VENUSTRACE: Diagnosis and Localization of Anomalies in Distributed LLM Training

Fangzheng Jiao[1], Menghao Zhang[1], Bolin Chen[1], Jiaxun Huang[1], Yanmin Jia[2],

Xiaohe Hu[2,3], Chunming Hu[1], Bohua Xu[4], Bowen Han[3]

[1]*Beihang University* [2]*Infrawaves* [3]*Shanghai Innovation Institute* [4]*China Unicom Research Institute*

Abstract

With the rapid scaling of large language model (LLMs) training clusters, troubleshooting presents significant challenges. Silent failures make it particularly difficult to detect and locate training *hangs* and *slowdowns* in a timely and accurate manner. Existing solutions only consider server localization and lack integrated analysis at the model training stage, leaving considerable scope for improvement in fault diagnosis. In this paper, we introduce VENUSTRACE, a lightweight profiling system designed specifically for large-scale LLM training. VENUSTRACE includes an online profiling tool that collects narrow-waisted data from NCCL, capturing essential features of the training process with minimal overhead. VENUSTRACE features an offline 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, VENUSTRACE can accurately pinpoint the root causes of hangs and slowdowns, precisely identifying when and where the issue happens. We open-source VENUSTRACE, and evaluate it through extensive testbed experiments and real-world deployments. Evaluation results show that VENUSTRACE 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—significantly accelerating the troubleshooting process for large-scale LLM training clusters.

1 Introduction

In recent years, large language models (LLMs) such as GPT-5 [24] and Grok-4 [32] have achieved remarkable success in a wide range of tasks including text generation, machine translation, autonomous driving. These models are typically trained over several months on large-scale computing clusters comprising tens of thousands, or even hundreds of thousands, of GPUs. The hardware configuration in such clusters commonly consists of NVIDIA GPUs, which are interconnected using high-performance networking technologies like InfiniBand (IB) [23] or RoCEv2 [10] that support Remote Direct Memory Access (RDMA). In large-scale distributed training systems, the failure of any individual component can degrade training performance, halt the ongoing training task, or even disrupt the entire training process. Unfortunately, the failure rate increases with system scale, as evidenced by fault statistics reported during the training of large-scale models such as Llama 3.1 [16] and OPT [35]. When failures occur within a cluster, identifying the faulty node and removing it from the resource pool is the primary remediation step. With increasing demands for resource utilization, recovery should not be limited to replacing faulty nodes and restarting tasks. It must also incorporate enhanced observability to indicate the exact training stage at which the failure occurs, thereby enabling the application of differentiated recovery strategies. Therefore, comprehensive monitoring and tracing of the cluster infrastructure become essential.

Based on our careful investigations, cluster issues can be broadly categorized into three types: *hang*, *slowdown*, and *fail-stop*. Among these issues, fail-stop issues, which result from software/hardware failures, are typically easier to pinpoint, such as GPU memory errors [19], ECC errors, GPU driver bugs. These anomalies generally trigger specific framework-level and system-level error logs, allowing for quick identification of the affected host or GPU. However, relying on system logs cannot resolve issues such as training task hangs or slowdowns, since neither produces explicit framework-level or system-level error logs. Therefore, when such issues arise, practitioners in the field typically rely on analysis tools or trace information for diagnosis. However, the former approach typically requires a significant amount of time even with the aid of testing scripts, and the training task must be stopped. As for the profiling tools, such as PyTorch Profiler [25] and Nsight [21], they have a substantial impact on the training performance and are generally not enabled for extended periods during live training sessions.

Recent studies have also conducted detailed investigations into failure localization. JIT [8] proposes a watchdog-based

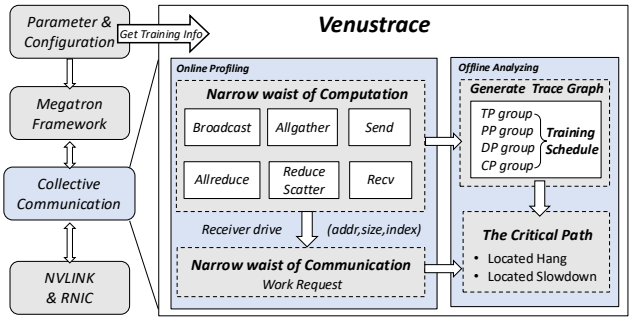


Figure 1: Overview of VENUSTRACE system.

approach to identify hanging issue, while Minder [3] identifies anomalies by analyzing system metrics and training feature models. Other work [5, 31, 34] exploits the mathematical characteristics of NCCL calls to detect and localize anomalies. MegaScale [12] and its follow-up study [14] collect intrusive statistics through CUDA events and employ simulators to analyze expected execution times, thereby identifying slowdowns within the system. These approaches actively collect informative signals from the system and exploit the regularity of training to discover and localize anomalies. However, while they provide answers regarding when a failure occurs and which server experience the anomaly, they do not analyze the training stage (which stage) at which a task fails—an aspect that is crucial for subsequent recovery strategies.

In this paper, we present VENUSTRACE, a system designed to accurately troubleshoot performance anomalies for LLM training based on the characteristics of computation and communication. VENUSTRACE aim to achieve three key design goals. First, VENUSTRACE should be capable of accurately and promptly identifying the machine and the specific stage at which the issue occurs. Second, to handle unexpected problems during training, VENUSTRACE must operate online, and capture critical information about the error state as soon as an issue arises. Finally, VENUSTRACE must be lightweight and have minimal impact on training, collecting information efficiently with zero impact on the training process.

As shown in Figure 1, the training process is divided into two parts for monitoring: computation and communication. Both of these parts can be reflected through collective communication, which acts as the *narrow waist* in parallel training. In particular, VENUSTRACE intercepts the NCCL communication APIs (e.g., `Broadcast`, `AllReduce`, `AllGather` and etc.) to capture the computation status, and counts Work Request (WR) involved in the data transmission to evaluate communication rate and status. Afterwards, leveraging the concept of time-based tracing in distributed systems, VENUSTRACE generates an offline trace graph for LLM training, identifies critical paths [13], and detects performance degradations along these paths, thus achieving precise issue localization. We implement a prototype of VENUSTRACE, and make the source code publicly available [15]. Extensive experiments on real testbeds show that VENUSTRACE achieves 100% precision in

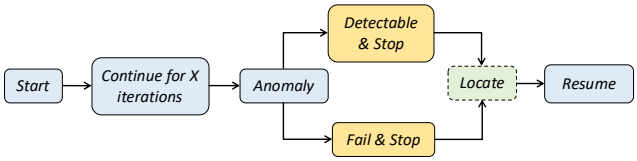


Figure 2: A typical training flowchart. System anomalies may lead to training failures and exits, or they may require specific detection methods to identify the anomalies and proactively trigger a restart.

detecting hang issues, providing sub-second detection granularity. In identifying slowdown nodes, VENUSTRACE improves the accuracy by 29.44% compared to state-of-the-art solutions, covering 100% of the detection scenarios. Moreover, VENUSTRACE offers interpretable output conclusions, effectively answering the critical *when* and *where* questions. The profiling overhead is only 0.16%, ensuring almost no impact on the training tasks. We have also deployed VENUSTRACE across two training clusters in our partner company, processing tens of thousands of runs to enable real-time detection and localization of issues, and helping diagnose several complex performance problems.

2 Background and Motivation

2.1 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 [18] and DeepSpeed [27] commonly employ various hybrid parallel strategies to coordinate GPU resources.

Data Parallelism (DP) distributes training data across GPUs, each maintaining a complete model copy 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. *Sequence Parallelism (SP)* divides input sequences across GPUs, enabling parallel processing of long sequences with lower memory requirement per device. However, it requires communications at sequence boundaries for state exchanging, and is less effective for shorter sequences. *Context*

Parallelism (CP) allocates computations across GPUs based on token contexts, enabling parallelization of computations such as attention. However, CP imposes high communication overhead for exchanging attention weights and hidden states, requiring dynamic load balancing for uneven context sizes.

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.

2.2 Types of Failures in LLM Training

As the scale of model training continues to increase, various issues may arise in real-world large-scale training scenarios. Shanghai AI Laboratory [9] reports that errors can stem from infrastructure, frameworks, or scripts. According to our production experience, once the training task successfully starts, it indicates that the framework software is functioning as expected and issues such as inefficient code or incorrect configurations have been resolved during the testing phase. As the training progresses, subsequent issues typically arise from two factors: errors in the underlying software (e.g., CUDA operation errors) or hardware failures (e.g., GPU memory issues). From this perspective, a training process that initially operates without issues can encounter three types of problems: *fail-stop*, *slowdown*, and *hang*.

The fail-stop issue refers to training termination due to hardware failure, which usually reveals clear hardware indicators, such as XID Error [22], ECC error, or optical module error. The slowdown and hang issues refer to situations where training performance degrades unexpectedly or the process freezes at a particular stage. Among these, fail-stop issues are relatively straightforward to detect and resolve, as the program typically terminates and throws an exception. For troubleshooting this issue, it is sufficient to analyze the error stack traces generated by the framework or the hardware to pinpoint the malfunctioning machine. However, slowdown and hang issues are more difficult to diagnose, as the framework does not generate explicit error messages, making it challenging to pinpoint the underlying location and stage.

As shown in Figure 2, the time consumed by failures mainly consists of detection and localization. To minimize the resource waste caused by each failure, it is crucial to reduce the detection and localization time for these failures. Therefore, the best approach is to quickly locate the faulty machine, replace it, and restart the training. Subsequent problem analysis should only be performed on the faulty machine offline. To illustrate these issues, we present a case from our production environment.

During a large-scale training task utilizing 4096 GPUs, we observed that performance dropped from 404 TFLOPs to 330 TFLOPs. To isolate the root cause, we conducted group-wise screening and reduced the sampling interval of monitoring metrics. After several hours of investigation, we identified that one specific GPU intermittently entered a brief "SW slowdown" state, during which its performance dropped abruptly. This state happened every few minutes and was very brief. After excluding the affected machine from the cluster, the performance drops ceased entirely. This case highlights the need for efficient tools that can log data immediately when anomalies occur, enabling rapid fault localization without extensive manual effort.

To conclude, large-scale cluster training requires rapid detection (*when*) and precise identification of the failure machine (*which server*) and stage (*which stage*), to facilitate subsequent component replacement and training recovery.

2.3 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. More general traditional tools, such as Nsight [21] or Torch Profiling [25], 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 [8] 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 [12, 14] 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 [14], 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 [31], Aegis [5], and Holmes [34] 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.

Summary. To conclude, 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.

2.4 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. By analyzing these metrics, the root cause of the issue can be precisely pinpointed. Moreover, as mentioned earlier, since hangs and slowdowns often occur randomly during training, the profiling method must be able to operate online alongside the training process in order to capture abnormal states as they occur. Therefore, in addition to being effective, the profiling method must be lightweight and introduce minimal overhead to the training process.

Observation for Challenge 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.

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.

Observation for Challenge 2. To promptly and accurately pinpoint the machine and stage at which the issue occurs, it is essential to untangle the complex interdependencies between each rank in the training process to trace the root cause. 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 3, each with its own unique characteristics and timing. For example, the TP group typically employs `AllReduce`, `AllGather` and `ReduceScatter` communication (with SP)), 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.

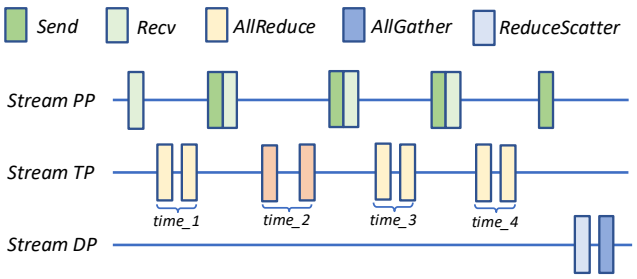


Figure 3: NCCL API calls with Megatron-LM [18] in one iteration. Different APIs are called at different times on different streams.

3 VENUSTRACE Design

3.1 VENUSTRACE Overview

VENUSTRACE 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. As a lightweight tool for large-scale LLM training, VENUSTRACE requires no modifications to the training framework and facilitates nearly zero-overhead performance analysis and monitoring.

Figure 4 illustrates the framework of VENUSTRACE. VENUSTRACE consists of two primary components: *online profiling* and *offline analysis*. The core functionality of the online profiling module is to collect performance-related information during training. It is further divided into two submodules: API interception and WR measurement. The API interception module captures and logs API call information, while the WR measurement module tracks the issuing and completion counts of WR operations. The offline analysis module uses a critical path [13] analysis approach to construct a trace tree from the collected training data. This enables a detailed performance analysis of each iteration.

Initially, VENUSTRACE 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. The two profiling modules then gather information on API calls and data transmission during training. Finally, the offline analysis traces 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.

3.2 Online profiling

3.2.1 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 6 shows the TP process 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 7 shows the execution schedule of the forward and backward passes of a training batch from the perspective of PP. 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.

From the analysis above, we can see that the timing of communication calls is a good indicator of the computational state. In Megatron-LM [18], 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.

3.2.2 WR measurement

RDMA is widely adopted in current collective communication libraries. In the context of RDMA data transfer, WR refers to a Work Request. Regardless of the type of collective communication API, data transmission relies on RDMA's WR to complete the transfer. Therefore, WR also serves as a key narrow waist in the communication. A typical data transfer process is outlined in Figure 5.

⓪: During the initialization and chain-building process,

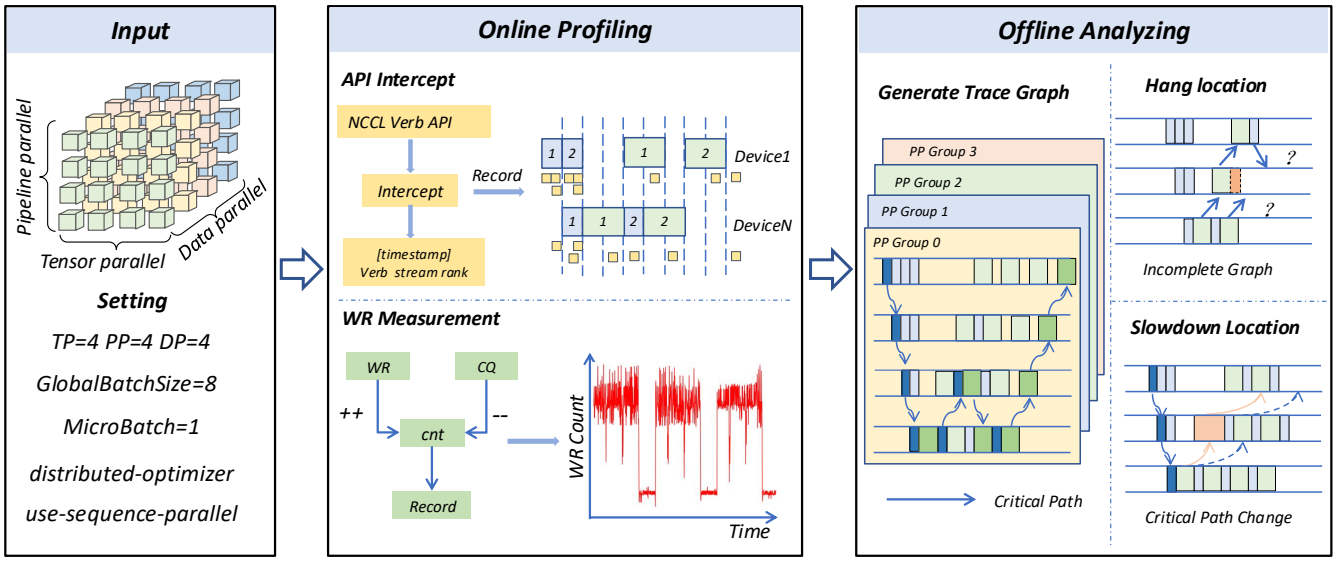


Figure 4: VENUSTRACE Framework.

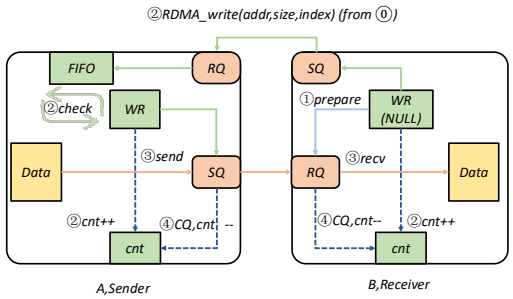


Figure 5: A typical data transfer process in NCCL.

NCCL exchanges the GID, QPN, and FIFO address information between both parties.

①: The data transfer process is initiated by the receiver. When a piece of data needs to be received, the receiver issues an empty WR and prepares the buffer to receive the data.

②B: The receiver writes the address of the receive buffer, the size of the data to be received, and the index information into the sender's FIFO using an RDMA write.

②A: The receiver continuously reads the FIFO queue and checks for updates to the index information. When it receives information from the sender, it issues a WR based on the received details.

③A: The data is sent according to the contents of the WR.

④: When the WR is issued, the counter `cnt` is incremented. When a completion queue (CQ) entry is completed, the counter `cnt` is decremented.

It is important to note that a WR can be posted to the send queue before earlier WRs are processed, meaning the actual data transfer may not begin immediately. To minimize the delay between posting a WR and starting the data transfer, we introduce a sliding window in the WR measurement. By default, this window covers w (e.g., 64) data trunks in transfer, with a duration of data transfer within the window typically on the order of tens of microseconds. This approach allows us to measure the throughput on an RNIC port with high precision.

3.3 Offline Analysis

Trace model. Through the online profiling module, we obtain information on both computation and communication throughout the model training. The next step is to accurately identify and locate anomalies based on this information. By WR measurement, the communication bandwidth can be accurately determined. This enables assessment of whether the communication is functioning properly and whether the achieved bandwidth meets the expected performance targets. In contrast, anomaly detection in computation is more complex. Each rank has different dependencies within various communication groups, making it more challenging to handle computation-related issues.

The execution process of a PP group is sequential, with each communication waiting for the batch computation within the corresponding TP group to complete. The DP groups consist entirely of communication processes, which occur synchronously after the batch computations on all ranks have completed. Therefore, constructing the overall training execution flow from the perspective of the PP group schedule can capture both the batch computation on each rank and treat each TP group's computation as a whole within the process. From the PP schedule as shown in Figure 7, we can see that some computations have bubbles, which implies that assessing a rank's task execution based solely on execution time is unreasonable, as there is *slack time* to allow flexibility in these computations.

Therefore, by analyzing the training configuration file and following the 1F1B rule, we can obtain the schedule for each iteration, which represents the forward and backward compu-

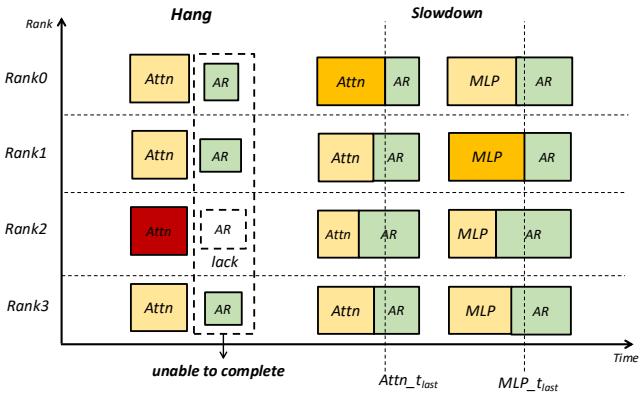


Figure 6: Hang and slowdown detection in TP group.

tation dependencies for each batch. We leverage the dependency relationships in the schedule to transform the task of identifying computation anomalies into a *critical path problem*. Based on this, we construct a directed graph $G(V,E)$, where Forward or Backward are the vertices V, and E represents the dependencies between them. 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`. Once $G(V,E)$ and T_i are determined, we can apply critical path algorithms (detailed in Appendix A) to identify the dependency path that directly impacts the final performance in this iteration. This ensures that analyzing the points on the critical path is meaningful for locating the anomalous graph nodes.

With the critical path identified, the next step is to assess whether the performance of each graph node in this path aligns with expectations. We employ a method of recording and updating the mean value to make this decision. Starting from the third iteration, we compare the current value to the mean, and if the difference is within a threshold α (e.g., 5%), we update the mean; if the difference exceeds another threshold β (e.g., 50%), the node's execution time is considered 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 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, as discussed above, slowdowns occur only in a PP groups explicitly, so we record the computation time consumed by each PP group within one iteration. We then focus graph construction and analysis on those PP groups with

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
```

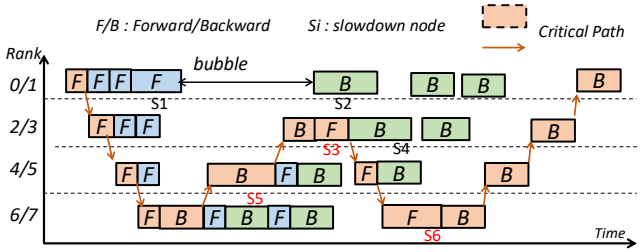


Figure 7: A typical critical path in TP=2 PP=4.

longer computation times. In large-scale training, scaling mainly relies on expanding DP. For example, in the LLaMA 405B model, the PP group has only 16 ranks but 128 DP groups. This approach allows us to analyze only a PP group that exceed the slowdown threshold, further reducing the overhead of graph generation.

Locate hang issue. The algorithm for detecting hang is shown in Algorithm 1. The offline thread maintains an expected execution time for each batch. When a hang occurs, the offline work thread will not receive any data from the online profiling for a certain period. When this duration, denoted as t, exceeds n (e.g., 2) times the expected time, the worker thread immediately notifies the manager thread for further processing.

At the same time, upon detecting a sustained high WR count, the WR measurement system sends an empty probe packet to assess network status. This packet contains no payload, and its transmission causes the WR count to rise. In a healthy network, the count first increases and then decreases. In contrast, if a network problem exists, the WR count will continue to increase without decreasing, which can be used to assess the health of the network. If the hang is caused by a network issue, the data transmission failure will result in the WR system sending the faulty network path to the manager thread. The manager thread uses the result of WR measurement to determine if the issue is network-related.

Since a hang has occurred, the data received by the man-

ager thread is incomplete. The manager thread will generate a corresponding *incomplete graph* based on the incomplete data. A detailed hang issue is presented in Appendix B. In this incomplete graph, the dependencies between nodes are analyzed to identify the first incomplete node. If it is a network issue, locating the missing node helps confirm the stage of failure. If it is a computational issue, further analysis is required to check for missing API calls in the parallel groups associated with the incomplete node. Missing API calls indicate that the preceding computation did not finished, allowing the identification of the faulty computational node.

Locate slowdown issue. The slowdown detection algorithm is shown in Algorithm 2. The offline work thread collects the online profiling data and classifies it according to the respective parallel groups. Once the log for each iteration is complete, the manager thread is triggered to process the data for that iteration. At the same time, the WR measurement system continuously monitors the network's bandwidth and notifies the manager thread if the bandwidth decreases by more than a factor of m (e.g., 20%).

If the network is functioning normally, the manager thread identifies the PP group with the longest execution time. Using the 1F1B schedule and the duration of each node, the manager thread constructs a trace graph and calculates the critical path. Each point on the critical path is then evaluated by comparing its execution time with the expected duration for that node. Any node on the critical path with a duration exceeding its expected value is identified. This slowdown node corresponds to a TP group or DP group. The manager thread further compares the synchronous communication calls within the group to find the slowest call, which leads to identifying the rank and stage corresponding to the slow node.

For example, there are two PP groups in this training session, and a total of four batches need to be computed. We identify the longest PP group based on the total computation time, as shown in Figure 7. Next, we apply the critical path algorithm and identify the key nodes responsible for the slowdown. Nodes from S1 to S6 exhibit performance degradation. However, since some of these nodes are not part of the critical path or are influenced by bubbles that cause fluctuations, they do not contribute to the overall performance drop.

In this case, the key nodes are S3, S5, and S6. Taking S5 as an example, we trace the corresponding TP group and identify the slowest node in the `AllReduce` operation. As shown in Figure 6, this allows us to locate the rank responsible for the slowdown in S5.

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 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
```

4 Implementation

We implemented the NCCL API interception and WR measurement features using approximately 1000 lines of C++ code based on NCCL v2.21.5. We also developed a Trace Manager for log data analysis, trace tree generation, and anomaly monitoring, consisting of about 1500 lines of C++ code. The result visualization was handled using Python. All the source code is publicly available [15].

Online profiling. The entire online profiling functionality is implemented within NCCL, and to use it, one simply needs to replace the corresponding NCCL with our modified version. By intercepting the NCCL API interfaces, we are able to record the API type, time, size, stream, and other relevant information when an NCCL verb is called. This part of the implementation requires only the insertion of an asynchronous log processing function into the relevant interfaces. We track the change in the number of WRs by recording the `ib_send` and `ib_test` functions during WR submission and completion. Since this value changes dynamically, we use a 50-size sliding window approach to accurately calculate bandwidth usage. For WR anomalies, our timeout is set to 500ms, and

Table 1: 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

the threshold for performance degradation is 20%.

Offline analysis. To understand and construct the trace model, we first load the relevant Megatron-LM configurations into the analysis program. The configurations we used are listed in Table 1.

We assign a worker thread to each rank to asynchronously process the data for that rank. All worker threads share a block of memory, which is used to store TP, PP, and DP data for a single iteration. Each worker thread is responsible for processing the received log information and, in combination with the overall model settings, populating the iteration data accordingly. After each iteration's data is filled or when an anomaly is detected, the worker thread notifies the Manager thread. We set $n = 100\%$ to determine whether a timeout occurs. Upon receiving notifications from all rank workers, the Manager thread immediately backs up the current data, clears it, and notifies all rank threads to continue processing. The data copy is then used by a new processing thread to generate the graph and perform anomaly monitoring. We set $m = 50\%$ to identify whether a node experiences a slowdown.

5 Evaluation

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

- How effective is VENUSTRACE in troubleshooting hang and slowdown issues (§5.2)?
- What advantages does the design of VENUSTRACE have compared to other approaches (§5.3)?
- How does VENUSTRACE perform in real-world environments (§5.4)?

5.1 Experimental Setup

Our hardware testbed consists of 256 Supermicro GPU servers. Each server is equipped with 8 NVIDIA H100 GPUs, 8 NVIDIA Mellanox ConnectX-7 SmartNICs (400GbE), 2 AMD EPYC 9575F 64-Core Processor 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 VENUSTRACE offline 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 the forward computation, backward computation, and data transmission phases. In our test model, a single forward computation takes approximately 40ms, so we set the slowdown delay time randomly between 50ms and 30000 ms.

In the evaluation, we use True/False and Positive/Negative classifications to assess the detection results. For the slowdown problem, we use the performance degradation of the training iterations as the ground truth to identify slowdowns. Recall and precision are then used to compare the detection effectiveness of different methods for both hang and slowdown issues, where Recall = $\frac{TruePositive}{TruePositive+FalseNegative}$, Precision = $\frac{TruePositive}{TruePositive+FalsePositive}$, and Accuracy = $\frac{TruePositive+TrueNegative}{TruePositive+TrueNegative+FalsePositive+FalseNegative}$.

As discussed in §2, numerous studies have addressed fault localization for large-scale model training. For our evaluation, we focus on approaches [3, 5, 12, 14, 34] 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 [31] presents a further analysis of the heatmap method in MegaScale. Since Aegis [5] and Holmes [34] employ similar methodologies with Holmes providing a more refined implementation, we did not explicitly compare these two works.

5.2 Overall Effectiveness

Hang and Slowdown detection. For hang localization, as shown in Figure 8, VENUSTRACE 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 8, VENUSTRACE achieves an F1-score of 0.95 in slowdown detection, outperforming Holmes, Minder and MegaScale methods which scored 0.93, 0.89, and 0.84.

Analysis of responsiveness. In terms of responsiveness, VENUSTRACE records expected execution times, enabling

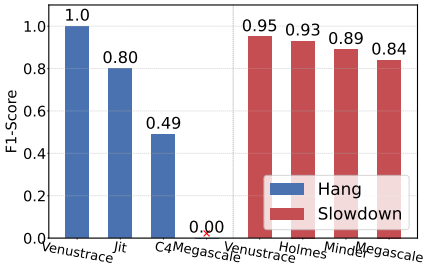


Figure 8: F1-Score of different methods.

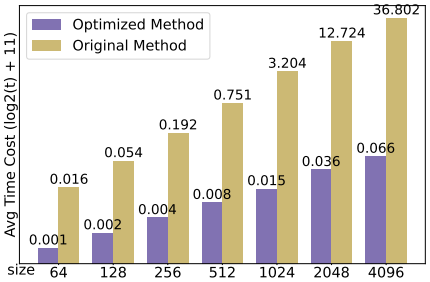


Figure 9: Alg. overhead at scale.

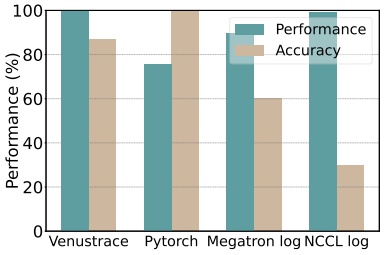


Figure 10: Instrumentation overhead.

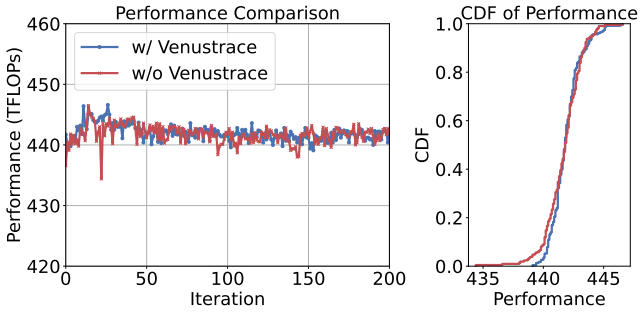


Figure 11: Training performance comparison.

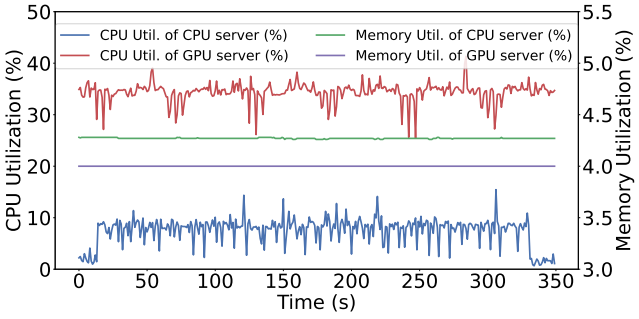


Figure 12: CPU and memory usage.

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, VENUSTRACE leverages real-time network bandwidth monitoring to accurately localize network anomalies. Regarding computational slowdowns, analysis is triggered at the end of an iteration. VENUSTRACE 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 overhead times from 64 ranks training (TP2&PP4) up to 4096 ranks training, with results shown in Figure 9. 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 offline analysis overhead. To compare the training overhead in real-world large-scale scenarios, we conduct a comparison of system performance with and without VENUSTRACE in a 2048 GPUs training task. As shown in Figure 11, we can observe that VENUSTRACE imposes nearly no overhead on the training process. The training performance with VENUSTRACE enabled is similar to when it is disabled, with almost identical performance distribution in Cumulative Distribution Function (CDF). After enabling VENUSTRACE, the sustained iterative performance experiences only a marginal degradation of 0.16This demonstrates that VENUSTRACE can collect critical information with virtually zero overhead, without affecting performance. We also monitor CPU and memory usage during the training process. As shown in Figure 12, we can see that the CPU and memory usage on the GPU server remained very stable during training, with no significant spikes. VENUSTRACE'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 12, we can see that the offline module utilizes approximately 8% of the CPU resources and 4.4% of the memory resources.

5.3 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 10, compared to traditional analysis tools such as PyTorch profiling, VENUSTRACE 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. VENUSTRACE leverages the two *narrow waist* of computation and communication, ensuring

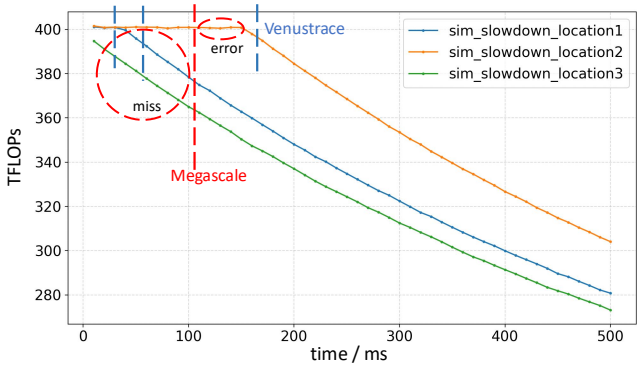
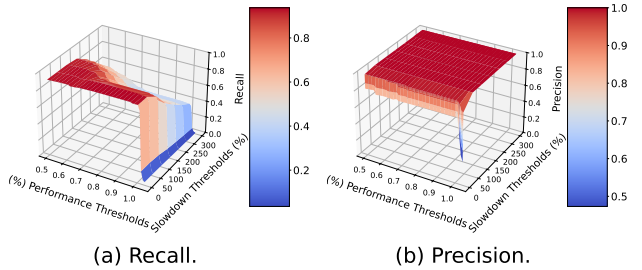


Figure 13: Simulating slowdown time at different locations.

both the effectiveness and minimality of the collected data. VENUSTRACE implements an asynchronous ring buffer to ensure that data collection does not affect training. To conclude, compared to these approaches, VENUSTRACE 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 13. 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. VENUSTRACE employs a critical path method by analyzing the training schedule to identify the graph node that ultimately impact training performance. This approach allows for flexible threshold determination. In contrast, MegaScale uses a fixed threshold for a certain phase. If the threshold is set too high, it results in excessive misses, while a too low threshold leads to more errors. VENUSTRACE bases its analysis on the critical path performance, enabling it to adapt dynamically according to the degree of training performance degradation, thus providing greater flexibility.

Parameter settings in VENUSTRACE. To illustrate the relationship between the judgment threshold, performance degradation, and VENUSTRACE '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 14, 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, 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



(a) Recall. (b) Precision.

Figure 14: Recall and Precision with different thresholds in slowdown detection.

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.

5.4 Case study

We have also deployed VENUSTRACE across two real-world LLM training clusters in our partner company, which 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 VENUSTRACE'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. After isolating the node, restarting the training, and continuing to monitor, the machine restart phenomenon was resolved.

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 VENUSTRACE, we were able to quickly pinpoint the root cause of the hang.

Case 2: Insufficient CPU Allocation on Single Machine. Another interesting 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 VENUSTRACE 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 an environmental deployment error.

6 Discussion

Use of CPU time for profiling. Unlike other high-precision profiling tools that primarily rely on CUDA events for timing, VENUSTRACE uses CPU timestamps. This decision is based on several considerations. First, collecting data at the GPU level allows for precise tracking of each API call, but it introduces additional complexity to the CUDA framework. Second, in an ideal scenario, model training follows a periodic iterative process, which exists both on the GPU and the CPU, and the behavior observed on the CPU exhibits regular patterns across iterations. Lastly, communication APIs are always invoked from the CPU side after the computation has been completed, ensuring a strong ordering constraint that supports the validity of this approach.

Analyzing multiple anomalies with VENUSTRACE. The basic criterion for VENUSTRACE 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, VENUSTRACE 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, VENUSTRACE 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 VENUSTRACE. While this paper presents experiments based on Megatron-LM [18] framework and RDMA networks, the design and methodology are applicable to other LLM training frameworks (e.g., DeepSpeed [27]) 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 VENUSTRACE to other LLM training frameworks and TCP networks as our future work.

7 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 [1, 20, 28, 33] to help identify potential issues within clusters. By testing different cases, hardware or software issues within the cluster can be discovered and localized [1, 20, 28]. Superbench [33] adopts a more efficient testing method to analyze the cluster's performance under different workloads, thereby identifying potential 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 [2, 7, 8, 11, 26, 29, 30] focus on minimizing the time overhead associated with faults in LLM training. Bamboo [29], Oobleck [11], ReCycle [7] and Torch Elastic [26] provide fault tolerance for training across different dimensions, utilizing redundant computations and nodes. CheckFreq [17], Check-N-Run [6], Gemini [30] and DLRover [2] reduce checkpoint overhead by adjusting its frequency or overlapping it with computation, enabling more frequent and lower-overhead checkpointing. JIT [8], 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 VENUSTRACE, better fault detection and recovery can be achieved.

LLM training logging and instrumentation. MegaScale [12] enhances system visibility by instrumenting at the framework level. DLRover [2] uses XPU_timer for fine-grained information collection during training, capturing the entire call stack relationship to analyze training issues. Aegis [5] uses NCCL API logs to perform basic identification of failure and slowdown issues. C4 [4] 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.

8 Conclusion

This paper designs VENUSTRACE, a lightweight profiling system for LLM training. To address the issue of high overhead in traditional profiling, VENUSTRACE 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. This enables profiling on the CPU side with almost zero overhead. Given the complex dependencies in distributed LLM training, VENUSTRACE 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 VENUSTRACE 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.

References

[1] Alibaba. Aicb. https://github.com/aliyun/aicb, 2024.

[2] Alibaba. Dlrover. https://github.com/intelligent-machine-learning/dlrover, 2024.

[3] Yangtao Deng, Xiang Shi, Zhuo Jiang, Xingjian Zhang, Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, Gaohong Liu, et al. Minder: Faulty machine detection for large-scale distributed model training. In *USENIX NSDI 25*, pages 505–521, 2025.

[4] Jianbo Dong, Bin Luo, Jun Zhang, Pengcheng Zhang, Fei Feng, Yikai Zhu, Ang Liu, Zian Chen, Yi Shi, Hairong Jiao, et al. Boosting large-scale parallel training efficiency with c4: A communication-driven approach. *arXiv preprint arXiv:2406.04594*, 2024.

[5] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, et al. Evolution of aegis: Fault diagnosis for {AI} model training service in production. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 865–881, 2025.

[6] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. {Check-N-Run}: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, 2022.

[7] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 211–228, 2024.

[8] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1110–1125, 2024.

[9] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, 2024.

[10] InfiniBand Trade Association. InfiniBand Architecture Specification Release 1.4 Annex A17: RoCEv2, 2020.

[11] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.

[12] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[13] James E Kelley Jr and Morgan R Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173, 1959.

[14] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, et al. Understanding stragglers in large model training using what-if analysis. In *USENIX OSDI 25*, 2025.

[15] Megatrace. Venustrace: Diagnosis and Localization of Anomalies in Distributed LLM Training. https://github.com/sii-research/Megatrace.git, 2025.

[16] Meta. Introducing Llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, 2024.

[17] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. {CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[18] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[19] NVIDIA. Nvidia gpumemory error management. https://docs.nvidia.com/deploy/pdf/a100-gpu-mem-error-mgmt.pdf, 2022.

[20] NVIDIA. Mlperf benchmarks. https://www.nvidia.com/en-us/data-center/resources/mlperf-benchmarks/, 2024.

[21] NVIDIA. Nvidia nsight systems. https://developer.nvidia.cn/nsight-systems, 2024.

[22] NVIDIA. Xid error. https://docs.nvidia.com/deploy/xid-errors/index.html, 2024.

[23] NVIDIA. InfiniBand. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/infiniband, 2025.

[24] OpenAI. https://openai.com/zh-Hant/index/introducing-gpt-5/, 2025.

[25] PyTorch. Pytorch profiler. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html, 2024.

[26] PyTorch. Torch distributed elastic. https://pytorch.org/docs/stable/distributed.elastic.html, 2024.

[27] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[28] Stanford. Dawnbench. https://github.com/stanford-futuredata/dawn-bench-entries, 2018.

[29] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.

[30] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381, 2023.

[31] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. Greyhound: Hunting fail-slows in hybrid-parallel training at scale. In *USENIX ATC 25*, pages 731–747, 2025.

[32] xAI. https://x.ai/news/grok-4, 2025.

[33] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, et al. {SuperBench}: Improving cloud {AI} infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 835–850, 2024.

[34] Zhiyi Yao, Pengbo Hu, Congcong Miao, Xuya Jia, Zuning Liang, Yuedong Xu, Chunzhi He, Hao Lu, Mingzhuo Chen, Xiang Li, et al. Holmes: Localizing irregularities in {LLM} training with mega-scale {GPU} clusters. In *USENIX NSDI 25*, pages 523–540, 2025.

[35] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

A Critical Path Algorithm

Algorithm 3 shows the critical path algorithm to identify the critical path.

Algorithm 3: Calculate Critical Path

// Input: Graph $G=(V,E)$, T where each Node $i \in V$ has start time St_i, end time Et_i and execution time t_i,
// Output: Critical path P_{critical} and Node completion time T_{Node}
// Initialize: $EST_i = 0$, $LST_i = \infty$, $\forall i \in V$
// Step 1: Calculate Earliest Start Time (EST) and Earliest Finish Time (EFT)
for *each Node* $i \in V$ **do**
  for *each predecessor Node* $j \in Predecessors(i)$ **do**
    $EST_i = \max(EST_i, EFT_j)$ // Calculate Earliest Start Time
  end
  $EFT_i = EST_i + t_i$ // Calculate Earliest Finish Time
end
// Step 2: Calculate Latest Start Time (LST) and Latest Finish Time (LFT)
for *each Node* $i \in V$ *in reverse order (from endpoint to start)* **do**
  for *each successor Node* $j \in Successors(i)$ **do**
    $LST_i = \min(LST_i, LST_j - t_i)$ // Calculate Latest Start Time
  end
  $LFT_i = LST_i + t_i$ // Calculate Latest Finish Time
end
// Step 3: Calculate Slack Time and Identify Critical Path
for *each Node* $i \in V$ **do**
  $Slack_i = LST_i - EST_i$ // Calculate Slack Time
  if $Slack_i = 0$ **then**
    $P_{\text{critical}} \leftarrow P_{\text{critical}} \cup \{i\}$ // Add Node i to critical path
  end
end
// Step 4: Calculate Node Completion Time
$T_{\text{Node}} = \max_{i \in P_{\text{critical}}} EFT_i$ // Node completion time is the latest finish time on the critical path

B Trace graphs

Figure 15 and Figure 16 illustrate the trace graphs we generated for detecting hang and slowdown issues.

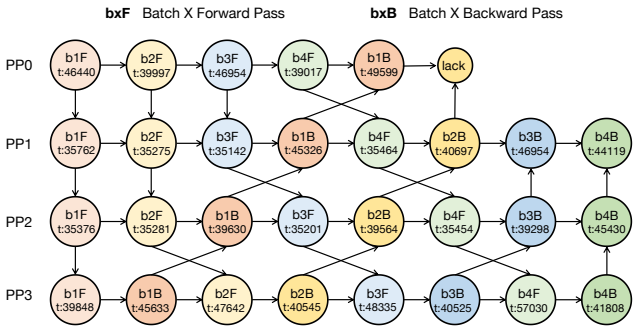


Figure 15: A hang trace graph instance.

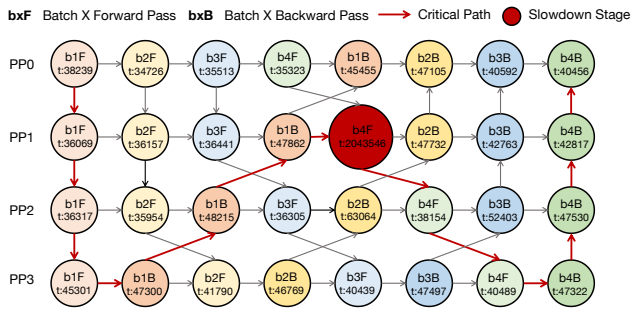


Figure 16: A slowdown trace graph instance.