RESEARCH-ARTICLE

Effective and Efficient Distributed Temporal Graph Learning through Hotspot Memory Sharing

LONGJIAO ZHANG, Zhejiang University, Hangzhou, Zhejiang, China

RUI WANG, Zhejiang University, Hangzhou, Zhejiang, China

TONGYA ZHENG, Hangzhou City University, Hangzhou, Zhejiang, China

ZIQI HUANG, Zhejiang University, Hangzhou, Zhejiang, China

WENJIE HUANG, Zhejiang University, Hangzhou, Zhejiang, China

XINYU WANG, Zhejiang University, Hangzhou, Zhejiang, China

View all

Published: 01 May 2025

Effective and Efficient Distributed Temporal Graph Learning through Hotspot Memory Sharing

Longjiao Zhang
Zhejiang University
zhljJoan@zju.edu.cn

Rui Wang*
Zhejiang University
High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
rwang21@zju.edu.cn

Tongya Zheng
Zhejiang Key Laboratory of Big Data Intelligent Computing, Hangzhou City University
doujiang_zheng@163.com

Ziqi Huang
Zhejiang University
ziqi@zju.edu.cn

Wenjie Huang
Zhejiang University
wjie@zju.edu.cn

Xinyu Wang
Zhejiang University
wangxinyu@zju.edu.cn

Can Wang
Zhejiang University
wcan@zju.edu.cn

Mingli Song
Zhejiang University
brooksong@zju.edu.cn

Sai Wu
Zhejiang University
wusai@zju.edu.cn

Shuibing He
Zhejiang University
heshuibing@zju.edu.cn

ABSTRACT

Memory-based temporal graph neural network (MTGNN) models are effective for predicting temporal graphs by using node memory and message-passing modules to capture temporal and structural information, respectively. However, distributed training for large graphs presents challenges such as accuracy loss and decreased efficiency due to remote features and memory transmission. Despite improvements in MTGNN system optimizations, issues like dynamic load imbalances, communication overhead, and memory staleness persist. To tackle these challenges, we introduce MemShare, a distributed MTGNN system. MemShare introduces a novel shared node memory paradigm that utilizes a small subset of shared nodes across machines and GPUs to reduce distributed communication for memory management. It incorporates techniques like shared nodes-centric graph partitioning, shared nodes-aware boundary decay sampling, and shared nodes-targeted synchronous smoothing aggregation. Experiments show that MemShare outperforms existing distributed MTGNN systems in accuracy and training efficiency.

PVLDB Reference Format:
Longjiao Zhang, Rui Wang, Tongya Zheng, Ziqi Huang, Wenjie Huang, Xinyu Wang, Can Wang, Mingli Song, Sai Wu, Shuibing He. Effective and Efficient Distributed Temporal Graph Learning through Hotspot Memory Sharing. PVLDB, 18(9): 3093 - 3105, 2025.
doi:10.14778/3746405.3746430

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zhljJoan/MemShare.

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746430

1 INTRODUCTION

Temporal graph data, which captures evolving node and edge relationships, is increasingly utilized in applications like social networks [1, 2], transportation networks [3, 4], and financial transactions [5, 6]. Temporal graph neural networks (TGNNs) [7, 8] are crucial for learning representations from these graphs, modeling complex temporal and structural relationships. Among them, memory-based TGNNs (MTGNNs) [7–10] effectively capture evolving dynamics through integrated node memory and message-passing modules to enhance predictive accuracy

Efficient MTGNN training and inference depend on frameworks like TGL [11], which abstracts training into core components and optimizes temporal graph storage and batch parallelism for faster training. As graph sizes increase, distributed parallel MTGNN computing becomes essential, utilizing multiple GPUs or machines for training [12–17]. These frameworks divide the graph into subgraphs, with each GPU managing a portion and handling data retrieval, memory updates, and message aggregation in parallel. However, communication bottlenecks can arise from distributed feature retrieval and memory aggregation across remote GPUs and machines [14, 16]. To reduce communication costs, various optimizations have been explored in distributed MTGNN training, including memory replicas to eliminate communication during batch training [17], distributed cache strategies to lower data retrieval overhead [15, 16, 18], and parallelizing communication and computation through background thread execution to minimize synchronization delays [13, 14, 18]. Optimizations also focus on graph partitioning to minimize edge cuts, achieve load balance, and reduce communication and synchronization costs [13, 14, 16].

Despite advancements in distributed MTGNN frameworks, several limitations persist. Firstly, existing graph partitioning algorithms struggle to maintain a dynamically balanced distribution as the graph evolves, leading to high synchronization waiting costs during training. While DisTGL [16] attempts to mitigate this by keeping track of an average timestamp for each partition to maintain a balanced temporal distribution, it fails to evenly distribute the

number of events among partitions within each batch. For example, when applying DisTGL to the WikiTalk dataset [19], the highest training load can exceed the average by more than two times in each batch on average. Secondly, communication overhead significantly affects training time, particularly during remote feature fetching and memory aggregation. In methods like MSPipe [18], it can exceed 56.7% of total time with models like TGN [7], as temporal dependencies prevent background processing. Additionally, distributed training increases batch sizes, leading to greater information loss and memory staleness [20]. Delayed synchronization of cached node memory further worsens these issues, impacting model accuracy and convergence [16, 18, 21].

To address the above challenges, we present MemShare, an efficient distributed MTGNN training and inference system. MemShare introduces a novel *shared node memory paradigm* that entails sharing a small subset of hot nodes across machines to reduce distributed communication and memory management costs. Expanding on this paradigm, we design several key techniques, i.e., shared node-centric graph partitioning, shared node-aware boundary decay sampling, and shared node-targeted synchronous smoothing aggregation. In summary, our contributions are as follows:

- We introduce a novel shared node memory paradigm that involves sharing a small subset of high-degree nodes across machines. Our research has revealed that this small subset of high-degree nodes serves as key connectors for the remaining nodes in the graph, significantly reducing the number of cut edges across subgraphs and minimizing communication overhead during MTGNN training.
- We propose three key optimizations based on the shared node memory paradigm: (1) a shared nodes-centric graph partitioning method that maximizes local neighbor attention weights while dynamically balancing workloads; (2) a boundary decay sampling technique that reduces communication by lowering remote neighbor sampling; and (3) a smoothing memory aggregation strategy that synchronizes shared nodes only during significant memory changes, minimizing overhead while maintaining freshness. Additionally, we propose pipeline-based adaptive parameter tuning and analyze communication patterns for various shared node configurations.
- We implement MemShare and conduct extensive experiments to demonstrate its effectiveness. Our results show that MemShare achieves the dominant highest accuracy, with training and inference speeds improved by several times, compared to the cutting-edge distributed methods TGL [11], DistTGL [17], and MSPipe [18].

2 BACKGROUND AND MOTIVATION

2.1 Temporal Graph Neural Network

Dynamic graphs. We focus on the event-based organization for dynamic graphs, specifically continuous-time dynamic graphs (CTDGs) [22]. A CTDG can be defined as a sequence of interaction events $G = (e(u_i, v_i, t_i))_{i=1}^{n}$ where each event $e(u_i, v_i, t_i)$ is a directed temporal edge from node u_i to node v_i at timestamp t_i. For simplicity, we represent a dynamic graph as $G = (\boldsymbol{V}, \boldsymbol{E}, \boldsymbol{X}_N, \boldsymbol{X}_E)$,

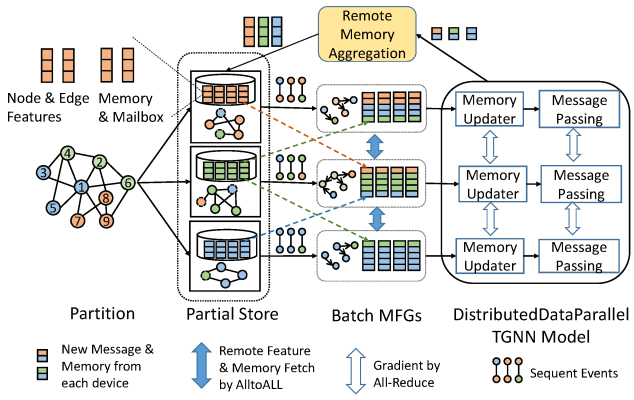


Figure 1: Distributed MTGNN training workflow.

where $\boldsymbol{V}$ and $\boldsymbol{E}$ are the node set and edge set, and $\boldsymbol{X}_N$ and $\boldsymbol{X}_E$ are the feature matrices of nodes and edges.

Memory-based temporal graph neural networks (MTGNN). MTGNN models capture both structural and temporal characteristics in dynamic graphs [7–10]. Each node u has a memory state $s(u, t)$ to track prior interactions, with t as the timestamp of the latest event involving u. The node memory is continuously updated with new events. MTGNN models involve two main operations:

(1) Node memory updating: When an event $e(t) = (u, v, t)$ occurs, messages m_e^{uv} are generated for nodes u and v at timestamp t:

$$m_e^{uv} = \text{MSG}(s(u, t^-)||s(v, t^-)||e(u, v, t)||\phi(t - t^-)). \quad (1)$$

Here, $s(u, t^-)$ and $s(v, t^-)$ are the recent memory vectors for nodes u and v before timestamp t. $\phi(\cdot)$ is a time encoding function, $||$ denotes concatenation, and MSG function generates final messages.

These messages are utilized to update the memories of the corresponding nodes:

$$s(u, t) = \text{UPDATE}(s(u, t^-), m_e^{uv}). \quad (2)$$

The UPDATE function typically involves an RNN [8] or GRU [7], with some models incorporating an attention mechanism [9].

(2) Temporal message propagation: After node memory updating, the message passing layer in MTGNN computes node embeddings $h_u(t)$ and $h_v(t)$ by aggregating their temporal neighbors. Let $N_u(t)$ be the temporal neighbor set for node u at timestamp t. The temporal message-passing process is described as follows:

$$\tilde{N}_u(t) = \text{SAMPLE}(N_u(t)), \quad (3)$$

$$g_u(t) = \text{AGGREGATE}(\{(s(v, t^-)||e_{uv}(\tau)||\phi(t - \tau)) \quad (4)$$

$$|(v, \tau) \in \tilde{N}_u(t)\}), \quad (5)$$

$$h_u(t) = \text{COMB}(s(u, t^-), g_u(t)). \quad (6)$$

Here, the SAMPLE function selects neighbors using sampling techniques like recent[7], uniform[23], weighted[24, 25], or adaptive methods[26]. $\phi(\cdot)$ encodes the time interval, and AGGREGATE aggregates neighbors using functions like GAT [23]. The COMB function is usually a linear layer that updates the node's memory with the aggregated neighbor information [11].

2.2 Distributed MTGNN Training

Distributed training for MTGNNs faces challenges due to dynamic node memory synchronization. While prior work (e.g., TGL [11], SPEED [12]) focused on single-machine multi-GPU setups, scaling to larger graphs necessitates multi-machine collaboration for parallel batch training. As illustrated in Figure 1, the graph is partitioned across machines, each handling a subgraph with nodes, edges, features, and memory. Training occurs chronologically, with each machine processing a batch of edges from its subgraph. They sample negative samples, select temporal neighbors, and retrieve nodes, edges, static features, and memory from local and remote machines. The memory updater generates messages and updates memory for positive root nodes, which are aggregated by index and timestamp with remote machines. Message passing updates node embeddings, with aggregated messages stored for future training. Remote feature fetching and memory aggregation introduce communication bottlenecks [13, 14]. Recent approaches optimize graph partitioning (e.g., balancing edge cuts and timestamps [16]) or dynamically repartition batches [13]. Some utilize caching for remote features and memory [15, 16], though stale memory can degrade accuracy [16, 18]. DisTGL [16] synchronizes caches based on miss thresholds, while DistTGL [17] replicates memory to avoid synchronization, incurring redundancy. MSPipe [18] and Sven [13] overlap communication and computation via background threads, reducing synchronization delays.

2.3 Limitation on Existing Training System

Limitation #1: High synchronization waiting overhead due to dynamic load imbalances. In the distributed MTGNN training process, each trainer sequentially processes a data subset per batch and synchronizes memory after each batch. Existing graph partitioning methods overlook data distribution within batches, leading to dynamic load imbalances. This results in some trainers facing significantly higher computational overhead than the average, causing increased synchronization waiting times. To illustrate this issue, we tested computation load balancing using the METIS [27] graph partitioning method and the recent temporal-aware method from DisTGL [16] on the StackOF [28] dataset. As shown in Figure 2, the x-axis represents the batch index, while the y-axis indicates the average and maximum computational load across eight GPUs. As batch IDs increase, computational load rises due to accumulating temporal neighbors. Both METIS and DisTGL exhibit significant load imbalances, especially in early training batches, with maximum loads often exceeding twice the average. This imbalance stems from their holistic graph partitioning approach, which fails

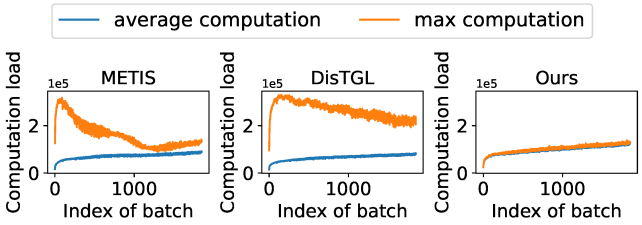


Figure 2: Dynamic load imbalance in existing graph partition.

Table 1: Time cost breakdown (seconds/percentage).

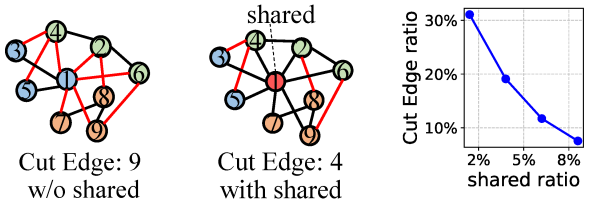
Stage	MSPipe	DisTGL*	Ours
Sampling	59.4/7.56%	38.5/4.91%	65.6/17.79%
Remote Fetching	107.5/13.68%	536.1/68.30%	99.0/26.85%
Memory Updating	48.0/6.11%	46.6/5.94%	46.6/12.63%
Message Passing	40.29/5.13%	43.2/5.50%	46.3/12.57%
Backward	192.6/24.51%	95.4/12.15%	97.7/26.50%
Remote Aggregation	338.0/43.02%	25.1/3.20%	13.49/3.66%

to distribute events within batches evenly. In contrast, our proposed shared nodes-centric graph partitioning method achieves a balanced distribution of computational load across eight partitions, maintaining the highest computation volume closely aligned with the average in each batch. This balance improves training efficiency in our MemShare framework (see §3.2).

Limitation #2: High communication overhead in memory fetching and updating. Despite various optimizations, communication overhead remains the main bottleneck in training time. We tested MSPipe [18] and DisTGL* [16] [1] on the GDELT [29] dataset with a batch size of 3000, using the TGN [7] model across two machines with eight GPUs and a 10 Gbps network. As shown in Table 1, communication overhead is significant; in MSPipe, remote memory aggregation via all-gather accounted for 43.02% of training time, while DisTGL's all-to-all feature fetching took 68.30%. These communication phases are more time-consuming than computation tasks and cannot be effectively overlapped due to memory dependencies. MSPipe reduces fetching costs by redundantly storing data, but incurs high synchronization costs during remote aggregation. In contrast, DisTGL lowers synchronization costs through edge partitioning but increases remote fetch communication. Our approach reduces remote fetches using shared hot node memory and node-aware boundary decay sampling, cutting remote fetching time from 536.1 seconds to 99.0 seconds (about 1/5). We also minimize synchronization costs through dynamic balanced graph partitioning and historical memory techniques, reducing remote aggregation time to 13.49 seconds (see §3.3).

Limitation #3: Memory staleness from delayed synchronization. Delays in cache-related memory synchronization [12, 16] lead to issues such as update lag, remote information loss, and insufficient event capture, worsening event discontinuity, and degrading model accuracy [21]. A key problem is the obsolescence of node memory inputs due to infrequent updates from remote machines [16]. While strategies like staleness thresholds and similarity-based aggregation [16, 18] aim to address this, the lack of access to the latest information from remote nodes results in persistent update lag. Furthermore, in distributed settings, local node memories may miss valuable data from remote machines, as updates rely solely on local events [7, 8, 16]. Additionally, aggregating events across multiple devices reduces updates per batch for the MTGNN model, further exacerbating event discontinuity and accuracy loss [17, 20, 21]. To address these challenges, we propose a novel shared nodes-targeted synchronous smoothing aggregation method to effectively reduce memory staleness and improve accuracy (see §3.4).

[1] DisTGL is not open-sourced; we extended TGL for multi-machine environments and reproduced its core streaming temporal-aware edge partitioning method.



(a) w/o shared nodes (b) with shared nodes (c) Edge-cut Counting

Figure 3: Decreased number of cross edges via shared nodes. In (a) and (b), nodes of different colors are allocated to different partitions, with red nodes being shared nodes. Red-colored edges represent cross-edges.

3 SHARED NODE MEMORY PARADIGM

3.1 Overview

Observation. In distributed MTGNN training, primary communication costs arise from remote feature fetching and memory aggregation, largely due to cut edges between subgraphs [30]. Traditional graph partitioning methods [14, 16, 27, 31] often fail to minimize these cut edges due to complex node connections.

High-degree nodes serve as critical connectors, especially in power-law graphs where a few nodes have exceptionally high degrees [32, 33]. By sharing a small set of high-degree nodes, we can significantly reduce cut edges. As shown in Figure 3(a) and Figure 3(b), sharing node v_1 decreases the cut edges (highlighted in red) among three partitioned subgraphs from nine to four.

We experimented with partitioning the GDELT dataset into eight subgraphs using the METIS [27] method, measuring the cut edge ratio with varying shares of high-degree nodes. Results in Figure 3(c) indicate that sharing the top 10% of high-degree nodes reduced the cut edge ratio from over 30% to 8%. In a power-law graph, it can be shown that the number of cut edges generated after sharing high-degree nodes will not exceed:

$$\mathrm{EC} \leq \frac{1}{|E|} \sum_{q=0}^{|V|(1-k)-1} m(k + \frac{q}{|V|})^{\frac{1}{1-\alpha}}, \tag{7}$$

where k is the ratio of shared high-degree nodes, m is the minimum node degree in the graph, and α is a parameter that indicates the skewness of the graph. For a detailed proof, refer to [12].

Main idea. We propose a shared node memory paradigm that allows high-degree nodes to exist in multiple subgraphs, enhancing local event capture [12]. This significantly reduces communication costs in distributed MTGNN training by minimizing access to critical nodes. While SPEED [12] shares weighted top-k nodes to reduce replication in single-machine multi-GPU (SM-MG) training, our focus is on multi-machine multi-GPU (MM-MG) scenarios, where communication costs from remote data access and synchronization are significant challenges. First, we aim to optimize shared node-centric graph partitioning to minimize edge cuts while ensuring dynamic load balancing. The node-cut method in SPEED is inadequate due to remote access and synchronization overhead. Second, sharing hot nodes' memory may still lead to frequent remote access, which SPEED does not address. Lastly, frequent synchronization for aggregation adds communication overhead in multi-machine setups, unlike SPEED's efficient single-machine approach.

Overview. To address the challenges mentioned, we design a shared node memory paradigm in multi-machine scenarios. We first introduce a shared nodes-centric graph partitioning method, which prioritizes maximizing the attention weight of local neighbors to enhance neighbor locality within partitions and achieve dynamic load balancing. Additionally, we propose a shared nodes-aware boundary decay sampling technique to reduce communication volume by decreasing the sampling probability of remote neighbors. We also introduce a shared nodes-targeted synchronous smoothing aggregation method that synchronizes shared nodes' memory only when substantial changes happen, thereby reducing communication overhead while maintaining memory freshness. These three techniques are elaborated on in the subsequent subsections.

3.2 Shared Nodes Centric Graph Partitioning

We propose a unique shared nodes-centric graph partitioning method that integrates neighbor-weight awareness and load-balancing coefficients to address dynamic load imbalances.

Neighbor weight awareness. Existing graph partitioning methods [14, 16, 27, 31] primarily aim to minimize cut edges, often overlooking the varying importance of neighbors based on their timestamps. In TGNN training, the attention weight of an edge event diminishes as more events accumulate, leading to exponentially distributed weights for neighbors at different timestamps [12, 25, 34]. The attention aggregation weight of a temporal neighbor (v, τ) contributing to the embedding of node u at timestamp t is defined as follows:

$$att_{uv}(t, \tau) \propto \mathrm{e}^{\delta(\tau - t)}, \tag{8}$$

where $att_{uv}(t, \tau)$ represents the attention aggregation weigh and δ is an exponential decay factor.

We aim to maximize neighbor aggregation information within local subgraphs to minimize reliance on remote data, targeting:

$$\text{maximize} \sum_{(u,\cdot,t)\in E} \sum_{(v,\tau)\in N_u^L(t)} att_{uv}(t, \tau), \tag{9}$$

where $N_u^L(t)$ is the local neighborhood of node u.

To efficiently approximate optimal graph partitioning solutions, we apply the streaming heuristic algorithm paradigm [12, 16]. With m trainers, the graph $G(\boldsymbol{V}, \boldsymbol{E}, \boldsymbol{X}_N, \boldsymbol{X}_E)$ is divided into m subgraphs $G_0, G_1, \ldots, G_{m-1}$, with each trainer managing their assigned subgraph. Events with their features are allocated to the partition with the highest sum of attention weights among neighboring partitions. Edge $e(u, v, t)$ is assigned to partition p with the highest score according to the formula:

$$p = \arg\max_{p_i} [Att(u, t, p_i) + Att(v, t, p_i) + 1] \cdot F_{BAL}(p_i), \tag{10}$$

where $Att(u, t, p_i)$ is the sum of estimated attention weights from u's previously assigned neighbors in partition p_i, and $F_{BAL}(p_i)$ is the balance constraint coefficient. $Att(u, t, p_i)$ is estimated as:

$$Att(u, t, p_i) = \sum_{(v,\tau)\in N_u(t), e(u,v,\tau)\in E_{p_i}} \mathrm{e}^{\delta(\tau - t)}, \tag{11}$$

where δ is set as $\frac{1}{\max_{e(u,v,t)\in E} t}$.

Dynamic load balance. In Equation 10, the balance constraint coefficient $F_{BAL}(p_i)$ is designed for dynamic load balance:

$$F_{BAL}(p_i) = BN(p_i) \cdot BE(p_i) \cdot BT(p_i), \tag{12}$$

$$BN(p_i) = 1 - \frac{|\boldsymbol{N}_{p_i}| - \min_{p_j<m} |\boldsymbol{N}_{p_j}|}{\epsilon + \max_{p_j<m} |\boldsymbol{N}_{p_j}| - \min_{p_j<m} |\boldsymbol{N}_{p_j}|}, \tag{13}$$

$$BE(p_i) = 1 - \frac{|\boldsymbol{E}_{p_i}| - \min_{p_j<m} |\boldsymbol{E}_{p_j}|}{\epsilon + \max_{p_j<m} |\boldsymbol{E}_{p_j}| - \min_{p_j<m} |\boldsymbol{E}_{p_j}|}, \tag{14}$$

$$BT(p_i) = \exp\left(\frac{\min_{p_j<m} T_{p_j} - T_{p_i}}{\epsilon + \max_{p_j<m} T_{p_j} - \min_{p_j<m} T_{p_j}}\right), \tag{15}$$

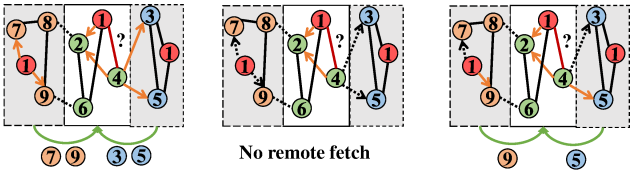
where $BN(p_i)$, $BE(p_i)$, and $BT(p_i)$ represent the scores for node balance, edge balance, and temporal balance, respectively. Specifically, $BN(p_i)$ and $BE(p_i)$ normalize deviations in node and edge counts in partition p_i to ensure balanced distributions, with $\boldsymbol{N}_{p_i}$ and $\boldsymbol{E}_{p_i}$ being the sets of nodes and edges in p_i. New events are prioritized for partitions with fewer nodes or edges to maintain minimal allocation differences over time, with ϵ preventing division by zero. $BT(p_i)$ measures the difference in event time relative to partition p_i, with T_{p_i} representing the timestamp of the last event assigned to partition p_i. New events are prioritized for partitions with smaller timestamps, ensuring a balanced timestamp distribution. The exponential function smooths out outliers.

Shared nodes-centric graph partitioning. In the graph partitioning process, we begin by selecting the top-k nodes with the highest node degrees as shared nodes present in every partition. Next, we iterate through all edge events one by one in temporal order to determine their partition assignment. Edges are treated as undirected in the partitioning process. When either u or v is encountered for the first time, it is assigned to partition p along with the current edge. Specifically, for a new edge $e(u, v, t)$, if it connects one shared node and one non-shared node with an assigned partition p_i, the edge will be assigned to partition p_i. For other new edges and their connected emerging non-shared nodes, they will be assigned to partition p_i based on Equation 10.

3.3 Boundary-Decay Sampling

Tradeoff in neighbor sampling. In the graph partitioning process, high-weighted neighbors are locally partitioned, while low-weighted neighbors are distributed across partitions. During MT-GNN training, common *k-recent sampling* involves remote neighbor sampling, leading to inefficient communication due to remote feature retrieval. Conversely, *local sampling* avoids remote access for high-weighted neighbors but introduces bias that can affect training accuracy. To address this tradeoff, we propose the *shared node aware θ-boundary decay sampling* method, which balances communication efficiency and accuracy by approximating the effects of sampling all recent neighbors and mitigating local sampling bias through aggregating a small subset of neighbors across partitions. See Figure 4 for a case comparison of the three sampling methods.

θ-Boundary decay sampling. For a temporal node (u, t) with a non-empty temporal neighbor set, we define $\tilde{N}_u^L(t)$ and $\tilde{N}_u^R(t)$ as the local recent neighbor set and remote recent neighbor set in the sampled subgraph, respectively. In our introduced θ-Boundary decay sampling method, we sample local neighbors from the most



(a) recent sample **(b) local sample** **(c) boundary decay**

Figure 4: Illustration of different distributed neighbor sampling methods, with nodes of different colors representing different partitions, and nodes in red indicating shared nodes.

recent k neighbors and remote neighbors based on a probability $p_{u,t}(v, \tau)$. The aggregation module output $\hat{g}_u(t)$ is expressed as:

$$\begin{aligned}\hat{g}_u(t) = &\sum_{(v,\tau)\in\tilde{N}_u^L(t)} att_{uv}(t,\tau)x_{uv}(t,\tau) \\ &+ \sum_{(v,\tau)\in\text{SAMPLE}(\tilde{N}_u^R(t))} \frac{1}{p_{u,t}(v,\tau)} att_{uv}(t,\tau)x_{uv}(t,\tau),\end{aligned} \tag{16}$$

where $att_{uv}(t, \tau)$ and $x_{uv}(t, \tau)$ are the aggregate weight and input embedding for the recent neighbor (v, τ) of (u, t), respectively. To address bias, $\frac{1}{p_{u,t}(v,\tau)}$ is applied. Studies [24, 25] suggest that the aggregate weights of recent neighbors typically follow exponential distributions, irrespective of input edge features and state vectors. To minimize variance, the sampling probability is proportional to these aggregate weights, favoring recently interacted neighbors, as illustrated in Equation 8. The sampling probability for a recent remote neighbor (v, τ) is calculated as:

$$p_{u,t}(v,\tau) = \min\left[\frac{\Theta \cdot e^{\delta(\tau-t)}}{\sum_{(v_i,\tau_i)\in\tilde{N}_u^R(t)} e^{\delta(\tau_i-t)}}, 1\right], \tag{17}$$

where $\Theta = \theta|\tilde{N}_u^R(t)|$. θ is the hyperparameter for boundary sample probability, while Θ limits the expected number of remote neighbor samples to be equal to θ times the total number of remote nodes. The impact of different θ values is examined in §5.5. To ensure an adequate sample size, local neighbor sampling for the root node is repeated until a sufficient number of neighbors are sampled.

Unbiasedness and variance analysis. We compare the unbiasedness and variance of our boundary-decay sampling with the common recent sampling [7, 9]. Firstly, the expected values of node embeddings are equivalent between the two methods: $\mathbb{E}(\hat{g}_u(t)) = \mathbb{E}(g_u(t))$, where u_t represents node u at time t, and $\hat{g}$ and g are the embedding functions for boundary-decay sampling and recent sampling, respectively. Secondly, the variance of boundary-decay sampling is bounded as: $\text{Var}(\hat{g}_u(t)) \leq \frac{|\mathcal{B}|\zeta^2}{\Theta}$, where $\zeta = \max_{u\in V}\|g_u(t)\|_2$ and $\mathcal{B}$ represents the neighbor set from other partitions, which is reduced by shared nodes among partitions.

3.4 Shared Node Memory Synchronization

Sharing high-degree nodes aids in reducing variance in boundary decay sampling, minimizing cross-partition feature fetching and costly write-backs of updated memory. However, synchronizing shared memory replicas presents challenges.

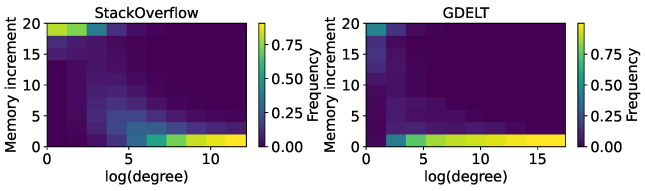


Figure 5: Average memory increment varying node degrees.

Memory stability in high-degree nodes. We observed that high-degree nodes show more stable memory increments than low-degree nodes after updates. Their numerous connections result in minor updates due to less new information. We analyzed memory changes in StackOverflow [28] and GDELT [29] datasets in TGN [7] with a batch size of 3000. Figure 5 displays the results under different node degrees, with the x-axis as log(degree) and the y-axis as the average memory increment $\|s(u,t) - s(u,t^-)\|_2^2$. Each cell indicates the frequency of memory changes within specific ranges corresponding to node degrees. The results indicate that highly connected nodes primarily exhibit memory changes in the lower increment range.

Aging inspection strategy. Based on this observation, we propose an aging inspection strategy that assesses the significance of memory changes over time and filters out data with minimal changes. We introduce an aging factor that quantifies cumulative changes in high-degree nodes. This aging factor is defined as:

$$\Delta s_{p_i}(u,t) = 1 - \text{cosine\_similarity}(s_{p_i}(u,t), \tilde{s}(u,t^-)), \tag{18}$$

where $\tilde{s}(u,t^-)$ denotes the shared memory synchronized by all trainers before updating node u at timestamp t, and $s_{p_i}(u,t)$ denotes the generated memory after the memory updater module on partition p_i. Memory is sent to other partitions for synchronization only if $\Delta s_{p_i}(u,t)$ exceeds a significant change detection threshold. The shared node memory is updated as:

$$\tilde{s}(u,t) = \text{Aggregate}\{s_{p_i}(u,t) \cdot \mathbb{I}(\Delta s_{p_i}(u,t) > \alpha), i \in [1,m]\}. \tag{19}$$

where α is a hyperparameter indicating the change detection threshold. The impact of varying α values is investigated in §5.5.

Smooth Aggregation. Although significant change detection lowers synchronization overhead, it results in prolonged update intervals for certain nodes, leading to the loss of recent data and outdated memory states [18, 21]. To tackle this issue, we introduce a smooth aggregation method that not only utilizes stable characteristics of highly connected nodes from global graph to predict future memory states but also retains the most recent locally captured memory.

Assuming $\hat{s}(u,t)$ represents the expected memory of node u at timestamp t, it can be estimated as:

$$\hat{s}(u,t) = \tilde{s}(u,t^-) + \Delta S(u, t - t^-), \tag{20}$$

where $\Delta S(u, t-t^-)$ represents the bias introduced by aging shared memory. Previous methods[21] employed a Gaussian Mixture Model to estimate this bias distribution. To improve upon this, we estimate $\Delta S(u, t-t^-)$ using moving average increments to reduce discontinuities. The memory of node u on trainer p_i is then updated as:

$$s_{p_i}(u,t) = \gamma \text{UPDATE}(s_{p_i}(u,t^-), m^{uv}_{e,p_i}) + (1-\gamma)\hat{s}(u,t), \tag{21}$$

where γ is a learnable parameter. The output of $s_{p_i}(u,t)$ serves as the input to the message passing layer, as depicted in Figure 6. This

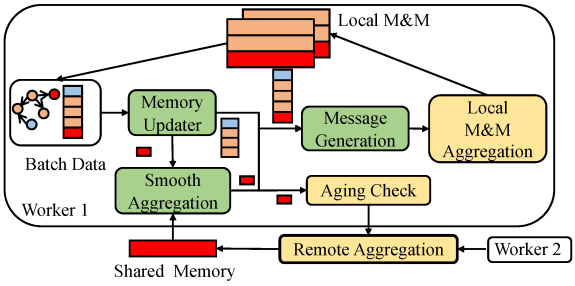


Figure 6: Smoothing aggregation of shared node memory.

method offers several advantages: it retains local recent characteristics, leverages high-weight information in partitioning, mitigates noise and abrupt changes in memory trends by smoothing $\Delta S(u, t-t^-)$, and addresses the staleness issue in large batch training, thereby enhancing prediction performance.

4 MEMSHARE FRAMEWORK

4.1 Pipeline-based Adaptive Parameter Tuning

Training pipeline parallelism. During the training process in MemShare, efficiency is enhanced through pipeline parallelism, where CPU sampling, GPU model computation, and communication phases overlap. Specifically, remote fetching commences after memory updater layer computation to prevent delayed memory updates. Synchronization overhead is reduced by aligning the communication stage with the subsequent batch's memory updater layer computation. The sampling stage runs continuously in the background, with the backward phase leveraging PyTorch's Distributed Data Parallel (DDP) operator for computation [35].

Adaptive tuning for boundary decay sample probability θ: The parameter θ is closely related to the communication volume during the remote fetch phase. To ensure pipeline efficiency, we balance the average time spent on the embedding computation (T_{emb}) and remote fetch communication (T_{fetch}), by updating θ iteratively after each batch during the early training epochs:

$$\theta^{(i)} = \theta^{(i-1)} \cdot \frac{T_{\text{emb}}}{T_{\text{fetch}}}, \quad \theta^{(i)} \in [0.01, 0.8] \tag{22}$$

where i is the iteration number. This equation balances the time spent on both phases, keeping θ within the range [0.01, 0.8], where it minimally affects final accuracy (as discussed in §5.5).

Adaptive tuning for significant change detection threshold α: The synchronization volume between α exhibits an approximate exponential relationship (see §5.5). Similar to θ, α is updated iteratively to balance the average time costs for the message update (T_{update}) and memory synchronization (T_{sync}):

$$\alpha^{(i)} = \alpha^{(i-1)} - \log(T_{\text{update}}) + \log(T_{\text{sync}}), \alpha^{(i)} \in [0,1] \tag{23}$$

The threshold α is maintained in the range $[0,1]$.

4.2 Theoretical Analysis for Memory Share

Communication analysis. To analyze the communication cost in distributed MTGNN training with our shared node memory paradigm, we introduce λ^k_π as the ratio of cut edges after partitioning

the graph using partition policy π with top-k nodes sharing. This ratio can be approximated as $\lambda_\pi^k \sim \sum_{e(u,v,t)\in E} \mathbb{I}[\mathcal{P}(u) \neq \mathcal{P}(v)]/|E|$, where $\mathcal{P}(u)$ is the partition of node u. Similarly, we define $\hat{\lambda}_\pi^k$ as the ratio of remote temporal neighbors for hot nodes, estimated as:

$$\hat{\lambda}_\pi^k \sim \frac{\sum_{e(u,\cdot,t)} \sum_{(v,\tau)\in\tilde{N}_u(t)} \mathbb{I}[\mathcal{P}(e(u,\cdot,t)) \neq \mathcal{P}(e(u,v,\tau))]}{\sum_{e(u,\cdot,t)} |\tilde{N}_u(t)|}, \quad (24)$$

where u is hot node and $\tilde{N}_u(t)$ is the set of recently sampled neighbors. $\mathcal{P}(e)$ is the partition of edge e. This equation calculates the likelihood of sampled neighbors for hot nodes in external partitions. We observe that $\hat{\lambda}_\pi^k$ is approximately equal to λ_π^0 for any k due to the uniform neighbor weight policy during graph partition process.

The communication overhead in distributed temporal graph learning mainly arises from the Remote Fetch and Memory Synchronization phases. The Remote Fetch communication volume includes transfers of node memory and node/edge features, while the Memory Synchronization communication volume involves broadcasting the updated memory of hot shared nodes to $m-1$ trainers. Let's denote C_k as the expected total communication volume with the top-k hot nodes, which is estimated as:

$$C_k \approx \underbrace{d_n \cdot n\theta\hat{\lambda}_\pi^k E_k(1-\frac{E_k}{2|\boldsymbol{E}|})}_{\text{Hots' } d_n \text{ Fetch vol.}} + \underbrace{d_n \cdot n\theta\lambda_\pi^k(2|\boldsymbol{E}|-E_k) + d_n\lambda_\pi^k|\boldsymbol{E}|}_{\text{Colds' } d_n \text{ Fetch vol.}}$$
$$+ \underbrace{d_e \cdot n\theta\hat{\lambda}_\pi^k E_k}_{\text{Hots' } d_e \text{ fetch vol.}} + \underbrace{d_e \cdot n\theta\lambda_\pi^k(2|\boldsymbol{E}|-E_k)}_{\text{Colds' } d_e \text{ Fetch vol.}} + \underbrace{d_{mem} \cdot (m-1)\eta E_k}_{\text{Mem Sync vol.}}. \quad (25)$$

where n and θ denote the number of sampled neighbors for each root node and the sampling probability of recent remote neighbors, as discussed in §3.3. d_n, d_e, and d_{mem} represent the dimensionality of node data, edge feature, and node memory, respectively. The parameter η denotes the actual communication ratio after aging inspection. E_k denotes the total number of event connected to the topk-k shared nodes. The actual communication volume is significantly smaller than $d_{mem} \cdot (m-1)E_k$ because only the latest memory per node is broadcast, with aging inspection filtering out substantial data, as detailed in §5.5. The volume of remote neighbor access for hot and cold nodes is estimated based on the ratio of remote neighbors to cut edges as a general probability.

Analysis of variance: The variance of the output embedding $\tilde{g}_u(t)$, primarily influenced by the message passing stage:

$$\mathrm{Var}[\tilde{g}_u(t)] \leq (1-\lambda_\pi^{k,u})\delta^2 + \frac{\lambda_\pi^{k,u}(\mu^2+\delta^2)}{\theta}. \quad (26)$$

Here, μ and δ^2 denote the expectation and variance of $x(\cdot)$, respectively. $\lambda_\pi^{k,u}$ equals to $\hat{\lambda}_\pi^k$ if u is hot node; otherwise, it is λ_π^k.

Balancing tradeoffs in sharing top-k hot nodes: In our shared node memory paradigm, a tradeoff exists in determining the proportion of hot nodes (top-k). A higher value of k can reduce communication volume and enhance accuracy in distributed MTGNN training and inference processes but also introduces additional storage costs for shared nodes. We aim to identify the optimal hyperparameter settings for top-k by analyzing its impact on communication volume and model accuracy. Regarding communication volume, we simulate the estimation of communication overhead on GDELT with 8 partitions and display the stacked communication volume in Figure 7. As top-k increases, the overhead for node data fetching decreases significantly due to the nonlinear decreasing trend of the cut edge ratio. However, the communication volume for edge feature fetching shows gradual changes as k reduces cut edges but increases external partition access for hot nodes. Additionally, the memory synchronization volume increases gradually with almost no extra overhead. Regarding model accuracy, shared memory for hot nodes reduces memory change discontinuity, enhancing model accuracy. To minimize variance, top-k selection can be achieved at $\xi^{\frac{1-\beta}{2-\beta}}$, where β is the exponent of the power-law distribution (typically $\beta \in [2,3]$) and ξ is approximately 0.46. Sharing top-k neighbors incurs a storage cost of $O((k+1/m)|\boldsymbol{N}|)$. Based on Figure 12 and the aforementioned analysis, we recommend keeping k no more than 0.1. Sharing some nodes can still be advantageous by reducing cut edges on graphs without hot nodes, except for low average degree graphs. Our work supports the case when top-k is 0, providing an acceleration scheme. The supplementary material will provide proofs of the above analysis due to space limitations.

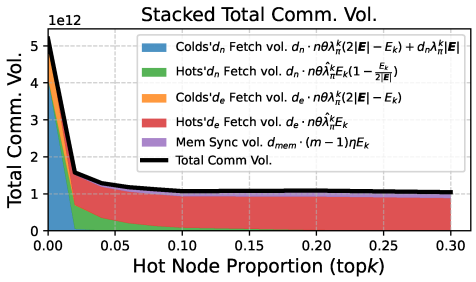


Figure 7: The stacked communication volume with different hot node proportions (top-k) shows how each part contributes to the total communication increase, as indicated by the subscripts in Equation 25. As top-k rises, the cost of fetching node data from other partitions decreases significantly due to a lower cut edge ratio, which dominates total communication when top-k is small. However, larger top-k values increase the communication overhead for fetching edge features of shared nodes. Therefore, it is recommended to set k at 0.1 or below for optimal benefits, as higher values offer limited advantages.

4.3 Optimizations and Implementation

Non-redundant message flow graph (MFG). Traditional MFG structures in TGL [11] often exhibit significant redundancy [13, 36]. Neighbor nodes sampled from different root nodes receive unique IDs, even with identical node IDs and timestamps. This redundancy causes unnecessary calculations in the memory updater model. To resolve this, we have removed duplicate node IDs and timestamps while maintaining their original sampling order, enabling the expansion of multilayer structures.

Localism negative sampler. In the boundary decay sampling method (see §3.3), we adjust the number of negative samples for

local and remote sampling using the same probability. To address potential sampling bias between local and global negatives, we introduce weight compensation for negative samples from different partitions. The loss function incorporates weighted binary cross-entropy (BCE) loss:

$$\text{BCELoss} = -\frac{1}{N}\sum_{i=1}^{N} w_i \cdot (y_i \cdot \log(\hat{y}_i) + (1-y_i) \cdot \log(1-\hat{y}_i)). \quad (27)$$

Let p_r be the sampling probability for remote negative destinations related to θ in Equation (17). For trainer i, N_d and N_d^L denote the total number of neighbors in whole graph and partition accessed by trainer i, respectively. The weight w_i is calculated as $\frac{N_d}{(1-p_r)\cdot N_d + p_r \cdot N_d^L}$, if its neighbors generated from the negative sampling method are stored locally, and $w_i = \frac{N_d}{N_d^L \cdot p_r}$ otherwise.

5 EXPERIMENTS

5.1 Experiment Setups

Testbed. Our experiments are conducted on four machines interconnected using 10G NICs, each equipped with two Xeon 6342R@ 2.8GHz processors, 1T DRAM, and four NVIDIA A40 (48G) GPUs. The CPU-to-GPU and GPU-to-GPU connectivity was configured using PCIe 4.0×16. To simplify notation, we use xMxG to denote x machines with x GPUs for training, e.g., 2M8G signifies the utilization of 2 machines with a total of 8 GPUs for training.

Datasets. We utilize four commonly used large temporal graph datasets, Each graph is chronologically divided into a training set (70%), a validation set (15%), and a test set (15%), as used in TGL [11] and ETC [20].

Test MTGNNs. We evaluate three representative MTGNN models: TGN [7], JODIE [8], and APAN [9]. Their implementations are customized versions of TGL [11]. The attention aggregator uses two heads for message passing, with node memory and hidden dimensions set to 100. APAN has a mailbox size of 10 mails, while other methods are set to 1 mail.

Comparison baselines. We compare MemShare with three state-of-the-art open-source MTGNN training frameworks: TGL, DistTGL, and MSPipe. Previous research has primarily focused on SM-MG scenarios, lacking support for MM-MG extensions. To address this, we developed TGL-dist, a multi-machine extension that optimizes communication through graph partitioning. We employed a dynamic balanced graph partitioning algorithm to mitigate load imbalance (see §3.2), with effects of different partition algorithms discussed in §5.4. Parameters were configured according to the original works. Since DistTGL only provides an open-source version of TGN, we implemented the JODIE and APAN models based on its framework.

Table 2: Dataset statistics, where d_v and d_e shows the dimensions of node and edge features.

Dataset	$\|V\|$	$\|E\|$	d_v	d_e
LASTFM [8]	1,980	1,293,103	172	172
WikiTalk [19]	1,140,149	7,833,139	172	172
StackOF [28]	2,601,977	63,497,049	172	172
GDELT [29]	16,681	191,290,882	413	186

Training setting. The training settings for each dataset are tailored to improve convergence and scalability. For the LASTFM dataset, we used 10 recent neighbors per node and an average batch size of about 1000 per device for 100 epochs. For the larger datasets, we increased the recent neighbors to 20 per node and set a batch size of around 3000 per device for 50 epochs, except for GDELT, which was trained for 10 epochs. During training and evaluation, mini-batches were created with an equal number of positive and negative node pairs. For MSPipe, the staleness mitigation ratio λ is set to 0.9, with a maximum delay of 10 epochs. The learning rates for DistTGL and MSPipe are proportional to the number of GPUs (#GPU) and $\sqrt{\#\text{GPU}}$, respectively as detailed in their source code. We conducted a grid search for the optimal learning rate from the set {0.0001, 0.0002, 0.0004}. Additionally, for MemShare, we set the top-k value to 0.1.

5.2 Model Accuracy Comparison

Convergence accuracy. We initially conduct a comprehensive analysis of model accuracy comparing MemShare with baseline frameworks across diverse models and datasets, with varying numbers of machines. We train the three TGNN models under the transductive setting, and use test average precision(AP) with best validation AP for accuracy metrics.TGL only supports SM-MG scenarios, so results for other setups are missing. Table 3 illustrates the consistent accuracy superiority of our MemShare across all test scenarios with identical device configurations. For instance, MemShare with the TGN model achieves a test AP of 94.18% on the LASTFM dataset using 4M16G, surpassing the nearest competitor, TGL-dist, by 12.97%. On average, MemShare enhances model training accuracy by 2.60%, 3.42%, 4.96%, and 4.29% over TGL, TGL-dist, DistTGL, and MSPipe, respectively. MemShare also demonstrates competitive or superior accuracy compared to the baseline results achieved by TGL on a single GPU. The accuracy improvements of MemShare result from several key factors: our hot nodes sharing reduces variance and bias from boundary decay sampling, which adds randomness for neighbor sampling and enhances model generalization. Additionally, the smooth aggregation method minimizes instability noise. The effects of each module on accuracy will be discussed in §5.4.

Convergence efficiency. We further compare the convergence efficiency by plotting the test average precision curves for TGL-dist, DistTGL, MSPipe, and MemShare on TGN model in Figure 8. The X-axis represents the total time cost for training, and the y-axis represents the corresponding test precision curves. Noticeably, MemShare achieves a significantly faster convergence rate and higher accuracy compared to baseline frameworks under the same time cost of training. Furthermore, MemShare exhibits a more stable convergence curve during training, which can be attributed to synchronous smoothing aggregation.

5.3 Efficiency Comparison

We perform a assessment of training and inference efficiency by comparing our MemShare with TGL-dist, DistTGL, and MSPipe across three models and four datasets. We vary the device configurations with 1M4G, 2M8G, and 4M4G setups.

Table 3: Test Average Precision (AP) across datasets and MTGNNs.

#.GPUs	Methods	TGN				JODIE				APAN			
		LASTFM	WikiTalk	StackOF	GDELT	LASTFM	WikiTalk	StackOF	GDELT	LASTFM	WikiTalk	StackOF	GDELT
1M1G	TGL[11]	80.23%	96.11%	95.74%	98.37%	62.02%	95.78%	96.20%	98.31%	56.59%	90.43%	87.52%	96.12%
1M4G	TGL[11]	78.20%	96.32%	95.47%	97.70%	61.16%	94.63%	94.94%	98.25%	57.11%	91.36%	88.86%	95.47%
	TGL-dist	80.69%	96.58%	95.85%	97.84%	58.84%	94.86%	95.10%	98.53%	54.49%	92.31%	86.97%	97.66%
	DistTGL[17]	76.83%	94.40%	93.38%	98.60%	56.28%	92.71%	93.71%	98.25 %	53.59%	90.06%	89.80%	97.29%
	MSPipe[18]	82.77%	95.52%	95.48%	98.32%	49.98%	91.23%	88.56%	98.31%	59.75%	91.47%	86.15%	93.61%
	Ours	**91.52%**	**97.88%**	**97.79%**	**98.92%**	**61.47%**	**95.28%**	**95.18%**	**98.62%**	**59.39%**	**94.26%**	**92.43%**	**97.90%**
2M8G	TGL-dist	80.73%	96.50%	95.65%	98.20%	56.48%	94.90%	94.29%	98.34%	54.06%	92.49%	88.34%	97.10%
	DistTGL[17]	74.19%	93.01%	93.28%	98.60%	55.70%	93.09%	92.32%	98.04%	54.44%	90.13%	88.29%	97.03%
	MSPipe[18]	77.39%	95.77%	95.95%	98.77%	57.03%	88.22%	94.01%	98.25%	60.79%	91.36%	89.89%	95.12%
	Ours	**92.99%**	**97.83%**	**97.84%**	**98.93%**	**63.78%**	**94.92%**	**94.37%**	**98.93%**	**62.24%**	**93.06%**	**94.48%**	**98.71%**
4M16G	TGL-dist	81.21%	96.45%	94.98%	98.15%	56.23%	94.11%	92.92%	98.12%	53.73%	91.76%	86.75%	97.66%
	DistTGL[17]	78.47%	91.59%	90.31%	98.62%	56.47%	91.56%	92.43%	97.86%	53.84%	89.87%	83.17%	96.09%
	MSPipe[18]	78.41%	95.86%	95.15%	98.64%	56.77%	90.66%	92.67%	98.30%	57.80%	89.54%	87.23%	92.65%
	Ours	**94.18%**	**97.96%**	**98.09%**	**99.17%**	**63.78%**	**94.92%**	**94.37%**	**98.93%**	**64.94%**	**93.53%**	**94.32%**	**99.00%**

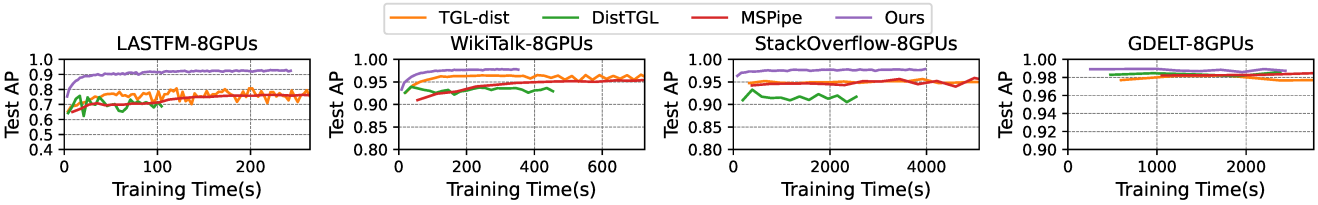


Figure 8: Convergence of TGN.

Training Efficiency. The training time cost and the corresponding test AP values are presented in Figure 9. Compared to TGL, TGL-dist, MSPipe and DistTGL, MemShare achieves an average speedup of 2.02× 2.34× 6.43× and 1.30× on average, respectively. Our method surpasses subgraph-parallel approaches in MM-MG while training time per epoch matches DistTGL,which avoids inter-machine dependencies and requires only gradient synchronization. However, DistTGL's epoch-level parallel design limits throughput during multi-machine inference. Our inference speed is 4.98× faster than DistTGL (see Figure 10). While DistTGL benefits from offline pre-sampling, it restricts training accuracy and convergence speed. With the same training time, our accuracy consistently exceeds that of DistTGL as Figure 8 shows. In SM-MG scenarios, we achieve comparable performance with other methods, even with the incorporation of a smooth aggregation module.

Training scalability. We assess the scalability of our MemShare with varying numbers of machines, comparing training throughput (time cost per epoch). As shown in Figure 9, when expanding from one to two machines, MemShare achieves speedups of 1.83× 1.67× 1.54× and 2.09× on LASTFM, WikiTalk, StackOF, and GDELT, respectively. With four machines, speedups increase to 3.13× 3.29× 2.06× and 3.60× for these datasets on the TGN model. Our advantages become increasingly evident as the number of machines escalates. TGL-dist and MSPipe often show no reduction in training times, and may even experience increased times due to significant communication overhead in distributed environments. We improve our approximate linear scalability by mitigating inter-machine communication through boundary decay sampling and smooth aggregation synchronization. Although DistTGL also achieves approximate linear scalability, it does not facilitate scaling for inference.

Inference Efficiency Comparison. To verify our inference efficiency, we measure the average time taken to perform a single inference pass of the TGN model on the test dataset, as Figure 10 presents. We conduct experiments with TGL-dist, DistTGL and MSPipe on various machines, each equipped with 4 GPUs. Our inference performance is on average 2.53× 3.42 × and 5.03× faster than DistTGL across different numbers of machines. And MemShare achieves 4.55×, 2.33×, and 3.43× on average speed up over TGL, TGL-dist, and MSPipe among various settings, respectively. DistTGL, while scalable during training, is limited to single-GPU inference. Our approach extends this to the MM-MG setup, improving performance despite significant communication overhead from global negative sampling for evaluation. Both TGL-dist and our method perform well with 4 GPUs due to the absence of feature pre-extraction. DistTGL and MSPipe use RAM for data access, whereas our method leverages VRAM, benefiting from higher GPU-to-GPU bandwidth.

5.4 Ablation study

Breakdown To evaluate the performance and efficiency of the key components of MemShare, we employ the naive TGL-dist as the reference baseline and progressively integrate the following

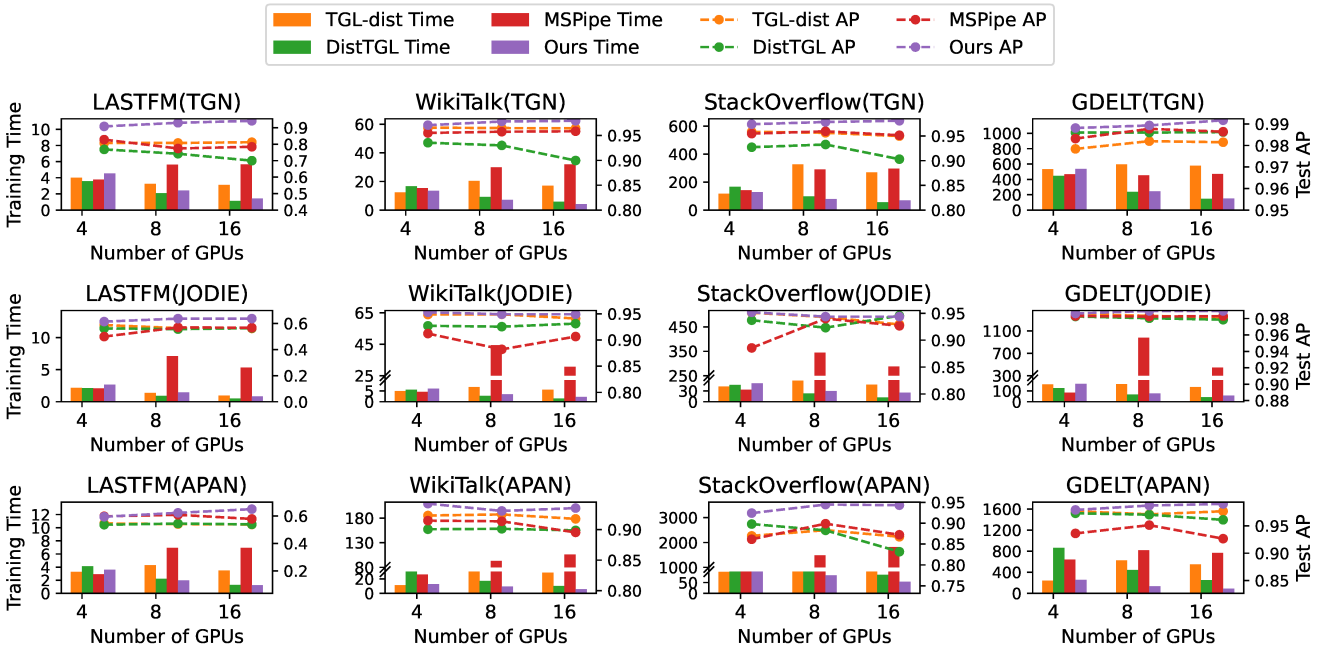


Figure 9: Test AP and training time (seconds) with different numbers of GPUs.

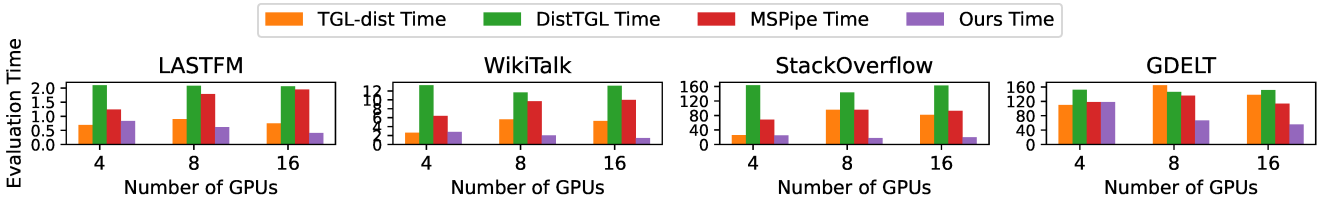


Figure 10: Inference time (seconds) with different number of GPUs on the TGN model.

components: shared hot nodes (shared), historical memory synchronization (his-mem), boundary decay sampling (b-decay), localism negative sampler (local-neg), and pipeline. This methodology enables the assessment of performance enhancements across each module, as illustrated in Figure 11. The experiments are conducted on two machines, equipped with eight GPUs. The x-axis represents the components stacked to MemShare, while the left y-axis indicates the training time per epoch and inference time. The right y-axis shows the test dataset's average precision after each integrated component. After integrating key components into the

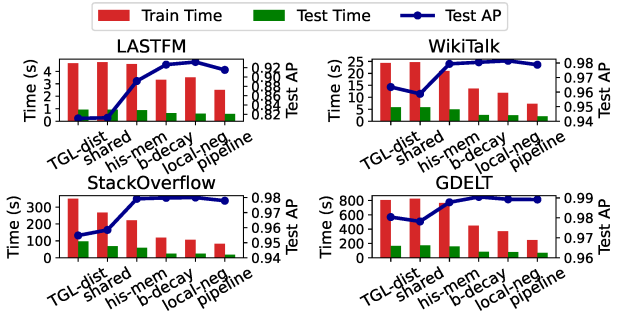


Figure 11: Breakdown of MemShare's key components.

baseline model, accuracy improvements were 3.24% for his-mem, 0.99% for b-decay, and 0.14% for local-neg, although the pipeline implementation slightly reduced test AP. Training speed increased by an average of 1.06× 1.12× 1.62× 1.11× and 1.45× across various settings, and inference throughput improved by 1.09× 1.12× and 1.91× for sharing nodes, his-mem, and b-decay, respectively. The localism negative sampler and pipeline mechanism do not affect the inference phase. The historical memory synchronization module achieved the most significant improvement, particularly on the LASTFM dataset, with 7.9% gain. The boundary decay sampling module provided the highest efficiency boost and also contributed to a modest increase in accuracy. Sharing nodes affects training and inference efficiency by changing partitioning without altering the training method or data distribution. While sharing hot nodes can lower fetch costs, it may not always improve speed due to higher memory synchronization costs. In contrast, sharing high-degree nodes reduces sampling variance in boundary decay sampling and enhances memory synchronization consistency.

Analysis of Graph Partitioning To evaluate MemShare's effectiveness, we compare it with four baseline graph partition methods: METIS [27], DisTGL's [16], METIS with shared nodes, and our method without shared nodes. METIS is implemented using the NetworkX [37] library. We implement DisTGL's method based on

Table 4: Training time (in seconds) per epoch of different graph partition methods on TGN.

Dataset	w/o shared nodes			w/ shared nodes	
	METIS	DisTGL	Ours	METIS	Ours
LASTFM	2.51	2.45	2.41	2.40	**2.37**
WikiTalk	19.95	17.72	8.57	8.74	**7.78**
StackOF	100.36	159.68	85.93	97.99	**74.98**
GDELT	1032.23	292.21	276.65	276.75	**267.73**

Table 5: Test average precision (AP) of different graph partition methods on TGN.

Dataset	w/o shared nodes			w/ shared nodes	
	METIS	DisTGL	Ours	METIS	Ours
LASTFM	79.46%	78.66%	78.20%	90.98	**93.96%**
WikiTalk	94.40%	95.23%	96.17%	96.52%	**97.04%**
StackOF	94.76%	93.89%	94.29%	97.23	**97.90%**
GDELT	98.46%	99.07%	98.94%	98.68%	**99.10%**

TGL, setting its load balance parameter γ to 1.5, as specified in their paper. The shared nodes consist of the top 10% high-degree nodes for both METIS with shared nodes and MemShare. For METIS with shared nodes, we partition the connected subgraph of the remaining nodes, randomly dividing edges between shared nodes across partitions and connecting remaining nodes to all shared nodes in each partition. To ensure a fair comparison, we maintain the same number of training samples, setting the boundary decay sampling parameter θ to 0.1 for all methods. The smooth aggregation change detection threshold α is set to 0.3 for methods with shared nodes to minimize variations during adaptive parameter tuning. Experiments are conducted on a TGN model using a 2M8G cluster.

Table 4 shows the training time per epoch for various TGN partition methods across four datasets. MemShare achieves the lowest training costs among these methods, with speedups of 2.06× and 1.51× over METIS and DisTGL, respectively, even without shared nodes, due to dynamic workload imbalances in those methods (see Figure 2). Additionally, MemShare offers a slight speedup of 1.01× to 1.15× compared to its version without shared nodes, thanks to reduced data-fetching needs. Finally, MemShare matches METIS's training speeds with shared nodes, indicating that our graph partitioning maintains training efficiency.

Table 5 further illustrates the effectiveness of MemShare compared to other graph partition baselines. MemShare achieves the highest AP among all partition methods, significantly outperforming methods without shared nodes by 15.1%, 1.8%, 3.6%, and 0.3% on the LASTFM, WikiTalk, StackOF, and GDELT datasets, respectively. Additionally, MemShare shows consistent improvements, averaging a 1.15% increase over METIS with shared nodes. This enhancement is due to the high-degree node sharing and neighbor weight-aware mechanisms, which effectively preserve more important neighbors within local partitions.

5.5 Analysis of Hyper-parameters

Analysis of top k Shared Nodes. To assess the impact of the top-k hot nodes shared memory, we analyzed different k ranging from 0 to 0.3 with the recent sample policy and boundary decay sample policy under historical smooth aggregation, because larger top-k values increase memory usage. The experiments are conducted on 2M8G clusters. Figure 12 shows the accuracy and training time across different settings. Firstly, training time generally decreases with an increasing number of shared nodes using recent sampling. Specifically, when k is set to 0.1, training speed improves by 1.2× and 1.6× for the WikiTalk and StackOF datasets, respectively. Secondly, training time initially rises and then stabilizes with boundary decay sampling. This rise is due to a high number of cut edges from limited hotspot data, resulting in insufficient samples at the same sampling probability. Additionally, boundary decay sampling is significantly lower in time costs compared to recent sampling. Moreover, increasing the proportion of shared hotspots enhances accuracy initially, but accuracy gradually declines as k increases. This is consistent with our theoretical analysis. A moderate amount of shared data reduces external neighbors, lowering edge sampling variance and improving historical aggregation. However, excessive sharing may cause minimal information loss between hot node.

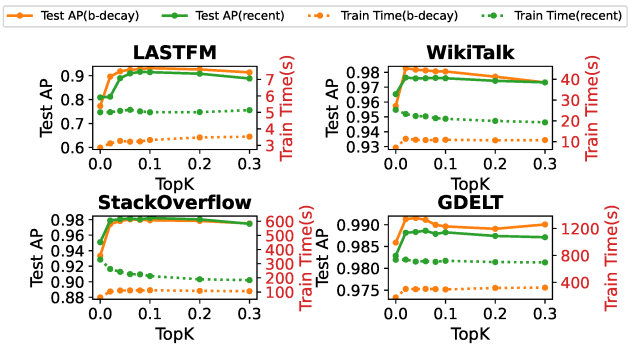


Figure 12: Comparison of TGN accuracy and training time (in seconds) per epoch across different shared nodes ratios with recent sampling and boundary decay sampling.

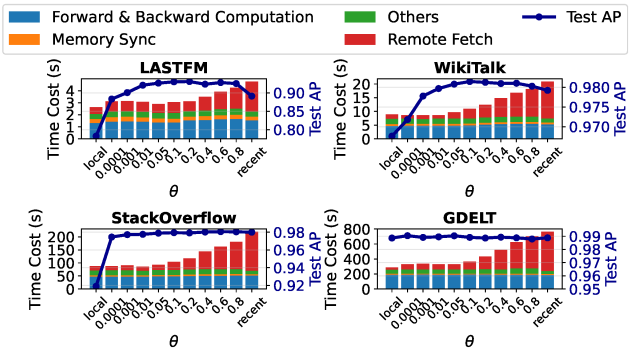


Figure 13: Comparison of time costs (in seconds) per epoch for each stage and test AP using recent sampling and boundary decay sampling with various hyperparameters θ without pipelining on TGN.

Analysis of Boundary Decay Sampling. To assess the impact of our boundary decay sampling method, we conducted a comparative analysis of our boundary decay sampling with various θ

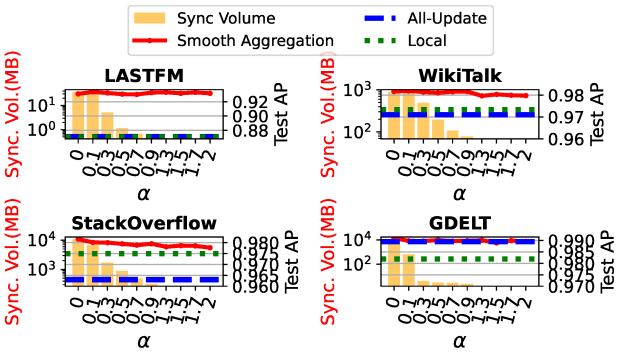


Figure 14: Comparison of TGN accuracy and synchronization volume across different synchronization aggregation methods, with varying significant change detection threshold α

which range from 0.01 to 0.8 against the traditional recent sampling methods and only local sampling method, i.e. θ=0. The experiments are conducted on TGN with a 2M8G cluster. Figure 13 illustrates time costs for each stage per epoch and the impact on accuracy for the recent sampling method and our boundary decay sampling method with various θ. Boundary decay sampling significantly reduces the time costs of the memory fetch stage, which accounts for the majority of the overhead. When $\theta = 0.1$, the remote fetch time costs are reduced by an average of 74%. As θ decreases, the impact on time consumption gradually diminishes until reaching a limit when θ=0.01, and the remote fetch time costs is reduced by up to 84% on average. Additionally, Boundary decay sampling not only demonstrates excellent time efficiency but also maintains accuracy, often outperforming the recent sampling method when $\theta > 0.01$. In most cases, boundary decay sampling can achieve a slight improvement compared to recent sampling, while the highest test accuracy achieves 3.9% improvement on the LASTFM dataset.

Analysis of Smooth Aggregation To evaluate our historical memory synchronization method, we compared it against traditional all-update synchronization, where each GPU collects all shared memory from others, and a no-synchronization approach (local). We tested significant change detection thresholds α in the range [0,2] on a TGN with a 2M8G cluster. Figure 14 illustrates the significant reduction in synchronization volume achieved by the aging inspection strategy compared to the vanilla all-update strategy. The hyperparameter α controls the update frequency of the shared node memory. Larger α results in less frequent synchronization. Notably, the communication volume significantly reduces since $\alpha = 0.3$, that leads to 92% and 99% reduction in communication volume compared to the all-update strategy on StackOF and GDELT, respectively. Furthermore, a slight increase in α exhibits an inverse exponential relationship with communication volume, reducing communication time costs by up to 60% compared to no aging inspection. Figure 14 also presents the test AP of smooth aggregation synchronization with various values of α, compared to the vanilla all-update strategy and the local strategy, consistently outperforming both baselines. Additionally, the varying values of α yield statistically insignificant results across all datasets, suggesting that smooth aggregation synchronization is robust to hyperparameter selection.

6 RELATED WORKS

Distributed Static GNN Training Systems. Many efforts focus on accelerating distributed training for static GNNs through graph partitioning algorithms that minimize cut edges and maintain workload balance [38, 39], caching strategies [39–43], and pre-fetch methods [40, 44] to enable simultaneous computation and communication. However, these approaches are not suitable for MTGNN training as they overlook node memory dependencies across temporal batches.

Distributed Temporal GNN Training Systems. Dynamic node memory synchronization is a major challenge for MTGNN training. Initial approaches focused on single-machine, multi-GPU setups using shared memory to reduce communication overhead, with TGL [11] and SPEED [12] introducing batch parallel and subgraph partitioning strategies, respectively. However, shared memory does not resolve communication overhead in multi-machine scenarios. Recent strategies include temporal-aware graph partitioning [16], node memory caching [16] with staleness mechanisms [18]. GNNFlow [15] sorely employs hash partitioning, while DisTGL [16] uses an edge streaming partition method based on average timestamp and neighbor counts. Sven [13] implements incremental repartitioning. To reduce synchronization overhead, DisTGL adjusts synchronization frequency with a cache miss threshold, and DistTGL [17] creates memory replicas. MSPipe [18] and Sven [13] parallelize communication by fetching memory in background threads.

Single-Machine Temporal GNN Training Systems. To bridge the gap between distributed and single-machine temporal GNN training, advancements have focused on dynamic node memory challenges. Some works aim to reduce information loss during training with large batch sizes by limiting duplicate nodes [20], using sliding windows [45], and applying memory-smoothing learning [21]. Other approaches enhance efficiency through caching [46] and data reuse [47] to minimize redundant computations.

7 CONCLUSION

This paper introduces MemShare, a distributed memory-based temporal graph neural network training system based on shared node memory. By incorporating shared node-centric graph partitioning, boundary decay sampling, and shared node-targeted synchronous smoothing aggregation, MemShare effectively addresses the challenges of high communication overhead in distributed training scenarios. Additionally, MemShare integrates localism negative sampling, pipelining parallelism, and data deduplication to enhance its overall effectiveness. Our results demonstrate that MemShare achieves the highest accuracy within the same machine cluster configuration, with overall improvements in training and inference efficiency compared to state-of-the-art distributed methods.

ACKNOWLEDGMENTS

This research is supported by the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (Grant No. SN-ZJU-SIAS-001), Hangzhou Joint Fund of the Zhejiang Provincial Natural Science Foundation of China (Grant No.LHZSD24F020001), the “Pioneer" R&D Program of Zhejiang (Grant No. 2024C01019), and the National Natural Science Foundation of China (Grant No. 62476238). The author gratefully acknowledges the support of Zhejiang University Education Foundation Qizhen Scholar Foundation.

REFERENCES

[1] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The world wide web conference*, pages 417–426, 2019.

[2] Shengjie Min, Zhan Gao, Jing Peng, Liang Wang, Ke Qin, and Bo Fang. Stgsna spatial–temporal graph neural network framework for time-evolving social networks. *Knowledge-Based Systems*, 214:106746, 2021.

[3] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *Expert systems with applications*, 207:117921, 2022.

[4] Saeed Rahmani, Asiye Baghbani, Nizar Bouguila, and Zachary Patterson. Graph neural networks for intelligent transportation systems: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 24(8):8846–8885, 2023.

[5] Soroor Motie and Bijan Raahemi. Financial fraud detection using graph neural networks: A systematic review. *Expert Systems with Applications*, 240:122156, 2024.

[6] Jianian Wang, Sheng Zhang, Yanghua Xiao, and Rui Song. A review on graph neural network methods in financial applications. *arXiv preprint arXiv:2111.15367*, 2021.

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

[8] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *ACM SIGKDD*, pages 1269–1278, 2019.

[9] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 international conference on management of data*, pages 2628–2638, 2021.

[10] Zhen Zhang, Jiajun Bu, Martin Ester, Jianfeng Zhang, Chengwei Yao, Zhao Li, and Can Wang. Learning temporal interaction graph embedding via coupled memory networks. In *Proceedings of the web conference 2020*, pages 3049–3055, 2020.

[11] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. Tgl: A general framework for temporal gnn training on billion-scale graphs. *Proceedings of the VLDB Endowment*, 15(8):1572–1580, 2022.

[12] Xi Chen, Yongxiang Liao, Yun Xiong, Yao Zhang, Siwei Zhang, Jiawei Zhang, and Yiheng Sun. Speed: Streaming partition and parallel acceleration for temporal interaction graph embedding. *arXiv preprint arXiv:2308.14129*, 2023.

[13] Yaqi Xia, Zheng Zhang, Donglin Yang, Chuang Hu, and et.al. Zhou. Redundancy-free and load-balanced tgnn training with hierarchical pipeline parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[14] Yaqi Xia, Zheng Zhang, Hulin Wang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Redundancy-free high-performance dynamic gnn training with hierarchical pipeline parallelism. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 17–30, 2023.

[15] Yuchen Zhong, Guangming Sheng, Tianzuo Qin, Minjie Wang, Quan Gan, and Chuan Wu. Gnnflow: A distributed framework for continuous temporal gnn learning on dynamic graphs. *arXiv preprint arXiv:2311.17410*, 2023.

[16] Ziquan Fang, Qichen Sun, Qilong Wang, Lu Chen, and Yunjun Gao. Distributed temporal graph neural network learning over large-scale dynamic graphs. In *Proceedings of the International Conference on Database Systems for Advanced Applications*, 2024.

[17] Hongkuan Zhou, Da Zheng, Xiang Song, George Karypis, and Viktor Prasanna. Disttgl: Distributed memory-based temporal graph neural network training. In *SC*, pages 1–12, 2023.

[18] Guangming Sheng, Junwei Su, Chao Huang, and Chuan Wu. Mspipe: Efficient temporal gnn training via staleness-aware pipeline. In *ACM SIGKDD*, pages 2651–2662, 2024.

[19] Wiki-talk. http://snap.stanford.edu/data/wiki-talk-temporal.html.

[20] Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. Etc: Efficient training of temporal graph neural networks over large-scale dynamic graphs. *Proceedings of the VLDB Endowment*, 17(5):1060–1072, 2024.

[21] Junwei Su, Difan Zou, and Chuan Wu. Pres: Toward scalable memory-based dynamic graph neural networks. In *The Twelfth International Conference on Learning Representations*.

[22] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. Foundations and modeling of dynamic networks using dynamic graph neural networks: A survey. *iEEE Access*, 9:79143–79168, 2021.

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

[24] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. Inductive representation learning in temporal networks via causal anonymous walks. In *International Conference on Learning Representations (ICLR)*, 2021.

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

[26] Gangda Deng, Hongkuan Zhou, Hanqing Zeng, Yinglong Xia, Christopher Leung, Jianbo Li, Rajgopal Kannan, and Viktor Prasanna. Taser: Temporal adaptive sampling for fast and accurate dynamic graph representation learning. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 926–937. IEEE, 2024.

[27] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[28] Stack-overflow. https://snap.stanford.edu/data/sx-stackoverflow.html.

[29] Kalev Leetaru and Philip A Schrodt. Gdelt: Global data on events, location, and tone, 1979–2012. In *ISA annual convention*, volume 2, pages 1–49. Citeseer, 2013.

[30] Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Scaling new heights: Transformative cross-gpu sampling for training billion-edge graphs. In *SC*, pages 1–15, 2024.

[31] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342, 2014.

[32] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX OSDI*, pages 17–30, 2012.

[33] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. Xpgraph: Xpline-friendly persistent memory graph stores for large-scale evolving graphs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1308–1325, 2022.

[34] Amauri Souza, Diego Mesquita, Samuel Kaski, and Vikas Garg. Provably expressive temporal graph networks. *Advances in neural information processing systems*, 35:32257–32269, 2022.

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

[36] Yufeng Wang and Charith Mendis. Tglite: A lightweight programming framework for continuous-time temporal graph neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1183–1199, 2024.

[37] NetworkX Developers. networkx-metis: Metis integration for networkx. https://github.com/networkx/networkx-metis.

[38] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. Distgnn: Scalable distributed training for large-scale graph neural networks. In *SC*, pages 1–14, 2021.

[39] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. Bytegnn: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment*, 15(6):1228–1242, 2022.

[40] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 417–434, 2022.

[41] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, and et.al. Peng. Bgl:gpu-efficient gnn training by optimizing graph data i/o and preprocessing. In *USENIX NSDI*, pages 103–118, 2023.

[42] Tim Kaler, Alexandros Iliopoulos, Philip Murzynowski, Tao Schardl, Charles E Leiserson, and Jie Chen. Communication-efficient graph neural networks with probabilistic neighborhood expansion analysis and caching. *Proceedings of Machine Learning and Systems*, 5:477–494, 2023.

[43] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. Flexgraph: a flexible and efficient distributed framework for gnn training. In *Proceedings of the European Conference on Computer Systems*, pages 67–82, 2021.

[44] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In *International Conference on Learning Representations*.

[45] Chaoyi Chen, Dechao Gao, Yanfeng Zhang, Qiange Wang, Zhenbo Fu, Xuecang Zhang, Junhua Zhu, Yu Gu, and Ge Yu. Neutronstream: A dynamic gnn training framework with sliding window for graph streams. *Proceedings of the VLDB Endowment*, 17(3):455–468, 2023.

[46] Shihong Gao, Yiming Li, Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. Simple: Efficient temporal graph neural network training at scale with dynamic data placement. *Proceedings of the ACM on Management of Data*, 2(3):1–25, 2024.

[47] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. Orca: Scalable temporal graph neural network training with theoretical guarantees. *Proceedings of the ACM on Management of Data*, 1(1):1–27, 2023.