USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Supercharging Packet-level Network Simulation of Large Model Training via Memoization and Fast-Forwarding

Fei Long, *Tsinghua University;* Kaihui Gao and Li Chen, *Zhongguancun Laboratory;* Dan Li and Yiwei Zhang, *Tsinghua University;* Fei Gui, *Zhongguancun Laboratory;* Yitao Xing, Wenjia Wei, and Bingyang Liu, *Huawei*

https://www.usenix.org/conference/nsdi26/presentation/long

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Supercharging Packet-level Network Simulation of Large Model Training via Memoization and Fast-Forwarding

Fei Long[⋆], Kaihui Gao[‡], Li Chen[‡], Dan Li[⋆], Yiwei Zhang[⋆], Fei Gui[‡],
Yitao Xing[†], Wenjia Wei[†], Bingyang Liu[†]
[⋆]*Tsinghua University* [‡]*Zhongguancun Laboratory* [†]*Huawei*

Abstract

Packet-level discrete-event simulation (PLDES) is a prevalent tool for evaluating detailed performance of large model training. Although PLDES offers high fidelity and generality, its slow performance has plagued networking practitioners. Existing optimization techniques either simplify the network model, resulting in large errors; or execute it in parallel using multiple processors, with an upper bound on speedup.

This paper explores an alternative optimization direction that reduces the computational loads of PLDES while maintaining high fidelity. Our key insight is that, in distributed LLM training, packet-level traffic behaviors often exhibit *repetitive contention patterns* and *steady-states* where flow rates stabilize, ignoring these redundant discrete events speeds up the simulation considerably and the error is negligible. We realize this idea by proposing Wormhole, a user-transparent PLDES kernel capable of automatically memoization for unsteady-states and skipping for steady-states. Wormhole adopts network partitioning, state memoization and reuse, and rate-based steady-state identification to accurately determine the periods of each flow's steady-state, while maintaining simulation consistency after fast-forwarding. Experiments demonstrate that Wormhole can achieve a 744× speedup over the original ns-3 (510× for MoE workload), with a bounded error of <1%. Applying current multithreading parallel techniques and Wormhole together allows a 1012× speedup, reducing the simulation time for one GPT-13B training under 128 GPUs from 9 hours to 5 minutes.

1 Introduction

Large-scale infrastructures of Large Language Models (LLMs), serving as the engine behind the development of generative AI, are currently witnessing an unprecedented surge in investments [18, 30, 59]. LLM training simulation stands as a crucial means to ensure the most effective use of investment [6, 13, 21, 35, 49, 62, 68, 75, 76], these simulators have become indispensable tools to explore the design space of model operator orchestration, parallel strategies, collective communication parameters, transport protocols, network topologies, *etc.*

The packet-level discrete event simulation (PLDES) [5, 19, 27, 64, 71, 81] is the predominant tool for LLM training simulations due to its high fidelity. PLDES meticulously executes the behavior of each packet to accurately simulate performance critical events such as queueing, packet loss, and computation-communication overlapping; these events are important for the design of LLM training systems.

However, detailed simulation leads to heavy computational load, which has been a key problem for PLDES in LLM training simulations [21, 81]. When simulating LLM training clusters, the network interconnects 10^3–10^6 GPUs [18, 24, 30, 61], and carries a large number of elephant flows (*GB* level), which generate a massive number of discrete events ($>O(10^{12})$) that should be executed strictly chronologically. As a result, existing PLDESs (*e.g.*, ASTRA-sim [63, 76] with ns-3 [64]) typically take several *weeks* to simulate one training iteration of GPT3-175B [7, 76](§2.1).

State of the arts. Currently, there exist two categories of technologies that aim to optimize the speed of the network PLDES. The first category involves coarse-grained modeling for networks. Specifically, flow-level simulators [4, 9, 36, 51, 53, 60, 65] and analytical models [35, 46] that solely calculate flow rates at network stability; AI-based methods employ deep neural networks to approximate network performance at different granularities: from individual switches [79], to subtopologies [32, 38, 81], and up to the entire network [14, 66]. Both ignore key packet-level events that contribute to performance fluctuations, showing an error margin of 10% to 25% in the LLM training scenario.

The second category enables parallel execution of PLDES using multiple cores or machines, which has recently received significant attention [5, 19, 21, 75]. While it achieves a certain speedup, the acceleration effects exhibit sublinear scaling as the number of CPU cores increases, eventually reaching an upper bound. Experimental results (§2.1) suggest that employing Unison [5] to simulate GPT3-175B can achieve a maximum

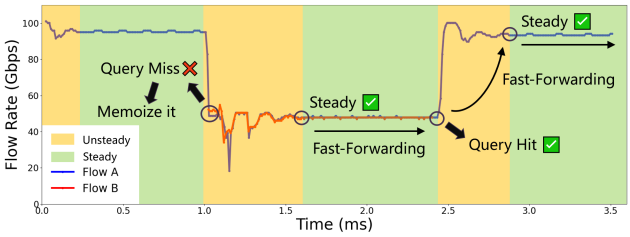


Figure 1: Illustrating unsteady-states, steady-states, state memoization, and simulation fast-forwarding in packet-level network simulation at the scenario of LLM training.

speedup of 10×, after which the acceleration decreases as the number of cores increases.

Research question. To further speed up PLDES, this paper addresses the question: *Can we reduce the computation load within PLDES while maintaining packet-level modeling and high fidelity?* A key advantage of reducing computational load of PLDES is in its orthogonality to existing multi-core parallelization techniques, enabling the acceleration of both approaches to compound.

Key insight. By analyzing the network communication patterns during LLM training, we observe that there are two categories of simulation load that can be reduced. (1) *Repeated Contention Patterns*: identical traffic conflict patterns (*e.g.*, last-hop incast, flow contention at core switch) result in reproducible rate evolution dynamics. The simulation of these episodes can be fast-forwarded by reusing historical results. (2) *Steady-state*: after congestion control algorithms (CCAs) [3, 44, 54, 83] convergence, flows enter a steady-state wherein transmission rates exhibit only minor periodic oscillations or remain constant. The PLDES of this repetitive regime is computationally redundant and can be similarly skipped.

LLM training, including GPT [7] and MoE [47] models, is prone to these two phenomena, which is particularly revealed by the behavior of data parallel (DP) flows. Specifically, during one iteration of LLM training, DP flows (>*GB* level) involve the synchronization of model parameters and gradients within different clusters, which may run periodically and converge to a stable rate. In Figure 1, we take two flows in one path during LLM training as an example to show fast-forward simulation. In the unsteady-state with rate fluctuation, if the exact same contention pattern has occurred in the history, we can reuse the historical data to fast-forward the simulation process to the steady-state; if it has not occurred in the history, then PLDES is executed as usual. Then, the two flows will go to steady-state, their rates tend to stabilize, and the queue length on the links also stabilizes without packet loss. Unless a new disruptive event (*e.g.*, flow enter/exit) occurs, this steady-state will continue. The simulation can fast-forward to the next unsteady-state without significantly affecting the simulation results, the FCT error if we skip the steady periods in Figure 1 is less than 1%.

Challenges. However, fast-forwarding both unsteady and steady-states to accelerate PLDES is not straightforward. Firstly, it requires the precise identification of the network regions in different states. If the whole network is regarded as a single region, it is difficult to match historical data and enter the global steady-state. Secondly, memoizing unsteady-states necessitates the extraction of key features that characterize flow contention patterns. This feature set must be representative enough and reproducible enough to enable the frequent reuse of historical data without compromising the accuracy of the simulation. Finally, defining and identifying the steady-state involves a delicate trade-off between the accuracy and the degree of acceleration achieved — a balance that is best guided by theoretical frameworks.

Our solution. This paper presents Wormhole, a user-transparent PLDES kernel capable of automatically memoization [15, 16, 52] and fast-forwarding, compatible with current PLDES parallelization technologies [5, 17, 19]. The main goals of Wormhole are to accurately identify and skip both the unsteady and steady-states, maintaining the consistency and correctness of the simulation. We propose three designs to address the aforementioned challenges.

❶ Network Partitioning Algorithm: To accurately identify the network regions in different states, we observe that the network tends to form non-interfering partitions in LLM training, which can be handled separately. To find them in advance, we propose a port-level network partitioning algorithm. This involves dividing the network into multiple connected graphs (*i.e.*, partitions) based on the ports through which flows go. Flows passing through the same port belong to the same partition. Consequently, the state of one partition is only determined by the flows within it.

❷ Memoization for Unsteady-states: To utilize repeated flow patterns, we employ the memoization technique [15, 16, 52] and build a database to record previously simulated scenarios. Specifically, we abstract a flow conflict graph (FCG) for one network partition, which captures the critical determinants of the unsteady-state processes. The database stores the first occurrence of unsteady-state processes, *i.e.*, the FCG (the *key*) and the snapshot at the end of the unsteady-state (the *value*). In the subsequent simulation, when one network partition enters an unsteady-state, it will query the database and reuse the historical data (if the query hits).

❸ Steady-state Identification Algorithm: By studying the dynamic equations of mainstream CCAs [3, 44, 54, 83], we propose a steady-state identification algorithm based on a unified metric — *sending rate*. If the rate fluctuation within a monitoring interval is below a predefined threshold, the flow enters a steady-state, and the average rate within this interval is utilized as the rate during the steady-state. Theoretical analysis confirms that when the sending rate is stable, other flow metrics are stable. Moreover, we analyze the error of the algorithm as well as the choice of its parameters.

We prototype Wormhole based on ns-3 [64] and address several practical challenges. 1) To maintain the impact of

steady regions on other flows, such as buffer occupancy, we pause the packets in the steady region's ports, thereby keeping the queue length constant until the steady-state ends. 2) To elegantly skip the simulation process of ns-3 without reconstructing its underlying architecture, when skipping a period for a partition, we increase the timestamps of the partition's events by ΔT, instead of clearing these events.

More specifically, we make the following contributions:

- We find that there are two types of computational load that can be omitted in LLM training simulation, resulting in substantially faster simulation with negligible error, and we are the first to introduce the concepts of *memoization* and *fast-forwarding* into PLDES.
- We propose a series of algorithms to identify as many fast-forward opportunities as possible, notably network partitioning algorithm, steady-state identification algorithm, and simulation skip mechanism.
- Based on the dynamic equations of CCAs, we theoretically analyze the bounded simulation error of Wormhole and threshold guidance.
- We perform extensive experiments to confirm the speed and accuracy of Wormhole, which can achieve a 744× speedup on simulating GPT3-175B workload (510× on MoE workload) over the original ns-3, with a bounded error of <1%. Applying Unison and Wormhole simultaneously allows a 1012× speedup on GPT3 workload (716× on MoE workload).

Limitations. In highly dynamic and random-flow-pattern scenarios such as public cloud and multi-tenant workloads [41,72,80], the traffic exhibits fewer repeating patterns and reaches a steady-state at lower frequency. Consequently, the benefit of Wormhole diminishes, with performance degrading to ns-3 baseline levels in worst-case scenarios, but without extra time cost and accuracy loss.
This work does not raise any ethical issues.

2 Background and Motivation

In this section, we first present the background of LLM training simulation, and examine the performance of existing simulators. We then analyze the traffic characteristics of LLM training, and motivate the design of Wormhole.

2.1 Network Simulation in LLM Training

Backgrounds. In recent years, the scale of LLM parameter changes rapidly to hundreds of billions and even trillions [7,12]. LLM training is conducted in a distributed manner across multiple high-performance computing nodes to expedite model convergence. To achieve efficient parallel training of LLMs, data parallelism (DP) [23,42], tensor parallelism (TP) [69], pipeline parallelism (PP) [25,56], sequence parallelism (SP) [34,43] and expert parallelism (EP) [12,47] are commonly employed as parallel acceleration methods. These methods generate multiple communication domains that perform collective operations according to the workload requirements of LLM training.

Thus, the efficiency of network communication significantly affects training performance. Typically, network simulation technology is required to precisely and cost-effectively optimize network communication, identifying the most suitable network configurations for LLM training. However, as the parameter count of LLMs increases, the scale of training clusters has also expanded to 10^3 –10^6 GPUs [18,24,30,61]. The performance of current state-of-the-art network simulators is difficult to meet requirements. Next, we conduct experiments to confirm this.

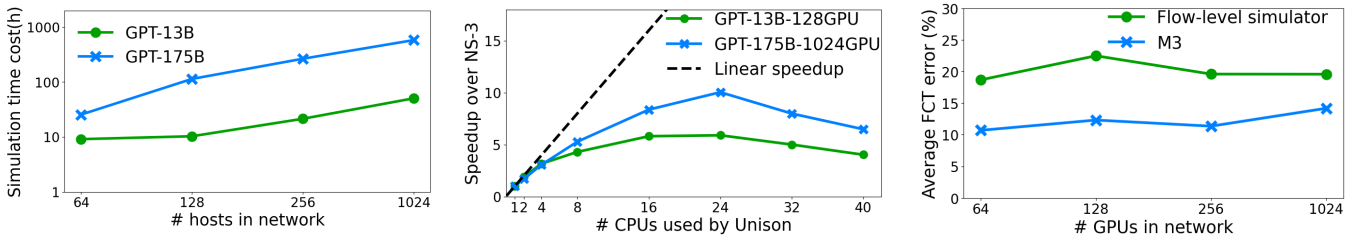
Experiment settings. To compare the speed and accuracy deficiencies of state-of-the-art methods, we simulate a LLM training workload iteration with DP and PP traffic on a 56-core server. The LLM training network is set to scale from hundreds to thousands of GPUs, generating $O(10^2)-O(10^4)$ of DP and PP flows.

Packet-level discrete-event simulation. The pure single-process packet-level simulation in ns-3 is full-fidelity but extremely inefficient. During the simulation process, a discrete event level of $O(10^{12})$ can be generated. As shown in Figure 2a, with the increase in cluster scale, the time cost also shows an exponential growth trend, and it takes several weeks to complete the simulation of large-scale clusters.

Parallel and distributed DES. UNISON [5] and DONS [19] use multithreading to execute PLDES in parallel on multiple CPUs. But they can only provide sublinear speedups due to the synchronization overhead, which always hits an upper bound, as shown in Figure 2b. Another downside is that it uses up a lot of CPU cores that could be used to run multiple independent experiments in parallel, which would exhibit a linear speedup, since LLM engineers always run multiple sets of experiments at once.

Flow-level simulation. Flow-level methods typically fall into two categories. The first computes stable flow rates via max-min or waterfilling allocation [4, 9, 36, 51, 53, 60, 65]. The second uses analytical models [35,46] or profiling-based techniques [6, 13, 37, 49] to approximate flow completion times. Both ignore packet-level dynamics such as queueing, congestion control, and transient losses, and producing ∼20% FCT error under dynamic LLM workloads (Figure 2c).

AI-based methods. AI-based methods [14, 32, 66, 79, 81] are designed to achieve faster performance estimation using neural network models. However, the acquisition of training data incurs high costs. Without a sufficiently large training set, they may not generalize to network scenarios. In Figure 2c, M3 [38] has an error of 10-15% in various scenarios, which is lower than that of flow-level simulators but still results in significant absolute deviations.

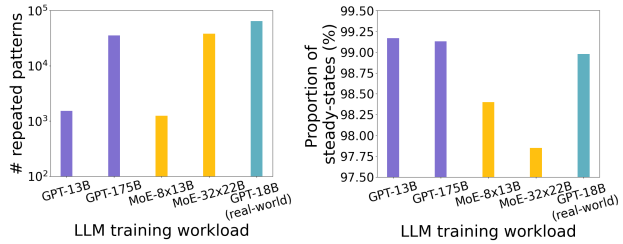


(a) Speed for ns-3 to simulate network communications of LLM training

(b) Upper bound on the speedup of the multi-threaded simulation approach

(c) Error of flow-level simulators and AI-based methods

Figure 2: The efficiency and accuracy of existing network simulation techniques for LLM training.



(a) Repeated contention patterns

(b) Proportion of steady-states

Figure 3: Repeated contention patterns and steady-states in LLM training.

Summary. Existing PLDES optimization techniques either simplify the network model, resulting in large errors, or execute it in parallel using multiple processors, with an upper bound on speedup. Practitioners are in urgent need of an optimization technique that reduces the computational burden of PLDES and maintains accuracy.

2.2 Repeated Flow Patterns in LLM Training

In large-scale model training, flow contention is a common occurrence due to the presence of diverse parallelization strategies. For instance, all-reduce flows in DP and point-to-point flows in PP may encounter contentions. A flow contention pattern is described by the set of flows that have overlap, as well as the links they travel through. We observe that the contention patterns are highly repetitive in LLM training. Under a fixed parallelization strategy, the same collective communication tasks are invoked multi-times in one training iteration, and thus the same contention patterns reappear in LLM training simulation.

To verify the prevalence of the repeated flow pattern, we conduct simulations on SimAI [75] for the training of GPT-13B and MoE-8×7B on 128 GPUs, as well as experiments for GPT-175B and MoE-32×22B on 1024 GPUs. We also analyze a real world trace of the training of a GPT-18B on 256 NVIDIA A100 GPUs. All these experiments employ the Rail-Optimized Fat-tree [57] topology.

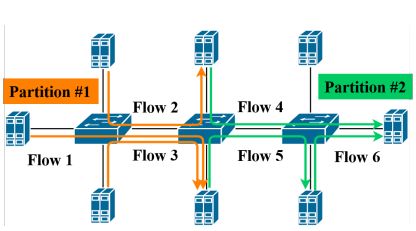
As Figure 3a shows, training a GPT-13B or MoE-8×7B on 128 GPUs yields flow pattern repetitions over 1200 times per iteration, where a total of 1633 flow contention patterns are identified. Training a GPT-175B or MoE-32×22B on 1024 GPUs yields nearly 40000 repetitions. The real-world case of training a GPT-18B on 256 GPUs yields 65870 flow contention pattern instances, which collapse into 1488 distinct patterns with over 60000 redundant occurrences.

2.3 Steady-state in LLM Training

Proportion of steady-states in LLM training. A steady-state in LLM training workloads refers to a temporal interval (steady period) during which network flows in a specific topological subset (steady region) exhibit stable transmission behaviors. LLM training uses efficient parallelization methods that generate a significant number of elephant flows of the same size within the cluster [20, 73]. Specifically, when employing data parallelism in LLM training, DP communication domains produce DP flows reaching GB levels.

Based on the above simulation experiments, we verify and quantify the existence of steady-states in LLM training. We give the formal definition of the steady-state in §3.1.2. As shown in Figure 3b, dense models (*e.g.*, GPT-175B) exceed 99% steady-state proportion, while MoE models, due to all-to-all traffic in EP, exhibit ∼97.5%; real-world traces demonstrate 98.82% steady-state proportion. We attribute the prevalence of steady-states in LLM training to the periodic exchange of parameters or gradients at fixed intervals required by parallelization methods and the constraint of flow paths to repetitive topological subsets by collective communication patterns.

Numerical analysis of simulation error. For the case in Figure 3b, we obtain the rate evolutions of all flows and perform a numerical analysis to compute the error of FCT after skipping steady-states. We identify the steady-state offline and use the average rate over this period as the constant sending rate during the steady-state. The speedup is calculated by dividing the total flow size by the amount of data sent during the steady period. The relative error of the FCT is determined by comparing the FCT derived from the original data with the estimated FCT. The results indicate that, by ignoring all events in steady-states, the simulation of LLM training can be accelerated by 120× on GPT (60× on MoE), with an average FCT error of only 1%. This high acceleration potential motivates Wormhole.



(a) An example of network partitions

(b) Examples of Flow Conflict Graph

Figure 4: Examples of network partitions and the corresponding Flow Conflict Graphs (FCGs).

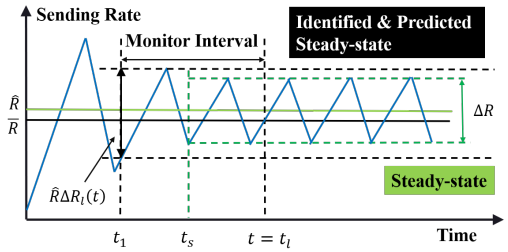


Figure 5: Flow sending rate during CCA convergence.

3 Wormhole Overview

In this section, we first define two important terms and concepts, *network partition* and *steady-state*, and then introduce the overall workflow of Wormhole. We present the skipping of unsteady-states in §4 and skipping of steady-states in §5.

3.1 Terminology & Formal Definitions

3.1.1 Port-level Network Partition

Data center networks are large in scale and have multiple paths, which are shared by a large number of traffic flows [39, 40]. If the network is treated as a whole when identifying the steady-state, the network will enter the steady-state only after all flows are stable, which is rare. Nevertheless, we observe that due to the locality nature of task deployment in data centers, the paths traversed by flows may not interfere with each other, forming multiple sub-networks.

In the scenario of LLM training, the deployment of DP and TP leads to the formation of several non-intersecting data parallel communication groups and tensor parallel communication groups within the network. Previous work [75] has also employed packet-level DES to simulate TP flows. The links between DP groups and TP groups do not overlap, as DP groups are generally deployed across different Points of Delivery (PoDs), while TP groups are deployed within a high-bandwidth domain. Moreover, both types of communication are confined within their respective communication domains. Consequently, the communication within DP groups and TP groups can be considered as occurring in different relatively independent sub-networks. Obviously, the states of these sub-networks do not affect each other, and identifying their states separately will avoid the steady-state being missed.

To capitalize on this characteristic, we formulate a formal definition of *network partitioning*.

Definition 1 (Network Partition)**.** *Flows sharing the same port (or link), along with the ports and links through which these flows traverse, construct a network partition.*

As demonstrated in Figure 4a, we segment flows and links so that no two flows in different partitions share common links. We define network partitions based on flow intersections at switch ports rather than entire switches. We reject switch-centric partitioning because individual ports on a switch exhibit minimal mutual interference despite co-location. By isolating flows that traverse shared ports into distinct partitions, our method achieves three key benefits: (1) finer-grained parallelism through contention-aware division, (2) elimination of hidden dependencies between logically separated flows, and (3) increased probability of finding steady-states. Notably, in MoE training workloads, spatial locality of all-to-all traffic and typical EP degrees (≤ 64) [12, 47] inherently constrain partition sizes no more than 128 GPUs.

3.1.2 Steady-state

Training LLMs in data center networks generates numerous elephant flows with sizes larger than 1GB, leading to a substantial number of discrete events. Conducting packet-level simulation for all these discrete events results in an enormous computational overhead, rendering the simulation process inefficient. We discover that after being regulated by congestion control algorithms, the flow rates of these elephant flows can stabilize within a relatively narrow range, exhibiting a trend of gradual convergence or periodicity. Furthermore, we posit that skipping these events during the simulation process has a negligible impact on the simulation outcomes.

To ensure the reliability and efficiency of the simulation process, the steady-state period of the flows must be both sustainable and predictable, characterized by repeatability and periodicity. When a flow is in its steady-state, various metrics associated with it should exhibit minimal fluctuations, confined within a narrow range. Therefore, we formally define the *steady-state* as follows:

Definition 2 (Traffic Steady-state)**.** *A flow is considered in a steady-state between times t_s and t_f if and only if the fluctuations of a set of key metrics are confined within a narrow range. Mathematically, this condition can be expressed as:*

$$\Delta X = \max_{t_s \leq t \leq t_f} \{X(t)\} - \min_{t_s \leq t \leq t_f} \{X(t)\} \quad (1)$$

$$\Delta X < \varepsilon_X \quad (2)$$

where X represents a suite of flow-related metrics, including sending rate R, congestion window size cwnd, round-trip time RTT, queue length Q, and in-flight bytes I. Here, ε_X is a small constant threshold of the metric X introduced to accommodate

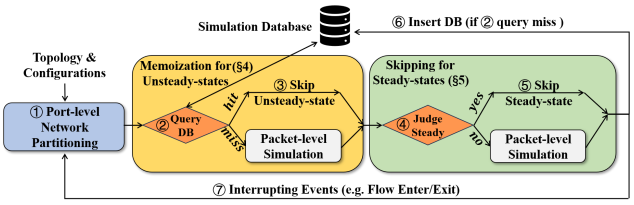


Figure 6: Wormhole's high-level workflow.

the inherent periodicity and minor oscillations observed in the metrics.

This definition of the steady-state captures the essential characteristics of flow behavior during periods of stability, ensuring that the metrics exhibit minimal variation over the specified interval. As shown in Figure 5, the steady-state is considered to start at t_s when ΔR reaches the threshold value. Here, the time interval for defining the steady-state should be based on the convergence of the congestion control algorithms. Additionally, a network partition is considered to be in steady-state if and only if all flows within it have attained steady-state; otherwise it is considered to be in *unsteady-state*.

3.2 Workflow

The workflow of Wormhole is illustrated in Figure 6. Upon user input of the topology, configurations, ① Wormhole first performs the network partitioning at port level (§4.1), which divides the flow into several disjoint sets, facilitating subsequent processing.

Then, ② Wormhole queries a simulation database based on the current partition information to check for the existence of simulation results for the identical scenario. The simulation database adopts the principle of memoization [15, 16, 52], recording once-simulated scenarios and their key outcomes as reusable knowledge (§4.4). If such results are available, ③ Wormhole bypasses the transient phase and transitions directly to the steady-state processing logic. Conversely, if no matching results are found, Wormhole proceeds with the packet-level simulation.

During the process of skipping the unsteady-state or employing packet simulation, ④ Wormhole employs a steady-state identification algorithm to determine whether the system has reached a steady-state. The simulation database adopts the principle of memoization, recording once-simulated scenarios and their key outcomes as reusable knowledge (§5.1.2). If a partition enters a steady-state, ⑤ Wormhole skips this steady-state period; otherwise, it continues with the packet-level simulation. When the steady-state of the network partition ends, or when the simulation of the network partition terminates at the packet-level granularity, ⑥ if Wormhole misses in the previous query, it encounters a novel simulation scenario and is required to insert the key information of the current simulation situation into the database for subsequent simulation processes (§4.3). Subsequently, ⑦ Wormhole continues to process relevant interruptive events, including flow enter and flow exit, which alter the current network partitioning pattern(§5.3). Therefore, the network partitioning algorithm needs to be invoked again to restart this cyclical process.

4 Memoization for Unsteady-states

In this section, we first present the definition of the *Flow Conflict Graph (FCG)* as the central abstraction for representing network partitions in unsteady-states. We then describe the mechanisms for storing unsteady processes using FCGs, followed by the procedure for and retrieving and bypassing unsteady-states through memoization.

4.1 Network Partitioning Algorithm

During the processing of interrupting events, the network partitions the entire network into several network partitions based on the topological overlap relationships of the flows. We achieve this partitioning by constructing a bipartite graph where vertices represent flows and links, creating edges between a flow vertex and a link vertex if the flow traverses that link. The depth-first search (DFS) [70] is then used to identify connected components (partitions) within this graph.

Appendix A describes the network partitioning algorithm used in Wormhole. Given that each node is visited at most once, the complexity of the algorithm is $O(N+M)$, where N represents the number of flows and M represents the number of links in the network. When a new flow starts, some existing network partitions may be merged. When an existing flow finishes, an existing network partition may be split. In response to the dynamic reconstruction characteristics of network partitions, we have proposed an incremental network partitioning algorithm, which is detailed in Appendix B.

4.2 Flow Conflict Graph

When a network partition operates in an unsteady-state, its flows are regulated by CCAs until the rates converge. Due to the diversity of partition topologies and the inherent complexity of CCA dynamics, the transient evolution of flow rates is analytically intractable. However, as discussed in §3.1.1, the training traffic of large-scale learning models often exhibits strong locality and symmetry, resulting in recurring partition structures across simulations. This recurrence motivates the use of memoization: the simulator can identify previously encountered cases and reuse the results instead of re-executing identical transient processes.

To support efficient detection and reuse of repeated cases, we need a lightweight yet expressive representation of simulation state. We define the *Flow Conflict Graph (FCG)* as such an abstraction. FCG models the conflict relationships among flows within a partition as an undirected graph. Vertices represent flows and edges denote link-sharing dependencies. If two flows traverse at least one common link, an edge is placed between them. To encode transient information, the vertex weights correspond to instantaneous flow rates, while the edge weights represent the number of overlapping links between

the corresponding flows. Note that FCG deliberately ignores the absolute path length of flows and their spatial positions in the topology as the resulting error is negligible.

This design ensures that FCGs capture both the structural and quantitative features of unsteady partitions, providing a formalized foundation for storage, retrieval, and reuse, which greatly reduces the storage footprint and accelerates retrieval. It creates a canonical representation that makes different simulation runs comparable, enabling reliable detection of recurring patterns.

4.3 Storing Unsteady-states

Upon the initialization of a partition, the simulator constructs an FCG that characterizes its starting condition, which is denoted as FCG_{start}. The steady-state identification algorithm (§5.1.2) subsequently monitors the evolution of the partition and records a corresponding FCG_{end} when a steady-state terminates or the partition encounters a flow completion event.

Simulation database then stores this process in a key-value format, where the key is the starting condition FCG_{start} and the value is a structured tuple containing the essential results of the transient phase:

$$\begin{aligned} &key\text{: } FCG_{start} \\ &value\text{: } (FCG_{end}, \{Size_f\}|_{f\in F}, T_{conv}) \end{aligned}$$

Here, FCG_{end} denotes the FCG at the end of the transient phase, $\{Size_f\}|_{f\in F}$ represents the aggregate transmission volume of flows f in the partition during the unsteady period, and T_{conv} is the measured convergence time.

Simulation database does not preserve the full temporal evolution of a network partition. Instead, it decomposes the process into unsteady and steady phases. This design follows the observation that flow sizes determine the duration of steady-states, whereas flow sizes remain independent of the transient convergence dynamics. Consequently, the database records only network snapshots at the entry and exit of unsteady phases, the steady-state transmission rate of each flow, and the associated convergence time. These quantities are sufficient for reconstructing FCTs at per-flow granularity within network simulations.

4.4 Retrieving and Skipping Unsteady-states

When a new partition is formed, the simulator constructs its FCG_{start} and queries the simulation database to determine whether an equivalent unsteady process has already been recorded. The lookup process first eliminates candidates with mismatched structural characteristics (*e.g.*, different numbers of vertices or edges). For the remaining candidates, a weighted graph isomorphism procedure is applied [11, 22]. If the simulation database returns a matching entry, the simulator bypasses the costly convergence phase of the CCAs. This mechanism reserves accuracy by using graph-based matching that encodes both structural and quantitative information, ensuring that reused results are semantically equivalent to recomputed ones. As a result, this mechanism maintains the correctness while maximizing reuse opportunities.

5 Fast-forwarding for Steady-states

In this section, we first present the steady-state identification algorithm. We then provide an analysis of parameter errors and guidelines for hyper-parameters selection. Finally, we discuss the termination conditions of the steady-state.

5.1 Steady-state Identification

Training LLMs in data center networks generates numerous elephant flows, where simulating every packet-level event incurs enormous computational overhead. Notably, once regulated by congestion control, these flows typically converge to stable or periodic rates. Therefore, identifying the steady-states of these flows and bypassing them during simulation can significantly reduce the computational time and resource consumption associated with the simulation process.

Next, we describe the metrics employed for steady-state identification and present a steady-state identification algorithm, along with the corresponding error analysis.

5.1.1 Metrics for Steady-state Identification

In practice, simultaneously calculating the fluctuation ranges of all metrics is redundant and unnecessary [48]. Therefore, we select the sending rate R as the indicator to assess the steady-state condition. The theory indicates that these metrics are stable when the sending rate R is stable, which is empirically validated through experiments in §7.3. Subsequently, we proceed to prove the following theorem.

Theorem 1. *Within the time interval $[t_s, t_f]$, the congestion control algorithm converges. Assuming the flow rate is stable, i.e., it satisfies $\Delta R < \varepsilon_R$, then for cwnd, RTT, Q, and I, they are also stable.*

$$\Delta R < \varepsilon_R, \quad t_s \le t \le t_f \tag{3}$$

This means when Equation 3 holds, there exist small constants ε_{cwnd}, ε_{RTT}, ε_Q, and ε_I such that the following inequalities hold, according to the definition of the steady-state of the flow and the stability of the metrics from Equation 1:

$$\Delta cwnd < \varepsilon_{cwnd}, \quad \Delta RTT < \varepsilon_{RTT}, \quad \Delta Q < \varepsilon_Q, \quad \Delta I < \varepsilon_I \tag{4}$$

We provide the proof in Appendix C. Theorem 1 indicates that when R is stable, other metrics are stable as well, further corroborating the stability of CCAs [3, 44, 54, 83].

5.1.2 Steady-state Identification Algorithm

In accordance with the definition of the steady-state, our goal is to construct an algorithm that can identify the steady-state during CCA converges, while minimizing the resulting error

at the same time. In Theorem 1, we identify and derive the desirable property that multiple metrics simultaneously achieve stability in the steady-state, which enables us to construct a unified steady-state identification algorithm based on a single metric: the sending rate R.

The steady-state identification algorithm is designed to accurately determine whether the flow has entered a steady-state based on a short segment of local data. This approach necessitates that the flow is considered to be in a steady-state once the fluctuations in the local data stabilize, and it remains stable until the flow completion or the occurrence of other interrupting events. Mathematically, this condition can be expressed as:

$$\Delta R<\theta, \quad when \max_{1\le k\le l}\{R(t_k)\}-\min_{1\le k\le l}\{R(t_k)\}<\theta \tag{5}$$

where θ and l are two appropriately set hyperparameters in advance, and $t_1<t_2<\cdots<t_l=t$ are a series of time instances at which the sampling of transmission rates occurs. Equation 5 holds as long as the CCA is convergent, as the CCA ensures that the flow rate remains constant over time or exhibits a sawtooth pattern upon convergence. With the theoretical guidance provided by Equation 5, steady-states can be accurately identified when l and θ are reasonable.

Identification algorithm. To determine whether a flow is in a steady-state, we compare the fluctuation of the flow rate with a predefined threshold. By maintaining a fixed length rate detection interval l, we can collect flow rate data over a specified period, which allows us to calculate the fluctuation of the flow rate. To enhance the criteria for determining the steady-state, we introduce a condition that the relative fluctuation must be smaller than a predefined threshold θ. For the rate detection interval $t_1<t_2<\ldots<t_l=t$, the fluctuation degree $\Delta R_l(t)$ of the flow rate is calculated as follows:

$$\Delta R_l(t)=\frac{\max_{1\le k\le l}\{R(t_k)\}-\min_{1\le k\le l}\{R(t_k)\}}{(\sum_{k=1}^{l}R(t_k))/l} \tag{6}$$

When the condition $\Delta R_l(t)<\theta$ is satisfied for the predefined threshold θ, the flow can be identified as having entered the steady period, as shown in Figure 5.

Rate estimation. After the steady-state has been identified, we should estimate the rate of the flow in the steady-state, which will affect the accuracy of the FCT. The max-min fair rate allocation algorithm [29] cannot be used because in multi-hop congestion scenarios, the converged flow rates may deviate from max-min fairness as revealed by a previous work [74]. The criterion of PLDES is to correctly simulate the unique process of various algorithms, rather than simulating the most ideal network situation. We set the rate during the steady period as the average rate over this interval:

$$\hat{R}=\frac{\sum_{k=1}^{l}R(t_k)}{l} \tag{7}$$

The finish time of the steady period is described in §5.3.

5.2 Error Analysis and Threshold Guidance

The steady-state identification algorithm utilizes $\Delta R_l(t)$ to approximate ΔR, and $\hat{R}$ to approximate the real steady-state average rate $\overline{R}$, which may introduce errors in the rate. We now analyze the upper bounds of the errors resulting from these approximations.

Error of estimating the sending rate. We first consider the error in estimating the real steady-state average rate $\overline{R}$ using Equation 7 within the steady-state interval by the following theorem.

Theorem 2. *The relative error of stable flow rate estimation is bounded. Specifically,*

$$\hat{\varepsilon_R}=\left|\frac{\hat{R}-\overline{R}}{\overline{R}}\right|<\frac{\theta}{1-\theta} \quad when\ \Delta R_l(t)<\theta \tag{8}$$

We provide the proof in Appendix D.

Error of estimating the steady period duration. In the absence of other interrupt events, the flow will transmit packets at a stable rate after entering the steady-state until all packets of the flow have been sent. We analyze the error of the steady-state duration of the flow through the following theorem:

Theorem 3. *The relative error of steady-state duration is bounded. Specifically,*

$$\hat{\varepsilon_T}=\left|\frac{\hat{T}-\overline{T}}{\overline{T}}\right|<\theta \quad when\ \Delta R_l(t)<\theta \tag{9}$$

where $\overline{T}$ and $\hat{T}$ denote the real and estimated flow steady-state duration.

We provide the proof in Appendix E.

Theorems 2 and 3 indicate that the steady-state identification algorithm possesses a controllable upper bound error in flow rate and duration of the steady period.

Guidelines for selecting θ and l. The selection of the hyperparameters θ and l influences the error and the efficiency of steady-state identification. We analyze the conditions for the values of θ and l in Appendix F.

5.3 End of Steady-state

After flows within a network partition reach a steady-state, we analyze and summarize three types of events that can interrupt the steady-state: 1. entry of new flows; 2. completion of existing flows; 3. rerouting of existing flows (due to link failures or load balancing strategies, *etc.*). These events dictate at

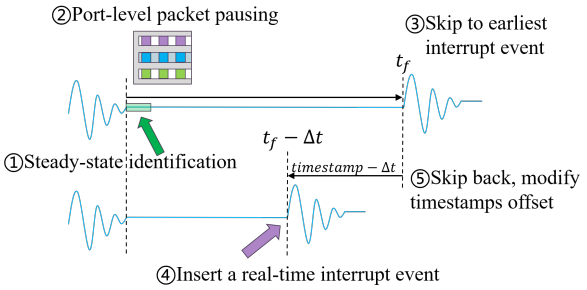


Figure 7: Implementation details of packet pausing and timestamp offsetting.

which time point the simulation process for that network partition can jump to, initiating packet-level simulations. These interrupt-type events may be known in advance or may occur in real time, and we discuss how Wormhole transitions out of a steady-state in each of these scenarios.

Predetermined interrupt-type events. When testing protocols such as congestion control algorithms or load balancing strategies offline, researchers typically construct traffic demands for a period of time in advance and input them into ns-3 [31, 44, 83], which initiates the sending of flows according to their start time. In addition, the timing of link failures can also be predetermined. We use a queue to store these events with known timestamps, where the minimum timestamp among them determines the end time of the network partition's steady-state.

Real-time interrupt-type events. When simulating network communications for LLM training or constructing real-time digital twins of networks, the arrival of flows is uncertain, and link failures cannot be known in advance. Only when these events are actually injected does Wormhole become aware of whether the steady-state should be terminated. In such scenarios, we present a skip-back mechanism: the simulation process of a network partition initially skips to the nearest known timestamp (t_1), when a real-time event occurs with a timestamp (t_2), and $t_2 < t_1$, the simulation process skips back to t_2, thereby ensuring temporal correctness and preventing simulation errors. §6.3 elucidates further details.

6 Implementation

In this section, we dive into the implementation details. Wormhole can be realized by simple secondary development on the existing network simulation engines, we base Wormhole on ns-3.17 [2] and Open-Usim [26], respectively. Our module consists of thousands of lines of code. Applying Wormhole to other PLDES simulators (*e.g.*, OMNeT++ [71], DONS [19]) is straightforward and ongoing. We also address several practical challenges during the implementation.

6.1 Multi-core Parallelization

We find that Wormhole is fully orthogonal to Unison [5], the state-of-the-art multithreading optimization for ns-3. Unison applies a typical DES parallelization strategy that the system automatically generates multiple logical processes (LPs) given the static network topology. During the simulation process, Unison calculates the load of each LP and adaptively assigns them to multiple threads, running on identical CPU cores. The LP scheduler aims to achieve a relative load balance on multiple cores.

Wormhole's network partition information can further optimize Unison's parallel efficiency. To take advantage of this opportunity, we propose a two-stage LP partitioning algorithm. Firstly, different LPs are formed according to the network partitions in §4.1, since there is no traffic interaction between them, which is more suitable for parallelism with multiple LPs. Unlike Unison, the minimum granularity of the network partitioning is network ports (or interfaces), while the minimum granularity in Unison is switches or hosts. This fine-grained partitioning reduces the necessity for data synchronization between LPs.

The simulation database implements lightweight concurrency control. Simulation-start queries are parallelized across threads, while simulation-end inserts are coordinated through fine-grained locks. This concurrency control mechanism is enabled by default in the Wormhole+Unison system in §7.

6.2 Packet Pausing in Switch Port

The first challenge is to exert steady-states' influence on other parts of the network. Other unstable flows certainly belong to different partitions and use different switch ports, so the interaction between these partitions predominantly manifests in the *shared buffer* of the switches. For instance, once flows within a network partition stabilize, they might continually occupy a portion of the shared buffer in a switch, diminishing the maximum buffer space available for other ports' flows. A naive approach is to let these packets be forwarded when the network partition enters a steady-state, because these stable flows will not generate new packets, and then the switch port will not occupy the buffer. However, inaccuracy will occur within this approach. For example, other flows bypassing the switch may encounter severe congestion and will deplete all shared buffer space. If the buffer size is incorrect, the packet loss timing will be wrong.

We propose an effective scheme to address this challenge. In the steady-state, all metrics within the network partition are almost constant. Therefore, as shown in Figure 7, upon entering steady-state, we pause packet processing for ports in this partition and keep the buffer occupancy constant until exiting steady-state. Since these flows continue to occupy the buffer, the maximum buffer size available to other flows will remain the same.

6.3 Offsetting the Timestamp

The second challenge is to elegantly skip the simulation process within ns-3 without completely reconstructing its underlying architecture. As Wormhole has settings for multiple network partitions, each potentially existing in distinct states

# GPUs	# GPT size, parallel	# MoE size, parallel
64	7B, TP8-DP4-PP2	8×7B, TP8-EP8-DP4-PP2
128	13B, TP8-DP4-PP4	8×13B, TP8-EP8-DP4-PP4
256	22B, TP8-DP8-PP4	8×22B, TP8-EP8-DP8-PP4
1024	175B, TP8-DP16-PP8	32×22B, TP8-EP8-DP16-PP8

Table 1: Parameters for LLM training workloads.

with varying durations, it is erroneous to effectuate simulation process jumps by altering the global simulation clock. Our approach adheres to the maintenance of the regular drive of the simulation clock whereas only adjusting the event timestamps of network partitions while entering a steady-state. For instance, when a network partition decides to skip to time T ahead, the timestamps of events (such as packet transmission and message forwarding within switch ports) associated with the specific network partition are increased by T, as shown in Figure 7. Meanwhile, the size and sequence number of these flows must also be modified accordingly.

Skip-back mechanism. In the presence of real-time interrupt-type events that cannot be predetermined, Wormhole will skip the network partition to a known recent interrupt event time (or a relatively large time if there are no predetermined events). Subsequently, when the real-time interrupt event is indeed input, Wormhole then navigates the network partition back to the current time. In other words, if a network partition has skipped to time T_1 and necessitates a revert to an earlier time T_2 (T_2<T_1). Prior to the simulation clock reaching T_1, events within this network partition are not processed, rendering the restoration of its state to T_2 uncomplicated.

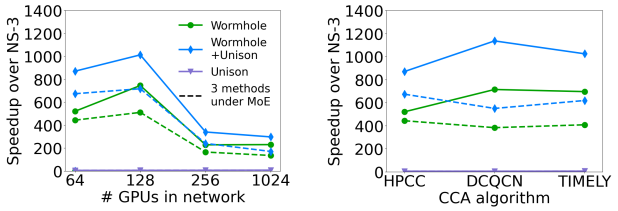
7 Evaluation

We evaluate Wormhole in simulation speed, accuracy, and sensitivity. We summarize our results as follows:

Key Results.

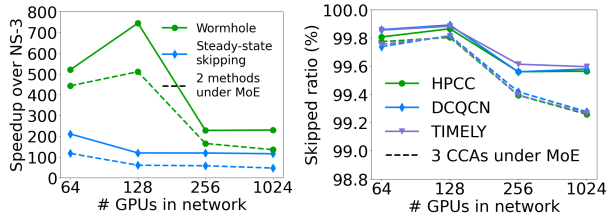
- **Simulation speed:** Wormhole is capable of simulating large-scale LLM training workloads. On a single core, it achieves speedups of 744× and 510× for GPT and MoE workloads compared to ns-3. When integrated with parallel DES, it achieves speedups of 1012× and 716× for GPT and MoE workloads using 16 CPU cores. The scalability of Wormhole is consistent with ns-3.
- **Accuracy:** Under various LLM training workloads and CCA scenarios, Wormhole maintains the average FCT error within 1% on both GPT and MoE workloads.
- **Sensitivity:** The monitoring metrics of steady-states identification are nearly equivalent. Wormhole exhibits insensitivity across varying hyper-parameters and topologies.

Alternatives. We select ns-3 [64], Unison [5], and flow-level simulator [8, 55] as the comparison alternatives, which represent a range of widely utilized types of simulators.



(a) Speedup under different network sizes (b) Speedup under different CCAs

Figure 8: Speedup for simulating LLM training.



(a) Speedup breakdown (b) Ratio of skipped events

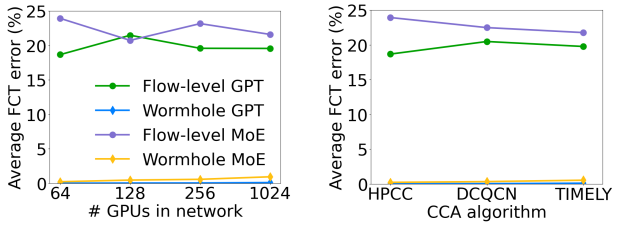
Figure 9: Speedup of different mechanism of Wormhole and ratio of skipped events for simulating LLM training.

Setup. The experimental server is a Linux-based server with 2 Intel Xeon CPUs (totaling 56 cores) and 128 GB of memory. Simulations conduct 4 Rail Optimized Fat-tree topologies [57] with varying sizes for training GPT-3 models or MoE of different scales for one iteration. To accurately model the scenario in LLM training networks where multiple NICs on each server may connect to different switches, we represent each GPU as a host in the simulations.

The training workloads consist of groups of DP, PP and EP flows, as existing works on LLM training simulation commonly neglects TP and SP flows [21, 63, 76]. The parallel configurations of GPTs of varying sizes follow the TP(SP)-DP-PP arrangement, whereas MoEs adopt the TP(SP)-EP-DP-PP arrangement, with specific parameters detailed in Table 1. The micro-batch size is set to 1, which is the smallest value that still permits pipeline parallelism, yielding a global batch size of DP×PP. We evaluate Wormhole across three CCAs: HPCC [44], DCQCN [83], and TIMELY [54]. In all experiments except for the sensitivity analysis, the parameters for Wormhole are θ=5% and l=2000.

7.1 Wormhole is Fast

Speed. We evaluate the simulation acceleration of Wormhole and Unison compared to the original ns-3. Figure 8a illustrates the performance of these simulators in LLM training scenarios of varying sizes under HPCC. While Unison achieves a maximum acceleration of less than 10×, Wormhole attains an acceleration of 227×-745× on GPT workloads (135×-510× on MoE workloads), and Wormhole+Unison achieves a peak acceleration of 1012× on GPT workloads (716× on MoE workloads). Specifically, Wormhole+Unison reduces GPT-13B training time on 128 GPUs from 9 hours to 5 minutes, achieving over 1000× speedup. Figure 8b presents the acceler-



(a) Average FCT error under different network sizes (b) Average FCT error under different CCAs

Figure 10: Accuracy for simulating LLM training.

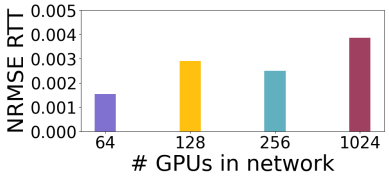


Figure 11: NRMSE of packets RTT in ns-3 and Wormhole.

ation effects of these simulators under different CCAs in the 64-GPU scenario. Results indicate that Wormhole achieves high acceleration across various CCAs, suggesting that Wormhole's optimization is broadly applicable for different CCAs. We analyze the number of network partitions in Appendix G and simulation database storage cost in Appendix H.

Acceleration breakdown. Figure 9a shows that skipping steady-states yields speedups exceeding 130× for GPT and 50× for MoE training, confirming that a significant proportion of steady-states in LLM training are redundant and can be safely skipped. Additionally, the average number of times each flow entered a steady-state was approximately 0.94. Figure 9b quantifies the skip rate across CCAs for LLM training, which shows that steady-state fast-forwarding combined with memoization skips >99.5% of events in GPT workloads and >99.2% in MoE workloads. Comparing the speedup ratios in Figure 9a, memoization alone delivers an additional 1.93×-8.43× acceleration after the dominant steady-state events are skipped, confirming that memoization remains a non-negligible performance booster. We conduct more acceleration breakdown experiments in Appendix I.

7.2 Wormhole is Accurate

Accuracy. We compare the simulation accuracy of Wormhole and a flow-level simulator relative to the original ns-3 in LLM training workloads. Specifically, we calculate the relative FCT error for each flow, and then compute the average of these relative errors. Figure 10a illustrates that Wormhole maintains an average FCT error within 1%, significantly lower than the ∼20% error of the flow-level simulator. Figure 10b presents the average FCT errors of Wormhole and the flow-level simulator across different CCAs. Wormhole without the unsteady-state memoization mechanism (i.e., skipping only steady-states) exhibits smaller errors than Wormhole itself.

Packet-level fidelity. We evaluate Wormhole on boarder metrics. Wormhole evaluation extends to broader metrics through Normalized Root Mean Square Error (NRMSE) computation for all packets of the first flow across scenarios, benchmarked against ns-3. Figure 11 shows that across multiple scenarios, the NRMSE values fall below 0.005, indicating minimal deviation and confirming end-to-end fidelity across additional performance dimensions.

7.3 Sensitivity Analysis

Identification metrics. The equivalence of sending rate R, in-flight bytes I, and queue length Q as metrics for steady-state identification algorithms has been theoretically established according to Theorem 1. To empirically validate this equivalence, we conduct experiments using HPCC with a 128-GPU scenario. Specifically, we employ R, I, and Q individually as metrics for steady-state detection and compare the speedup ratios and errors. As depicted in Figure 12a, the speedup ratios and errors obtained using these different metrics are closely aligned, which corroborates Theorem 1.

Identification thresholds. By fixing one hyper-parameter and varying the other, we examine the sensitivity of Wormhole to two hyper-parameters, l and θ. As show in Figures 12b&12c, as l increases or θ decreases, the criterion for entering the steady-state is more readily satisfied, resulting in a higher ratio of skipped events, which enhances the speedup but also increases the error. Therefore, when l and θ are within appropriate ranges, Wormhole exhibits insensitivity to the hyper-parameters.In practice, θ=5% is sufficient for most scenarios.

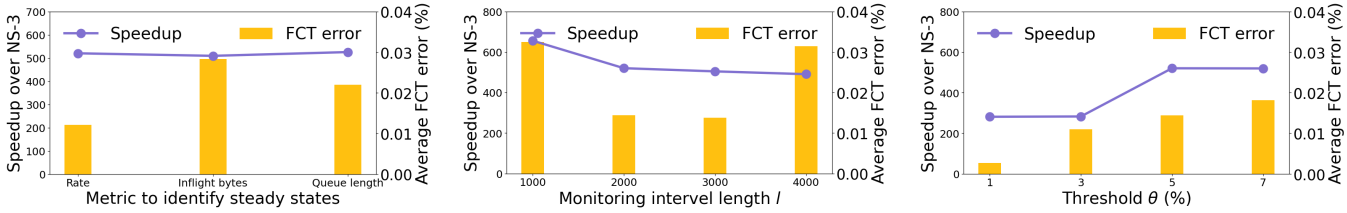
Network topology. We evaluate Wormhole on multiple network topologies, including the standard setup, Fat-tree [1], and Clos [10]. Results are shown in Figure 13, where ROFT denotes Rail-Optimized Fat-tree [57]. The variation in speedup ratios of Wormhole across different topologies does not exceed 13%, and the average FCT error remains within 1%. This indicate that the techniques employed by Wormhole are applicable to a broad range of data center topologies [26, 45, 78].

7.4 Real-trace Based Experiments

We evaluate Wormhole on a real-world LLM training trace. We collect operation-level collective communication latency via NVIDIA Nsight Compute [58] from training a GPT-18B on a 256-GPU ROFT cluster. The training employs TP8-DP16-PP2-VPP2 parallelism, micro batch size of 1, and global batch size of 512.

Speed. Figure 13a shows that Wormhole achieves a 97.75× speedup over ns-3, while Wormhole+Unison achieves 133.35×. The trace incorporates recomputation and hardware performance fluctuations, producing a more complex workload than idealized traces from SimAI [67, 75]. This complexity reduces Wormhole speedup relative to idealized scenarios, yet acceleration still approaches more than 100×.

Accuracy. Figure 13b presents that Wormhole demonstrates 3.02% end-to-end training time error, while ASTRA-sim+ns-3 [76] achieves 3.01%. At comparable accuracy for real-world



(a) Effects of the monitor metrics (b) Effects of the monitoring interval length l (c) Effects of the fluctuation threshold θ

Figure 12: Effects of the monitoring metrics and hyper-parameters in Wormhole.

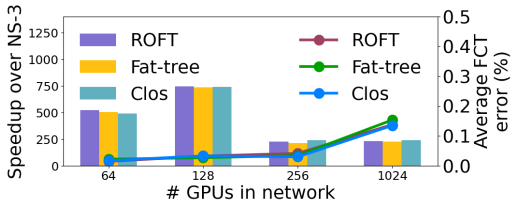
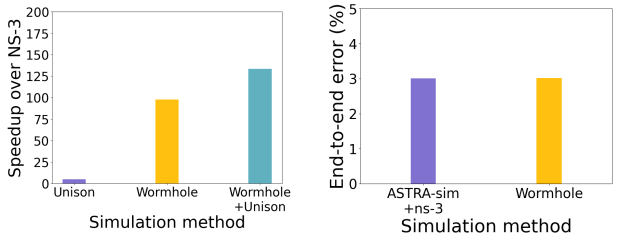


Figure 13: Effects of network topology in Wormhole.



(a) Speedup over different simulation methods. (b) End-to-end error over different simulation methods.

Figure 14: Speedup and error of real-trace based experiments.

model training time, Wormhole provides simulation speed far exceeding the baseline.

8 Related Work

Packet-level discrete-event simulators. PLDES plays a crucial role in LLM training by providing high-fidelity simulations of network behavior. Traditional DES, such as ns-2/3 [27, 64], OMNeT++ [71], and OPNET [50], accurately model packet transmission processes within networks. However, as network scale increases, the scalability of these simulators becomes a limiting factor. Parallel and distributed DES [5, 17, 19, 28] can enhance the efficiency of DES simulations, but the acceleration effects exhibit sub-linear scaling, eventually reaching an upper bound.

LLM training simulation. Simulation of large-scale LLM infrastructures has recently gained significant traction. Network simulators such as ASTRA-sim [63, 76], SimAI [75], and ATLAHS [68] build on PLDES like ns-3 and htsim, but remain bottlenecked by PLDES overhead. To improve scalability, some works [21, 33, 62] exploit GPU or FPGA parallelism, which accelerates execution but leaves the underlying computation unchanged and is orthogonal to Wormhole. Others adopt coarse-grained models that sacrifice fidelity, either through analytical approximations [35, 46] or profiling-based techniques [6, 13, 37, 49].

Flow-level simulation. Flow-level simulators [8, 55, 77, 82] employ flow-level abstraction to conduct network simulations and utilize the max-min fairness algorithm [29] to address bandwidth allocation. This modeling approach achieves a speedup by 2-3 orders of magnitude compared to PLDES, but it introduces significant error margins. Additionally, a considerable proportion of flow-level simulators [8, 77] focus primarily on specific types of network topologies or applications, thereby limiting their generalizability.

AI-based methods. AI-based network simulation methods [14, 32, 66, 79, 81] leverage machine learning to estimate the network performance, resulting in a trade-off with accuracy. They typically require extensive data collection to support model training, but exhibit limited generalizability beyond the training dataset.

9 Conclusion

We observe that, in distributed LLM training, packet-level traffic behaviors often exhibit *repetitive contention patterns* and *steady-states*, ignoring these redundant discrete events speeds up the simulation considerably and the error is negligible. To this end, we propose Wormhole, a user-transparent PLDES kernel capable of automatically memoization for unsteady-states and skipping for steady-states. Wormhole adopts network partitioning, state memoization and reuse, and rate-based steady-state identification to accurately determine the periods of each flow's steady-state. Experiments demonstrate that Wormhole can achieve a 744× speedup over the original ns-3 (510× for MoE workload), with a bounded error of <1%. Wormhole+Unison allows a 1012× speedup, reducing the simulation time for one GPT-13B training under 128 GPUs from 9 hours to 5 minutes.

10 Acknowledgement

We thank our shepherd Prof. Marco Chiesa and the anonymous NSDI reviewers for their constructive comments. Kaihui Gao and Li Chen are the corresponding authors. This work was supported by the Beijing Outstanding Young Scientist Program (No. JWZQ20240101008) and Zhongguancun Laboratory.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM CCR*, 2008.

[2] Alibaba-edu. High-precision-congestion-control. https://github.com/alibaba-edu/High-Precision-Congestion-Control, 2019.

[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM*, 2010.

[4] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: stability, convergence, and fairness. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):73–84, 2011.

[5] Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, and Guihai Chen. Unison: A parallel-efficient and user-transparent network simulation kernel. In *EuroSys*, 2024.

[6] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. vtrain: A simulation framework for evaluating cost-effective and compute-optimal large language model training. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 153–167. IEEE, 2024.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[8] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.

[9] Florin Ciucu and Jens Schmitt. Perspectives on network calculus: no free lunch, but still good value. In *ACM SIGCOMM*, pages 311–322, 2012.

[10] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[11] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[12] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[13] Yicheng Feng, Yuetao Chen, Kaiwen Chen, Jingzong Li, Tianyuan Wu, Peng Cheng, Chuan Wu, Wei Wang, Tsung-Yi Ho, and Hong Xu. Echo: Simulating distributed training at scale. *arXiv preprint arXiv:2412.12487*, 2024.

[14] Miquel Ferriol-Galmés, Jordi Paillisse, José Suárez-Varela, Krzysztof Rusek, Shihan Xiao, Xiang Shi, Xiangle Cheng, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Routenet-fermi: Network modeling with graph neural networks. *IEEE/ACM transactions on networking*, 31(6):3080–3095, 2023.

[15] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. *ACM SIGPLAN Notices*, 37(9):36–47, 2002.

[16] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, 2004.

[17] Richard M Fujimoto. *Parallel and distributed simulation systems*, volume 300. Citeseer, 2000.

[18] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 57–70, 2024.

[19] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *ACM SIGCOMM*, pages 167–181, 2023.

[20] Jinkun Geng, Dan Li, and Shuai Wang. Elasticpipe: An efficient and dynamic model-parallel solution to dnn training. In *Proceedings of the 10th Workshop on Scientific Cloud Computing*, pages 5–9, 2019.

[21] Fei Gui, Kaihui Gao, Li Chen, Dan Li, Vincent Liu, and Ran Zhang. Accelerating design space exploration for llm training systems with multi-experiment parallel simulation. In *USENIX NSDI*, 2025.

[22] Aric Hagberg, Pieter J Swart, and Daniel A Schult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.

[23] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[24] 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.

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Easy scaling with micro-batch pipeline parallelism. *proceeding of Computer Science> Computer Vision and Pattern Recognition*, 2019.

[26] Huawei. ns-3-ub: Unifiedbus network simulation framework. https://gitcode.com/open-usim/ns-3-ub, 2025.

[27] Teerawat Issariyakul, Ekram Hossain, Teerawat Issariyakul, and Ekram Hossain. *Introduction to network simulator 2 (NS2)*. Springer, 2009.

[28] Shafagh Jafer, Qi Liu, and Gabriel Wainer. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, 30:54–73, 2013.

[29] Jeffrey Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[30] 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.

[31] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-aware load balancing at the virtual edge. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 323–335, 2017.

[32] Charles W. Kazer, Jo ao Sedoc, Kelvin K.W. Ng, Vincent Liu, and Lyle H. Ungar. Fast network simulation through approximation or: How blind men can describe elephants. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 141–147, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Sajy Khashab, Hariharan Sezhiyan, Rani Abboud, Alex Normatov, Stefan Kaestle, Eliav Bar-Ilan, Mohammad Nassar, Omer Shabtai, Wei Bai, Matty Kadosh, et al. Nsx: Large-scale network simulation on an ai server. In *Proceedings of the 2nd Workshop on Networks for AI Computing*, pages 19–25, 2025.

[34] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[35] Joyjit Kundu, Wenzhe Guo, Ali BanaGozar, Udari De Alwis, Sourav Sengupta, Puneet Gupta, and Arindam Mallik. Performance modeling and workload analysis of distributed large language model training and inference. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*, pages 57–67. IEEE, 2024.

[36] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2001.

[37] Seonho Lee, Amar Phanishayee, and Divya Mahajan. Data-driven forecasting of deep learning performance on gpus. *arXiv e-prints*, pages arXiv–2407, 2024.

[38] Chenning Li, Arash Nasr-Esfahany, Kevin Zhao, Kimia Noorbakhsh, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. m3: Accurate flow-level performance estimation using machine learning. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 813–827, 2024.

[39] Dan Li, Yunfei Shang, Wu He, and Congjie Chen. Exr: Greening data center network with software defined exclusive routing. *IEEE Transactions on Computers*, 64(9):2534–2544, 2014.

[40] Dan Li, Yirong Yu, Wu He, Kai Zheng, and Bingsheng He. Willow: Saving data center network energy for network-limited flows. *IEEE Transactions on Parallel and Distributed Systems*, 26(9):2610–2620, 2014.

[41] Junfeng Li, Sameer G Kulkarni, KK Ramakrishnan, and Dan Li. Analyzing open-source serverless platforms: Characteristics and performance. *arXiv preprint arXiv:2106.03601*, 2021.

[42] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[43] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, 2023.

[44] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM special interest group on data communication*, pages 44–58. 2019.

[45] Heng Liao, Bingyang Liu, Xianping Chen, Zhigang Guo, Chuanning Cheng, Jianbing Wang, Xiangyu Chen, Peng Dong, Rui Meng, Wenjie Liu, et al. Ub-mesh: a hierarchically localized nd-fullmesh datacenter network architecture. *arXiv preprint arXiv:2503.20377*, 2025.

[46] Zhongyi Lin, Ning Sun, Pallab Bhattacharya, Xizhou Feng, Louis Feng, and John D Owens. Towards universal performance modeling for machine learning training on multi-gpu platforms. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[47] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[48] Tianfeng Liu, Kaihui Gao, Li Chen, Dan Li, Jin Guang, Xinyun Chen, Vincent Liu, Zhiyong Chen, Yiwei Zhang, Ni Jin, et al. Cceval: Accurately and confidently evaluating performance metrics of congestion control algorithms for datacenter networks.

[49] Guandong Lu, Runzhe Chen, Yakai Wang, Yangjie Zhou, Rui Zhang, Zheng Hu, Yanming Miao, Zhifang Cai, Li Li, Jingwen Leng, et al. Distsim: A performance model of large-scale hybrid distributed dnn training. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 112–122, 2023.

[50] Zheng Lu and Hongji Yang. *Unlocking the power of OPNET modeler*. Cambridge University Press, 2012.

[51] Marco Ajmone Marsan, Michele Garetto, Paolo Giaccone, Emilio Leonardi, Enrico Schiattarella, and Alessandro Tarello. Using partial differential equations to model tcp mice and elephants in large ip networks. *IEEE/ACM Transactions on Networking*, 13(6):1289–1301, 2005.

[52] Donald Michie. "memo" functions and machine learning. *Nature*, 218(5136):19–22, 1968.

[53] Vishal Misra, Wei-Bo Gong, and Don Towsley. Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 151–160, 2000.

[54] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[55] Pooria Namyar, Behnaz Arzani, Srikanth Kandula, Santiago Segarra, Daniel Crankshaw, Umesh Krishnaswamy, Ramesh Govindan, and Himanshu Raj. Solving {Max-Min} fair resource allocations quickly on large graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1937–1958, 2024.

[56] 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.

[57] NVIDIA. Superpod: Next generation scalable infrastructure for ai leadership. https://docs.nvidia.com/https:/docs.nvidia.com/dgx-superpod-reference-architecture-dgx-h100.pdf, 2023.

[58] NVIDIA Corporation. Nvidia nsight compute. https://docs.nvidia.com/nsight-compute/2023.3/ReleaseNotes/, 2023.

[59] OpenAI and SoftBank. Announcing the stargate project. https://openai.com/index/announcing-the-stargate-project/, 2025. Jan 22.

[60] Qiuyu Peng, Anwar Walid, Jaehyun Hwang, and Steven H Low. Multipath tcp: Analysis, design, and implementation. *IEEE/ACM Transactions on networking*, 24(1):596–609, 2014.

[61] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 691–706, 2024.

[62] Yicheng Qian, Ran Shu, Rui Ma, Yang Wang, Derek Chiou, Nadeen Gebara, Luca Piccolboni, Miriam Leeser, and Yongqiang Xiong. Miniature: Fast ai supercomputer networks simulation on fpgas. In *Proceedings of the 9th Asia-Pacific Workshop on Networking*, pages 114–120, 2025.

[63] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 81–92, 2020.

[64] George F Riley and Thomas R Henderson. The ns-3 network simulator. *Modeling and tools for network simulation*, pages 15–34, 2010.

[65] Thomas G Robertazzi. *Computer networks and systems: queueing theory and performance evaluation*. Springer Science & Business Media, 2000.

[66] Krzysztof Rusek, José Suárez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Routenet: Leveraging graph neural networks for network modeling and optimization in sdn. *IEEE Journal on Selected Areas in Communications*, 38(10):2260–2270, 2020.

[67] Adel Sefiane, Alireza Farshin, and Marios Kogias. Mlsynth: Towards synthetic ml traces. In *Proceedings of the 2nd Workshop on Networks for AI Computing*, pages 98–104, 2025.

[68] Siyuan Shen, Tommaso Bonato, Zhiyi Hu, Pasquale Jordan, Tiancheng Chen, and Torsten Hoefler. Atlahs: An application-centric network simulator toolchain for ai, hpc, and distributed storage. *arXiv preprint arXiv:2505.08936*, 2025.

[69] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[70] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[71] András Varga. A practical introduction to the omnet++ simulation framework. *Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem*, pages 3–51, 2019.

[72] Fangxin Wang, Ruilin Ling, Jing Zhu, and Dan Li. Bandwidth guaranteed virtual network function placement and scaling in datacenter networks. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2015.

[73] Songtao Wang, Dan Li, Yang Cheng, Jinkun Geng, Yanshu Wang, Shuai Wang, Shu-Tao Xia, and Jianping Wu. Bml: A high-performance, low-cost gradient synchronization algorithm for dml training. *Advances in Neural Information Processing Systems*, 31, 2018.

[74] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, TS Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: efficient, robust, and practical datacenter {CC} via deployable {INT}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 255–274, 2023.

[75] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, et al. {SimAI}: Unifying architecture design and performance tuning for {Large-Scale} large language model training with scalability and precision. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 541–558, 2025.

[76] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 283–294, 2023.

[77] Junjie Xie and Yuhui Deng. mtcloudsim: A flow-level network simulator for multi-tenant cloud. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 332–339. IEEE, 2016.

[78] Zihan Yan, Dan Li, Li Chen, Dian Xiong, Kaihui Gao, Yiwei Zhang, Rui Yan, Menglei Zhang, Bochun Zhang, Zhuo Jiang, et al. From atop to zcube: Automated topology optimization pipeline and a highly cost-effective network topology for large model training. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 861–881, 2025.

[79] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. Deepqueuenet: towards scalable and generalized network performance estimation with packet-level visibility. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 441–457, 2022.

[80] Ruozhou Yu, Guoliang Xue, Xiang Zhang, and Dan Li. Survivable and bandwidth-guaranteed embedding of virtual clusters in cloud data centers. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[81] Qizhen Zhang, Kelvin KW Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. Mimicnet: fast performance estimates for data center networks with machine learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 287–304, 2021.

[82] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. Scalable tail latency estimation for data center networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 685–702, 2023.

[83] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

Algorithm 1: Network partitioning algorithm

```
Function construct_bipartite_graph(flows):
    connections[1..n+2×m] = empty list
    for flow in flows do
        for link in flow.links do
            connections[flow.id].append(n+link.id);
            connections[n+link.id].append(link.id);
    return connections;
Function DFS(vertex, connections):
    set_visited(vertex);
    local_partition = empty list;
    if is_flow(vertex) then
        local_partition.append(vertex);
    for neighbor in connections[vertex] do
        if not visited(neighbor) then
            neighbor_partition = DFS(neighbor,
              connections);
            local_partition.extend(neighbor_partition);
    return local_partition;
Function network_partitioner(flows):
    connections = construct_bipartite_graph(flows);
    network_partitions = empty list;
    for flow in flows do
        if not visited(flow.id) then
            partitions = DFS(flow.id, connections);
            network_partitions.append(partitions);
    return network_partitions;
```

Algorithm 2: Incremental Partitioning Algorithm

```
Function on_new_flow_enter(newflow):
    affected_partitions = empty list;
    for partition in all_partitions do
        if newflow.path ∩ partition≠∅ then
            affected_partition.append(partition);
    if affected_partitions.size =0 then
        all_partitions.append(partition(newflow));
    else if affected_partitions.size =1 then
        affected_partition.append(newflow);
    else
        network_partitioner(affected_partitions.flow);
Function on_old_flow_leave(oldflow):
    affected_flows = empty list;
    for flow in partition.flows do
        if flow is not oldflow then
            affected_flows.append(flow);
    if affected_flows.size ≤ 1 then
        partition.erase(oldflow);
        if affected_flows.size = 0 then
            all_partitions.erase(partition);
    else
        network_partitioner(affected_flows);
```

Appendix

Appendices are supporting material that has not been peer-reviewed.

A Network Partitioning Algorithm

Network partitioning algorithm is implemented in Algorithm 1.

B Incremental Recalculate the Network Partitions

A challenge is to reconstruct the network partitions in a reasonable time and possibly speed up the recalculation process. Since Wormhole follows port-level partitioning, we observe that such a recalculation will be triggered if a new flow enters the network or a flow is about to finish and leave. Wormhole keeps track of all flows both activated and scheduled to enter so that during the simulation, Wormhole knows when the recalculation should be done.

It is worth noting that in a typical scenario of LLM training, such recalculation of partitions may occur frequently, whereas a certain rearrangement requires extra computing resources. Therefore, we propose a more efficient way to update partitions. Notice that many of the updates occur in the local part of the network topology and only affect a few other flows (or partitions). To take advantage of it, we follow a strategy of incremental updating. In other words, merely those affected parts will be updated.

The incremental algorithm is highly restricted in local parts rather than the overall topology. Hence, the worst case of the algorithm is degrading into applying Algorithm 1 on the basis of the entire network.

The incremental updating algorithm is implemented in Algorithm 2. The Algorithm is described in the following two cases.

Addition of new flow. If a new flow comes into the topology, we first compute the set of partitions *affected* on account of the specific new flow's path. By affected, meaning the new flow passes through the domain of some existing partitions. After that, different operations are carried out based on the number of partitions that are affected. More details are in Algorithm 2.

Completion of flow. Whenever an about-to-finish flow leaves the network, its own partition may split into several parts if the

only connection is the leaving flow. In this case, we calculate the remaining number of flows in the leaving flow's partition and do operations based on the exact number of remaining flows. If the remaining number is larger than 1, we use the Function *network_partitioner* in Algorithm 1 to reconstruct the local part of network. Otherwise, simply delete the flow in its partition.

C Proof of Theorem 1

Proof. When R is stable, it can be approximated by a fixed value $\overline{R}$ during computation, which is:

$$R(t)\approx\overline{R},\quad |R(t)-\overline{R}|<\varepsilon_R,\quad when\ t_s\le t\le t_f,\quad \Delta R<\varepsilon_R \tag{10}$$

This simplification is justified by the negligible variations in the rate during this period, rendering the rate effectively uniform across the interval. Such an approximation facilitates subsequent proofs and derivations by reducing the complexity of the analysis and providing a more tractable framework for understanding the system's behavior.

CWND. For *cwnd*, according to HPCC [44], we have

$$cwnd=R\times\overline{RTT} \tag{11}$$

where $\overline{RTT}$ represents smooth RTT. When Equation 3 holds, by applying Equation 1, we have

$$\begin{aligned}\Delta cwnd&=\max_{t_s\le t\le t_f}\{cwnd(t)\}-\min_{t_s\le t\le t_f}\{cwnd(t)\}\\&=\overline{RTT}(\max_{t_s\le t\le t_f}\{R(t)\}-\min_{t_s\le t\le t_f}\{R(t)\})\\&=\overline{RTT}\Delta R\\&<\varepsilon_R\overline{RTT}\\&:=\varepsilon_{cwnd}.\end{aligned} \tag{12}$$

This observation implies that as the rate R enters a steady-state, the fluctuations of *cwnd* will correspondingly diminish to within another threshold.

RTT. For RTT, TIMELY [54] updates the target rate using $\Delta RTT(t)=\frac{dRTT(t)}{dt}$ as follows:

$$R(t+\Delta t)=\begin{cases}R(t)+\delta & \text{if } \Delta RTT(t)\le 0\\ R(t)(1-\beta\frac{dRTT(t)}{dt}) & \text{if } \Delta RTT(t)>0\end{cases}$$

Using Equation 10, we have

$$\begin{aligned}&R(t+\Delta t)-R(t)\\&=\begin{cases}\delta & \text{if } \Delta RTT(t)\le 0\\ -\beta R(t)\frac{dRTT(t)}{dt}\approx-\beta\overline{R}\Delta RTT(t) & \text{if } \Delta RTT(t)>0\end{cases}\end{aligned}$$

when $\Delta RTT(t)=\frac{dRTT(t)}{dt}>0$, by using Equation 3, we have

$$\Delta RTT(t)=|\frac{R(t+\Delta t)-R(t)}{\beta\overline{R}}|<\frac{\varepsilon_R}{\beta\overline{R}}$$

When $\Delta RTT\le 0$, here $R(t)\le C$, we have

$$\Delta RTT(t)=\Delta q(t)=\frac{(R(t)+\delta-C)\Delta t}{C}\le\frac{\delta\Delta t}{C} \tag{13}$$

where C is the represents the bandwidth allocated upon convergence, and $q(t)$ is the queuing delay through the bottleneck queue. Although Equation 13 shows that the fluctuation may be time-related, and it seems that the calculation of ΔRTT might be related to the cumulative time in the steady period, the actual situation is not the case. Considering the speed adjustment mechanism of TIMELY AIMD, during the steady period, an increase in R that causes $\Delta RTT(t)\ge 0$ triggers a Multiplicative Decrease, followed by a series of consecutive Additive Increases until another increase in R again causes $\Delta RTT\ge 0$. Therefore, it can be considered that within the steady period, the maximum value of RTT comes from an increase in R that causes $\Delta RTT(t)\ge 0$, and the minimum value of RTT comes from this Multiplicative Decrease adjustment. Hence, using Equation 13, there exists a t' such that:

$$\Delta RTT=\Delta RTT(t')<\frac{\delta\Delta t'}{C}$$

where $\Delta t'$ is the interval between two rate adjustments during congestion, which is capped at a specific upper limit. Thus, we have:

$$\Delta RTT<max(\frac{\varepsilon_R}{\beta\overline{R}},\frac{\delta\Delta t'}{C}):=\varepsilon_{RTT} \tag{14}$$

for all $t_s\le t\le t_f$. Consequently, ΔRTT is confined within a constant upper limit.

Queue length. For queue length Q, according to DCTCP [3], we have

$$Q(t)=NW(t)-C\times RTT \tag{15}$$

where there are N flows on the bottleneck link, and $W(t)$ represents the window size. By calculating the fluctuation of Q using Equation 1, we have

$$\begin{aligned}\Delta Q&=\max_{t_s\le t\le t_f}\{Q(t)\}-\min_{t_s\le t\le t_f}\{Q(t)\}\\&=\max_{t_s\le t\le t_f}\{NW(t)-C\times RTT\}-\min_{t_s\le t\le t_f}\{NW(t)-C\times RTT\}\\&=N\times RTT(\max_{t_s\le t\le t_f}\{R(t)\}-\min_{t_s\le t\le t_f}\{R(t)\})\\&=N\times RTT\Delta R\\&<N\times RTT\varepsilon_R\\&:=\varepsilon_Q\end{aligned} \tag{16}$$

Consequently, the queue length Q is also stable when R is stable.

In-flight bytes. For in-flight bytes I, according to HPCC [44], we have

$$I=Q+R\times RTT_{base} \tag{17}$$

By calculating the fluctuation of I using Equation (1), we have

$$\begin{aligned}\Delta I &= \max_{t_s\le t\le t_f}\{Q(t)+R(t)RTT_{base}\}-\min_{t_s\le t\le t_f}\{Q(t)+R(t)RTT_{base}\}\\ &\le(\max_{t_s\le t\le t_f}\{Q(t)\}-\min_{t_s\le t\le t_f}\{Q(t)\})\\ &\quad+RTT_{base}(\max_{t_s\le t\le t_f}\{R(t)\}-\min_{t_s\le t\le t_f}\{R(t)\})\\ &=\Delta Q+RTT_{base}\Delta R\\ &<\varepsilon_Q+RTT_{base}\varepsilon_R\\ &:=\varepsilon_I \end{aligned} \tag{18}$$

This implies that when R remains stable, I also maintains stability.

In summary, by Equations 12, 14, 16 and 18, we have proved Theorem 1. □

D Proof of Theorem 2

Proof. During the steady period, by employing Equation 6, we obtain

$$|R(t)-\overline{R}|\le|\max_{1\le k\le l}\{R(t_k)\}-\min_{1\le k\le l}\{R(t_k)\}|=\hat{R}\Delta R_l(t)<\theta\hat{R} \tag{19}$$

for every single $R(t)$ in the steady period. Consequently, using Equation 19, the error in estimating $\overline{R}$ using $\hat{R}$ is given by:

$$\hat{\varepsilon_R}=|\frac{\hat{R}-\overline{R}}{\overline{R}}|=|\frac{\sum_{k=1}^{l}(R(t_k)-\overline{R})}{l\overline{R}}|\le\frac{\sum_{k=1}^{l}|R(t_k)-\overline{R}|}{l\overline{R}}<|\frac{l\theta\hat{R}}{l\overline{R}}|=\frac{\theta\hat{R}}{\overline{R}}$$

Thus we have

$$|\frac{\hat{R}}{\overline{R}}-1|<\theta\frac{\hat{R}}{\overline{R}}$$

By solving $\frac{\hat{R}}{\overline{R}}$, we have

$$\frac{1}{1+\theta}<\frac{\hat{R}}{\overline{R}}<\frac{1}{1-\theta} \tag{20}$$

Thus, applying Equation 20 we have

$$\hat{\varepsilon_R}=|\frac{\hat{R}}{\overline{R}}-1|<\max(\frac{\theta}{1+\theta},\frac{\theta}{1-\theta})=\frac{\theta}{1-\theta}$$

which demonstrates Theorem 2. □

E Proof of Theorem 3

Proof. Assuming the remaining data volume of the flow upon entering the steady-state is F, we have the following for the actual and estimated cases:

$$F=\frac{\int_{t_1}^{t_l}R(t)dt}{t_l-t_1}\overline{T}=\overline{RT}$$

$$F=\frac{\sum_{k=1}^{l}R(t_k)}{l}\hat{T}=\hat{R}\hat{T}$$

By combining these two equations, we obtain

$$\hat{T}=\frac{\overline{R}}{\hat{R}}\overline{T}$$

Using Equation 20, we have

$$1-\theta<\frac{\overline{R}}{\hat{R}}<1+\theta$$

Consequently, we have

$$\hat{\varepsilon_T}=|\frac{\hat{T}-\overline{T}}{\overline{T}}|=|\frac{\hat{T}}{\overline{T}}-1|=|\frac{\overline{R}}{\hat{R}}-1|<\theta$$

which demonstrates Theorem 3. □

F Threshold Guidance

Range of θ. For accurate identification of the steady-state, the fluctuation threshold within the identification interval should be slightly greater than the rate fluctuation within the steady-state, that is,

$$\theta\gtrsim\varepsilon_{relative} \tag{21}$$

Based on the DCTCP [3] model, the following formulas are derived:

$$D=(W^*+1)\frac{\alpha}{2},\quad \alpha\approx\sqrt{\frac{2}{W^*}},\quad W^*=\frac{C\times RTT+K}{N},$$
$$K>\frac{1}{7}(C\times RTT),\quad T_C=\sqrt{\frac{C\times RTT+K}{2N}}$$

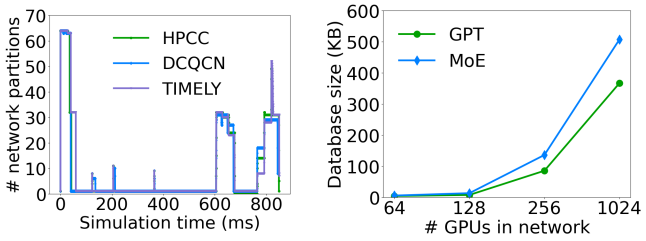
Thus, for $\varepsilon_{relative}$, we have

$$\begin{aligned}\varepsilon_{relative}&=\frac{D}{(max(W)-min(W))/2}=\frac{(W^*+1)\frac{\alpha}{2}}{(W^*+1)(1-\frac{\alpha}{4})}=\frac{2\alpha}{4-\alpha}\\ &\approx\frac{\alpha}{2}\approx\sqrt{\frac{1}{2W^*}}=\sqrt{\frac{N}{2(C\times RTT+K)}}<\sqrt{\frac{7N}{16(C\times RTT)}}\end{aligned}$$

When Equation 21 holds, the boundary for θ is determined as follows:

$$\theta\gtrsim\sqrt{\frac{7N}{16(C\times RTT)}} \tag{22}$$

When θ is less than $\varepsilon_{relative}$, the steady-state identification algorithm may never detect the steady-state, resulting



(a) Number of network partitions (b) Storage space of the database

Figure 15: Number of network partitions and database size in simulating LLM training.

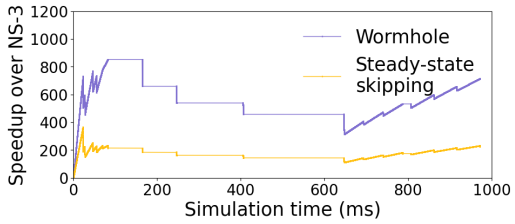


Figure 16: Benefit of Wormhole over simulation progress.

in no acceleration. Conversely, when θ is excessively large, unsteady-states may be misidentified as steady-states, leading to erroneous rate estimation. Hence, θ should be selected as a value slightly greater than, but close to, the relative rate fluctuation within the steady-state.

Range of l. To obtain an accurate $\Delta R_l(t)$ in Equation 6, the identification interval should be able to cover at least one period of the steady-state rate presentation, which requires l to be sufficiently large such that:

$$\Delta t(l) \geq T_C \tag{23}$$

which we have:

$$\Delta t(l) = t_l - t_1 \geq T_C > \sqrt{\frac{4C \times RTT}{7N}} \tag{24}$$

When $l < T_C$, the estimated rate will have a significant deviation. When l is excessively large, it will reduce the efficiency of entering the steady-state and decrease the acceleration ratio. Therefore, l should be set to an appropriate multiple of T_C.

G Experiment: Number of Network Partitions

Figure 15a shows the evolution of network partition counts over time in a 64-GPU scenario under various CCAs. Although the completion times of flows differ slightly among different CCAs, the results of the network partitioning algorithm remain essentially consistent. This suggests that the partitioning algorithm is independent of the specific CCA employed.

H Experiment: Database Storage Cost

As illustrated in Figure 15b, the simulation database stores only a limited amount of critical information, thereby significantly conserving storage space. In a scenario with 1024 GPUs, it occupies less than 100KB of space. Consequently, the database can be entirely put into memory, which enhances the efficiency of database queries and insertions operations. In addition, the scalability of Wormhole is consistent with ns-3.

I Experiment: Speedup over the Simulation Progress

Figure 16 illustrates the temporal progression of Wormhole speedup ratio over simulation time for a 64-GPU configuration. The speedup metric quantifies the quotient of events processed by ns-3 divided by events processed by Wormhole. DP flow scenarios with larger flow sizes and complex workload in the initial and final phases amplify Wormhole performance advantage, while PP flow scenarios with smaller flow sizes in the intermediate phase reduce the average speedup. Over time, memoization accumulates performance benefits.