PipeTGL: (Near) Zero Bubble Memory-based Temporal Graph Neural Network Training via Pipeline Optimization

Jun Liu
HUST[†]
liujun2023@hust.edu.cn

Bingqian Du[*]
HUST[†]
bqdu@hust.edu.cn

Ziyue Luo
OSU[‡]
luo.1457@osu.edu

Sitian Lu
HUST[¶]
st_lu@hust.edu.cn

Qiankun Zhang
HUST[§]
qiankun@hust.edu.cn

Hai Jin
HUST[†]
hjin@hust.edu.cn

ABSTRACT

Memory-based Temporal Graph Neural Networks (M-TGNNs) demonstrate superior performance in dynamic graph learning tasks. Their success attributes to a memory module, which captures historical information for each node and implicitly creates a memory dependency constraint among chronologically ordered minibatches. This unique characteristic of M-TGNN introduces new challenges for parallel training that have not been encountered before. Existing parallelism strategies for M-TGNN either sacrifice memory accuracy (minibatch parallelism and epoch parallelism) or compromise space efficiency (memory parallelism) to optimize runtime. This paper proposes a pipeline parallel approach for multi-GPU M-TGNN training that effectively addresses both inter-minibatch memory dependencies and intra-minibatch task dependencies, based on a runtime analysis DAG for M-TGNNs. We further optimize pipeline efficiency by incorporating improved scheduling, finer-grained operation reorganization, and targeted communication optimizations tailored to the specific training properties of M-TGNN. These enhancements significantly reduce GPU waiting and idle time caused by memory dependencies and frequent communication and result in zero pipeline bubbles for common training configurations. Extensive evaluations demonstrate that PipeTGL achieves a speedup of 1.27x to 4.74x over other baselines while also improving the accuracy of M-TGNN training across multiple GPUs.

PVLDB Reference Format:
Jun Liu, Bingqian Du, Ziyue Luo, Sitian Lu, Qiankun Zhang, Hai Jin. PipeTGL: (Near) Zero Bubble Memory-based Temporal Graph Neural Network Training via Pipeline Optimization. PVLDB, 18(8): 2722 - 2734, 2025.
doi:10.14778/3742728.3742760

[*]Corresponding author.
[†]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology.
‡ Department of Electrical and Computer Engineering, The Ohio State University.
[¶]School of Artificial Intelligence and Automation, Huazhong University of Science and Technology.
[§]School of Cyber Science and Engineering, Huazhong University of Science and Technology.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742760

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CGCL-codes/PipeTGL.

1 INTRODUCTION

Many real-world graphs inherently consist of nodes and edges that dynamically evolve over time, and capturing these dynamic patterns is crucial. For example, a temporal financial transaction network records the creation and dissolution of financial relationships and transactions as they happen, enabling the timely detection of financial crimes. *Temporal Graph Neural Networks* (TGNNs) [10, 16, 18, 19, 21, 28] have recently been introduced to effectively capture both structural and temporal relationships in dynamic graphs, demonstrating superior performance in tasks such as node classification, link prediction, and graph classification [18, 28].

Among the various TGNN models, memory-based models, such as TGN [18], DyRep [21], and APAN [26], stand out for their use of a node memory module [20] and have achieved state-of-the-art performance in modeling and analyzing dynamic graphs [32]. This memory module, typically a recurrent neural network, continuously records the information of historical events associated with each node. This recorded information, along with the node features, is utilized by graph neural network in each training iteration for recursive aggregation of structural information. Specifically, in M-TGNN training, minibatches are organized in chronological order of events. For each minibatch, the memory containing information from events in prior minibatches is loaded and updated for loss computation. The updated memory representation is then written back to the memory module and serves as the basis for processing subsequent minibatches. The temporal relationship in dynamic graph is thus maintained, *imposing a constraint across minibatches.*

This unique characteristic of the M-TGNN model introduces significant challenges to training efficiency, which are not typically encountered in most previous deep learning systems. In most existing DL systems, minibatches are mutually independent, allowing various parallelism strategies—such as data parallelism [11] and model parallelism [8, 14]—to speed up training and improve scalability. However, in the M-TGNN model, minibatches are mutually dependent due to the need to compute memory chronologically. As a result, these parallelism strategies cannot be applied effectively. Existing parallel training approaches for M-TGNN either compromise model performance or sacrifice system scalability. Specifically, minibatch parallelism ignores memory dependencies among in-flight minibatches, adversely affecting the performance of the learned model. Epoch parallelism increases gradient variance due to the repeated execution of the same minibatch and jeopardizes training convergence. Memory parallelism requires separate memory copies for each GPU, leading to substantial increase in memory space requirements, which may be prohibitively high.

Given the limitations of existing parallelism strategies for the M-TGNN model, we propose PipeTGL, an efficient parallel training system for M-TGNN in a multi-GPU setup. Our main contributions and key results are summarized as follows:

• **We identify bottlenecks in parallel M-TGNN training to inform the design of our parallel strategy.** We begin by profiling the runtime of each stage within a training iteration across various datasets, discovering that *approximately 30% of the runtime is spent on communications for feature and memory fetching.* To better understand the bottlenecks, we construct a DAG for M-TGNN training, taking into account both inter-minibatch dependencies, driven by memory continuity requirements, and intra-minibatch stage dependencies, driven by the diverse tasks within each training iteration. By combining the DAG with the profiled runtime, we analyze the critical path within this DAG and uncover that the *M-TGNN model's runtime is constrained by memory dependencies across minibatches, allowing the stages of a minibatch to be executed sequentially without significantly increasing overall runtime.*

• **Building on these bottlenecks, we devises a straightforward yet effective parallel execution plan,** deciding the execution orders and placement of all stages during training. Stages within a minibatch are executed sequentially on the same GPU to avoid additional inter-GPU communication, while different minibatches are distributed across GPUs using a round-robin algorithm. The memory fetching and model updating stages for a given minibatch must wait for the completion of memory updating and model updating in the preceding minibatch to maintain inter-minibatch dependencies. This naturally forms a *pipeline* across different GPUs and minibatches. To reduce pipeline bubbles, we integrate two techniques into the execution order optimization: (1) reorganizing memory-related operations and prioritizing memory tasks for target nodes to minimize waiting time due to memory dependencies, and (2) introducing a pre-sampling strategy that advances the sampling process by one iteration, allowing feature fetching communication to overlap with sampling and other computations.

• **We optimize the communication processes to minimize the associated time expenditure.** We identified that the bottleneck in feature fetching lies in the search process for locating the required nodes, rather than in the I/O itself. Consequently, directly overlapping feature-fetching communication with computation stages does not significantly reduce runtime. To address this, we offload the search task to a separate process, independent of the training process, allowing for better overlap and improved efficiency. Additionally, we replace D2H and H2D communication with D2D transfers for memory fetching of target nodes to further reduces memory dependency and communication runtime, based on data showing that more than 50% of target nodes recur across adjacent training minibatches in M-TGNN.

• We achieve an *(almost) zero-bubble pipeline parallel training for M-TGNN* that effectively addresses the memory continuity requirement. We implement PipeTGL on GNNFlow[31], incorporating all the optimization strategies discussed above. We evaluated PipeTGL against the existing M-TGNN training systems and demonstrated that PipeTGL achieves a speedup of 1.27x to 4.74x compared to all other baselines.

2 BACKGROUND AND MOTIVATION

2.1 Memory-based Temporal Graph Neural Network

In a dynamic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of nodes, while $\mathcal{E}$ is the set of edges, each edge is represented as (u, v, e_{uv}, t), indicating an event occurring between nodes u and v at time t, with features e_{uv}. M-TGNN stores the temporal information of each node $v \in \mathcal{V}$ in a memory vector s_v.

Batched Training. M-TGNN employs self-supervised learning to predict the occurrence of future edges (positive samples) based on historical events. To achieve this, the edges $\mathcal{E}$ are arranged chronologically and divided into minibatches, with each batch containing a sequence of consecutive edges. Thus, for $\mathcal{E} = \{E_1, E_2, \ldots, E_n\}$, the edges are partitioned into minibatches $\{\{E_1, E_2, \ldots, E_i\}, \{E_{i+1}, E_{i+2}, \ldots, E_j\}, \ldots, \{E_{k+1}, E_{k+2}, \ldots, E_n\}\}$, where $t_i \leq t_{i+1}$ for any two edges E_i and E_{i+1}. This allows the memory module to continuously capture historical events as training progresses, enabling the memory containing past events from previous minibatches to be utilized for predicting edges in subsequent minibatches.

Temporal Sampler. The nodes associated with training edges (both positive and negative samples) are termed *target nodes*. In an n-layer M-TGNN, each target node requires its n-hop neighbors, known as *supporting nodes*, to recursively compute its embedding. The sampler samples both negative samples and supporting nodes to form a computation graph for each minibatch.

Message Generating. Each time an event occurs, *messages* are generated for the associated nodes, serving as the basis for memory updates. For an event (u, v, e_{uv}, t), the messages for the involved nodes are:

$$\begin{aligned} m_u(t) &= \{s_u(t_u^-) | s_v(t_v^-) | \theta(t - t_u^-) | e_{uv}\}, \\ m_v(t) &= \{s_v(t_v^-) | s_u(t_u^-) | \theta(t - t_v^-) | e_{uv}\}. \end{aligned} \tag{1}$$

where $s_u(t_u^-)$ $(s_v(t_v^-))$ represents memories of node u (v) just before time t. The term $t - t_u^-$ denotes the time interval since the last memory update of node u, and θ is the time encoding function [28].

Memory Updating. After generating the message, each node updates its memory accordingly. The updated memory of node u is calculated as follows:

$$s_u(t) = UPT(s_u(t_u^-), m_u(t)). \tag{2}$$

The function $UPT(\cdot)$ can be any recurrent model, such as LSTM and GRU. It is important to note that M-TGNN models typically delay memory updates to subsequent minibatches to prevent information leakage, *i.e.*, to avoid using memory that includes events from the current minibatch to predict events within the same minibatch [12]. Therefore, instead of updating memory immediately, the messages from the current minibatch are stored and will be utilized in subsequent minibatch for memory updating.

Embedding Calculation. After updating the nodes' memory, M-TGNN employs a single-layer temporal attention mechanism [22] to aggregate information, including memory and features, from the neighbors of target nodes to generate the final embeddings, similar to the approach in static GNNs [7, 9].

The complete training process for a minibatch in M-TGNN is illustrated in Figure 1.

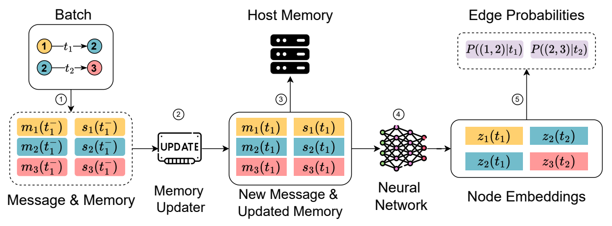


Figure 1: The training process of M-TGNN: ①Fetch the most recent memory and messages for nodes in computation graph. ②Generate the latest memory and messages. ③Save the latest memory and messages to the host memory. ④Compute each node's embedding using the latest memory and messages. ⑤Calculate the probability of each edge using embeddings for downstream tasks.

2.2 Multi-GPUs Training for M-TGNN

To enable parallel training of M-TGNN models in a multi-GPU environment, different systems have adopted three distinct approaches: **Minibatch Parallelism**, employed by TGL [32] and GNNFlow [31], bypasses the memory dependency between in-flight minibatches. In this approach, each in-flight minibatch uses the same node history and messages for model updates. Memory and messages for nodes involved in multiple in-flight minibatches are updated only once. In contrast, with sequential execution, each minibatch loads the memory and messages updated by the previous minibatch, with each minibatch corresponding to a separate update. As a result, this method can lead to information loss and memory staleness. Figure 2(a) illustrates this approach: In a training cluster with three GPUs, each GPU processes the $3i$-th, $(3i + 1)$-th, and $(3i + 2)$-th minibatches in round i using the same memory and messages from minibatches prior to round i in parallel, disregarding the temporal relationships among these minibatches.

Epoch Parallelism, introduced by DistTGL [33], allows multiple epochs to be trained simultaneously across different GPUs. As illustrated in Figure 2(b), in a 3-GPU cluster, three epochs run concurrently: when GPU0 finishes processing minibatch i of the first epoch, GPU1 immediately begins processing minibatch $i + 1$ of the same epoch, utilizing the updated memory from minibatch i. Meanwhile, GPU0 starts training minibatch i for the second epoch, using the memory loaded for the training of minibatch i in the first epoch. However, since the parameters used to calculate this memory have been updated after processing minibatch i in the first epoch, this introduces parameter staleness. Additionally, repeatedly applying gradients from the same set of positive target nodes can negatively impact the convergence of the model.

Memory Parallelism, as proposed by DistTGL [33], involves each GPU retaining a complete copy of the node memory, enabling training similar to a single GPU setup, with only the model parameters synchronized among GPUs. To prevent the same sample from being trained multiple times within a short period, the iterations computed by different GPUs are staggered. Figure 2(c) illustrates this memory parallelism approach. However, this method significantly increases memory overhead, which scales with the number of GPUs, making it potentially prohibitive compared to other parallelism strategies. For example, in a dynamic graph with 100 million nodes, the memory required to store a single copy of the node memory would be about 40GB. With memory parallelism across 8 GPUs, this requirement would balloon to 320GB, may well exceeding the host memory capacity of a single machine.

Current parallelism strategies for M-TGNN fall short: *either sacrifice model performance or limit scalability.*

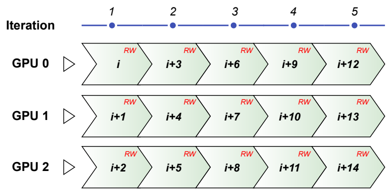
2.3 Challenges of Parallel Training for M-TGNN

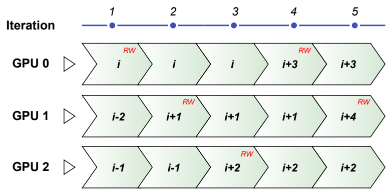
The limitations of current parallel training approaches for M-TGNNs prompt us to rethink and design more efficient, scalable methods that ensure memory continuity across multiple GPUs. In this process, we've identified several previously overlooked challenges.

▹ **No established guidelines to inform the design of parallel training strategy for M-TGNN models.** The existing parallelism strategy for M-TGNN is intuitively designed, prioritizing runtime optimization at the expense of memory continuity. However, the consequences of sacrificing memory continuity are not well understood. We first evaluate the performance of the M-TGNN model under varying levels of memory continuity and assess the significance of maintaining memory continuity. Figure 3 (a) evaluates the convergence accuracy and the number of epochs needed for convergence when training with varying numbers of GPUs, utilizing minibatch parallelism which disregards memory continuity among in-flight minibatches. Each GPU holds 1000 training samples, resulting in a minibatch size that is 1000 times the number of GPUs. As the number of GPUs increases, the lack of memory continuity becomes more pronounced, leading to reduced model accuracy and a significant increase in the number of epochs required for convergence. Therefore, *memory continuity is crucial for ensuring faster and better model convergence.* An intuitive solution to mitigate memory discontinuity in minibatch parallelism is to reduce the minibatch size, thereby minimizing the loss of information. However, decreasing the minibatch size would lead to reduced training throughput and lower GPU utilization (Figure 3 (b)).

Given the importance of memory continuity, no existing work systematically analyzes and identifies the bottlenecks in M-TGNN training when memory continuity is enforced, leaving the design of parallelism strategies with memory continuity guarantees without clear direction. To address this, we profile the runtime of each stage during a single iteration of the M-TGNN training process and formulate a directed acyclic graph (DAG) that accounts for both inter-minibatch memory dependencies and intra-minibatch stage dependencies, through which we identify the most time-consuming tasks that require optimization and the critical path that dictates the design and performance of the parallel execution plan.

▹ **No efficient parallel execution plan to decide the placement and execution order for all stages in the M-TGNN training.** The placement and execution order of stages are crucial factors in the runtime efficiency of the M-TGNN training system. For instance,


(a) minibatch parallelism


(b) epoch parallelism

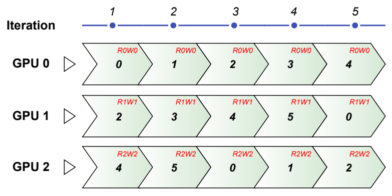
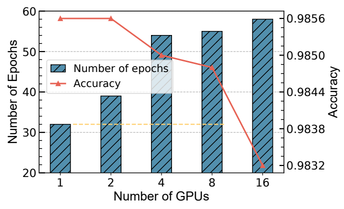

(c) memory parallelism

Figure 2: Overview of *minibatch paralleism*, *epoch parallelism* and *memory parallelism*. RW indicates GPU read and write memory in this iteration.


(a)

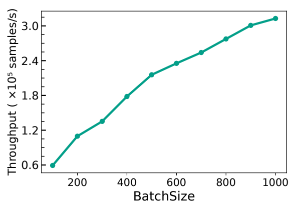

(b)

Figure 3: (a) Model convergence accuracy and the number of epochs required for convergence on the Reddit dataset with different numbers of GPUs, using a minibatch size of 1000 times the number of GPUs. (b) Model throughput during training on the Reddit dataset with varying minibatch sizes.

parallel execution of feature fetching and memory fetching for the same minibatch across different GPUs reduces runtime compared to sequential execution on the same GPU. However, the current M-TGNN training system does not optimize for placement and execution order, and addressing these challenges is inherently difficult due to their combinatorial nature and exponential solution space. Additionally, memory continuity requirements impose constraints across minibatches, further complicating the problem.

Building on our findings that the runtime of the training process is constrained by memory dependencies, we observed that sequential execution of stages within the same minibatch has little impact compared to using an optimized execution order. As a result, we arrange all stages of a single minibatch on the same GPU and execute them sequentially, while distributing different minibatches across GPUs using a simple round-robin method, ensuring that inter-minibatch dependencies are respected. Additionally, we integrate two execution order optimization techniques tailored to the unique characteristics of M-TGNN to minimize GPU idle time.

▹ **No communication optimization method specifically designed to address the unique communication challenges present in M-TGNN.** The communication process in M-TGNN exhibits distinct characteristics compared to those in other deep learning model training: first, frequent device-to-host and host-to-device communication occurs for fetching features and per-iteration renewed memory; second, simply overlapping communication with computation does not significantly reduce runtime. Understanding the communication properties of M-TGNN and developing corresponding optimization techniques are essential for enhancing system efficiency.

To uncover the underlying reasons, we decomposed the feature fetching process into two components—I/O operations and feature gathering—and measured the runtime for each separately. Our findings reveal that feature gathering consumes about half of the total feature fetching time, indicating that the bottleneck is present not only in the I/O operations but also in the feature gathering process. We further analyzed the memory fetching process and identified a pattern in M-TGNN's training data that can be leveraged to reduce memory fetching time. Based on these insights, we implemented communication optimizations to enhance the overlap between feature fetching and computation, while also accelerating the memory fetching process.

3 PERFORMANCE ANALYSIS

To identify bottlenecks in M-TGNN training and inform the design of an effective parallel strategy, we started by profiling intra-minibatch tasks in the M-TGNN model to identify the most time-consuming tasks that require optimization during each training iteration. We then formulated the inter-minibatch dependencies, necessary for memory continuity, and the intra-minibatch dependencies, driven by tasks per iteration, into a Directed Acyclic Graph (DAG) to analyze how these dependencies impact overall training efficiency.

3.1 Intra-Minibatch Analysis

In each iteration, the training process for a single minibatch involves the following stages: *graph sampling, feature fetching, memory fetching, memory updating, forward and backward propagation, model updating,*[1] as shown in Figure 4. In this process, *sampling* is the prerequisite task for *feature fetching* and *memory fetching* since the computation graph returned by the sampler determines the content needed from host memory. *Feature fetching* and *memory fetching* are the only two stages that can be executed in parallel. Next, the fetched features and memory are utilized in the *memory updating* phase, where the memory is updated and written back to the host memory for the next minibatch's computation. This updated memory then participates in loss and gradient computation, creating a dependency between *memory updating* and *forward and backward propagation.* The final step is *model updating*, where the gradients

[1]we use $s, ff, mf, mem\_up, f\&b, model\_up$ as shorthand for different stages.

from *forward and backward propagation* are applied to the model parameters.

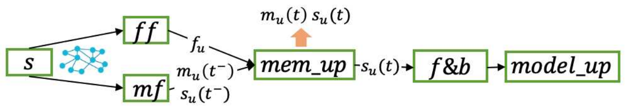


Figure 4: Intra-minibatch task dependency

Following that, we profile the duration of each stage within a single training iteration to gain a comprehensive understanding of the time spent on each stage. Table 1 presents the profiling results across various datasets.

Table 1: The runtime distribution of different stages.

Dataset	s	ff	mf	mem_up	f&b	model_up
WIKI	13.4%	19.4%	11.4%	9.5%	37.1%	9.2%
MOOC	13.1%	16.3%	10%	11.3%	39.4%	9.9%
REDDIT	12.4%	19.1%	13.5%	8.6%	37.2%	9.2%
LASTFM	13.5%	17.3%	10.9%	8.1%	40.5%	9.7%
GDELT	19.5%	19.9%	12.6%	8.4%	31.4%	8.2%

We make the following observations: **Observation #1**: Approximately 30% of the training time is devoted to communication (ff and mf); **Observation #2**: Over 50% of the iteration runtime is consumed by preparing input data, *i.e.*, obtaining the updated memory, for forward and backward computation. Therefore, to maximize the GPU's efficiency in performing actual computations for model updating, it is crucial to minimize communication time and reduce the time spent on preparing input data.

3.2 Intra- and Inter-Minibatch Analysis

When training multiple minibatches concurrently in a multi-GPU setup, two types of dependencies are involved: intra-minibatch dependency, where tasks within a single minibatch must be executed according to the specific order in Figure 4, and inter-minibatch dependency, which requires minibatch execution to adhere to memory continuity and model updating dependencies. To identify runtime bottlenecks, we represent the dependencies in M-TGNN training as a Directed Acyclic Graph (DAG), as shown in Figure 5. Across different minibatches, the memory continuity dependency requires that the *memory fetching* stage of minibatch $t+1$ can only commence after minibatch t has completed its *memory updating* stage. Additionally, the model updating dependency mandates that the *model updating* for minibatch $t+1$ cannot start until minibatch t has finished its *model updating* stage.

In a multi-GPU setup, training dependencies often require the introduction of model parameter staleness to prevent idle waiting times between GPUs [14] [29]. We adopt a similar approach to Pipedream [14], where during the *forward and backward propagation* phase, the GPU processing training batch t computes the loss and gradients using its locally available parameters w^{t-n+1}, with n being the number of GPUs. Therefore, the parameter update proceeds as follows:

$$w^{t+1} = w^t - \eta \cdot \nabla_{w^{t-n+1}} f(w^{t-n+1}) \tag{3}$$

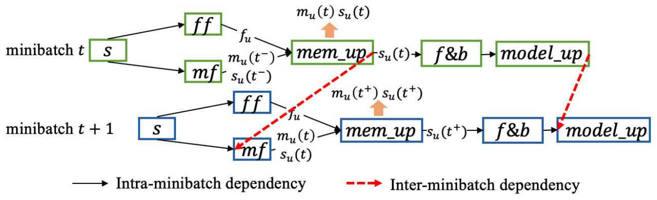


Figure 5: DAG for M-TGNN training

where η is the learning rate, f is the loss function optimized by M-TGNN and w_{t+1} represents the model parameters after the $t+1$-th iteration/training minibatch.

By integrating the DAG with the profiled runtime for each stage, we analyze the earliest possible execution time for each stage, as indicated by the numbers in parentheses (#) above each stage in Figure 6. The duration of each preceding stage is labeled on the arrows of the first minibatch, reflecting the average values from Table 1. Figure 6 illustrates a cluster of three GPUs under the assumption that the system starts at time 0, with each color representing a different minibatch. However, the insights gained from this analysis can be generalized to any system configuration. Our observations are as follows: **Observation #3**: **The system's runtime is constrained by memory dependencies.** The *memory fetching* stage of each minibatch depends on the completion of *memory updating* from the preceding minibatch, placing it consistently on the critical path. This property is demonstrated by the earliest possible execution time for the *memory fetching* stage, which is highlighted in red; **Observation #4**: **Sequentially executing stages within a single minibatch has minimal impact on the overall system runtime.** Within a single minibatch, *feature fetching* and *memory fetching* are the only two stages that can be executed in parallel. However, since there is often a waiting period after *feature fetching* before *memory fetching* can commence, there is no need to execute them in parallel. The only minibatch where *memory fetching* can be executed in parallel with *feature fetching* is the first minibatch, as it has no preceding memory dependencies. The purple numbers in (#) indicate the earliest execution time for each stage when they are executed in parallel, while the grey numbers represent the times when they are executed sequentially. The difference between them is negligible; **Observation #5**: **The GPU processing minibatch** i **completes its tasks earlier than the GPU handling minibatch** $i+1$**.** This conclusion trivially holds when each GPU processes one minibatch at a time due to the memory and model dependencies across minibatches.

4 THE DESIGN OF PIPETGL

4.1 Parallel Execution Plan for M-TGNN

The insights gained from the performance analysis of the M-TGNN model offer a straightforward yet effective strategy for determining the placement and execution order of all stages. Stages within a single minibatch can be executed sequentially on the same GPU, as the runtime benefit of parallel execution is minimal (**Observation #4**). For different minibatches, a simple round-robin algorithm can

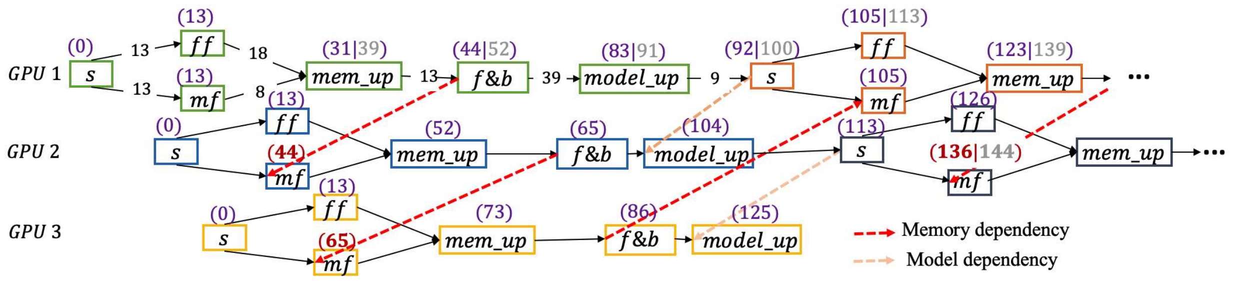


Figure 6: Runtime analysis for M-TGNN training. Milliseconds are the units used for each number.

be used for placement, ensuring that GPUs handling earlier minibatches finish sooner, allowing subsequent minibatches to start earlier (**Observation #5**). Additionally, the memory and model dependencies for *memory fetching* and *model updating* must be carefully respected. This strategy naturally creates a parallel *pipeline* across GPUs, as illustrated in Figure 8(a).

The efficiency of pipeline implementation is significantly influenced by the idle time of each device, referred to as pipeline bubbles. These bubbles seem inevitable due to the necessity of communication during feature and memory fetching, as well as maintaining memory continuity. We conduct an in-depth examination of the M-TGNN training process, uncovering hidden characteristics that can be leveraged to mitigate these bubbles through two execution order optimization techniques.

Reorder of the memory operations. We identified an opportunity to further reduce the waiting time caused by memory dependency (bottlenecks from **Observation #2** and **Observation #3**) by considering memory related operations at a finer granularity. Recall that for each training minibatch, the computation graph generated during the sampling stage includes both target and supporting nodes. Current practice in existing systems [31, 33] involves fetching and updating memory for **all** nodes in computation graph simultaneously. However, we observed that *only the target nodes have dependencies for subsequent minibatches.* This is because only the positive target nodes correspond to new events that occurred in the current minibatch t and require their updated memories $s_u(t)$ and messages $m_u(t)$ to be written back to the CPU. The other nodes in the computation graph of this minibatch are not associated with any new events, meaning there is nothing new to write back for these nodes. As a result, there exists no dependencies between these nodes and subsequent minibatches. Therefore, **prioritizing memory-related operations for the target nodes** can further reduce waiting time due to memory dependencies, as illustrated in Figure 7.

Based on this insight, we prioritize the memory fetching and updating stages for target nodes, combining them into a single stage called *memory operations on target nodes (t_mem_op)*. Similarly, the memory fetching and updating stages for supporting nodes are also combined into a single stage referred to as *memory operations on supporting nodes (s_mem_op)*. Figure 8 (a) (b) compares the schedule and waiting times before and after the reordering of memory-related operations.

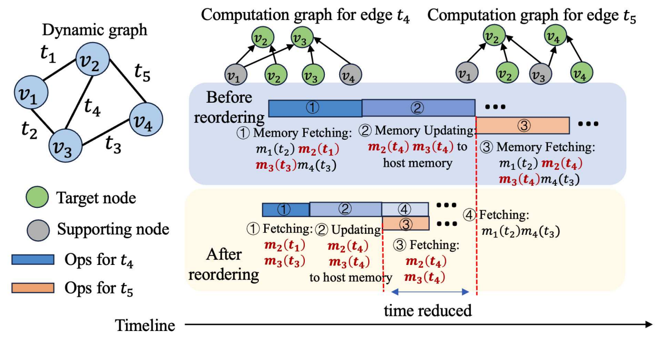


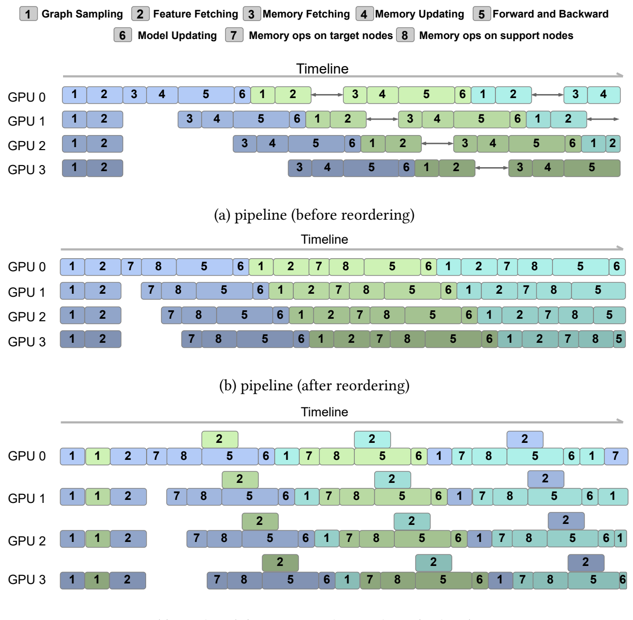
Figure 7: A comparison of the system's performance before and after prioritizing memory-related ops for target nodes.

Pre-sampling and pre-fetching for each minibatch. We further optimize the execution order by introducing a *pre-sampling* approach that decouples the dependency between the sampling stage and subsequent feature fetching communication stage, enabling overlapping between communication and computation. Specifically, we move the sampling of minibatch t to iteration $t-n$ (where n represents the number of GPUs), with the sampling for the first n iterations completed before training begins. With pre-sampling, *pre-fetching* features [2] for each iteration becomes feasible and can be scheduled to run in parallel with computation tasks whenever possible. The training paradigm after enabling pre-sampling and pre-fetching is shown in Fig. 8(c). In this setup, feature fetching for the next minibatch, based on pre-sampling results, is performed concurrently with the forward and backward stages of the current minibatch.

4.2 Communication Optimization for M-TGNN

To address the communication bottleneck identified in **Observation #1**, we closely examined the communication process of M-TGNN and made the following observations: (1) Although pre-sampling and pre-fetching allow for the parallel execution of feature fetching and computation stages, using different CUDA streams for training and feature fetching does not effectively achieve parallel execution; (2) Significant time-consuming H2D and D2H communication occurs due to the need to fetch up-to-date memory, and the feasibility of overlapping memory fetching and computation is

[2]Pre-fetching memory is not feasible due to cross-minibatch memory dependencies.



(a) pipeline (before reordering)

(b) pipeline (after reordering)

(c) pipeline (after pre-sampling and pre-fetching)

Figure 8: The execution plan before and after different optimization techniques, with blank spaces indicating idle time. Each GPU processes a distinct set of minibatches, with different colors and shades representing the various minibatches.

limited by the requirement for memory continuity. We analyzed the root causes of these issues and proposed targeted optimizations to address them.

Offloading feature Gathering with a separate process. To investigate the unexpected system performance when overlapping feature fetching communication and computation, we divided the feature fetching stage into two distinct operations: feature gathering, which involves searching for and locating the corresponding features for the required nodes, and I/O operations, which handle the transfer of the gathered features from host memory to GPU memory. We then measured the runtime of each operation separately, with the evaluation results presented in Figure 9(a). The fact that feature gathering accounts for about 30% to 50% of the runtime suggests that a separate process is necessary to fully overlap feature fetching with other computation tasks. Using separate CUDA streams can only overlap I/O operations with computation, leaving half of the feature fetching runtime sequentially executed with other tasks. Therefore, we introduce an additional process to handle the feature gathering process for each minibatch, maximizing the overlap between communication and computation stages.

Direct transfer of memory between adjacent GPUs. We analyze the construction of the computation graph in M-TGNN with the goal to identify common characteristics that can be leveraged to reduce the communication time due to memory fetching. Figure 9(b) illustrates the repetition rate of target nodes across consecutive minibatches for various datasets, showing that *approximately 34%-62% of target nodes repeatedly appear between adjacent batches.* This property offers an opportunity to replace a portion of the D2H

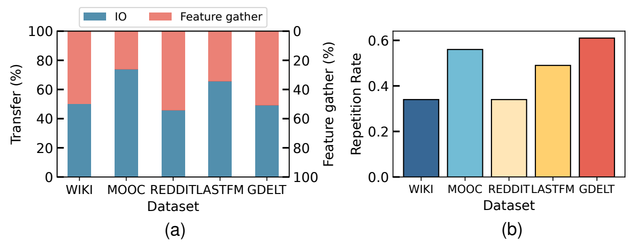


Figure 9: (a) The proportion of I/O operations relative to feature gathering in the feature fetching process. (b) The repetition rate of target nodes between consecutive minibatches.

and H2D communications for memory fetching with faster D2D memory transmissions, further reducing memory dependency and accelerating the overall training process. During the *memory ops on target nodes* stage, three steps occur: (1) fetching memory and messages corresponding to historical events for all target nodes; (2) computing the updated memory and messages based on events in the current minibatch; and (3) writing the updated memory and messages back to host memory to ensure that the subsequent minibatch can access the most up-to-date data. Under the D2H and H2D communication mode, the *memory ops on target nodes* for a new minibatch cannot begin until the previous minibatch has completed the third step of writing back to host memory. However, since 34%-62% of target nodes are repeated across adjacent minibatches, each GPU can directly retrieve the memory and messages of duplicate nodes from the GPU that handled the previous minibatch as soon as the GPU with previous minibatch completes the second step, instead of waiting for the third step to finish. Additionally, since the memory of repeated target nodes is immediately updated in the next minibatch, there is no need to write them back to the host. As a result, only the target nodes that do not appear in the next minibatch need to be written back in the third step, further reducing the execution time of the write-back process. This process is illustrated in Figure 10.

Due to the self-supervised nature of M-TGNN, the target nodes for each minibatch are fixed and we can identify in advance, before training begins, which nodes appear repeatedly across adjacent minibatches. This allows us to determine which nodes' memory should be transferred to the next GPU via D2D and which should be fetched from the host memory. Therefore, this optimization technique does not introduce any additional overhead, as the decisions regarding memory transfer can be made prior to training.

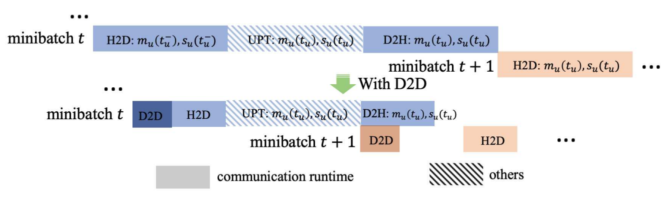

minibatch t
H2D: $m_u(t_u^-), s_u(t_u^-)$
UPT: $m_u(t_u), s_u(t_u)$
D2H: $m_u(t_u), s_u(t_u)$
minibatch $t+1$
H2D: $m_u(t_u), s_u(t_u)$
With D2D
D2D
H2D
communication runtime
others

Figure 10: A comparison of the communication time on memory fetching without and with D2D.

4.3 Quantitative Analysis of PipeTGL

Based on the design of PipeTGL, we establish the relationship between bubble size and the number of GPUs n involved in training, offering an intuitive understanding of PipeTGL's performance. We use p_i to denote the time duration of different stages i, where i can represent s, ff, t_mem_op, s_mem_op, $f\&b$, or $model\_up$. Memory-related operations (i.e., memory fetching and updating) are reorganized into memory operations on target nodes (t_mem_op) and memory operations on supporting nodes (s_mem_op) in PipeTGL. The profiled duration p_i for each stage is shown in Table 2.

Table 2: The runtime distribution after memory reordering.

Dataset	s	ff	t_mem_op	s_mem_op	f&b	model_up
WIKI	13.4%	19.4%	11.4%	9.5%	37.1%	9.2%
MOOC	13.1%	16.3%	10%	11.3%	39.4%	9.9%
REDDIT	12.4%	19.1%	13.5%	8.6%	37.2%	9.2%
LASTFM	13.5%	17.3%	10.9%	8.1%	40.5%	9.7%
GDELT	19.5%	19.9%	10.6%	10.4%	31.4%	8.2%

The idle time of GPU 0 during the processing of its second minibatch (denoted as minibatch α) is $\max(t_2 - t_1, 0)$, where $t_1 = p_s + p_{t\_mem\_op} + p_{s\_mem\_op} + p_{f\&b} + p_{model\_up} + p_s$, $t_2 = p_s + n * p_{t\_mem\_op}$. t_2 represents the time when the last GPU completes the dependent memory operations of target nodes for minibatch $\alpha - 1$, while t_1 is the time when GPU 0 has finished all preceding tasks allocated to it prior to the memory related operations for minibatch α (Notably, due to the pre-sampling and pre-fetching optimization, the ff stage of one minibatch on a GPU can be fully overlapped with the $f\&b$ stage of the previous minibatch allocated to the same GPU). Therefore, $\max(t_2 - t_1, 0)$ measures the duration of GPU 0's idle time, if any, while waiting for updated memory from minibatch $\alpha - 1$. Next, we show that $\max(t_2 - t_1, 0)$ precisely quantifies the pipeline bubble time.

Theorem 4.1. *The pipeline bubble time experienced by each GPU during a single minibatch in PipeTGL is precisely* $\max(t_2 - t_1, 0)$, *where* $t_1 = p_s + p_{t\_mem\_op} + p_{s\_mem\_op} + p_{f\&b} + p_{model\_up} + p_s$, $t_2 = p_s + n * p_{t\_mem\_op}$.

Proof. The proof can be completed by induction. Suppose the system starts at time 0. The idle time for GPU 0 while processing its second minibatch is straightforward to compute: $\max(t_2 - t_1, 0)$ [3], where $t_1 = p_s + p_{\text{t\_mem\_op}} + p_{\text{s\_mem\_op}} + p_{\text{f\&b}} + p_{\text{model\_up}} + p_s$, and $t_2 = p_s + n \cdot p_{\text{t\_mem\_op}}$.

Assume that this idle time holds for GPU i when processing minibatch j. Let τ_1 represent the time when GPU i finishes its tasks prior to the memory operation for minibatch j, and let τ_2 be the time when GPU $i - 1$ completes the $t_{\text{mem\_op}}$ for minibatch $j - 1$. The idle time for GPU i is then $\max(\tau_2 - \tau_1, 0) = \max(t_2 - t_1, 0)$.

Next, we consider whether this relationship holds for GPU $i + 1$ processing minibatch $j + 1$. The time when GPU $i + 1$ can begin its preceding tasks before the memory operation for minibatch $j + 1$ depends on when GPU i completes its t_mem_op stage for the minibatch allocated to it before minibatch j. This time is given by $\tau_1 - p_s - p_{\text{model\_up}} - p_{\text{f\&b}}$.

Thus, we can compute the time when GPU $i + 1$ finishes its preceding tasks before the memory operation for minibatch $j + 1$. This time, denoted $\mathcal{T}_1$, is given by: $\mathcal{T}_1 = \tau_1 - p_s - p_{\text{model\_up}} - p_{\text{f\&b}} - p_{\text{s\_mem\_op}} + p_{\text{t\_mem\_op}} + p_{\text{s\_mem\_op}} + p_{\text{f\&b}} + p_{\text{model\_up}} + p_s = \tau_1 + p_{\text{t\_mem\_op}}$.

Similarly, the time when GPU i finishes its t_mem_op stage for minibatch j, allowing GPU $i + 1$ to begin its $t_{\text{mem\_op}}$ for minibatch $j + 1$, is: $\mathcal{T}_2 = \max(\tau_1, \tau_2) + p_{\text{t\_mem\_op}}$.

The idle time for GPU $i + 1$ while processing minibatch $j + 1$ is:

$$\max(\mathcal{T}_2 - \mathcal{T}_1, 0) = \max(\max(\tau_1, \tau_2) - \tau_1, 0) = \max(t_2 - t_1, 0). \quad (4)$$

This completes the proof. □

Remark The bubble size in Theorem 4.1 demonstrates that as the number of GPUs n increases, the bubble time also increases, a trend that aligns with our empirical findings in 5.4. Furthermore, by substituting the stage durations from Table 2 into the bubble size equation, we observe that PipeTGL achieves nearly zero bubble size when GPU number $n \leq 8$, which is typically sufficient for dynamic graph training.

4.4 Scalability of PipeTGL

4.4.1 Distributed Training for M-TGNN. PipeTGL performs effectively in single-machine, multi-GPU setups, as it leverages high-speed GPU-to-GPU communication links (e.g., NVLink or PCIe). Directly extending the design of PipeTGL—such as round-robin scheduling and direct memory transfer between adjacent GPUs—to a distributed, multi-machine, multi-GPU setup is both unnecessary and potentially detrimental to system performance, especially in clusters without high-speed inter-machine networking. However, **PipeTGL can be safely extended to a multi-machine, multi-GPU distributed setup, provided the dataset fits within a single machine.** This requirement is also consistent with existing multi-machine approaches, such as DistTGL [33].

• **When the dataset fits within a single machine,** the system can maintain at least one complete copy of the node memory on each machine. This allows for leveraging memory parallelism across machines while utilizing the pipeline method in PipeTGL within a single machine/a pipeline group, eliminating the need for node memory synchronization across trainers and enabling the use of more GPUs without increasing the number of GPUs or bubbles within each pipeline group. Let p represent the number of machines and k denote the number of node memory copies to maintain. To eliminate costly node memory synchronization across machines, we require $k \geq p$. The number of node memory copies, k, can be determined based on the hardware configuration.

• **When the dataset is too large to fit on a single machine,** partitioning the dataset becomes necessary. In this case, each machine can only store a portion of the nodes, making memory synchronization across machines unavoidable. Although PipeTGL can be employed across all participating GPUs to maintain memory continuity, optimizing its performance in a multi-machine, multi-GPU scenario with substantial cross-machine memory synchronization overhead remains an open challenge. We leave this as future work.

[3] t_mem_op and s_mem_op account for memory communication; however, we do not explicitly consider them as GPU idle time due to their short duration and the rapid D2D memory transfer.

5 EVALUATION

We implemented PipeTGL based on GNNFlow [31] with PyTorch 1.13.1 [17] and DGL 1.1.1 [25]. All communication between GPUs and machines was handled by the distributed package provided by PyTorch, with NCCL as the backend.

Datasets We evaluated PipeTGL with several widely-used dynamic graph datasets of varying scales. Detailed information about these datasets is presented in Table 3.

• **WIKI** is a user-internet page bipartite graph, with each edge corresponding to a user modifying a page. The features of each edge are derived from the edit document and encoded into a 172-dimensional vector.

• **MOOC** tracks student activities within online courses, where nodes represent either students or courses, and edges indicate interactions between students and courses.

• **REDDIT** captures interactions between users and sub-reddits over the course of a month, with edge features derived from the text of the posts.

• **LASTFM** tracks users' song listening activity over the course of a month, with nodes representing either users or songs.

• **GDELT** is a knowledge graph that records global events from 2016 to 2020, updated every 15 minutes, with each edge representing an individual event.

The task for all five datasets is temporal link prediction.

Table 3: Dataset statistic. The |V| and |E| denote the number of nodes and edges. d_v and d_e represent the dimensions of node features and edge features, respectively. * indicates that the dataset contains incomplete features, which have been generated using random data. t_{max} denotes the maximum edge timestamp.

Dataset	|V|	|E|	d_v	d_e	t_{max}
WIKI [16]	9,227	157,474	128^*	172	2.7e6
MOOC [16]	7,144	411,749	128^*	-	2.6e7
REDDIT [16]	10,984	672,447	128^*	172	2.7e6
LASTFM [10]	1,980	1,293,103	128^*	-	1.3e8
GDELT [32]	16,682	191,290,882	413	182	1.6e8

Models We employ the TGN-attn model [18], which achieves state-of-the-art performance, for our evaluation, same as DistTGL [33]. The model's learning rate was scaled linearly with the number of GPUs, as described in previous works [31, 33]. We followed the same hyperparameter choices as TGN [18], setting the memory dimension for each node to 100, the number of heads in the multi-head attention layer to 2, and the dropout probability to 0.2. For each target node, we sampled its 10 most recent neighbors as supporting nodes to compute its embedding. To balance throughput and information loss, the minibatch size for each dataset was set according to its scale, consistent with the settings used in DistTGL [33]. For smaller datasets (WIKI and MOOC), the minibatch size was set to 600, while for medium-sized datasets (REDDIT and LASTFM), it was set to 1000. Due to GPU capacity constraints, the minibatch size for the large dataset (GDELT) was set to 3200. During training, 80% of the minibatches were used as training data, 10% as validation data, and 10% as test data. The final accuracy for all experiments was calculated based on the test dataset.

Environment All experiments were conducted on machines equipped with NVIDIA GeForce RTX 4090 GPUs, an AMD EPYC 7542 processor, 256 GB of host memory. The default bandwidth between machines was 100 Gbps.

Baselines We compare PipeTGL with existing distributed M-TGNN training systems:

• DistTGL [33], designed to facilitate the distributed training of M-TGNN models. In multi-machine scenarios, DistTGL employs memory parallelism across machines, while within a single machine, it utilizes either minibatch parallelism (denoted as DistTGL_1) or epoch parallelism (denoted as DistTGL_2).

• GNNFlow [31], a general system that supports both static GNNs and temporal GNNs training. Although PipeTGL is built atop GNNFlow, the underlying training approaches differ significantly. GNNFlow uses data parallelism across multiple GPUs without guaranteeing memory dependency handling. To ensure a fair comparison, we implement memory parallelism across machines in GNNFlow, reducing the heavy memory synchronization overhead, while preserving its native training method on a single machine.

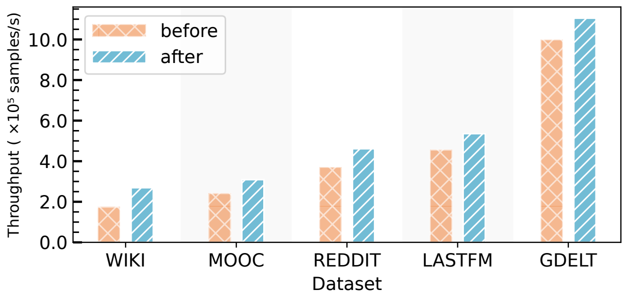


Figure 11: A comparison of throughput before and after enabling communication optimization.

5.1 Overall Performance Comparison

We first compare the end-to-end performance of PipeTGL against the baselines. The experiments were conducted using 2 machines, each equipped with 4 GPUs. PipeTGL leverages memory parallelism across machines and implements pipeline processing with corresponding optimizations on each machine. This approach ensures that each machine retains a complete copy of the features, minimizing slow inter-machine memory synchronization. Each method was trained for 100 epochs to ensure the convergence of the M-TGNN model. Table 4 presents the average precision achieved by each baseline, along with the speedup achieved by PipeTGL when it surpasses the highest accuracy obtainable by the baselines. The average precision of PipeTGL in Table 4 reflects either the precision PipeTGL achieved after convergence or the precision at the point when the corresponding speedup was recorded, as specifically noted.

Overall, PipeTGL achieved the highest average precision across **all** datasets and demonstrated a speedup ranging from **1.27× to**

Table 4: The average precision (AP) after convergence of each baseline, along with the corresponding speedup achieved by PipeTGL. × indicates the speed-up, calculated by dividing the average precision convergence time of each baseline by the time it takes for PipeTGL's precision to surpass the convergence precision of the baseline. − indicates that although its fitting time is shorter than that of PipeTGL, it suffers from a certain loss in accuracy. We found that PipeTGL can achieve the same level of accuracy in less time.

	WIKI		MOOC		REDDIT		LASTFM		GDELT	
	AP(%)	Speedup	AP(%)	Speedup	AP(%)	Speedup	AP(%)	Speedup	AP(%)	Speedup
DistTGL_1	98.00	4.57× (320s)	89.51	2.67× (433s)	98.52	1.52× (256s)	80.14	1.27× (843s)	98.01	1.97× (418s)
DistTGL_2	97.81	4.5× (315s)	89.61	2.88× (466s)	98.00	2.7× (454s)	73.25	- (237s)	97.82	- (205s)
GNNFlow	98.15	4.74× (332s)	87.38	3.57× (579s)	98.50	1.72× (289s)	80.15	1.52× (1005s)	94.31	1.61× (341s)
PipeTGL	98.28	1× (70s)	90.51*	1× (162s)*	98.54	1× (168s)	80.30	1× (661s)	98.07	1× (212s)

*indicates that the accuracy reported is not PipeTGL's convergence accuracy, but rather the accuracy at the point when PipeTGL's speedup is calculated. The convergence accuracy for the MOOC dataset achievable by PipeTGL is 93.41%, with a fitting time of 463 seconds.

4.74× compared to baselines. Specifically, for datasets with fewer nodes, such as WIKI and MOOC, the issue of information loss with the baselines is more pronounced. As a result, the benefits of PipeTGL are more significant. For WIKI, PipeTGL achieved over a 4.7× speedup while maintaining high convergence accuracy, and for MOOC, it delivered more than a 3.5× speedup with consistently higher convergence accuracy. Even on larger-scale datasets, PipeTGL demonstrated a significant acceleration effect, achieving speedups of up to 2.7× across all three datasets, while maintaining high convergence accuracy. Specifically, PipeTGL achieves higher convergence precision with speedups ranging from 1.52× to 2.7× on the REDDIT dataset. On the LASTFM dataset, epoch parallelism (DistTGL_2) fails to converge to satisfactory accuracy due to higher gradient variance, while PipeTGL delivers up to a 1.52× speedup with even better model performance. For the GDELT dataset, PipeTGL reaches a similar accuracy level to the convergence performance of DistTGL_2 in about 60 seconds, after which the accuracy improvement of DistTGL_2 slows down—a limitation that epoch parallelism does not overcome. Compared to GNNFlow, PipeTGL achieves higher accuracy in less time, delivering a 1.61× speedup. Although minibatch parallelism (DistTGL_1) can achieve comparable performance, it requires roughly twice the time of PipeTGL.

For minibatch parallelism (DistTGL_1) and epoch parallelism (DistTGL_2), the negative impact of epoch parallelism on convergence accuracy is more significant, leading to slightly lower accuracy compared to minibatch parallelism due to the large gradient variance introduced by this method. In contrast, PipeTGL maintains high convergence accuracy thanks to our memory continuity assurance. While minibatch parallelism achieves faster training speeds than epoch parallelism by directly ignoring memory dependencies among in-flight minibatches, PipeTGL outperforms both approaches by leveraging an optimized execution plan and communication strategy, achieving significantly faster training speeds while still respecting memory continuity. Compared to GNNFlow, which also neglects memory continuity, PipeTGL achieves significantly improved model performance and a speedup ranging from 1.52× to 4.74×. This demonstrates that, despite being built on top of GNNFlow, PipeTGL's memory continuity-assured pipeline training method and its optimizations to eliminate pipeline bubbles make it superior to GNNFlow.

5.2 The Effect of Memory Reordering

To reduce the waiting time caused by memory continuity requirements across minibatches, we reorder and prioritize memory-related operations for target nodes. Next, we evaluate the memory-dependent time during training using a machine with 4 GPUs, both before and after implementing memory reordering. Table 5 presents the evaluation results. For **all** datasets, memory reordering effectively reduces GPU execute time for update memory from by approximately **50%**, demonstrating the effectiveness of memory reordering in reducing pipeline bubbles.

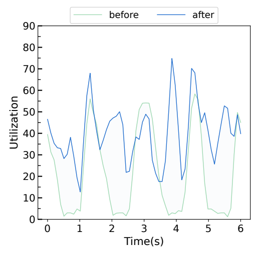
Table 5: The proportion of the memory-dependent time before and after memory reordering.

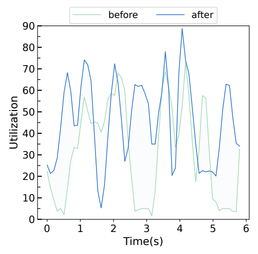
Dataset	WIKI	MOOC	REDDIT	LASTFM	GDELT
before reordering	20.9%	21.3%	22.1%	19%	21.0%
after reordering	11.4%	10.0%	13.5%	10.9%	10.6%

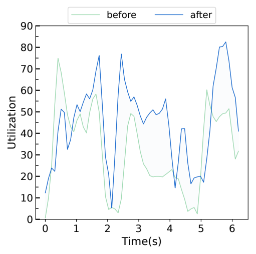
5.3 The Effect of Communication Optimization

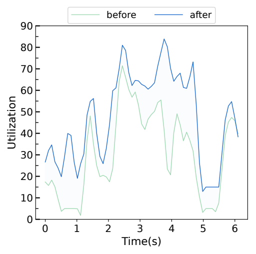
The *pre-sampling* technique employed by PipeTGL enables the overlapping of communication and computation. Additionally, PipeTGL replaces a portion of the H2D and D2H communication for memory fetching with direct D2D communication. Next, we evaluate the system's performance both before (with all communication optimization options in PipeTGL disabled) and after applying the communication optimization techniques in PipeTGL to demonstrate their effectiveness.

We compared GPU utilization before and after enabling communication optimizations across various datasets, as illustrated in Figure 12. A continuous segment of the training process was randomly selected, and GPU utilization was sampled at a frequency of 0.1 seconds. As shown in Figure 12, our optimization method increased peak GPU utilization by over 10% and significantly improved average GPU utilization. Specifically, for the five datasets—WIKI, MOOC, REDDIT, LASTFM, and GDELT—the average GPU utilization before


(a) WIKI


(b) MOOC


(c) REDDIT


(d) LASTFM

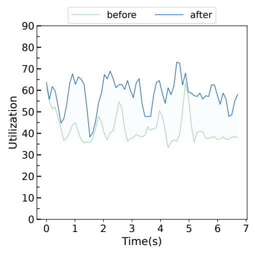

(e) GDELT

Figure 12: A comparison of GPU utilization before and after enabling communication optimization.

communication optimization is 18.35%, 29.21%, 27.31%, 26.57%, and 42.12%, respectively. After enabling communication optimization, the average GPU utilization increases to 33.00%, 40.53%, 39.79%, 39.07%, and 58.14%, respectively. Additionally, we measured system throughput before and after enabling the communication optimization techniques, as shown in Figure 11. The results demonstrate that our communication optimization method improved system throughput by approximately 25%, significantly enhancing the efficiency of PipeTGL.

5.4 Overall Bubble Rate of PipeTGL

We evaluate the size of the pipeline bubble throughout the entire training process for PipeTGL, which incorporates all the optimization techniques. The evaluations were conducted using varying numbers of GPUs, comparing the bubble size before and after optimization under various datasets. The results, shown in Figure 13, indicate that PipeTGL achieves nearly **zero pipeline bubble**, with less than 8% GPU idle time during the entire training process. Additionally, it can be observed that the bubble size increases with the number of GPUs, as more memory dependencies are introduced. This observation aligns with the theoretical analysis of pipeline bubble size for PipeTGL presented in Theorem 4.1.

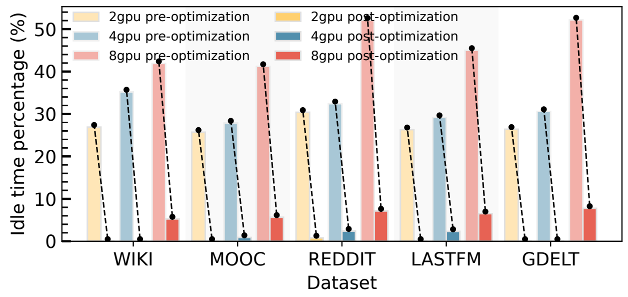


Figure 13: The proportion of idle time to total training time with different numbers of GPUs.

5.5 Convergence Results

We compare the convergence process across various parallelism strategies for M-TGNNs. Specifically, we trained the REDDIT dataset for 50 epochs using 4 GPUs with four different parallel methods to assess their convergence efficiency. The results are shown in Fig.14(a). The x-axis represents the training epoch number, allowing us to eliminate the runtime differences and enable a fair comparison of convergence performance. Among the various parallelism strategies for M-TGNN training, epoch parallelism shows the poorest convergence performance due to high gradient descent variance. This is also why DistTGL prioritizes minibatch parallelism and memory parallelism over epoch parallelism. Minibatch parallelism, while faster than epoch parallelism, suffers from slower convergence and lower accuracy than PipeTGL and memory parallelism because it neglects the memory continuity between in-flight minibatches. In contrast, PipeTGL and memory parallelism exhibit similar convergence speed and accuracy, but PipeTGL requires significantly less storage space. As a result, PipeTGL offers the best trade-off between hardware resource efficiency and convergence performance.

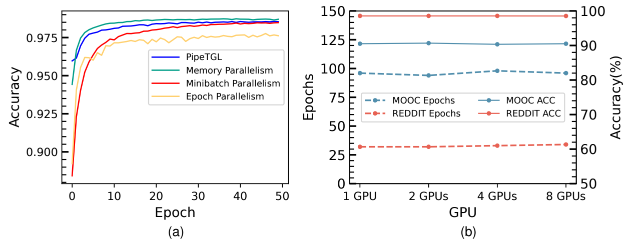


Figure 14: (a) Convergence curves for different parallel methods on the REDDIT dataset. (b) Epochs requested for fitting and accuracy under varying number of GPUs.

5.6 Performance Analysis Under Scaling Conditions

5.6.1 The impact of training cluster configuration. We evaluate the system performance of PipeTGL across various training cluster setups, measuring the required training epochs for convergence, the model accuracy post-convergence, and the corresponding runtimes. Since graphs with varying characteristics may display distinct convergence properties, we use the MOOC and REDDIT datasets to assess the impact of training setups on different types of graphs. The MOOC dataset exhibits a dense structure, whereas the REDDIT dataset is comparatively sparse. The results in Fig. 14(b) show that PipeTGL maintains consistent convergence epochs and accuracy across different GPU configurations. This demonstrates that PipeTGL effectively prevents the increase in convergence epochs and the decrease in accuracy as the number of GPUs rises, thanks to its memory continuity guarantee. In contrast, the system performance of minibatch parallelism shows an increase in required training epochs as the number of GPUs grows, due to increased information loss, as illustrated in Fig. 3. Table 6 presents the end-to-end runtimes of PipeTGL with varying numbers of GPUs. More GPUs introduce greater parallelism, which generally leads to shorter training times. However, PipeTGL can achieve runtimes similar to the baselines even when using only a quarter of the GPUs compared to the baseline setup for the MOOC and REDDIT datasets (as shown in the 2-GPUs results in Table 6, which are comparable to the baselines in Table 4), demonstrating the superiority of PipeTGL. Furthermore, we observe that the speedup in training efficiency is lower than the proportional increase in the number of GPUs. This is due to increased contention for PCIe lanes between GPUs and a rise in the number of bubbles in the pipeline, as the GPU count grows, as outlined in Theorem 4.1.

Table 6: End-to-end runtimes with varying numbers of GPUs.

Number of GPUs	1 GPU	2 GPUs	4 GPUs	8 GPUs
MOOC	768s	470s	274s	162s
REDDIT	496s	291s	198s	168s

5.6.2 Average waiting time per minibatch. We evaluated the GPU waiting time caused by memory dependencies for the MOOC and REDDIT datasets across varying numbers of GPUs, with the results shown in Fig. 15. In the 4GPU*2 machine setup, we leverage the pipeline design proposed by PipeTGL within each machine and memory parallelism across machines. When the number of GPUs in a single pipeline group is 4 or fewer, the GPU waiting time due to mini-batch dependencies is 0%. However, when the number of GPUs increases to 8, the waiting times for the MOOC and REDDIT datasets rise to 5.66% and 7.14%, respectively, due to the extended critical path. These results also align with our calculations in Section 4.3. Based on this, we recommend limiting the number of GPUs in a single pipeline group to 4 or fewer in PipeTGL to minimize this overhead.

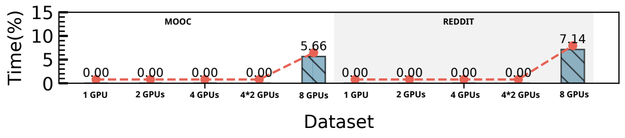


Figure 15: The waiting time due to memory dependency under different numbers of GPUs, where the vertical axis represents the proportion of waiting time in the total training time.

6 RELATED WORKS

Research on dynamic graph neural network learning can be divided into two main areas, each focusing on a different type of dynamic graph: discrete-time dynamic graphs (DTDGs) [15, 16, 30] and continuous-time dynamic graphs (CTDGs) [18, 19, 28]. In the context of DTDGs, several studies [1, 2, 4, 6, 24] focus on accelerating the training process at the system level. PiPAD [24] offers a method for aggregating repeated regions across multiple snapshots and parallelizes communication and computation through pipelining. EDEG [1] introduces a gradient checkpointing technique to reduce the memory requirements for dynamic graphs and proposes a graph-difference-based transfer method for DTDGs. However, these approaches are not applicable to CTDGs.

In CTDGs, several studies [3, 5, 12, 13, 20, 28, 32] also aim to accelerate the training process of TGNNs from both algorithmic and system perspectives. TGL [32] provides a general framework for training on CTDGs. Orca [12] introduces a cache-based method that reduces the time required for forward propagation in M-TGNNs by reusing stale embeddings. TGOpt [27] focuses on minimizing redundant computations in TGAT [28]. However, these approaches do not address distributed training.

DistTGL [33] and GNNFlow [31] introduce several distributed training methods, but their approaches have various limitations. In terms of pipelining, PipeDream [14] was among the first to introduce the concept, providing a foundational analysis of its impact on the training process. PipeGCN [23] further developed this idea by introducing a method for using pipelining in GNNs to parallelize data transfer and computation. However, due to the unique characteristics of M-TGNNs, these methods cannot be directly applied.

7 CONCLUSION

In this work, we introduce PipeTGL, a parallel training system meticulously designed to address the challenges of M-TGNN training under strict memory continuity requirements. We conduct an in-depth analysis of the limitations inherent in existing M-TGNN training systems, pinpointing critical bottlenecks and unique challenges associated with training M-TGNNs under memory dependencies. Leveraging the insights, PipeTGL employs a pipeline parallel approach that strategically integrates a suite of optimization techniques. These optimizations encompass critical aspects such as node placement, execution order, and communication strategies, all working in concert to accelerate the training process. Through these enhancements, PipeTGL not only achieves significant speedups but also ensures the elimination of pipeline bubbles, resulting in a more efficient training system that outperforms traditional methods.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (NSFC) (Grant No. 62302187 and 62302183), the Natural Science Foundation of Hubei Province (Grant No. 2024AFB047), Open Foundation of Key Laboratory of Cyberspace Security, Ministry of Education of China (Grant No. KLCS20240401), and CCF-DiDi GAIA Collaborative Research Funds (Grant No. CCF-DiDi GAIA 202412).

REFERENCES

[1] Venkatesan T Chakaravarthy, Shivmaran S Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, and Shashanka Ubaru. 2021. Efficient scaling of dynamic graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[2] Chaoyi Chen, Dechao Gao, Yanfeng Zhang, Qiange Wang, Zhenbo Fu, Xuecang Zhang, Junhua Zhu, Yu Gu, and Ge Yu. 2023. NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams. *Proceedings of the VLDB Endowment* 17, 3 (2023), 455–468.

[3] Gangda Deng, Hongkuan Zhou, Hanqing Zeng, Yinglong Xia, Christopher Leung, Jianbo Li, Rajgopal Kannan, and Viktor Prasanna. 2024. TASER: Temporal Adaptive Sampling for Fast and Accurate Dynamic Graph Representation Learning. *arXiv preprint arXiv:2402.05396* (2024).

[4] Kaihua Fu, Quan Chen, Yuzhuo Yang, Jiuchen Shi, Chao Li, and Minyi Guo. 2023. BLAD: Adaptive Load Balanced Scheduling and Operator Overlap Pipeline For Accelerating The Dynamic GNN Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[5] Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. ETC: Efficient Training of Temporal Graph Neural Networks over Large-scale Dynamic Graphs. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1060–1072.

[6] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. Dynagraph: dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–10.

[7] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems* 30 (2017).

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in Neural Information Processing Systems* 32 (2019).

[9] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[10] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data mining*. 1269–1278.

[11] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 583–598.

[12] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Orca: Scalable Temporal Graph Neural Network Training with Theoretical Guarantees. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[13] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Zebra: When temporal graph neural networks meet temporal personalized pagerank. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1332–1345.

[14] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[15] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. 2021. Transfer graph neural networks for pandemic forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 4838–4845.

[16] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 5363–5370.

[17] Adam Paszke. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[18] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637* (2020).

[19] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 519–527.

[20] Junwei Su, Difan Zou, and Chuan Wu. 2024. PRES: Toward Scalable Memory-Based Dynamic Graph Neural Networks. *arXiv preprint arXiv:2402.04284* (2024).

[21] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. Dyrep: Learning representations over dynamic graphs. In *Proceedings of International Conference on Learning Representations*.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 30 (2017).

[23] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv preprint arXiv:2203.10428* (2022).

[24] Chunyang Wang, Desen Sun, and Yuebin Bai. 2023. PiPAD: pipelined and parallel dynamic GNN training on GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 405–418.

[25] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[26] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, and Zhenyu Guo. 2021. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 International Conference on Management of Data*. 2628–2638.

[27] Yufeng Wang and Charith Mendis. 2023. TGOpt: Redundancy-aware optimizations for temporal graph attention networks. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 354–368.

[28] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962* (2020).

[29] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. 2021. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems* 3 (2021), 269–296.

[30] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. 2019. T-gcn: A temporal graph convolutional network for traffic prediction. *IEEE Transactions on Intelligent Transportation Systems* 21, 9 (2019), 3848–3858.

[31] Yuchen Zhong, Guangming Sheng, Tianzuo Qin, Minjie Wang, Quan Gan, and Chuan Wu. 2023. GNNFlow: A Distributed Framework for Continuous Temporal GNN Learning on Dynamic Graphs. *arXiv preprint arXiv:2311.17410* (2023).

[32] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: A general framework for temporal GNN training on billion-scale graphs. *arXiv preprint arXiv:2203.14883* (2022).

[33] Hongkuan Zhou, Da Zheng, Xiang Song, George Karypis, and Viktor Prasanna. 2023. DistTGL: Distributed Memory-Based Temporal Graph Neural Network Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.