Send/Recv`) phases, ensuring that the evaluation spans the complex dependencies of the entire training graph. In our test model, a single forward computation takes approximately 40ms, so we inject random delays ranging from 50ms to 30s to simulate slowdowns.

We evaluate detection accuracy using standard binary classification metrics: Precision, Recall, and F1-Score. For the slowdown problem, we use the performance degradation of the training iterations as the ground truth to identify slowdowns. Recall = $\frac{TruePositive}{TruePositive+FalseNegative}$, Precision = $\frac{TruePositive}{TruePositive+FalsePositive}$, and Accuracy = $\frac{TruePositive+TrueNegative}{TruePositive+TrueNegative+FalsePositive+FalseNegative}$.

As discussed in §II, numerous studies have addressed fault localization for large-scale model training. For our evaluation, we focus on approaches [11]–[13], [15], [16] that have been deployed in real-world large-scale environments and provide a detailed comparison of their accuracy and timeliness. Among them, the What-if [14] presents a further analysis of the heatmap method in MegaScale. Since Aegis [12] and Holmes [13] employ similar methodologies with Holmes providing a more refined implementation, we did not explicitly compare these two works.

B. Overall Effectiveness

Hang and Slowdown detection. For hang localization, as shown in Figure 7, MEGATRACE attains an F1-score of 1.00, achieving complete detection of all hang occurrences. Since JIT and C4 are specifically tailored to computation and communication issues respectively, they exhibit limited recall while maintaining 100% precision within their targeted domains, yielding F1-scores of 0.80 and 0.49. Alternative methods such as MegaScale lack hang detection capability and consequently obtain an F1-score of zero. For slowdown localization, we employ a critical path algorithm for trace graph analysis. Owing to computational bubbles, only nodes located on the critical path influence training performance. Thus, accurate identification of slow nodes along this path is essential for high localization precision. As illustrated in Figure 7, MEGATRACE achieves an F1-score of 0.95 in slowdown detection, surpassing state-of-the-art baselines: Holmes(0.93), Minder(0.89), and MegaScale(0.84).

Crucially, the evaluation results demonstrate that MEGATRACE goes beyond merely isolating the faulty node. It concurrently and accurately outputs the specific execution stage of the anomaly. In all detected hang and slowdown cases reported above, we observe that MEGATRACE correctly pinpoints the exact phase where the failure occurred—distinguishing, for instance, between a stalled `AllReduce` and a delayed `MatMul` . This stage-level visibility enables precise fault attribution: communication failures point to network issues, while computation anomalies isolate GPU hardware or kernel faults. Unlike server-level baselines, this granularity provides immediate, actionable insights for diagnosis.

Analysis of responsiveness. In terms of responsiveness, MEGATRACE records expected execution times, enabling it to promptly identify which server has encountered a hang when the actual execution exceeds the anticipated duration. It subsequently correlates this information with training phase data to pinpoint the exact stage where the hang occurred. This entire process is completed within seconds. For communication-related issues, MEGATRACE leverages real-time network bandwidth monitoring to accurately localize network anomalies. Regarding computational slowdowns, analysis is triggered at the end of an iteration. MEGATRACE first identifies the slowest PP group, then locates the longest-running forward or backward operation within that group. Finally, it pinpoints the slowest node within the corresponding TP group. This hierarchical approach significantly narrows the analysis scope, allowing the localization of slow nodes within the time frame of a single iteration plus the analysis overhead. We benchmark the full analysis as a baseline method against the optimized approach that analyzes only the slowest groups, simulating

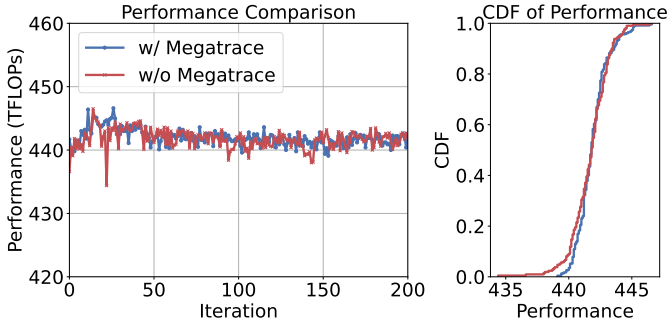


Fig. 10: Training performance comparison.

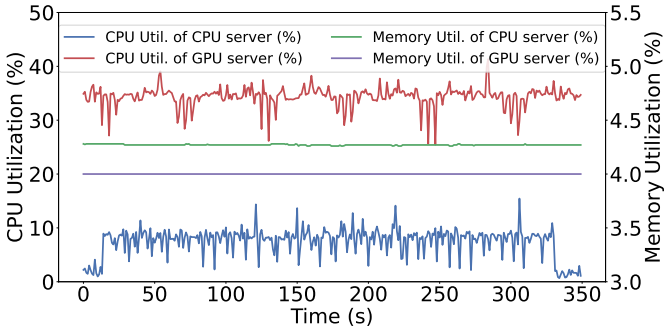


Fig. 11: CPU and memory usage.

overhead times from 64 ranks training (TP2&PP4) up to 4096 ranks training, with results shown in Figure 8. The results show a significant difference in processing time between the two methods, demonstrating that this optimization effectively reduces the time overhead of the analysis algorithm in large-scale training scenarios.

Online profile and asynchronous analysis overhead. To compare the training overhead in real-world large-scale scenarios, we conduct a comparison of system performance with and without MEGATRACE in a 2048 GPUs training task. As shown in Figure 10, we can observe that MEGATRACE imposes 0.16% overhead on the training process. The training performance with MEGATRACE enabled is similar when it is disabled, with almost identical performance distribution in Cumulative Distribution Function (CDF). After enabling MEGATRACE, the sustained iterative performance experiences only a marginal degradation of 0.16%. We also monitor CPU and memory usage during the training process. As shown in Figure 11, we can see that the CPU and memory usage on the GPU server remained very stable during training, with no significant spikes. MEGATRACE's analysis program is deployed on a CPU server. To evaluate the overhead of the analysis program, we also monitor the CPU and memory usage of this machine. As shown in Figure 11, we can see that the asynchronous module utilizes approximately 8% of the CPU resources and 4.4% of the memory resources.

C. Efficiency Explained

Profile efficiency. The design of the two *narrow waist* phases ensures that the collected data primarily reflects the most critical timing information of the system. As shown in Figure 9,

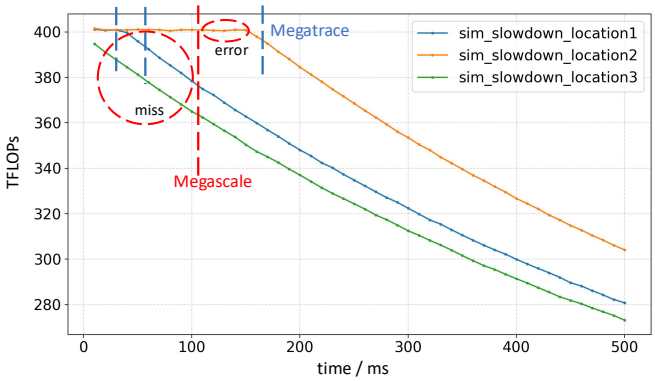
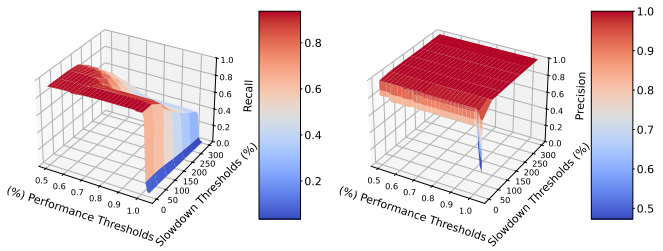


Fig. 12: Simulating slowdown time at different locations.

compared to traditional analysis tools such as PyTorch profiling, MEGATRACE reduces the granularity of kernel-level information, significantly lowering system overhead. Megatron-LM can enable data collection such as *--log-straggler* to identify slowdowns that consistently occur on a single machine. however, the overhead remains significant due to data collection and barrier synchronization. NCCL's logging feature has low overhead, but the information it provides is limited, requiring additional optimization and analysis methods to extract more useful insights. MEGATRACE leverages the two *narrow waist* of computation and communication, ensuring both the effectiveness and minimality of the collected data. MEGATRACE implements an asynchronous ring buffer to ensure that data collection does not affect training. To conclude, compared to these approaches, MEGATRACE achieves lower overhead and higher accuracy in log instrumentation.

Validity of the critical path. We mutate slowdowns of varying durations at different positions in the training process, with results shown in Figure 12. The figure illustrates that due to the existence of bubbles, the tolerance for slowdowns varies at different positions during training. At some positions (e.g., location 1), even if a slowdown occurs promptly, it does not affect the overall training performance. However, at other positions (e.g., location 2), any performance degradation immediately leads to a decline in training performance for that iteration. MEGATRACE employs critical path analysis to pinpoint nodes that effectively impact training throughput. Mapping latency to execution stages enables MEGATRACE to filter out "false alarms" absorbed by pipeline bubbles. In contrast to MegaScale's [15] reliance on fixed thresholds, which risks excessive misses or false positives, MEGATRACE dynamically adjusts its criteria based on critical path performance, ensuring accurate detection under varying conditions.

Parameter settings in MEGATRACE. To illustrate the relationship between the judgment threshold, performance degradation, and MEGATRACE 's effectiveness, we scan different threshold values and calculate the recall and precision for slowdown detection under each condition. Based on the results shown in Figure 13, a comprehensive analysis suggests that to improve both recall and precision, the criterion for identifying slowdown should first be set below 90% of the average performance to avoid triggering detections due to minor jitters,



(a) Recall. (b) Precision.

Fig. 13: Recall and Precision with different thresholds in slowdown detection.

which may arise from normal system fluctuations. Regarding the slowdown threshold, setting it too high results in fewer detections, negatively impacting recall, while setting it too low decreases precision and increases false positives. Therefore, the optimal performance for slowdown detection is achieved when the threshold is set within a time increase range of 50% to 100%. If the performance threshold is set above the average performance, the recall rate will drop sharply because the criterion for identifying slowdowns becomes too strict — minor jitters are then classified as slowdowns, even though such fluctuations may stem from normal system variability. Therefore, for effective detection, the performance threshold should be set below 90% of the average performance.

D. Case study

We have also deployed MEGATRACE across two real-world LLM training clusters from Infrawaves, and helped diagnose several complex performance problems.

Case 1: Cluster training hang issue. In November 2024, during a 3840-GPU training task, we encountered intermittent training hangs, with durations ranging from 30 minutes to 8 hours, followed by machine restarts. Through MEGATRACE's localization and analysis, we identified that the issue was caused by the absence of the corresponding `Recv` operation on GPU2 of one node, which occurred during the forward pass of the third batch. Upon isolating the faulty node and resuming training, the instability was eliminated.

We conducted several hours of single-node stress testing and detected an XID 109 error, which had not appeared during the 3840-GPU training. This confirmed that the node had a hardware fault, which was responsible for the issue. In large-scale training scenarios, conducting several hours of testing on all GPUs is impractical. However, by using MEGATRACE, we were able to quickly pinpoint the root cause of the hang.

Case 2: Insufficient CPU Allocation on Single Machine. Another issue occurred in a Docker testing environment. We had a 256-GPU training job that was under performing, with the iteration execution time being much longer than expected. From the results of using PyTorch profiling, the majority of the time overhead was attributed to communication. After deploying MEGATRACE to the environment, we found that the critical path of each iteration always passed through a particular GPU server, node24. Upon inspecting node24, we discovered that only 8 CPU cores were allocated when the Docker container was created, while other machines had 192 CPU cores. This resulted in poor training performance on node24, which was due to a deployment error.

VI. DISCUSSION

Analyzing multiple anomalies with MEGATRACE. The basic criterion for MEGATRACE to identify training anomalies is by analyzing the generated trace graph. It determines the location of the abnormal points through a comprehensive analysis of API call relationships and network status monitoring. For slowdown issues, MEGATRACE can identify the nodes that truly impact training performance using the critical path analysis approach. Therefore, it can detect multiple slow nodes that lie along the critical path. For hang issues, MEGATRACE determines the first problematic node by analyzing the residual graph during hang. Typically, when a hang issue occurs, the program stops working at one node, so multiple nodes simultaneously experiencing a hang is not common. If a slowdown is mixed with a hang issue, our method can still identify the hang and slowdown nodes, but the critical path based on the residual graph may lead to false positives. We leave the exploration of this issue as our future work.

Extensibility of MEGATRACE. While this paper presents experiments based on Megatron-LM [19] framework and RDMA networks, the design and methodology are applicable to other LLM training frameworks (e.g., DeepSpeed [20]) and traditional TCP networks. To adapt to these scenarios, the scheduling and communication strategies of the frameworks should be identified firstly, and then TCP communication can also be leveraged as a narrow waist of computation to analyze the system's operation. We leave the extension of MEGATRACE to other LLM training frameworks and TCP networks as our future work.

VII. RELATED WORK

Besides the most relevant works discussed in the main text, our work is also inspired by the following topics.

Early issue detection through testing. There are various cluster testing toolsets [24]–[27] to help identify potential issues within clusters. By testing different cases, hardware or software issues can be discovered and localized [24], [25], [27]. Superbench [26] adopts a more efficient testing method to analyze the cluster's performance under different workloads, thereby identifying problems in pre-launch cluster setups. However, these tools work before the training task is launched, and when issues arise during training, these tools usually cannot help pinpoint the cause of the anomalies.

LLM training fault tolerance. Many works [10], [28]–[33] focus on minimizing the time overhead associated with faults in LLM training. Bamboo [30], Oobleck [32], ReCycle [33] and Torch Elastic [28] provide fault tolerance for training across different dimensions, utilizing redundant computations and nodes. CheckFreq [34], Check-N-Run [35], Gemini [29] and DLRover [31] reduce checkpoint overhead by adjusting its frequency or overlapping it with computation, enabling more frequent and lower-overhead checkpointing. JIT [10], on the

other hand, uses backup data from other nodes to avoid the need for saving checkpoints, allowing for rollback to the previous batch state after a failure. Nevertheless, all of these efforts rely on timely and effective fault localization tools, which cannot be satisfied by current simple timeout-based detection mechanisms. By complementing these works with rapid and low-overhead fault localization tools like MEGATRACE, better fault detection and recovery can be achieved.

LLM training logging and instrumentation. MegaScale [15] enhances system visibility by instrumenting at the framework level. DLRover [31] uses XPU_timer for fine-grained information collection during training, capturing the entire call stack relationship to analyze training issues. Aegis [12] uses NCCL API logs to perform basic identification of failure and slowdown issues. C4 [36] enhances NCCL to monitor communication states at multiple layers. However, these solutions often face challenges such as excessive instrumentation, high performance overhead, or limited coverage, thus offering only partial solutions to the problems.

VIII. CONCLUSION

This paper designs MEGATRACE, a lightweight profiling system for LLM training. To address the issue of high overhead in traditional profiling, MEGATRACE leverages unique characteristics of the current training process, treating communication calls as the narrow waist for computation, and data transfer WRs as the narrow waist for communication. Given the complex dependencies in distributed LLM training, MEGATRACE uses a critical path algorithm based on the training schedule to identify the steps that truly affect training performance, effectively pinpointing the root causes of slowdowns and hangs. Extensive evaluation demonstrates that MEGATRACE is timely, effective, and low-cost. It can be easily deployed in existing training environments, assisting in troubleshooting and improving the average computational resource utilization in end-to-end model training.

ACKNOWLEDGMENTS

We thank anonymous ICDCS reviewers for their valuable comments. This work is supported in part by the National Key Research and Development Program (No. 2024YFB4505604), the National Natural Science Foundation of China (No. 62402025), and Huawei-BUAA Joint Lab. Menghao Zhang and Chunming Hu are the corresponding authors.

REFERENCES

[1] OpenAI, https://openai.com/zh-Hant/index/introducing-gpt-5/, 2025.
[2] xAI, https://x.ai/news/grok-4, 2025.
[3] NVIDIA, "InfiniBand," https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/infiniband, 2025.
[4] InfiniBand Trade Association, "InfiniBand Architecture Specification Release 1.4 Annex A17: RoCEv2," 2020.
[5] Meta, "Introducing Llama 3.1: Our most capable models to date," https://ai.meta.com/blog/meta-llama-3-1/, 2024.
[6] S. Zhang, S. Roller, N. Goyal, and et al., "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.
[7] NVIDIA, "Nvidia gpumemory error management," https://docs.nvidia.com/deploy/pdf/a100-gpu-mem-error-mgmt.pdf, 2022.
[8] PyTorch, "Pytorch profiler," https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html, 2026.
[9] NVIDIA, "Nvidia nsight systems," https://developer.nvidia.cn/nsight-systems, 2024.
[10] T. Gupta, S. Krishnan, R. Kumar, and et al., "Just-in-time checkpointing: Low cost error recovery from deep learning training failures," in *EuroSys*, 2024, pp. 1110–1125.
[11] Y. Deng, X. Shi, Z. Jiang, and et al., "Minder: Faulty machine detection for large-scale distributed model training," in *NSDI*, 2025, pp. 505–521.
[12] J. Dong, K. Qian, P. Zhang, and et al., "Evolution of aegis: Fault diagnosis for {AI} model training service in production," in *NSDI*, 2025, pp. 865–881.
[13] Z. Yao, P. Hu, C. Miao, and et al., "Holmes: Localizing irregularities in {LLM} training with mega-scale {GPU} clusters," in *NSDI*, 2025, pp. 523–540.
[14] T. Wu, W. Wang, Y. Yu, and et al., "Greyhound: Hunting fail-slows in hybrid-parallel training at scale," in *ATC*, 2025, pp. 731–747.
[15] Z. Jiang, H. Lin, Y. Zhong, and et al., "{MegaScale}: Scaling large language model training to more than 10,000 {GPUs}," in *NSDI*, 2024, pp. 745–760.
[16] J. Lin, Z. Jiang, Z. Song, and et al., "Understanding stragglers in large model training using what-if analysis," in *OSDI*, 2025.
[17] J. E. Kelley Jr and M. R. Walker, "Critical-path planning and scheduling," in *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, 1959, pp. 160–173.
[18] Megatrace, "open source of megatrace," https://github.com/sii-research/Megatrace.git, 2026.
[19] D. Narayanan, M. Shoeybi, J. Casper, and et al., "Efficient large-scale language model training on gpu clusters using megatron-lm," in *SC*, 2021, pp. 1–15.
[20] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *SIGKDD*, 2020, pp. 3505–3506.
[21] Q. Hu, Z. Ye, Z. Wang, and et al., "Characterization of large language model development in the datacenter," in *NSDI*, 2024, pp. 709–729.
[22] NVIDIA, "Xid error," https://docs.nvidia.com/deploy/xid-errors/index.html, 2024.
[23] ——, "Nvidia cuda profiling tools interface," https://developer.nvidia.com/cupti, 2026.
[24] Stanford, "Dawnbench," https://github.com/stanford-futuredata/dawn-bench-entries, 2018.
[25] NVIDIA, "Mlperf benchmarks," https://www.nvidia.com/en-us/data-center/resources/mlperf-benchmarks/, 2024.
[26] Y. Xiong, Y. Jiang, Z. Yang, and et al., "{SuperBench}: Improving cloud {AI} infrastructure reliability with proactive validation," in *ATC*, 2024, pp. 835–850.
[27] Alibaba, "Aicb," https://github.com/aliyun/aicb, 2024.
[28] PyTorch, "Torch distributed elastic," https://pytorch.org/docs/stable/distributed.elastic.html, 2024.
[29] Z. Wang, Z. Jia, S. Zheng, and et al., "Gemini: Fast failure recovery in distributed training with in-memory checkpoints," in *SOSP*, 2023, pp. 364–381.
[30] J. Thorpe, P. Zhao, J. Eyolfson, and et al., "Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}," in *NSDI*, 2023, pp. 497–513.
[31] Alibaba, "Dlrover," https://github.com/intelligent-machine-learning/dlrover, 2024.
[32] I. Jang, Z. Yang, Z. Zhang, and et al., "Oobleck: Resilient distributed training of large models using pipeline templates," in *SOSP*, 2023, pp. 382–395.
[33] S. Gandhi, M. Zhao, A. Skiadopoulos, and C. Kozyrakis, "Recycle: Resilient training of large dnns using pipeline adaptation," in *SOSP*, 2024, pp. 211–228.
[34] J. Mohan, A. Phanishayee, and V. Chidambaram, "{CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing," in *FAST*, 2021, pp. 203–216.
[35] A. Eisenman, K. K. Matam, S. Ingram, and et al., "{Check-N-Run}: A checkpointing system for training deep learning recommendation models," in *NSDI*, 2022, pp. 929–943.
[36] J. Dong, B. Luo, J. Zhang, and et al., "Boosting large-scale parallel training efficiency with c4: A communication-driven approach," *arXiv preprint arXiv:2406.04594*, 2024.