UNISON: A Parallel-Efficient and User-Transparent Network Simulation Kernel

Songyuan Bai[★], Hao Zheng[★], Chen Tian[★], Xiaoliang Wang[★], Chang Liu[★], Xin Jin[†], Fu Xiao[△], Qiao Xiang[◇], Wanchun Dou[★], Guihai Chen[★]

[★]State Key Laboratory for Novel Software Technology, Nanjing University, China [†]Peking University, China
[△]Nanjing University of Posts and Telecommunications, China [◇]Xiamen University, China

Abstract

Discrete-event simulation (DES) is a prevalent tool for evaluating network designs. Although DES offers full fidelity and generality, its slow performance limits its application. To speed up DES, many network simulators employ parallel discrete-event simulation (PDES). However, adapting existing network simulation models to PDES requires complex reconfigurations and often yields limited performance improvement. In this paper, we address this gap by proposing a parallel-efficient and user-transparent network simulation kernel, UNISON, that adopts fine-grained partition and load-adaptive scheduling optimized for network scenarios. We prototype UNISON based on ns-3. Existing network simulation models of ns-3 can be seamlessly transitioned to UNISON. Testbed experiments on commodity servers demonstrate that UNISON can achieve a 40× speedup over DES using 24 CPU cores, and a 10× speedup compared with existing PDES algorithms under the same CPU cores.

CCS Concepts: • **Networks** → **Network simulations**; • **Computing methodologies** → **Discrete-event simulation**; *Massively parallel and high-performance simulations.*

Keywords: Network simulation, Parallel discrete-event simulation, Data center networks

ACM Reference Format:
Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, Guihai Chen. 2024. UNISON: A Parallel-Efficient and User-Transparent Network Simulation Kernel. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3627703.3629574

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629574

1 Introduction

Discrete-event simulation (DES) is a prevalent tool for evaluating network designs among various simulation approaches. Popular network simulators including ns-3 [30], OMNeT++ [31] and ns.py [18] are all based on DES. Many influential studies on networks, such as DCTCP [4], DCQCN [44], TIMELY [27], PowerTCP [2] and ABM [1] have used these DES-based simulators to provide convincing evaluation results. The full-fidelity accuracy and packet-level details of DES make it the *de facto* ground truth for network performance estimation [9, 36, 40, 42, 43]. The extensive generality of DES also enables researchers to understand network behaviors under buggy configurations (*e.g.*, BGP route leak), heavy workload (*e.g.*, DDoS attacks) and other extreme scenarios (*e.g.*, TCP incast), which are costly to produce and analyze in the real world.

Although DES offers full fidelity and generality, its slow performance limits its application. As the scale of current networks evolves, such as expanding data center networks (DCNs), the simulation time for these giant networks is unacceptably long for DES. Based on our experience, simulating a 1536-host fat-tree connected with 100Gbps links for only 0.1 seconds using the DES kernel of ns-3 on a modern server CPU can take over a day to complete. Moreover, deploying technologies into these giant networks will counter phenomena that a small-scale simulation or testbed cannot reflect [12]. For example, the performance of congestion control algorithms suffers considerable downgrades in large-scale DCNs with a large amount of bursty traffic [14]. Therefore, a full-scale DES for these giant networks is necessary to understand their performance.

To speed up the DES, many network simulators offer parallel discrete-event simulation (PDES). PDES requires the simulated network topology to be spatially divided into multiple partitions before the simulation starts. Each partition is called a logical process (LP). A synchronization mechanism between LPs is required to ensure the correctness of the whole simulation progress [10, 23]. As the simulated network is divided into multiple LPs, PDES can improve the simulation speed by processing these LPs in parallel.

However, adapting existing network simulation models to PDES requires complex reconfigurations and often yields limited performance improvement. Therefore, many researchers still prefer the slow DES instead of PDES [23]. In this paper,

we perform a deep profiling of existing PDES algorithms based on ns-3. We argue that there are two major limitations of the existing PDES algorithms for networks:

- *Complex manual configurations.* Existing algorithms require extensive modifications to model code, due to the manual partition on the simulated network and the aggregation of results collected by each LP. The configurations are inflexible, rely on experience, and can directly compromise the parallel performance.
- *Slow speedup.* The existing static partition scheme and synchronization mechanisms of PDES are not well-suited for emerging low-latency, high-bandwidth, and large-scale networks, leading to significant synchronization overhead and parallel inefficiency.

These two limitations make PDES inflexible and infeasible to scale up [29, 32]. To avoid this problem, recent works cleverly borrow advances in machine learning (ML) to make the simulation faster with GPU [9, 36, 40, 42]. Although these ML-based data-driven methods can obtain satisfactory execution efficiency in well-trained scenarios, their generality is still limited. For new scenarios and algorithms, it is difficult to perform the data-driven methods since there is no sufficient data to train the ML models. Moreover, they still suffer from 7 to 12 hours-long training time and can only obtain approximated results [40, 42].

In this paper, we address this gap by proposing a parallel-efficient and user-transparent network simulation kernel, named Unison. The main idea of Unison is to perform automatic, fine-grained partition and dynamic, load-adaptive scheduling optimized for network scenarios in PDES. For fine-grained partition, Unison divides the simulated network into fine-grained LPs automatically before the simulation starts to improve cache efficiency and for further scheduling. With this approach, Unison frees users from complex manual configurations when setting up the simulated network. For load-adaptive scheduling, Unison decouples the relationship between LPs and processor cores, leaving the load-balancing job among each core to its scheduler. The scheduler of Unison dynamically balances the workload with several heuristic methods based on network characteristics. With this approach, all processor cores can finish their events *in unison*, resulting in an efficient parallelization under various topologies and traffic patterns.

We prototype Unison based on ns-3 and address several practical challenges. We improve the performance and usability of Unison by introducing a lock-free execution workflow on a shared-memory architecture, which is fully transparent to users. With this approach, Unison can support dynamic topologies (*e.g.*, reconfigurable DCN [8]) and obtain global statistics (*e.g.*, FCT). By introducing a tie-breaking rule for simultaneous events, these collected statistics are also deterministic and reproducible. To scale Unison on a cluster, we also implemented a hybrid simulation kernel for distributed simulation. As a preview of our work, Figure 1 shows that Unison can achieve over 10× speedup compared with the existing PDES algorithms, turning over two days long DES into less than 2 hours. Moreover, all existing network simulation models included with ns-3 can be seamlessly transitioned to Unison to obtain performance improvements.

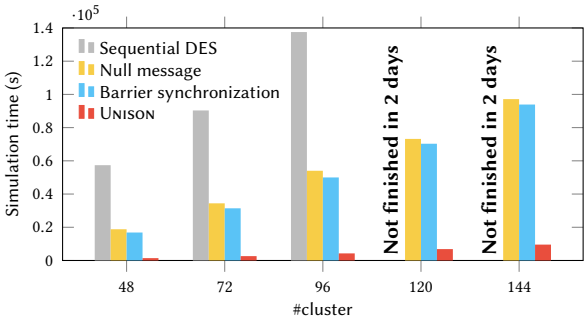


Figure 1. Simulating fat-trees with 100Gbps links under incast traffic for 0.1s using Unison, alongside sequential DES and other PDES algorithms [7, 10]. All PDES algorithms use the same number of processor cores which is set to the number of clusters. Each cluster contains 16 hosts.

This paper contributes to the field of network simulation and network performance estimation through the following:

- A deep profiling of existing PDES algorithms, highlighting important observations about the root cause of their complex configurations and unsatisfactory performance: static manual partition on the simulated network (§3).
- A fine-grained partition scheme and a load-adaptive scheduling strategy for PDES, optimized with network scenarios, achieving a 40× speedup over DES using 24 CPU cores, and a 10× speedup over existing PDES algorithms (§4).
- A user-transparent and lock-free implementation of Unison on ns-3, addressing several practical challenges to further improve its performance and usability (§5).
- An extensive evaluation for Unison compared with existing PDES algorithms and data-driven approaches, demonstrating its high performance, scalability and usability under various topologies and traffic patterns (§6).

Our implementation of Unison is open-sourced[1]. This work does not raise any ethical issues.

2 Background

In this section, we first provide a background on discrete-event network simulation (§2.1). We then discuss two acceleration methods: newly emerged data-driven approaches (§2.2) and existing PDES algorithms (§2.3).

2.1 DES for Networks

In network simulation, a network topology is modeled as a graph with nodes and links. Each node represents a host or

[1] https://github.com/NASA-NJU/UNISON-for-ns-3

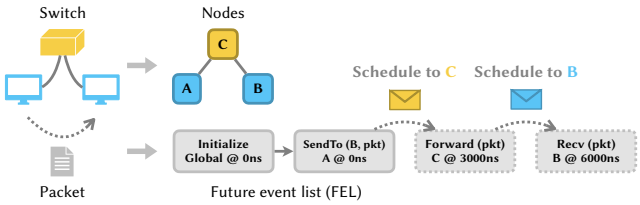


Figure 2. A toy example of DES. The topology is modeled as three nodes A, B, and C. The packet sending from A to B is modeled as a series of discrete events.

a switch. The sending, forwarding, and receiving of a packet between nodes is modeled as a series of discrete events stored in the future event list [34], as illustrated in Figure 2.

Discrete event. In network simulation, a discrete event consists of a timestamp, a node ID, and a callback function, indicating when, where, and what happens respectively. In the callback function, state variables of the corresponding node (*e.g.*, queue length) may be modified, and new events may be scheduled into the future event list [10]. The processing time of a discrete event is determined by the complexity of its callback function.

Future event list. To maintain a correct order of events, the simulator uses a future event list (FEL) to store discrete events. The FEL is a priority queue, with the event having the smallest timestamp at the head of the queue [10].

Sequential simulation. As the simulator runs, it pops an event from the head of the FEL and processes it. When the event is processed, the simulator advances its clock to the timestamp of the event [10]. The simulator terminates if it encounters a stop event or if the FEL is empty (*i.e.*, all events have been processed). Since events in the FEL are ordered by timestamps, causality and correctness are guaranteed.

However, the sequential approach is depressingly slow when simulating large-scale networks. For example, there are over 550 million events per simulated second for a $k = 8$ fat-tree with 100Gbps links, which takes nearly 2 days to simulate for only 1 second based on our observation.

2.2 Data-Driven Approaches

Different from the full-fidelity DES, data-driven simulators such as MimicNet [42] and DeepQueueNet [40] use ML to approximate behaviors of large networks. They replace a set of nodes in the simulated network with a simplified black box to be trained. In the black box, deep neural networks (DNN) are used to approximate the protocol stack. The training data is generated via a small-scale simulation or by collecting traces from physical devices. However, the applicability and performance of these approaches are still limited.

Limited usability. Due to the simplification of the black box, these data-driven approaches have limited use cases

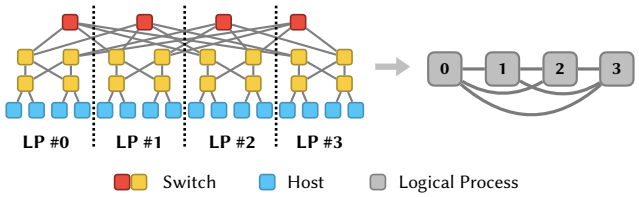


Figure 3. The static partition scheme of a $k = 4$ fat-tree.

compared with DES. For MimicNet, it is only applicable to fat-tree topologies [42]. For DeepQueueNet, it ignores state variables of the protocol stack, making it unable to simulate stateful behaviors such as TCP congestion control [40]. Moreover, they are only applicable to networks at the stable point. MimicNet cannot model skewed traffic between fat-tree clusters [42] while DeepQueueNet cannot reflect temporal dynamic behaviors of networks [40].

Long training time. In addition to their limited usability, training the DNN in the black box also takes a long time. For MimicNet, it takes about 7 hours to train a single fat-tree cluster. When the workload or the structure of the cluster changes, the model must be retrained [42]. For DeepQueueNet, training a single type of switch can take 12 hours, yet all types of switches in the model must be trained before simulation [40].

Still relying on DES. For machine learning, sufficient training data is required to achieve satisfactory accuracy, but it is costly to obtain such data from real physical devices. Most new scenarios, such as implementing a new switch or designing a new protocol, do not have such previously known data. As a result, these data-driven methods still rely on DES to obtain labeled training and testing data. MimicNet uses DES to simulate a cluster with full fidelity to train other mimics [42], while DeepQueueNet uses ns.py to obtain training data for its switches [40]. Therefore, DES is still an irreplaceable tool in network study. Improving the performance of DES can further facilitate these data-driven approaches.

2.3 PDES Algorithms

The sequential DES can be parallelized by dividing the network topology spatially into logical processes (LPs). Figure 3 is an example of dividing a $k = 4$ fat-tree into 4 LPs. Each LP runs the sequential DES with its own FEL. A synchronization mechanism is required to ensure that the causality (*i.e.*, the order of timestamps) of events is not violated [10, 23].

In network simulation, due to the inevitable propagation delay of links, there is always a time window during which packets are on the wire. For the receiving LP, processing events in its FEL within the time window (*i.e.*, before these packets arrive) does not violate the causality of these events, as illustrated in Figure 4. Based on this concept, a lookahead value can be calculated for each LP, which is the shortest delay of all links connected to other LPs [10, 23].

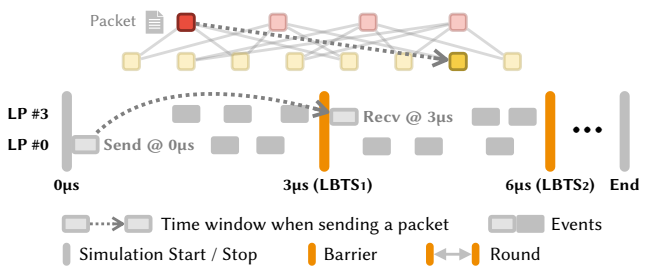


Figure 4. The time window and the barrier synchronization algorithm, assuming the link delay between core switches and aggregation switches is 3μs.

The default PDES algorithm implemented in ns-3 is the barrier synchronization algorithm [10, 34]. In this algorithm, all LPs are executed in rounds separated by barriers. At the start of each round, each LP calculates its time window, called the lower bound on the time stamp (LBTS) as

$$LBTS = \min\{N_i\} + lookahead \quad (1)$$

where N_i represents the timestamp of the next event of the i-th LP. LPs can safely execute events whose timestamps do not exceed the LBTS. Then, all LPs have to perform a barrier synchronization, waiting for other LPs to complete before entering the next round [10], as shown in Figure 4.

In addition to this algorithm, OMNeT++ and ns-3 also implement PDES with the null message algorithm [7]. In this algorithm, LPs are synchronized locally via null messages instead of global barriers. Similar to the barrier synchronization algorithm, this algorithm uses the lookahead value to ensure causality as well.

Another approach for PDES is to allow for the violation of causality while providing a rollback mechanism [10, 20]. However, it requires a significant re-architecture of existing simulators due to the implementation of state saving for rollbacks or user-provided rollback methods [20], which is also not user-transparent for the latter case. Most network simulators including ns-3 and OMNeT++ do not implement this approach. Therefore, we focus on the previous two algorithms for optimization and comparison.

3 Why Don't We Use PDES in Practice

Open-source artifacts of network designs rarely use PDES for evaluation in recent 3 years except a few [11, 25, 40, 42]. Through the following analysis, we find that the main obstacles to the application of PDES are complex manual configurations (§3.1) and slow speedup (§3.2).

3.1 Complex Manual Configurations

When adapting a DES model of network to PDES, the following additional steps are required.

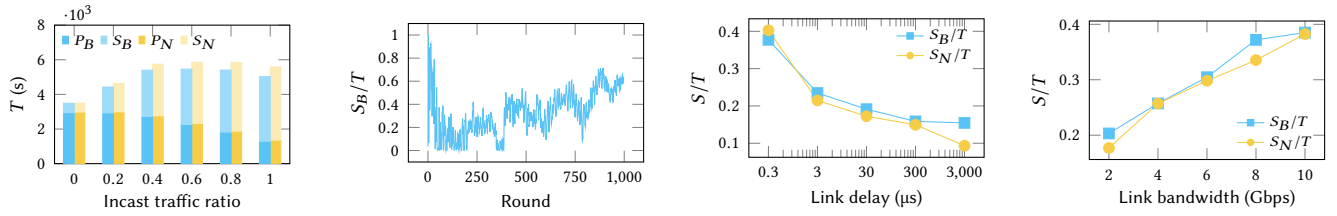
Dividing the network topology. As mentioned in §2.3, PDES requires us to divide the network topology manually into a set of LPs. For efficient parallelization, it is necessary to balance the workload of each LP. However, since the topology and workload in different simulation models can vary, there is no panacea for achieving optimal partition results in every scenario [39, 41]. This means that the partition is inflexible and heavily relies on experience. For example, in the case of a k-ary fat-tree topology [3] with a balanced workload, we can perform a symmetric partition where each pod is treated as an LP, and the core layer is evenly distributed among the clusters, as shown in Figure 3. In this case, the number of LPs is fixed at k, meaning that the maximum possible speedup is also fixed at k fold. If the current hardware resources do not have enough slots to launch k processes, or if we wish to further increase the speedup of the model, we must perform another partition, which can cause parallel inefficiency due to the spatial asymmetry of the new partition scheme. Additionally, if the simulated traffic pattern is not balanced among the LPs (*e.g.*, an incast scenario), even a symmetric partition scheme may not reduce the waiting time. In this case, heuristic static partition methods that try to produce a balanced partition require users to provide hints or pre-run the whole simulation to acquire such hints [31, 38], which leads to extra time cost and also complex configurations.

Table 1. LOC change when adapting sequential DES models to PDES in ns-3.

Model	Fat-tree	BCube	Spine-leaf	2D-torus
LOC addition	36	44	40	33
LOC deletion	21	16	18	20

Collecting the result. PDES network simulators use ghost nodes to obtain global topology knowledge [35]. However, global data knowledge is still not achievable because the application code and the tracing code are separated into different LPs, and each LP cannot see the ongoing traffic of other LPs. As a result, it is difficult to track a flow when it passes through different LPs. To alleviate this issue, we have to either use inter-process communications to pass statistics or temporarily save the results of each LP and manually aggregate them after the simulation is finished. Both of these solutions will require extra effort to create lengthy configuration scripts. More importantly, the simultaneous events during the simulation can make the results indeterministic [21], making it difficult to reproduce the previous results and debug any issues.

Carrying out the aforementioned steps will result in significant code changes, as shown in Table 1. The code modification is complex and error-prone since it requires the user to distinguish different LPs and carefully consider available CPU cores, proper topology division and the specific data collection strategy. Therefore, we conclude that the existing PDES is not user-friendly on existing network simulators because of the manual partition-and-collection process.



(a) P, S, M under skewed traffic **(b)** S/T under balanced traffic **(c)** S/T of different delay **(d)** S/T of different bandwidth

Figure 5. Performance of existing PDES algorithms when simulating a $k = 8$ fat-tree for 0.1 seconds. The subscript B stands for the barrier synchronization algorithm, N stands for the null message algorithm.

3.2 Slow Speedup

Even if users successfully adapted their model to PDES, the speedup is still unsatisfactory. To find out the performance bottleneck of PDES, we divide the total running time T of a single LP into three parts: processing time P, synchronization time S, and messaging time M. So we have $T = P + S + M$. P counts when the LP is processing events from its FEL. S counts when the LP is waiting for other LPs after event processing is finished[2]. M counts when the LP is receiving events from other LPs.

We insert the time profiling code into ns-3 implementations of the two PDES discussed in §2.3, and record the P, S and M of each LP. Through in-depth profiling, we conclude the following three important observations.

Observation 1. *The synchronization time gradually dominates as the traffic inhomogeneity increases.*

We run a $k = 8$ fat-tree with 100Gbps link bandwidth and 3µs link delay for 0.1 second. We divide the topology symmetrically according to Figure 3, where each cluster is an LP. Figure 5a shows the time composition in this experiment. Note that we omit M in Figure 5a since it takes less than 5% of T. We find as the traffic inhomogeneity increases, S becomes a bottleneck on both algorithms. Under an extremely skewed traffic pattern, S can take up over 70% of the total execution time. This is because the LP of the victim host becomes the slowest one in the incast scenario. Other LPs must wait for it to finish before entering the next round.

Observation 2. *The processing time of each LP is highly unbalanced in a transient time window, even if the traffic pattern is balanced in macro.*

Even if the traffic pattern is load-balanced, S still takes about 20% of the total time. To figure out, this time we measure the P and S in each round of the barrier synchronization algorithm under load-balanced traffic. Interestingly, we found that the processing time in each round of each LP is highly unbalanced, resulting in the long synchronization time in a transient time window, as Figure 5b shows. This implies that even heuristic static partition methods that try to produce a balanced partition based on load estimation still have considerable synchronization time.

Observation 3. *The synchronization time is long for low-latency, high-bandwidth and large-scale networks.*

When simulating data center networks, due to the low propagation delay of the links, the time window between two synchronization barriers is smaller. Along with their high bandwidth and traffic throughput, the synchronization time (S) caused by load variation during a transient time window is huge. This is proven by Figure 5c where we simulate a 10Gbps fat-tree with different link delay and Figure 5d where every host sends a total of 128Gbps traffic in fat-trees with 30µs link delay and different link bandwidth. In addition, the large scale of such data center networks will produce more LPs, further increasing the synchronization time. Based on our experience, the S ratio can reach over 50% for a 100Gbps, $k = 16$ fat-tree with 1µs link delay under balanced traffic.

Through the above discussions, we find that the existing synchronization mechanisms based on static partition do not fit well with bursty, dynamically unbalanced traffic and the current high-bandwidth, low-latency, and large-scale networks. Alongside the manual partition discussed in §3.1, we conclude that the static manual partition is the root cause for both complex configurations and slow speedup.

4 Unison Design

In this section, we introduce a new network simulation kernel, Unison, to address the limitations of existing PDES algorithms in §3. The design objectives of Unison are:

- *Parallel-efficient.* The new kernel should reduce the synchronization time efficiently and be applicable to different traffic patterns and different topologies.
- *User-transparent.* The new kernel should be transparent to users for topology partition and result aggregation. It should also let users flexibly specify the number of processor cores to be used for speedup with zero configurations.

We first provide an overview of the workflow of Unison. Then we discuss the fine-grained partition algorithm and the

[2]For the barrier synchronization algorithm, S also includes the time cost of MPI collective communications. However, the cost for such communications is negligible since it is only used to calculate the LBTS.

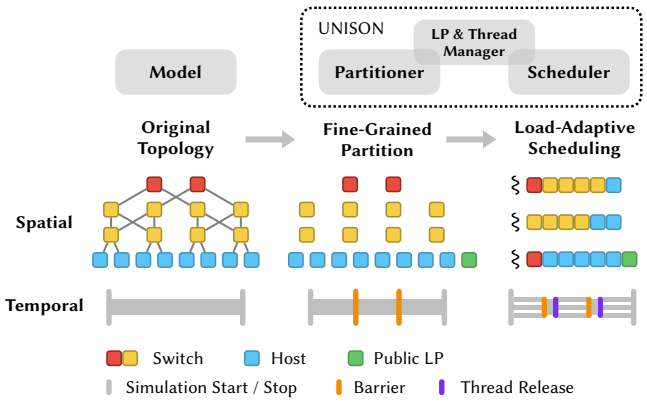


Figure 6. The architecture and workflow of UNISON.

load-adaptive scheduling algorithm that enables its parallel-efficient performance and user-transparent characteristics.

4.1 System Overview

To achieve the design objectives, UNISON decouples the relationship between LPs and processor cores. UNISON then performs fine-grained partition (§4.2) and load-adaptive scheduling (§4.3) workflow, which is illustrated in Figure 6.

In this workflow, UNISON first performs a 2-dimensional partition automatically for user transparency. This divides the topology into a set of LPs, and divides the simulated time into multiple rounds, within which LPs are parallelly processable according to a modified barrier synchronization algorithm. For parallel efficiency, UNISON divides the topology with fine granularity. Fine-grained partition is beneficial to the scheduler by creating more LPs with smaller sizes, making it more likely to produce a balanced result. Meanwhile, since LPs have to process their FEL in time order, fine-grained partition can reorder spatially related but temporally isolated simulated packets of a node by extracting them from a larger LP and grouping them to be processed together in a smaller LP to increase cache affinity. In network simulation, a simulated flow will correspond to consecutive simulated packets (*i.e.*, events) within a time window. Inspired by the cache boost of network stacks and applications by reordering packets in real-world NIC [13], grouping these simulated packets can also improve the performance of network simulation. Since the node IDs of consecutive events are not always the same for each LP, each LP will frequently bounce between different nodes. As the number of LPs increases, each LP will have a smaller range of nodes, thus decreasing cache misses.

After partitioning, the load balancing job is left for the scheduler to handle. The scheduler uses a thread pool to process decoupled LPs. To address the synchronization time issues discussed in Observation 1, we have to figure out the optimally balanced workload of each thread, which can be abstracted as the multiway number partitioning problem. Although the optimal solution is NP-hard, UNISON employs an approximation algorithm that uses the longest job first policy combined with a heuristic approach for network simulation. To address the transient unbalanced workload issue discussed in Observation 2 and Observation 3, the scheduler balances the workload of each thread by assigning LPs to different cores dynamically in each round according to this approximation algorithm.

By combining these techniques above, we directly eradicate the root cause for limitations of PDES discussed in §3. This is because the whole partition and scheduling process is transparent to users, allowing them to easily simulate models in parallel without further configurations. Meanwhile, cache misses are reduced by fine-grained partition, and the mutual waiting time among threads is minimized by load-adaptive scheduling, resulting in efficient parallelization.

4.2 Fine-Grained Partition

The objective of this stage is to automatically divide the simulation model both spatially and temporally. For spatial partition, it should be efficient for scheduling while preserving parallelism under different link delays. For temporal partition, the causality should be guaranteed by utilizing lookahead, while supporting specific demand for network simulation such as dynamic topologies.

Spatial Partition. The partition algorithm is presented in Algorithm 1. The algorithm takes a network topology as input, assigns each node its LP identifier, and outputs the

Algorithm 1: FINE-GRAINED-PARTITION

```
Data: Network topology: G(V, E)
begin
    lookahead-lowerbound ← MEDIAN-DELAY(E);
    LP-count ← 0;
    q ← MAKE-EMPTY-QUEUE();
    for v in V do
        if not v.visited then
            LP-count ← LP-count + 1;
            ENQUEUE(q, v);
            while not EMPTY(q) do
                v ← DEQUEUE(q);
                v.LP-id ← LP-count;
                v.visited ← true;
                for (u, v) in E do
                    if not u.visited and LINK-DELAY(u, v)
                     < lookahead-lowerbound then
                        ENQUEUE(q, u);
                    end if
                end for
            end while
        end if
    end for
    return LP-count;
end
```

total number of LPs created. First, a lower bound on the lookahead value is set to the median of all link delays in the topology. This is because cutting off links with a low propagation delay value will lead to a small degree of parallelism. The lower bound is set to the median instead of the average to make sure at least half of the links will be cut off for fine granularity. Then, for every link in the topology, if the link delay value is greater than or equal to the lower bound on the lookahead value, the link should be logically cut off. Cutting off a link logically is used to produce LPs. It will not affect the connectivity in the simulation model. Finally, every node in each connected component forms an LP. Note that the links in this algorithm refer to stateless links, which have no state variables associated with them (*i.e.*, point-to-point links and full duplex Ethernet links). This is because two LPs with different simulated clock times may read and change the state variables simultaneously or in a reversed order in real time, violating causality.

Here is a simple illustration of the partition scheme. Assuming the delay of links between hosts and aggregation switches of the topology in Figure 6 (*i.e.*, the links at the bottom) is set to zero, Unison will produce 10 LPs in total. In contrast, for a space-symmetric static partition scheme, the optimal partition result is to cut the topology in half from the middle, yielding only 2 LPs.

In addition to these automatically generated LPs, Unison introduces a public LP to handle global events. Global events are events that can potentially affect all LPs immediately. They are scheduled by users before the simulation begins or by another global event. Typical global events include changing the topology dynamically, stopping the simulator at a given time, and printing the simulation progress. Existing PDES approaches only support limited global events like stopping the simulator, which are duplicated by every LP to make sure they can coordinate at the same time. For Unison, since the memory is shared across LPs, global events have to be handled just once. This is what the public LP is for. It is equivalent for the public LP to have a zero-delay connection to other LPs. This implies that the lookahead value of the public LP is always zero, only global events with the same timestamp can be handled in the same round.

Temporal Partition. In the temporal dimension, we use a modified barrier synchronization algorithm, which divides the whole simulated time into multiple rounds, within which LPs are parallelly processable. The window size of each round is calculated using the lookahead value. To accommodate the public LP, the equation for calculating the window size, or LBTS, is modified as

$$LBTS = \min\left\{N_{\text{pub}},\ \min\{N_i\} + lookahead\right\} \tag{2}$$

where N_{pub} is the timestamp of the next event of the public LP. N_i is the timestamp of the next event of the i-th LP. Equation (2) considers the constraint introduced by global events. Since global events can potentially affect all LPs, the current round must be interrupted when the next global event in the public LP occurs.

Moreover, since the topology is not static anymore, the lookahead value in Equation (2) can be changed during the simulation when link delay changes, adding or removing a link. Therefore, in addition to Equation (2), Unison will recompute the lookahead value if a topology change occurs when processing the public LP.

4.3 Load-Adaptive Scheduling

The objective of this stage is to assign LPs to multiple parallelly executed threads $\mathcal{T}$ while balancing workloads among these tasks, under the assumption that threads are running on identical CPU cores (*i.e.*, with the same clock frequency). Each LP must be processed exactly once in each round. Assume that the job size, or processing time, of the i-th LP in round r is known in advance and denoted as $P_{i,r}$. This assumption can be satisfied by our proposed scheduling metrics in this section. A thread $t \in \mathcal{T}$ in this round has a set of LPs $\mathcal{L}(t,r)$ assigned to it. Therefore, the total processing time of thread t can be expressed as $\sum_{i\in\mathcal{L}(t,r)} P_{i,r}$. Our goal is to determine the optimal assigning strategy $\mathcal{L}(\cdot,r)$ to minimize the largest total processing time (*i.e.*, minimize $\max_{t\in\mathcal{T}} \sum_{i\in\mathcal{L}(t,r)} P_{i,r}$) for every round r.

Scheduling algorithms. For a given round, the objective can be abstracted to scheduling a set of jobs on $|\mathcal{T}|$ identical machines such that the makespan is minimized, which is equivalent to the identical machine scheduling problem, or multiway number partitioning problem, which is NP-hard [17]. One of the approximation algorithms [15] is the longest job first policy: schedule the longest job to the thread with the smallest load currently. Unison employs this policy and stores LPs in a priority queue according to their estimated processing time $P_{i,r}$ in descending order. When a thread is idle or finished with its previously assigned LP, it pops the queue to get the current longest LP and process that one. With this approach, scheduling n LPs only takes $O(n \log n)$ steps, plus the complexity of acquiring $P_{i,r}$ in each step, with a worse-case approximation ratio of $4/3$ [15].

Scheduling metrics. The exact processing time of each LP, or $P_{i,r}$, is impossible to know in advance. Therefore, estimation is required to use the approximation algorithm. Finding an accurate and low-complexity estimation method becomes a challenge. We propose several heuristic methods suitable for network simulation to estimate this value efficiently.

One way is to evaluate the number of pending events in the next round of each LP, as LPs with more events tend to have a longer processing time. In network simulation, most event scheduling is related to packet transmission. These scheduled events will typically have a delay that is equal to the lookahead, falling into the next round exactly, which is already illustrated in Figure 4. Therefore, we can count the

number of events scheduled to be received by each LP in the next round (*i.e.*, pending events), which can be simply done in linear time. Note that the longest job policy described above only considers the partial order of job size, rather than the exact job size of each LP, so this method will work.

A second heuristic method is to use the processing time of the previous round, $P_{i,r-1}$, as an estimate. This is because network simulation programs often have a high degree of temporal locality. The propagation delay, or lookahead, is far less than the transmission delay of the entire flow. As a result, the processing time of each LP in consecutive rounds tends to remain stable (although highly unbalanced), which we can notice from Figure 13a. The time complexity of this method is only constant, so it is faster than the previous one. Moreover, this method is more accurate than the first one and is used by default in Unison if a high-resolution system clock is available. §6.3 will further evaluate the performance of different scheduling metrics.

Scheduling periods. The scheduler itself will introduce performance costs as well. Although getting the scheduling metrics is optimized to constant time using $P_{i,r-1}$, sorting n LPs by their estimated processing time still takes $O(n \log n)$ steps. When simulating a large network topology, combined with the fine-grained partition scheme, there will be tons of LPs to schedule. To alleviate the cost, Unison runs the scheduler periodically rather than every round.

Due to the temporal locality of network simulation, which we have described above, the processing time of each LP tends to remain stable in consecutive rounds. Therefore, the schedule results should also tend to remain stable. In order to balance the schedule accuracy and the schedule cost, Unison let the schedule period grow logarithmically with respect to the number of LPs. A topology with n LPs has a schedule period of $\lceil \log_2 n \rceil$. §6.3 will further evaluate the performance impact when choosing different scheduling periods.

5 Implementation

In this section, we delve into the implementation detail of Unison. The implementation is based on ns-3.36.1 and consists of approximately 3100 lines of code. Applying Unison to other network simulators based on DES (*e.g.*, OMNeT++) is straightforward and is undergoing. We also address several practical challenges during the implementation.

The first challenge is to resolve thread-safety issues while minimizing the overhead. To achieve this, Unison has been carefully implemented to ensure lock-free execution, allowing the recording of global statistics seamlessly (§5.1).

The second challenge is to achieve determinism and scalability. We introduce a tie-breaking rule of simultaneous events for deterministic, reproducible simulation. we also implement a hybrid simulation kernel with Unison for scalable, distributed simulation (§5.2).

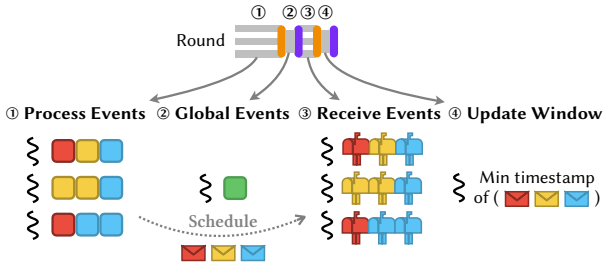


Figure 7. Four phases of the lock-free execution stage.

5.1 Lock-Free Execution

The objective of the lock-free execution is to allow threads to process LPs in a lock-free manner, either through the use of atomic operations or by circumventing the need for synchronization. Using such a manner can reduce P by alleviating the cost of accessing shared data, which happens frequently during the simulation. It can also reduce S by reducing the overhead of the scheduler, and reduce M by reducing the overhead of cross-LP event insertion.

To achieve this objective, Unison divides each round into four phases as depicted in Figure 7. During phase changes, a barrier or a thread release is required, which are both implemented using atomic operations.

Processing events. In this phase, Unison first runs the scheduler to assess the priority of LPs and assign LPs to threads as described in §4.3. However, when scheduling inter-LP events, thread-safety issues arise as the FEL of the target LP can be manipulated by multiple threads. To avoid this situation, Unison uses mailboxes as a cache for inter-LP events. Before the simulation starts, each LP creates a mailbox for any LP it has a connection with. Inter-LP events are first stored in one of the mailboxes of the target LP. These cached events will be subsequently inserted into the FEL during the receiving event phase, making the whole event scheduling process asynchronous and lock-free.

In addition to inter-LP event scheduling, the underlying architecture of ns-3 is not thread-safe as well. We perform the following modifications to ns-3 for thread safety:

- *Reference counting.* Reference counting [19] is used for automatic memory management. We replace the reference counter of ns-3 objects, packet tags and packet buffers with an atomic integer counter. We also disable the lookup cache of aggregated objects.
- *Buffer recycling.* Packet buffers and metadata are recycled via a global linked list to reduce memory allocation calls. We disable this mechanism by allocating/freeing memory every time a new buffer is created/deleted.
- *Flow monitor.* The flow monitor is widely used for flow tracking [6]. It saves statistics of tracked packets and flows into maps shared across nodes. Since new flows and packets can be inserted into the map concurrently, we use atomic operations to make these maps thread-safe.

- *NIx-vector routing*. NIx-vector routing speeds up route decisions with a neighbor index cache [33]. The cache is shared globally and will be re-calculated if it is dirty. We replace the dirty state variable with a boolean atomic variable and use atomic operations to protect the cache.

Handling global events. In this phase, Unison checks the public LP for any global events at the current time and processes them with the main thread. Unison will recompute the lookahead value if a topology change occurs when processing global events as discussed in §4.2.

Receiving events. In this phase, LPs are reallocated to threads according to their priority in the first phase. Each LP then retrieves all events from its mailboxes and inserts them into the FEL. The FEL orders the inserted events automatically by their timestamps. In the case of two events having the same timestamp, they are ordered using the tie-breaking rule (§5.2) for deterministic simulation.

Updating the window. The last phase is to calculate and update the LBTS of every LP of the next round according to Equation (2). If there are no more events to be executed for all LPs, the simulation terminates.

5.2 Improving Usability

Deterministic simulation. To address the indeterminacy caused by simultaneous events, Unison introduces the following tie-breaking rule. When scheduling an inter-LP event, the timestamp of the sender LP and the event ID (which indicates the number of events created in the current LP) are sent along with the event itself. If the timestamps of two events are the same, the event with a smaller timestamp of the sender LP is processed first. If the sending timestamps are still the same, the event from the LP with the smaller ID will be processed first. If two events are still scheduled by the same LP, then the event with a smaller event ID will be processed first. As all events in the mailboxes have a total order, Unison is guaranteed to have a deterministic result.

Scalable hybrid simulation. For scalability across multiple hosts, we also implemented a hybrid simulation kernel with Unison. In this approach, the network topology is first divided into several large partitions according to the barrier synchronization algorithm (§2.3) to map each host. Each host then runs its large partition using Unison for further fine-grained partition as illustrated in Figure 6.

For correct synchronization, the receiving event stage is modified to handle inter-host events. After intra-host events are received from the mailboxes, inter-host events are then received by the main thread. When updating the window, the smallest timestamp of the next event of every local LP is calculated first, followed by an all-reduce operation to calculate the global smallest timestamp. Given the global smallest timestamp, the time window of the next round is finally updated using Equation (2).

6 Evaluation

In this section, we evaluate Unison in comparison with prior PDES algorithms and ML-based data-driven approaches. We mainly focus on the following questions:

- Can Unison efficiently reduce the synchronization time while providing a transparent interface for users? (§6.1)
- How is the accuracy of Unison compared with sequential DES and data-driven approaches? (§6.2)
- How do the partition and scheduling algorithms introduced in §4.2 and §4.3 improve the performance of Unison? (§6.3)

Our evaluation uses an optimized build of ns-3.36.1 in three different testbeds:

- *Testbed 1.* To compare the overall performance of Unison with PDES, we run both Unison and other PDES algorithms over six identically configured machines. Each machine has 256GB RAM and two 12-core 2.2GHz CPUs with hyper-threading off.
- *Testbed 2.* To demonstrate the behavior of packets in detail which requires recording P, S and M defined in §3.2, we run both Unison and other PDES algorithms on one machine which has 256GB RAM and two 8-core 2.0GHz CPUs with hyper-threading off.
- *Testbed 3.* For ML-based data-driven approaches, we run their evaluations on one machine with two 28-core 2.0GHz CPUs and 512GB RAM, alongside two NVIDIA A100 GPUs with 40GB high bandwidth memory.

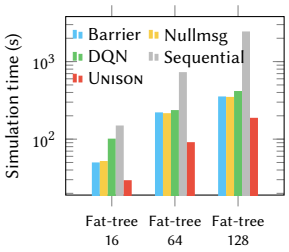
6.1 Unison is Fast and Transparent

With the automatic fine-grained partition scheme and the load-adaptive scheduling, Unison resolves the complicated configuration and slow speedup of PDES simultaneously.

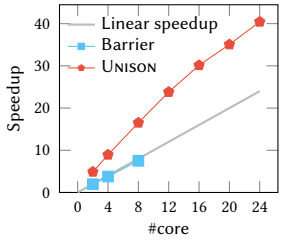
Unison is faster than existing approaches. As shown in Figure 1, we first evaluate Unison, the sequential DES, and PDES algorithms under extremely skewed traffic (*i.e.*, incast traffic ratio is 1). Unison can achieve over 10× speedup with the same number of CPU cores compared with the existing PDES algorithms.

We further evaluate the performance of Unison against an ML-based data-driven simulator, DeepQueueNet [40], under balanced traffic (*i.e.*, incast traffic ratio is 0). We use three fat-tree topologies[3] according to the configuration of DeepQueueNet [40]. The link is 100Mbps with a delay of 500μs. For traditional PDES algorithms, we evenly divide the topologies as shown in Figure 3, which leads to 4 LPs on fat-tree 16, 8 LPs on fat-tree 64 and fat-tree 128. For Unison, we launch 16 threads for all three topologies. For DeepQueueNet, since it supports parallel DNN inference on multiple GPUs, we use both GPUs in Testbed 3. We use

[3]The fat-tree 16 consists of 4 clusters, each of which has 4 hosts (*i.e.*, $k = 4$). The fat-tree 64 consists of 8 clusters, each of which has 8 hosts. The fat-tree 128 consists of 16 clusters, each of which has 8 hosts (*i.e.*, $k = 8$).

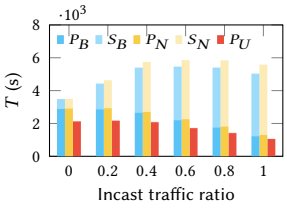


(a) Comparing UNISON against existing PDES algorithms and data-driven approaches.

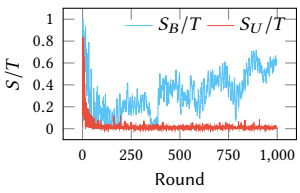


(b) Simulating a 100Gbps, $k = 8$ fat-tree for 1 second with flexible speedup configurations.

Figure 8. Parallel performance and flexibility of UNISON.



(a) P, S, M of UNISON



(b) S/T in each round

Figure 9. The combination effect of the fine-grained partition and the load-adaptive scheduling. The subscript U stands for UNISON. The data of the existing PDES algorithms is the same as those in Figure 5.

the example device model provided by DeepQueueNet and only evaluate its inference time. As the number of injecting packets increases, we find that the DeepQueueNet implementation given by [40] will crash due to the GPU memory overflow. Therefore, we only inject 0.32, 1.28 and 2.56 million packets into the three topologies, respectively. As shown in Figure 8a, UNISON and other PDES simulations are faster than data-driven approaches at a small scale. As the scale of the fat-tree grows, UNISON outperforms DeepQueueNet with full-fidelity simulation and achieves over 13× speedup for sequential DES using 16 threads. It happens because that DeepQueueNet operates at the packet level. Therefore, its simulation time is proportional to the number of packets. The parallel-efficient UNISON can achieve parallelism as high as DNN while avoiding long training time and providing full-fidelity simulation.

UNISON is transparent to users. Since the whole partition and scheduling process of UNISON is automatic, all operations in UNISON kernel are transparent to users. Users only need to import the UNISON kernel, and the simulator's parallelism can be freely adjusted without modifying the original DES model code. We enable UNISON in several ns-3 examples [30], including queue discipline benchmarks, NIx-vector routing examples, multicast examples and a RIP routing model which changes the topology by teardown links during the simulation to observe its convergence. All of these examples are accurately simulated without any modifications to their model code.

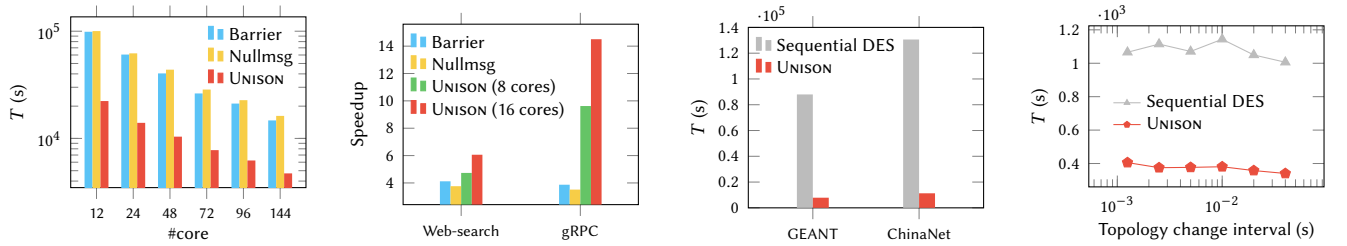
The previous experiment on fat-tree has demonstrated that UNISON can launch 16 threads while the manual partition scheme of the other PDES algorithms can only produce up to 8 LPs for fat-tree 128. To further demonstrate UNISON's flexibility, we increase the number of threads to 24 to fully utilize the hardware resources of Testbed 1. We reset the link bandwidth to 100Gbps and the link delay to 3µs to simulate a high-performance DCN for 1 second. The $k = 8$ fat-tree, which can only be divided symmetrically into 2, 4 and 8 LPs for other PDES algorithms, can be simulated with 24 cores and achieve over 40× super-linear speedup, as shown in Figure 8b. UNISON turns two-day long (45.2 hours) sequential DES into 1.1 hours. The super-linear speedup can be explained by the cache boost of fine-grained partition, which will be further evaluated in §6.3. The flexibility of UNISON can fully utilize hardware resources without the need to manually re-divide the topology.

UNISON eliminates the synchronization time. To understand the astonishing speedup brought by UNISON, we run the fat-tree of $k = 8$ introduced in §3.2 and record P, S, M of each thread (instead of each LP). Here, the processing time P is counted for both phases of processing events and handling global events. And the messaging time M is counted for both phases of receiving events and updating the window.

As illustrated in Figure 9a, the S of UNISON is surprisingly less than 2% of the total time in every case, and the M is less than 0.3% of the total time, which is hardly visible in Figure 9a, so we omit them. Meanwhile, the P is also less than the other two algorithms by about 20%, which is a benefit from the cache boost. These two factors together make UNISON achieve 5× speedup relative to other PDES algorithms. We also measure the S of UNISON under the balanced traffic in the first 1000 rounds of the simulation. As shown in Figure 9b, the ratio of S is mainly under 1% in every round. Compared with Figure 5b, we can see that UNISON solves the issue of highly unbalanced processing time in a transient time window. Therefore, we can understand that the performance gain of UNISON is mainly caused by eliminating the synchronization time with adaptive scheduling optimized for network simulation, while reducing the processing time by leveraging the cache effect with fine-grained partition.

UNISON is effective in various scenarios. To understand the generality of UNISON across different scenarios, we add four new topologies: 2D-torus [28], BCube [16] and two wide area networks from the Internet Topology Zoo [22] with two traffic patterns: web-search [4] and gRPC [27]. We also add a reconfigurable DCN model [8] to analyze the performance impact of dynamic topologies to UNISON.

For 2D-torus, we set the topology size to 48×48, the link bandwidth to 10Gbps, and the link delay to 30µs. For a node located at the i-th row, j-th column, we assign an ID of $i+48j$ to the node. For other PDES algorithms that require manual



(a) 2D-torus **(b)** BCube **(c)** Wide-area network **(d)** Reconfigurable DCN

Figure 10. The generality of UNISON under different topologies and traffic patterns.

partition, we evenly divide the range of $[0, 48 \times 48]$ and treat nodes whose ID falls in each sub-array as an LP. We run the simulations with 48, 72, 96 and 144 cores for 1 second, generate traffic to take up 30% of the bisection bandwidth and record the simulation time. As shown in Figure 10a, UNISON outperforms the other two algorithms by nearly 4×. The sequential DES is not finished in 5 days and only progresses the simulated time to 0.6 seconds. UNISON can turn this into less than 1.5 hours with 144 cores.

For BCbue, we set $n = 8$ and construct the topology using 2 levels. We set the link bandwidth to 10Gbps, the link delay to 3µs, and generate traffic according to web-search and gRPC distribution, plus incast traffic. Together these traffic takes 30% load of the bisection bandwidth. We treat each BCube_0 as an LP (so there are 8 LPs) and distribute the top-level switches evenly into these LPs for other PDES algorithms. As shown in Figure 10b, UNISON achieves the highest speed among all algorithms. Under the gRPC traffic, UNISON achieves nearly 10× speedup with 8 cores and 15× speedup with 16 cores.

For wide area backbone networks GEANT and ChinaNet, we use RIP dynamic routing with 50% traffic load generated according to web-search distribution. Since finding a symmetric division for these versatile topologies is impossible, we opt out of other PDES algorithms. We launch 8 threads for UNISON. As illustrated in Figure 10, UNISON achieves over 10× super-linear speedup relative to sequential DES.

For reconfigurable DCN, we use 4 cores to simulate a 10Gbps, $k = 4$ fat-tree under balanced traffic for 1 second. After a given interval, we replace all core switches with one optical switch and then replace back, by changing the link connectivity between ToR and core switches. The configurations are similar to TDTCP [8]. As shown in Figure 10d, the simulation time of UNISON and the default sequential simulator both slightly increase as the topology change frequency increases. Therefore, we conclude that the performance impact of UNISON under dynamic topologies is negligible.

6.2 UNISON is Accurate and Deterministic

UNISON is accurate and deterministic based on our several optimizations in practice, including the lock-free implementation, the tie-breaking rule and the support for global thread-safe measurement.

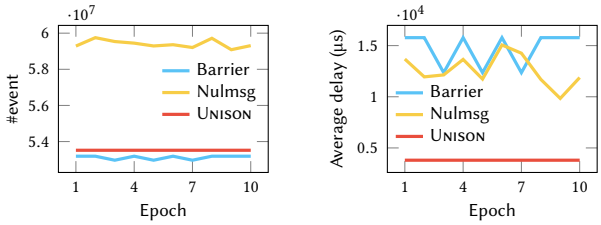
UNISON is as accurate as DES. To evaluate the accuracy of UNISON and MimicNet, a data-driven approach [42], we simulate TCP New Reno with RED queue for 5 seconds in both 2-cluster and 4-cluster fat-trees. The number of hosts per rack is 2, so each cluster has 4 hosts. The link speed is 100Mbps with 500µs delay, which is similar to the setup given in MimicNet [42]. We generate the same traffic for UNISON and MimicNet, which is sampled from the web-search distribution and takes up 70% of the bisection bandwidth. In addition, each time a flow is generated, its destination has a 10% chance of being changed into a random host in the very right cluster. For UNISON, we record statistics with the FlowMonitor module of ns-3 and use the default sequential DES kernel of ns-3 as the baseline. For MimicNet, we train the ingress, egress and inter-mimic models according to their default configurations using the same traffic generation method but a different random seed[4], and use its underlying OMNeT++ as the baseline.

As shown in Table 2, MimicNet can accurately predict the 2-cluster fat-tree. However, its accuracy dropped for RTT and throughput when simulating the 4-cluster fat-tree. This is because MimicNet only trains one cluster and uses this cluster to predict other clusters' performance, which is not suitable for traffic that does not scale proportionally in this incast situation. For UNISON, due to the different tie-breaking rules of simultaneous events (instead of the sequential DES, which always processes the earliest created

[4]The training seed is 0, and the evaluation seed is 9. The hyperparameter tuning feature of MimicNet is not utilized for the default configuration.

Table 2. The Accuracy of UNISON compared with existing data-driven approaches under different fat-tree scales. The unit of FCT and RTT is milliseconds. The unit of throughput is Mbps. All the data are averages.

Simulator	2-cluster			4-cluster		
	FCT	RTT	Thr.	FCT	RTT	Thr.
OMNeT++	617.03	8.47	7.10	418.69	10.22	3.14
MimicNet	607.17	7.38	7.44	412.58	8.02	4.56
Rel. Error	**1.6%**	**12.9%**	**4.8%**	**1.5%**	**21.5%**	**45.2%**
ns-3 default	650.89	15.14	6.81	480.84	10.49	6.50
UNISON	632.52	15.03	6.99	472.45	11.39	6.41
Rel. Error	**2.8%**	**0.7%**	**2.6%**	**1.7%**	**8.5%**	**1.4%**



(a) Event count **(b)** End-to-end delay

Figure 11. The Determinism of Unison.

event first), there is only a slight difference. It is notable that we cannot compare the accuracy between sequential DES and Unison, since both outcomes are possible if we handle all the simultaneous events in random order.

Meanwhile, existing PDES algorithms in ns-3 exhibit a significant inconsistency with the default sequential DES. Their results are either erroneous or from a single cluster. This is because they are unable to collect global statistics for the FlowMonitor module unless we collect and gather them manually. Therefore, we do not include them in Table 2 for comparison.

We further adapt and run the existing DCTCP evaluation [4] with Unison, which achieves 2.5× speedup with 4 threads compared with sequential DES. Unison successfully reproduced the simulation results including per-flow throughput, Jain index and average queue delay described in its publication.

Unison is deterministic. The tie-breaking rule introduced by Unison guarantees that the simulation is deterministic. To this end, we simulate a $k = 4$ fat-tree 10 times and record the event count of different algorithms. As shown in Figure 11a, the event count of Unison is always the same. Yet for the other two PDES algorithms, their event counts will fluctuate when running the simulation for another time, leading to inconsistent results, as illustrated in Figure 11b. Moreover, since another two algorithms fail to measure global statistics, their results are neither precise nor accurate. We also run Unison with 1 to 16 threads. The event count and simulation results of Unison also match *exactly* regardless of the number of threads used.

6.3 Micro Benchmarks

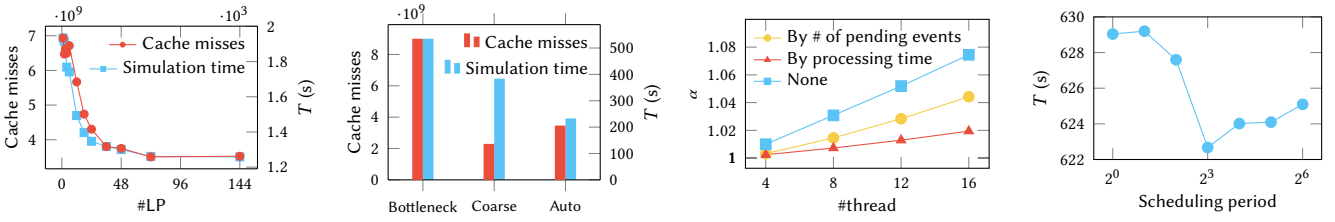
We evaluate the benefits of fine-grained partition and optimize configurations for different networking scenarios used in load-adaptive scheduling.

Fine-grained partition reduces cache misses. As shown in Figure 8b and Figure 9a, Unison can achieve super-linear speedup because of the cache-friendly design of Unison. To further validate the claim, we use Unison but manual partition for a 12×12 torus network discussed in §6.1, with 1 thread. As illustrated in Figure 12a, the number of cache misses decreases as the number of LPs increases, resulting in a faster simulation time of 1.5× with the most fine-grained partition (*i.e.*, 144 LPs where each node is an LP).

To explore how different partition schemes affect the performance, we use Unison but manual partition to simulate the DCTCP example [4] in §6.2 with 4 threads. There is a bottleneck link carrying huge traffic between sender clusters and receiver clusters in this model. We use Unison's fine-grained partition, but avoid cutting off this bottleneck link in our first manual partition scheme. We also avoid cutting off links in the clusters in our second manual partition scheme (*i.e.* coarse-grained). As shown in Figure 12b, fine-grained partition reduces cache misses from frequent interleaving when cutting off a link carrying huge traffic. It also produces balanced scheduling and reduces the simulation time compared with coarse-grained partition, despite a small increase in cache misses when using multiple threads.

Evaluation on scheduling metrics and periods. To measure the impact of different metrics used in the load-adaptive scheduling, we define a slowdown factor, α, which is the sum of the actual completion time of each round divided by the sum of the idealistic round time. The idealistic round time is calculated by assuming that the scheduler knows the exact processing time of each LP in advance, and schedules these LPs according to their exact processing time.

We simulate a $k = 8$ fat-tree. Figure 12c shows that the default metric used by Unison, which is the processing time of the last round, stands out by reducing the slowdown factor by 6% compared with no scheduling (*i.e.*, random priorities of LPs) as the number of threads increases to 16, and achieves



(a) Cache misses under different partition granularities **(b)** Cache misses under different partition schemes **(c)** Slowdown factors under different scheduling metrics **(d)** Simulation time under different scheduling periods

Figure 12. Micro benchmarks of fine-grained partition and load-adaptive scheduling.

only 2% more of the round time compared with the idealistic situation.

We evaluate the performance impact of different scheduling periods since there is a trade-off between scheduling benefits and overheads (*i.e.*, caused by sorting LPs). As shown in Figure 12d, the performance of Unison gets better as the period increases to 16. Further increases in the scheduling period will cause degraded performance. Based on our experience, a period of 16 is large enough to handle 2^{16} LPs with minimal impact on performance.

7 Discussion

Memory Overhead. In addition to the parallel efficiency and user transparency, Unison also reduces the memory overhead of PDES. This is because the network topology and flow information is shared among LPs via multithreading. Therefore, the memory usage of Unison is comparable with the default sequential DES.

Applicability and Generality. One of the limitations on the applicability of Unison is that, it cannot handle models that only contain stateful links such as wireless channels, since they cannot be cut off for fine-grained partition. In addition, for a large model with a low traffic load, the speedup of Unison is less significant, which is about the same as other PDES approaches. However, it is fast to simulate such a model even with sequential DES due to a small number of events and a small degree of parallelism.

Heterogeneous parallel simulation. The scheduling algorithm of Unison assumes that each processor core has the same clock frequencies. For parallel simulation on processor cores with different clock frequencies, a more general scheduling strategy has to be considered. Furthermore, Unison only utilizes CPU cores. For GPU and FPGA-based devices, other approaches are required to utilize their potential computation power and parallelism.

Future work. We will apply Unison to other network simulators including OMNeT++ and ns.py. We are also going to explore heterogeneous parallel simulation and emulation by investigating the use of other computation components, such as FPGAs and programmable switches.

8 Related Work

Network performance estimation. In addition to DES, flow-level mathematical modeling and end-to-end performance estimators [9, 24, 26, 36] can be used in network performance estimation as well. However, they treat the whole network as a black box, which cannot provide detailed visibility [40]. Data-driven approaches are currently replacing their roles. However, as already discussed in §2.2, existing data-driven approaches still have limited usability, long training time, approximated results and rely on DES to collect training data for new scenarios [40, 42].

Another recent work eliminates the need for training by transforming the original topology into many link-level topologies [43]. It then simulates these topologies using DES in parallel and aggregates the results. However, it still relies on DES and can only be used to estimate the tail latency.

Zero-configuration fast network PDES. We identified that both ns-3 and OMNeT++ communities have attempted to achieve zero-configuration network PDES by using a shared-memory approach. The ns-3 community has attempted a multithreaded approach [37]. However, they primarily focus on thread safety issues and their costs, ignoring cache effects, scheduling strategy, determinism and scalability. In contrast, our work provides an extensive enhancement and a greater speedup upon this.

The proposal of OMNeT++ [5] is to identify concurrently processable events in the FEL via a colorization algorithm running on a worker thread, allowing other threads to grab these processable events. However, their proposal relies on the distance matrix of every LP, which would occupy a significant amount of memory ($O(n^2)$ if the number of LPs is n) for large models, and it has not been implemented yet.

Another recent work uses a data-oriented design to reduce cache misses in network simulation [11]. They adopt existing PDES algorithms to cope with large-scale simulations, but the profiling and optimization of PDES are not in their design scope. Moreover, it requires a full re-architecture of existing network simulators to cope with the data-oriented paradigm, which means the entire network protocol stack and applications have to be redesigned to use their simulator. In contrast, our work retains compatibility with existing ns-3 frameworks, requires zero configurations, and can be easily adapted to other network simulators based on DES.

9 Conclusion

Existing PDES algorithms for network simulation are not widely used in practice due to their complex configuration and limited performance gains. This paper introduces a new network simulation kernel, Unison, which is parallel-effect and user-transparent. Unison addresses these limitations by adapting fine-grained partition and load-adaptive scheduling. Our evaluations demonstrate that Unison is fast, transparent, accurate and deterministic across various scenarios.

Acknowledgments

We would like to thank our shepherd, Qizhen Zhang, and the anonymous EuroSys '24 reviewers for all their constructive feedback and comments. This research is supported by the National Key R&D Program of China under Grant Numbers 2022YFB2901502, and the National Natural Science Foundation of China under Grant Numbers 62072228, 62172204 and 62325205.

References

[1] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. 2022. ABM: active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 36–52.

[2] Vamsi Addanki, Oliver Michel, and Stefan Schmid. 2022. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 51–70.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review* 38, 4 (2008), 63–74.

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.

[5] OMNeT++ Technical Articles. 2020. Zero Configuration Automatic Parallel Simulation. https://docs.omnetpp.org/articles/zero-conf-parsim/

[6] Gustavo Carneiro, Pedro Fortuna, and Manuel Ricardo. 2009. Flowmonitor: a network monitoring framework for the network simulator 3 (ns-3). In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. 1–10.

[7] K. Mani Chandy and Jayadev Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on software engineering* 5 (1979), 440–452.

[8] Shawn Shuoshuo Chen, Weiyang Wang, Christopher Canel, Srinivasan Seshan, Alex C Snoeren, and Peter Steenkiste. 2022. Time-division TCP for reconfigurable data center networks. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 19–35.

[9] Miquel Ferriol-Galmés, Jordi Paillisse, José Suárez-Varela, Krzysztof Rusek, Shihan Xiao, Xiang Shi, Xiangle Cheng, Pere Barlet-Ros, and Albert Cabellos-Aparicio. 2023. RouteNet-Fermi: Network Modeling With Graph Neural Networks. *IEEE/ACM Transactions on Networking* (2023).

[10] Richard M Fujimoto. 2000. *Parallel and distributed simulation systems*. Vol. 300. Citeseer.

[11] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. 2023. DONS: Fast and Affordable Discrete Event Network Simulation with Automatic Parallelization. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 167–181.

[12] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 519–533.

[13] Hamid Ghasemirahni, Tom Barbette, Georgios P Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q Maguire Jr, and Dejan Kostić. 2022. Packet order matters! Improving application performance by deliberately delaying packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 807–827.

[14] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E Anderson. 2022. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 779–805.

[15] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

[16] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. 2009. BCube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. 63–74.

[17] Juris Hartmanis. 1982. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review* 24, 1 (1982), 90.

[18] Huawei 2012 Labs. 2020. ns.py. https://github.com/TL-System/ns.py

[19] Paul Hudak. 1986. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. 351–363.

[20] David R Jefferson. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 3 (1985), 404–425.

[21] Vikas Jha and Rajive Bagrodia. 2000. Simultaneous events and lookahead in simulation protocols. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 10, 3 (2000), 241–267.

[22] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The internet topology zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775.

[23] Georg Kunz. 2010. Parallel discrete event simulation. In *Modeling and Tools for Network Simulation*. Springer, 121–131.

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

[25] Hejing Li, Jialin Li, and Antoine Kaufmann. 2022. SimBricks: end-to-end network system evaluation with modular simulation. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 380–396.

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

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

[28] Yasuyuki Miura, Kentaro Shimozono, Shigeyoshi Watanabe, and Kazuya Matoyama. 2013. An adaptive routing of the 2-D torus network based on turn model. In *2013 First International Symposium on Computing and Networking*. IEEE, 587–591.

[29] David M Nicol. 1993. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM (JACM)* 40, 2 (1993), 304–333.

[30] nsnam. 2017. ns-3. https://www.nsnam.org

[31] OpenSim Ltd. 2018. OMNeT++. https://omnetpp.org

[32] Alfred Park, Richard M Fujimoto, and Kalyan S Perumalla. 2004. Conservative synchronization of large-scale network simulations. In *Proceedings of the eighteenth workshop on Parallel and distributed simulation*. 153–161.

[33] George F Riley, Mostafa H Ammar, and Richard Fujimoto. 2000. Stateless routing in network simulations. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No. PR00728)*. IEEE, 524–531.

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

[35] George F Riley, Talal M Jaafar, Richard M Fujimoto, and Mostafa H Ammar. 2004. Space-parallel network simulations using ghosts. In *18th Workshop on Parallel and Distributed Simulation, 2004. PADS 2004.* IEEE, 170–177.

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

[37] Guillaume Seguin. 2009. Multi-core parallelism for ns-3 simulator. *INRIA Sophia-Antipolis, Tech. Rep* 106 (2009), 110.

[38] András Varga, Yasar Ahmet Sekercioglu, and Gregory K Egan. 2003. A practical efficiency criterion for the null message algorithm. In *European Simulation Symposium 2003*. SCS Europe Publishing House, 81–92.

[39] Hao Wu, Richard M Fujimoto, and George Riley. 2001. Experiences parallelizing a commercial network simulator. In *Proceeding of the 2001 Winter Simulation Conference (Cat. No. 01CH37304)*, Vol. 2. IEEE, 1353–1360.

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

[41] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. 1998. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the twelfth workshop on Parallel and distributed simulation*. 154–161.

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

[43] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. 2023. Scalable Tail Latency Estimation for Data Center Networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 685–702.

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

A Processing Time

We record the processing time P of the first 10000 rounds of the barrier synchronization algorithm and Unison, as shown in Figure 13. In Figure 13a, we can notice that the processing time of each LP (the Y-axis) is highly different, yet the processing time of consecutive rounds of a single LP (the X-axis) changes more smoothly. In Figure 13b, Unison can perfectly balance the skewed workload in Figure 13a while reducing the processing time with cache boost.

B Artifact Appendix

B.1 Abstract

This artifact includes Unison's implementation based on ns-3.36.1, along with experiments for profiling and performance comparison between Unison and existing PDES approaches.

B.2 Description & Requirements

B.2.1 How to access. The artifact is publicly visible at https://github.com/NASA-NJU/Unison-for-ns-3. The evaluated artifact is persistently indexed by https://doi.org/10.5281/zenodo.10077300.

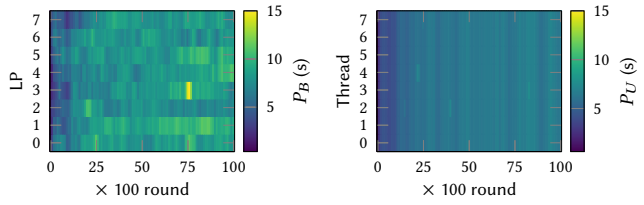
B.2.2 Hardware dependencies. You should have at least 144 CPU cores and 512GB of memory to run all experiments. These computation resources can come from either a single host or multiple identically configured hosts within a cluster.

B.2.3 Software dependencies. A Unix-like operating system installed with Python 3.8 or above, Git, CMake, Linux Perf, OpenMPI library and NFS server/client is required. The minimum compiler versions supported by Unison are g++-7, clang-10 or Xcode 11.

B.2.4 Benchmarks. Our artifact requires WAN network topology data from the Internet Topology Zoo [22] and traffic CDF data from previous publications [4, 27]. We already packed these data into the artifact.

B.3 Set-up

You should first get the source code of Unison by cloning its GitHub repository to your host. Then, switch to the `unison-evaluations` branch and follow the `README.md` file in that branch to set up.



(a) Barrier synchronization **(b)** Unison

Figure 13. Sum of P in every consecutive 100 rounds for a $k = 8$ fat-tree.

B.4 Evaluation workflow

B.4.1 Major Claims. Here we list all of our major claims and their corresponding figures and experiments below.

Claim 1. *Unison can achieve 10× speedup over existing PDES approaches.* This is proven by Exp 1 and Exp 2 whose results are reported in Figure 1.

Claim 2. *The synchronization time of existing PDES approaches gradually dominates as the traffic inhomogeneity increases.* This is proven by Exp 3 whose results are reported in Figure 5a.

Claim 3. *The synchronization time ratio is high in a transient time window for existing PDES approaches, even if the traffic pattern is balanced in macro.* This is proven by Exp 4 whose results are reported in Figure 5b.

Claim 4. *The synchronization time is long for low-latency and high-bandwidth networks for existing PDES approaches.* This is proven by Exp 5 and Exp 6 whose results are reported in Figure 5c and Figure 5d.

Claim 5. *Unison can significantly reduce the synchronization time to near zero.* This is proven by Exp 7, Exp 8 whose results are reported in Figure 5a and Figure 9b.

Claim 6. *Unison exhibit super-linear speedup and its parallelism is flexible to set.* This is proven by Exp 9, Exp 10 and Exp 11 whose results are reported in Figure 8b.

Claim 7. *Unison is also fast with other topologies and under different traffic patterns.* This is proven by Exp 12, Exp 13 and Exp 14 whose results are reported in Figure 10b.

Claim 8. *The output of Unison is deterministic under multiple runs while other PDES approaches are not.* This is proven by Exp 15 whose results are reported in Figure 11.

Claim 9. *Fine-grained partition of Unison can reduce cache misses which can further reduce the simulation time.* This is proven by Exp 16 whose results are reported in Figure 12a.

Claim 10. *The default scheduling metric of Unison performs better than others and without scheduling.* This is proven by Exp 17 whose results are reported in Figure 12c.

Notably, our artifact does not fully validate Figure 8a and Table 2, since they both 1) have to modify and evaluate other artifacts of ML-based data-driven simulators [40, 42] and 2) require expensive GPUs to run these ML-based simulators.

The validation of the figures not mentioned in this section, and part of the Figure 8a and Table 2 is included in our artifact. You can find instructions to run these in our repository. However, they are less important so we do not treat them as major claims. Specifically, Figure 10a, Figure 10c and Figure 10d imply the same as Claim 7 but takes a longer time to run. Figure 12b implies the same as Figure 12a but in the corner case. Figure 12d also reflects the optimization of the load-adaptive scheduling in Claim 10 but the effect is less significant than Figure 12c.

B.4.2 Experiments. It is easy to run experiments of our artifact. We provide a script `exp.py` and all you need to do is pass an argument indicating the experiment name. Here we list all the experiments required to validate our major claims. We also list the corresponding name and its expected machine time for each experiment.

Exp 1 (`fat-tree-distributed`, 7d)**.** This experiment will simulate 48-cluster to 144-cluster fat-trees under incast traffic on multiple hosts, assuming the number of cores used in total is equal to the number of clusters and each host has 24 cores. If your hosts have a different number of cores (*e.g.*, 16), you can change the cluster parameter to 32, 48, 64, *etc.* 24 clusters (cores) should be enough to produce the 10× relative speedup result. Therefore if you do not have so many cores, you can run a small-scale experiment with 8 to 24 clusters, but the relative speedup is only about 6× at 8 clusters.

Exp 2 (`fat-tree-default`, 4d)**.** This experiment will use the default sequential simulation kernel to run Exp 1. If you have changed any parameters of Exp 1, please adjust these parameters in this experiment accordingly.

Exp 3 (`mpi-sync-incast`, 18h)**.** This experiment runs a $k = 8$ fat-tree with existing PDES algorithms using 8 cores under different incast traffic ratios, and records the average P, S and M of every LP. It is expected that S will increase to over 70% of the total time as the incast traffic ratio increases.

Exp 4 (`mpi-sync`, 1h)**.** This experiment runs a $k = 8$ fat-tree with the barrier synchronization algorithm using 8 cores under balanced traffic, and records the S ratio of each round. It is expected that the S ratio will fluctuate and will be above 20% for most of the time.

Exp 5 (`mpi-sync-delay`, 20min)**.** This experiment runs a $k = 8$ fat-tree with existing PDES algorithms using 8 cores under different link delay, and records the average S ratio of every LP. It is expected that the S ratio will decrease as the link delay increases.

Exp 6 (`mpi-sync-bandwidth`, 10min)**.** This experiment runs a $k = 8$ fat-tree with existing PDES algorithms using 8 cores under different link bandwidth, while keeping the same traffic load, and records the average S ratio. It is expected that the S ratio will increase as the link bandwidth increases.

Exp 7 (`mtp-sync-incast`, 3h)**.** This experiment runs a $k = 8$ fat-tree with Unison using 8 threads under different incast traffic ratios, and records the average P, S and M of every thread. It is expected that S is less than 5% for every case.

Exp 8 (`mtp-sync`, 40min)**.** This experiment runs a $k = 8$ fat-tree with Unison using 8 threads under balanced traffic, and records the S ratio of each round. It is expected that the S ratio will be near zero in almost every round.

Exp 9 (`flexible`, 1d)**.** This experiment runs a $k = 8$ fat-tree with Unison using 2-24 threads. It is expected that 24 threads can achieve about 2.5× speedup relative to 8 threads. If you do not have so many cores, you can run a small-scale experiment (a $k = 4$ fat-tree) with 2-8 threads.

Exp 10 (`flexible-barrier`, 3d)**.** This experiment runs a $k = 8$ fat-tree with the barrier synchronization algorithm using 2-8 cores. It is expected that the barrier synchronization algorithm is slower than Unison in Exp 9 under the same number of cores. If you do not have so many cores, you can run a small-scale experiment (a $k = 4$ fat-tree) with 2-4 cores.

Exp 11 (`flexible-default`, 1d)**.** This experiment will use the default sequential simulation kernel to run Exp 9 and Exp 10. If you change any parameters of Exp 9 and Exp 10, please adjust these parameters in this experiment accordingly.

Exp 12 (`bcube`, 40min)**.** This experiment runs a 3-level $n = 8$ BCube with Unison using 8 and 16 threads. It is expected that 16 threads can achieve about 1.3-1.6× speedup relative to 8 threads.

Exp 13 (`bcube-old`, 2h)**.** This experiment runs a 3-level $n = 8$ BCube with Unison existing PDES algorithms 8 cores. It is expected that the existing PDES algorithms are slower than Unison in Exp 12.

Exp 14 (`bcube-default`, 1d)**.** This experiment will use the default sequential simulation kernel to run Exp 12 and Exp 13. If you change any parameters of Exp 12 and Exp 13, please adjust these parameters in this experiment accordingly.

Exp 15 (`deterministic`, 4h)**.** This experiment runs a $k = 8$ fat-tree with Unison and existing PDES algorithms 20 times. It records the number of processed events and flow statistics. It is expected that Unison's event count and flow statistics are the same across multiple runs, while other PDES algorithms are not.

Exp 16 (`partition-cache`, 1d)**.** This experiment runs a 12×12 torus with Unison using only 1 thread. However, the automatic partition of Unison is disabled and we change the granularity of the partition (*i.e.*, number of LPs) manually. It is expected that Unison's cache miss and simulation time are reduced while the partition granularity increases.

Exp 17 (`scheduling-metrics`, 4h)**.** This experiment runs a $k = 8$ fat-tree with Unison with different scheduling metrics. It is expected that the slowdown factor of Unison's default scheduling metric `ByExecutionTime` is the smallest among others.