MSPipe: Efficient Temporal GNN Training via Staleness-Aware Pipeline

Guangming Sheng
The University of Hong Kong
Hong Kong, China
gmsheng@connect.hku.hk

Junwei Su*
The University of Hong Kong
Hong Kong, China
junweisu@connect.hku.hk

Chao Huang
The University of Hong Kong
Hong Kong, China
chaohuang75@gmail.com

Chuan Wu
The University of Hong Kong
Hong Kong, China
cwu@cs.hku.hk

ABSTRACT

Memory-based Temporal Graph Neural Networks (MTGNNs) are a class of temporal graph neural networks that utilize a node memory module to capture and retain long-term temporal dependencies, leading to superior performance compared to memory-less counterparts. However, the iterative reading and updating process of the memory module in MTGNNs to obtain up-to-date information needs to follow the temporal dependencies. This introduces significant overhead and limits training throughput. Existing optimizations for static GNNs are not directly applicable to MTGNNs due to differences in training paradigm, model architecture, and the absence of a memory module. Moreover, these optimizations do not effectively address the challenges posed by temporal dependencies, making them ineffective for MTGNN training. In this paper, we propose MSPipe, a general and efficient framework for memory-based TGNNs that maximizes training throughput while maintaining model accuracy. Our design specifically addresses the unique challenges associated with fetching and updating node memory states in MTGNNs by integrating staleness into the memory module. However, simply introducing a predefined staleness bound in the memory module to break temporal dependencies may lead to suboptimal performance and lack of generalizability across different models and datasets. To overcome this, we introduce an online pipeline scheduling algorithm in MSPipe that strategically breaks temporal dependencies with minimal staleness and delays memory fetching to obtain fresher memory states. This is achieved without stalling the MTGNN training stage or causing resource contention. Additionally, we design a staleness mitigation mechanism to enhance training convergence and model accuracy. Furthermore, we provide convergence analysis and demonstrate that MSPipe maintains the same convergence rate as vanilla sampling-based GNN training. Experimental results show that MSPipe achieves up to 2.45× speedup without sacrificing accuracy, making it a promising solution for efficient MTGNN training. The implementation of our paper can be found at the following link: https://github.com/PeterSH6/MSPipe.

CCS CONCEPTS

• **Computing methodologies → Artificial intelligence**; • **Information systems → Information systems applications**.

KEYWORDS

Temporal Graph Neural Networks; Distributed Training; Efficient Training; Minimal Staleness Bound

ACM Reference Format:
Guangming Sheng, Junwei Su, Chao Huang, and Chuan Wu. 2024. MSPipe: Efficient Temporal GNN Training via Staleness-Aware Pipeline . In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24), August 25–29, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3637528.3671844

1 INTRODUCTION

Many real-world graphs exhibit dynamic characteristics, with nodes and edges continuously evolving over time, such as temporal social networks [26, 40] and temporal user-item graphs in recommendation systems [18, 44]. Previous attempts to model such dynamic systems have relied on static graph representations, which overlook their temporal nature [20, 29, 30, 46, 48]. Recently, temporal graph neural networks (TGNNs) have been developed to address this limitation. TGNNs are designed to incorporate time-aware information, learning both structural and temporal dependencies. Consequently, TGNNs facilitate more accurate and comprehensive modeling of dynamic graphs [8, 15, 25, 27, 31–34, 38, 41, 47].

Among the existing TGNN models, MTGNNs like TGN [25], APAN [38], JODIE [15], and TIGER [47] have achieved state-of-the-art performance on various tasks, notably link prediction and node classification [23]. Their success can be attributed to the node memory module, which stores time-aware representations, enabling the capture of intricate long-term information for each node. The training process of MTGNNs involves the following steps: First, node memory states and node/edge features from sampled subgraphs are loaded and inputted into the MTGNN model. In the model, the *message* module sequentially processes incoming events to generate message vectors. Subsequently, the *memory* module utilizes these

[*] Corresponding author

KDD '24, August 25–29, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-0490-1/24/08
https://doi.org/10.1145/3637528.3671844

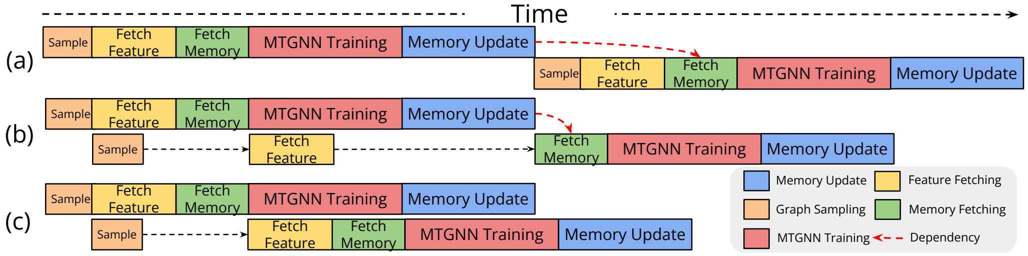


Figure 1: Memory-based TGNN training. (a) represents the general training scheme; (b) shows the pre-sampling and pre-fetching optimization; (c) is the case of breaking the temporal dependency, where the TGNN training stage is executed uninterruptedly.

message vectors along with the previous memory states to generate new memory vectors. Then, the *embedding* module combines the latest memory vectors with structural information to generate temporal embeddings for the vertices. At the end of each iteration, the updated memory states are written back to the memory module storage in the CPU's main memory, as illustrated in Figure 1.

Significant Overhead of The Memory Module in TGNNs. Despite their impressive performance, training memory-based TGNNs at scale remains challenging due to the *temporal dependency* induced by the memory module. This temporal dependency arises from the memory fetch and update operations across different iterations. Specifically, the latest memory state of a node cannot be fetched until the update of the node memory module in the previous iteration is completed. This dependency is illustrated by the red arrow in Figure 1, indicating that subsequent iterations rely on the most recently updated node memory from previous iterations. The memory module functions as a recursive filtering mechanism, continually distilling and incorporating information from historical events into the memory states. Respecting this temporal dependency incurs significant overhead in memory-based TGNN training, accounting for approximately 36.1% to 58.6% of the execution time of one training iteration, depending on the specific models. However, preserving this temporal dependency is essential for maintaining the model's performance. Therefore, it's imperative to enhance the training throughput while effectively modeling the temporal dependency without compromising the model's accuracy.

Limitation of Static GNN Optimizations. There is a line of research [10, 13, 22, 36, 49] focused on optimizing the training of static GNNs. However, the temporal dependencies specific to MTGNNs, arising from the memory module, pose unique challenges. As a result, these works are inadequate for handling such temporal dependencies and are ineffective for MTGNN training. For instance, when applying pre-sample and pre-fetch optimizations from ByteGNN [49] and SAILENT [13], the memory fetching in the next training iteration must wait until the memory update in the current iteration is completed, as shown in Figure 1(b). This waiting period diminishes training efficiency. Moreover, approaches like PipeGCN [36] and SAILENT [22] address the substantial communication overhead caused by inter-layer dependencies in multi-layer GNNs using the full graph training paradigm. However, these approaches may not be applicable to MTGNNs, which typically utilize a single layer and employ sample-based subgraph training. Hence, there is an urgent need for a general parallel execution framework enabling more efficient and scalable distributed MTGNN training.

To address these gaps, we introduce MSPipe, a general and efficient training system for memory-based TGNNs. MSPipe leverages a minimal staleness bound to accelerate MTGNN training while ensuring model convergence with theoretical guarantees.

Training Pipeline Formulation. To identify the bottlenecks in MTGNN training, we present a formulation for the MTGNN training pipeline. Through an analysis of initiation and completion times across various training stages, we decompose MTGNN training into distinct stages. This formulation enables a comprehensive analysis of training bottlenecks. Leveraging this formulation, we conduct a thorough profiling of distributed MTGNN training. Our analysis highlights the potential for optimizing the bottlenecks arising from the memory module and its temporal dependencies.

Tackling the Temporal Dependencies. We propose two key designs to enhance training throughput while preserving model accuracy. **(1)** We break the temporal dependencies by introducing staleness in the memory module, as shown in Figure 1(c). However, determining an appropriate staleness bound requires careful tuning and lacks generalizability across diverse MTGNN models and datasets. Setting a small bound may hinder system throughput, while a large bound can introduce errors in model training. To overcome this challenge, we design a minimal staleness algorithm that determines a precise staleness bound and effectively schedules the training pipeline accordingly. The resulting minimal staleness bound ensures uninterrupted execution of MTGNN training stages. Moreover, it allows for the retrieval of the node memory vectors that are as fresh as possible, effectively minimizing staleness errors. **(2)** To further improve the convergence of MSPipe, we propose a lightweight staleness mitigation method that leverages the node memory vectors of recently updated nodes with the highest similarity, which effectively reduces the staleness error.

Theoretical guarantees. Although previous works have analyzed the convergence rate of static GNN training [3, 6, 7], the consequences of violating temporal dependencies have not yet been explored. Therefore, we present an in-depth convergence analysis for our proposed methods, validating their effectiveness.

In summary, we make the following contributions in this paper:

- We propose a general formulation for the MTGNN training pipeline, allowing us to identify training bottlenecks arising from the memory module. Based on the formulation, MSPipe strategically determines a minimal staleness bound to ensure uninterrupted MTGNN training while minimizing staleness error, thereby maximizing training throughput with high accuracy.

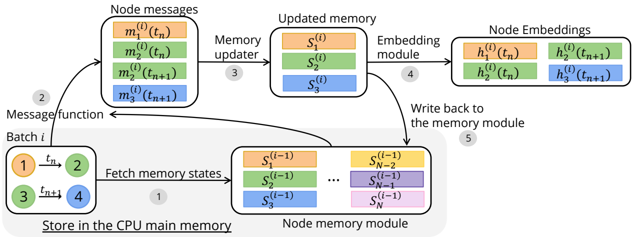


Figure 2: Memory-based TGNN Training Stages. The node memory states are stored in the CPU memory to ensure consistency among multiple training workers and reduce GPU memory contention. The MTGNN model is stored in the GPU.

- We propose a lightweight similarity-based staleness mitigation strategy to further improve the model convergence and accuracy.
- We provide a theoretical convergence analysis, demonstrating that MSPipe does not sacrifice convergence speed. The convergence rate of our method is the same as vanilla MTGNN training (without staleness).
- We evaluate the performance of MSPipe through extensive experiments. Our results demonstrate that MSPipe outperforms state-of-the-art MTGNN training frameworks, achieving up to 2.45× speed-up and 83.6% scaling efficiency without accuracy loss.

2 PRELIMINARY

Dynamic Graphs. We focus on event-based representation for dynamic graphs. A dynamic graph can be represented as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, ..., N\}$ is the node set and $\mathcal{E} = \{y_{uv}(t)\}, u, v \in \mathcal{V}$ is the event sets [25, 28, 41]. The event set $\mathcal{E}$ represents a sequence of graph events $y_{uv}(t)$, indicating interactions between nodes u and v at timestamp $t \geq 0$.

Temporal Graph Neural Network. Among the variety of TGNNs, memory-based TGNNs achieve superior accuracy in modeling temporal dynamics in graph-structured data [15, 23, 25, 38, 47]. Memory-based TGNNs maintain a node memory vector s_v for each node v in a dynamic graph that memorizes long-term dependencies. The memory update and training paradigms can be formulated as:

$$\begin{aligned} m_v^{(i)} &= msg\left(s_v^{(i-1)}, s_u^{(i-1)}, y_{uv}(t), \Delta t\right) \\ s_v^{(i)} &= mem\left(s_v^{(i-1)}, m_v^{(i)}\right) \\ h_v^{(i)} &= emb\left(s_v^{(i)}, s_u^{(i)} \mid u \in \mathcal{N}(v)\right) \end{aligned} \tag{1}$$

where $m_v^{(i)}$ represents a message generated by the graph event related to v that occurs at training iteration i, $s_v^{(i)}$ is the memory states and $h_v^{(i)}$ is the embedding of node v in iteration i and Δt represents the time gap between the last updated time of the memory state $s_v^{(i-1)}$ of node v and the occurrence time of the current graph event $y_{uv}(t)$. $\mathcal{N}(v)$ is the 1-hop temporal neighbours of nodes v. The message module *msg* (e.g., MLP), memory update module *mem* (e.g., RNN), and embedding module *emb* (e.g., a single layer GAT) are all learnable components. Note that all the operations described above collectively form the *MTGNN training stage*, which is executed on the GPU. The updated memory vectors $s_v^{(i)}$ will be written back to the node memory storage in the CPU main memory. The detailed training workflow is illustrated in Figure 2.

Table 1: Training time breakdown of TGN model.

Dataset	Sample	Fetch feature	Fetch memory	Train MTGNN	Update memory
REDDIT [15]	9.5%	12.6%	5.7%	46.9%	25.3%
WIKI [15]	6.6%	5.8%	5.8%	51.5%	30.3%
MOOC [15]	9.7%	3.0%	2.5%	53.1%	31.7%
LASTFM [15]	11.5%	9.1%	8.5%	43.0%	26.8%
GDELT [52]	17.6%	12.8%	10.5%	37.5%	21.6%

3 MSPIPE FRAMEWORK

We introduce MSPipe, a stall-free minimal-staleness scheduling system designed for MTGNN training (Figure 1(c)). Our approach identifies the memory module as the bottleneck and leverages pipelining techniques across multiple iterations to accelerate training. We determine the minimal number of staleness iterations necessary to prevent pipeline stalling while ensuring the retrieval of the most up-to-date memory states. However, incorporating the minimal staleness bound into the training pipeline introduces resource competition due to parallel execution. To mitigate this, we present a resource-aware online scheduling algorithm that controls the staleness bound and alleviates resource contention. Additionally, we propose a lightweight similarity-based memory update mechanism to further mitigate staleness errors and obtain fresher information.

3.1 MSPipe mechanism

Significant memory operations overhead. We consider a 5-stage abstraction of memory-based TGNN training, i.e., graph sampling, feature fetching, memory fetching, MTGNN training, and memory update. We conduct detailed profiling of the execution time of each stage, with time breakdown shown in Table 1. Memory operations incur substantial overhead ranging from 36.1% to 58.6% of one iteration training time for different MTGNN models, while sampling and feature fetching do not, due to the 1-layer MTGNN structure. In Figure 1(b), memory fetching depends on memory vectors updated at the end of the last iteration, and has to wait for the relatively long MTGNN training and memory updating to finish

Pipline mechanism. A natural design to accelerate the training process involves decoupling the temporal dependency between the memory update stage in one training iteration and the memory fetching stage in the subsequent iteration, by leveraging stale memory vectors in the latter. Figure 1(c) provides an overview of the training pipeline, where computation (e.g., MTGNN training) is parallelized with fragmented I/O operations including feature fetching, memory fetching, and memory update. The advanced memory fetching stage introduces a certain degree of staleness to the node memory module, causing the MTGNN model to receive outdated input. Mathematically, MSPipe's training can be formulated as follows (modifications from Eqn. 1 are highlighted in blue):

$$\begin{aligned} m_v^{(i)} &= msg\left(\tilde{s}_v^{(i-k)}, \tilde{s}_u^{(i-k)}, y_{uv}(t), \Delta t\right) \\ \tilde{s}_v^{(i)} &= mem\left(\tilde{s}_v^{(i-k)}, m_v^{(i)}\right) \\ h_v^{(i)} &= emb\left(\tilde{s}_v^{(i)},\ \tilde{s}_u^{(i)} \mid u \in \mathcal{N}(v)\right) \end{aligned} \tag{2}$$

where $\tilde{s}_v^{(i)}$ represents the memory vector of node v in training iteration i updated based on stale memory vector in iteration $i - k$,

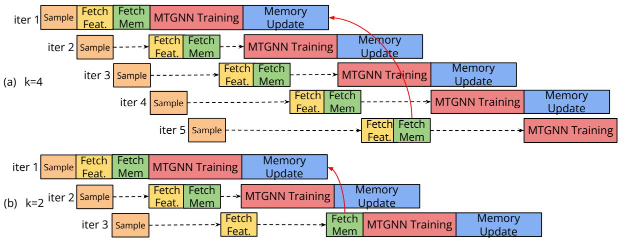


Figure 3: Pipeline execution. The dashed black arrow represents the bubble time. The red arrow denotes memory fetching to retrieve memory vectors updated k iterations before.

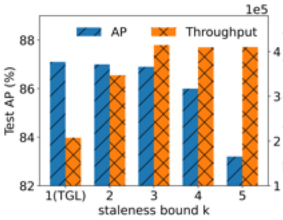
and $h_v^{(i)}$ is the embedding of node v. MSPipe uses the memory vector from k iterations before the current iteration to generate messages and train the model.

In the example pipeline in Figure 3, we have staleness bound $k = 2, 4$, indicating that MSPipe retrieves memory vectors updated two and four iterations before, respectively. Previous GNN frameworks [22, 36] use a predefined staleness bound to address different dependencies. *We argue that randomly selecting a staleness bound is inadequate. A small or large staleness bound may affect system performance or introduce errors in model training.* To support our argument, we conduct experiments on the LastFM dataset [15], training TGN model [25]. As shown in Figure 4, applying the smallest staleness bound (e.g., $k = 2$) leads to training throughput degradation, while employing a larger staleness bound (e.g., $k = 4, 5$) impacts model accuracy. To address this, we introduce a pipeline scheduling policy that determines the minimal staleness bound that maximizes system throughput without affecting model convergence.

3.2 Stall-free Minimal-staleness Pipeline

To maximize MTGNN training throughput, our objective is to enable the GPU to seamlessly perform computation (i.e., MTGNN training stage) without waiting for data preparation, as depicted in Figure 3(a). We seek to determine the minimal staleness bound k and perform resource-aware online pipeline scheduling to avoid resource contention. This approach enables maximum speed-up without stalling the MTGNN training stage and ensures model convergence. To accurately model resource contention, we analyze the resource requirements of different stages. Figure 5 demonstrates that feature fetching and memory fetching contend for the copy engine and PCIe resources during the copy operation from host to device. However, no contention is encountered during the memory update stage, as it involves a copy operation from device to host [5]. Additionally, we adopt a GPU sampler with restricted GPU resource allocation to avoid competition with the MTGNN training stage.

The start and end time modeling at different stages. The problem of ensuring uninterrupted execution of the MTGNN training stage with minimal staleness can be transformed into determining the start time of each training stage. Therefore, it's essential to model the range of starting and ending times for different stages. Let $b_i^{(j)}$ and $e_i^{(j)}$ denote the start time and end time of stage j in iteration i. The execution time of stage j, denoted as $\tau^{(j)}$, can be collected in a few iterations of profiling. The end time $e_i^{(j)}$ can be computed by adding execution time $\tau^{(j)}$ to the start time $b_i^{(j)}$, stated as $e_i^{(j)} = b_i^{(j)} + \tau^{(j)}$. There are three cases for computing $b_i^{(j)}$ to ensure sequential execution and avoid resource competition:



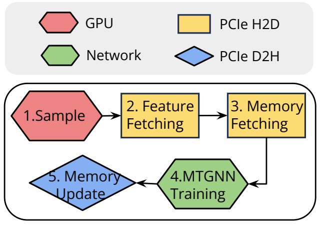


Figure 4: Model accuracy and training throughput at different staleness bounds.

Figure 5: Different resource requirements (by color/shape) of 5 training stages.

1) For the first stage, the sampler can initiate the sampling of a new batch immediately after the completion of the previous sample stage. This can be expressed as $b_i^{(1)} = e_{i-1}^{(1)} = b_{i-1}^{(1)} + \tau^{(1)}$.
2) In the second stage, feature fetching competes for PCIe and copy engine resources with memory fetching (stage 3) in the previous iteration. Hence, feature fetching cannot begin until both the memory fetching from the previous iteration and the sampling stage (stage 1) from the current iteration have been completed, as illustrated in Figure 3. Consequently, the start time is determined as $b_i^{(2)} = \max\left\{e_i^{(1)}, e_{i-1}^{(3)}\right\} = \max\left\{b_i^{(1)} + \tau^{(1)}, b_{i-1}^{(3)} + \tau^{(3)}\right\}$.
3) The remaining stages adhere to sequential execution order. Taking the MTGNN training stage as an example, it cannot commence until both the memory fetching from the current iteration and the same stage (i.e., MTGNN training) from the previous iteration have finished. The start time for these three stages are formulated as $b_i^{(j)} = \max\left\{e_i^{(j-1)}, e_{i-1}^{(j)}\right\} = \max\left\{b_i^{(j-1)} + \tau^{(j-1)}, b_{i-1}^{(j)} + \tau^{(j)}\right\}$. By combining the above results, we obtain the following equations:

$$b_i^{(j)} = \begin{cases} e_{i-1}^{(j)} & j = 1 \\ \max\left\{e_i^{(j-1)}, e_{i-1}^{(j+1)}\right\} & j = 2 \\ \max\left\{e_i^{(j-1)}, e_{i-1}^{(j)}\right\} & j \in [3, 5] \end{cases} \tag{3}$$

$$e_i^{(j)} = b_i^{(j)} + \tau^{(j)} \qquad j \in [1, 5] \tag{4}$$

Minimal-staleness bound. Given the start and end time ranges of different stages, we observe a time gap between the start time of stage $b_i^{(j)}$ and the end time of the previous stage $e_i^{(j-1)}$, referring to the bubble time in Figure 3. This motivates us to advance or delay the execution of a stage to obtain a fresher node memory state. To maximize training throughput with the least impact on model accuracy, our objective is to determine the minimal staleness bound k_i, ensuring that MSPipe fetches the most up-to-date memory vectors that are k_i iterations prior to the current iteration i, without causing pipeline stalling. To tackle this optimization process, we must satisfy the following three constraints:
C1: We ensure that memory updates for the $i - k_i$th iteration are completed before fetching memory states in the ith iteration, which can be expressed as $e_{i-k_i}^{(5)} \geq b_i^{(3)}$.

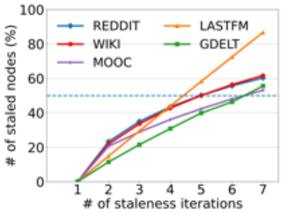


Figure 6: Percentage of nodes that use staled memory vectors under different numbers of staleness iterations

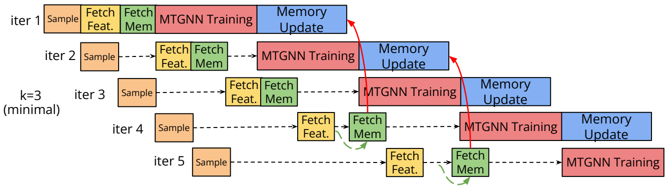


Figure 7: Resource-aware online schedule with minimal staleness bound is 3. The scheduler delays the memory fetch by utilizing the bubble time and avoids resource competence from different stages. The dashed green and black arrows represent the delay time and the bubble time respectively. The red arrow denotes fetching the memory states updated k iterations before.

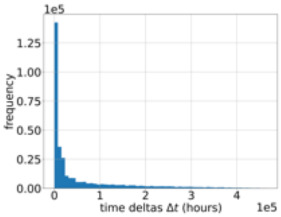


Figure 8: Distribution of Δt in WIKI dataset. Other datasets follow a similar power-law distribution.

C2: To enable incessant execution of MTGNN training stages on the GPU, we should guarantee that delaying the memory fetching stage does not stall the subsequent MTGNN training stage. This condition can be formulated as $e^{(5)}_{i-k_i} \leq b^{(4)}_i - \tau^{(3)}$, where $b^{(4)}_i - \tau^{(3)}$ represents the delayed starting time of the memory fetching stage.

C3: We apply an upper bound $k_{\max}$ on the staleness bound based on a key observation: *During each iteration, the memory module updates only a small subset of nodes’ memory vectors.* Consequently, it is only the memory vectors of these specific nodes that become stale when they are fetched prior to the memory update stage. Figure 6 demonstrates the increase in the percentage of stale nodes with larger staleness iterations. We select an upper bound $k_{\max}$ to ensure the percentage of stale nodes will not exceed 50%. Combining all above, we can formulate the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & k_i \\ \text{subject to} \quad & e^{(5)}_{i-k_i} \geq b^{(3)}_i, \\ & e^{(5)}_{i-k_i} \leq b^{(4)}_i - \tau^{(3)}, \\ 1 \leq k_i < \min\{i, k_{\max}\}&, i = 1, \ldots, E. \end{aligned}$$

Here E is the total number of iterations in an epoch. By iterating through each iteration, the above problem can be solved in $O(E)$.

Resource-aware online pipeline schedule. Once the minimal staleness iteration number k_i has been determined, we can schedule the training pipeline by deciding the commencement time of each stage. This scheduling problem can be modeled as a variant of the “bounded buffer problem” in producer-consumer systems [19]. Here, the buffer length corresponds to the number of staled iterations k_i, with the memory update stage acting as a slow consumer and the memory fetching stage as a fast producer. To ensure efficient training, the scheduler ensures that the training stages from different iterations do not compete for the same hardware resources and strictly adhere to a sequential execution order. By leveraging the minimal staleness iteration numbers k_i, the scheduler monitors the staleness state of each iteration and defers the memory fetching stage until the minimal staleness condition is satisfied, ensuring that subsequent MTGNN training stages are not impeded to maximize training throughput. This is achieved by effectively utilizing the bubble time to delay the memory fetching stage, as illustrated in Figure 7. The detailed pseudocode can be found in Appendix D.3.

3.3 Similarity-based Staleness mitigation

Nodes in the dynamic graph only update their memory states based on events directly involving them. Therefore, the nodes that are not involved in any graph events for a long duration will maintain stationary memory states, which would result in stale representations [15, 25]. MSPipe may aggravate this problem although minimal staleness is introduced. To improve model convergence and accuracy with MSPipe, we further propose a staleness mitigation strategy by aggregating memory states of recently active nodes with the highest similarity, which are considered to have similar and fresher temporal representations, to update the stale memory of a node. When node v’s memory has not been updated for time Δt, longer than a threshold γ, we update the stale memory of the node, $\tilde{s}^{(i-k_i)}_v$, by combining it with the averaged memory states of a set of most similar and active nodes $\Omega(v)$. An active node is defined to be the one whose memory is fresher than that of node v and Δt is smaller than γ. To measure the similarity between different nodes, we count their common neighbors which are reminiscent of the Jaccard similarity [16]. We observe that Δt follows a power-law distribution shown in Figure 8, which means that only a few Δt values are much larger than the rest. We accordingly set γ to p quantile (e.g., 99% quantile) of the Δt distribution to reduce staleness errors. We apply the following memory staleness mitigation mechanism in the memory fetching stage:

$$\hat{s}^{(i-k_i)}_v = \lambda \tilde{s}^{(i-k_i)}_v + (1-\lambda)\frac{\sum_{u\in\Omega(v)} \tilde{s}^{(i-k_i)}_u}{|\Omega(v)|}$$

where $\hat{s}^{(i-k_i)}_v$ is the mitigated memory vector of node v at iteration $i - k_i$, and λ is a hyperparameter in $[0, 1]$. The mitigated memory vector will then be fed into the memory update function to generate new memory states for the node:

$$\hat{s}^{(i)}_v = mem\left(\hat{s}^{(i-k_i)}_v, m^{(i)}_v\right)$$

4 THEORETICAL ANALYSIS

We analyze the convergence guarantee and convergence rate of MSPipe with respect to our bounded node memory vector staleness. By carefully scheduling the pipeline and utilizing stale memory vectors, we demonstrate that our approach incurs negligible approximation errors that can be bounded. We provide a rigorous analysis

Table 2: AP of dynamic link prediction and speedup. The best and second-best results are emphasized in bold and underlined. The AP difference smaller than 0.1% is considered the same. The results are averaged over 3 trials with standard deviations.

Model	Dataset	REDDIT		WIKI		MOOC		LASTFM		GDELT	
		AP(%)	Speedup	AP(%)	Speedup	AP(%)	Speedup	AP(%)	Speedup	AP(%)	Speedup
TGN	TGL	**99.82(0.01)**	1×	**99.43(0.03)**	1×	**99.42(0.03)**	1×	87.21(1.90)	1×	**98.23(0.05)**	1×
	Presample	**99.80(0.01)**	1.16×	**99.43(0.03)**	1.12×	**99.40(0.03)**	1.16×	87.12(1.51)	1.36×	98.18(0.05)	1.32×
	MSPipe	**99.81(0.02)**	**1.77×**	99.14(0.03)	**1.54×**	99.32(0.03)	**1.50×**	86.93(0.89)	**2.00×**	**98.25(0.06)**	**2.36×**
	MSPipe-S	**99.82(0.01)**	1.72×	99.39(0.03)	1.52×	**99.48(0.03)**	1.47×	**87.93(1.26)**	1.96×	**98.29(0.04)**	2.26×
JODIE	TGL	**99.63(0.02)**	1×	**98.40(0.03)**	1×	**98.64(0.01)**	1×	73.04(2.89)	1×	98.01(0.07)	1×
	Presample	**99.62(0.03)**	1.10×	**98.41(0.03)**	1.14×	**98.61(0.03)**	1.09×	72.96(2.68)	1.37×	98.04(0.05)	1.73×
	MSPipe	**99.62(0.02)**	**1.55×**	97.24(0.02)	**1.65×**	**98.63(0.02)**	**1.50×**	71.7(2.84)	**1.87×**	98.12(0.08)	**2.28×**
	MSPipe-S	**99.63(0.02)**	1.50×	97.61(0.02)	1.54×	**98.66(0.02)**	1.48×	**76.32(2.45)**	1.79×	**98.23(0.05)**	2.23×
APAN	TGL	**99.62(0.03)**	1×	**98.01(0.03)**	1×	**98.60(0.03)**	1×	73.37(1.59)	1×	95.80(0.02)	1×
	Presample	**99.65(0.02)**	1.38×	**98.03(0.03)**	1.06×	**98.62(0.03)**	1.30×	73.24(1.70)	1.49×	95.83(0.04)	1.71×
	MSPipe	**99.63(0.03)**	**2.03×**	96.43(0.04)	**1.78×**	98.38(0.02)	**1.91×**	72.41(1.21)	**2.37×**	95.94(0.03)	**2.45×**
	MSPipe-S	**99.64(0.03)**	1.96×	97.12(0.03)	1.63×	**98.64(0.03)**	1.77×	**76.08(1.42)**	2.19×	**96.02(0.03)**	2.41×

of the convergence properties of our approach, which establishes the theoretical foundation for its effectiveness in practice.

Theorem 4.1 (Convergent result, informal). *With a memory-based TGNN model, suppose that **1)** there is a bounded difference between the stale node memory vector $\tilde{s}_v^{(i)}$ and the exact node memory vector $s_v^{(i)}$ with the staleness bound ϵ_s, i.e., $\left\|\tilde{s}_v^{(i)} - s_v^{(i)}\right\|_F \leq \epsilon_s$ where $\|\|_F$ is the Frobenius norm; **2)** the loss function $\mathcal{L}$ in MTGNN training is bounded below and L-smooth; and **3)** the gradient of the loss function $\mathcal{L}$ is ρ-Lipschitz continuous. Choose step size $\eta = \min\left\{\frac{2}{L}, \frac{1}{\sqrt{t}}\right\}$. There exists a constant $D > 0$ such that:*

$$\min_{1 \leq t \leq T} \left\|\nabla \mathcal{L}(W_t)\right\|_F^2 \leq \left[2\mathcal{L}(W_0) - \mathcal{L}(W^*) + \rho D\right] \frac{1}{\sqrt{T}},$$

where W_0, W_t and W^* are the initial, step-t and optimal model parameters, respectively.

The formal version of Theorem 4.1 along with its proof can be found in Appendix A. Theorem 4.1 indicates that the convergence rate of MSPipe is $O(T^{-\frac{1}{2}})$, which shows that our approach maintains the same convergence rate as vanilla sampling-based GNN training methods ($O(T^{-\frac{1}{2}})$ [3, 6, 7]).

5 EXPERIMENTS

We conduct experiments to evaluate the proposed framework MSPipe, targeting answering the following research questions:

- Can MSPipe outperform state-of-the-art baseline MTGNN training systems on different models and datasets? (Section 5.2)
- Can MSPipe maintain the model accuracy and preserve the convergence rate? (Section 5.2 and 5.3)
- How do the key designs in MSPipe contribute to its overall performance, and what is its sensitivity to hyperparameters? (Section 5.4 and 5.5)
- How are the memory footprint and GPU utilization when applying staleness in MSPipe?(Section 5.6)

5.1 Experiment settings

Testbed. The main experiments are conducted on a machine equipped with two 64-core AMD EPYC CPUs, 512GB DRAM, and four NVIDIA A100 GPUs (40GB), and the scalability experiments are conducted on two of such machines with 100Gbps interconnect bandwidth.

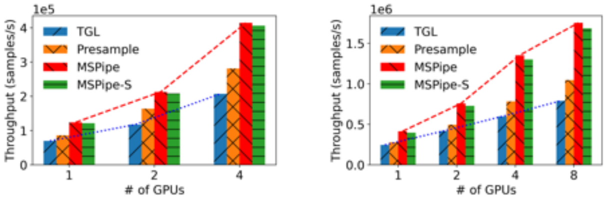
Table 3: The detailed statistics of the datasets. $|d_v|$ and $|d_e|$ show the dimensions of node features and edge features.

Dataset	$\|V\|$	$\|E\|$	$\|d_v\|$	$\|d_e\|$	Duration
Reddit [15]	10,984	672,447	0	172	1 month
WIKI [15]	9,227	157,474	0	172	1 month
MOOC [15]	7,144	411,749	0	128	17 months
LastFM [15]	1,980	1,293,103	0	128	1 month
GDELT [52]	16,682	191,290,882	413	186	5 years

Datasets and Models. We evaluate MSPipe on five temporal datasets: REDDIT, WIKI, MOOC, LASTFM [15] and a large dataset GDELT [52]. Table 3 summarizes the statistics of the temporal datasets. On each dataset, we use the same 70%-15%-15% chronological train/validation/test set split as in previous works [25, 41]. We train 3 state-of-the-art memory-based TGNN models, JODIE [15], TGN [25] and APAN [38]. The implementations of TGN, JODIE, and APAN are modified from TGL [52] which was optimized by TGL to achieve better accuracy than their original versions.

Baselines. We adopt **TGL** [52], a state-of-the-art MTGNN training system, as the synchronous MTGNN training baseline. We also implement the **Presample** (with pre-fetching features) mechanism similar to SAILENT [13] on TGL as a stricter baseline, which provides a parallel sampling and feature fetching scheme by executing them in advance. We implement **MSPipe** on PyTorch [21] and DGL [37], supporting both single-machine multi-GPU and multi-machine distributed MTGNN training. **MSPipe-S** is MSPipe with staleness mitigation from similar neighbors with λ set to 0.95. Noted that MSPipe does not enable the staleness mitigation by default. The implementation details of MSPipe can be found in Appendix D.

Training settings. To ensure a fair comparison, we used the same default hyperparameters as TGL, including a learning rate of 0.0001, a local batch size of 600 (4000 for the GDELT dataset), and hidden dimensions and memory dimensions of 100. We train each dataset for 100 epochs, except for GDELT, which was trained in 10 epochs. We sampled the 10 most recent 1-hop neighbors for all datasets and constructed mini-batches with an equal number



(a) LASTFM (b) GDELT

Figure 9: Scalability of training TGN.

of positive and negative node pairs for sampling and subgraph construction during training and evaluation. The experiments are conducted under the transductive learning setting and we use average precision for evaluation metrics. For a more comprehensive analysis of various batch sizes, we provide detailed experiments in Appendix E.4.

5.2 Expedited Training While Maintaining Accuracy

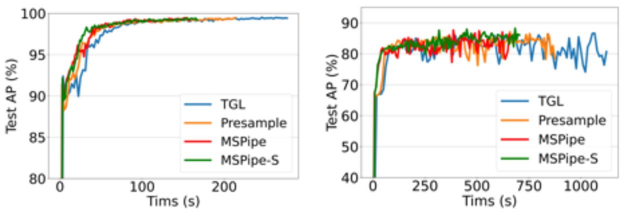
The results in Table 2 show that MSPipe improves the training throughput while maintaining high model accuracy. AP in the table stands for average model precision evaluated on the test set.

Training Throughput. We observe that MSPipe is 1.50× to 2.45× faster than TGL, and achieves up to 104% speed-up as compared to the Presample mechanism. MSPipe obtains the best speed-up on GDELT, which can be attributed to the relatively smaller proportion of execution time devoted to the MTGNN training stage compared to other datasets (as shown in Table 1). This is mainly because MSPipe effectively addresses the primary bottlenecks in MTGNN training by breaking temporal dependencies between iterations and ensuring uninterrupted progression of the MTGNN training stage, thereby enabling seamless overlap with other stages. Consequently, the total training time is predominantly determined by the uninterrupted MTGNN training stage. Notably, a smaller MTGNN training stage results in a larger speed-up, further contributing to the superior performance of MSPipe.

Model Accuracy. MSPipe without staleness mitigation can already achieve comparable test average precision with TGL on all datasets, with a marginal degradation ranging from 0 to 1.6%. This can be attributed to the minimal staleness mechanism and proper pipeline scheduling in MSPipe.

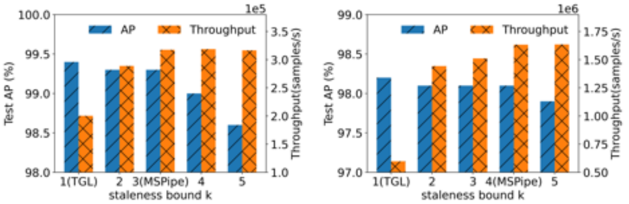
Staleness Mitigation. With the proposed staleness mitigation mechanism, MSPipe-S consistently achieves higher average precision than MSPipe across all models and datasets. Notably, MSPipe-S achieves the same test accuracy as TGL on REDDIT and MOOC datasets, while surpassing TGL's model performance on LastFM and GDELT datasets. MSPipe-S introduces a minimal overhead of only 3.73% on average for the staleness mitigation process. This demonstrates the efficiency of the proposed mechanism in effectively mitigating staleness while maintaining high-performance.

Scalability. Figure 9 presents the training throughput with different numbers of GPUs on LastFM and GDELT datasets. MSPipe achieves not only consistent speed-up but also up to 83.6% scaling efficiency on a machine, which is computed as the ratio of the speed-up achieved by using 4 GPUs to the ideal speed-up, outperforming other baselines. We also scale TGN training on GDELT to two machines with eight GPUs in Figure 9(b). Without explicit optimization for inter-machine communication, MSPipe still outperforms the baselines and exhibits better scalability.



(a) WIKI (b) LASTFM

Figure 10: Convergence of TGN training. x-axis is the wall-clock training time, and y-axis is the test average precision.



(a) MOOC (b) GDELT

Figure 11: Throughput and AP on different staleness bound

GPU sampler Analysis. Although MSPipe utilizes a GPU sampler for faster sampling, we found that our sampler is 24.3% faster than TGL's CPU sampler for 1-hop most recent sampling, which accounts for only 3.6% of the total training time as shown in Table 7 in Appendix C.4. Therefore, the performance gain is primarily attributed to our pipeline mechanism and resource-aware minimal staleness schedule but not to the acceleration of the sampler.

5.3 Preserving Convergence Rate

To validate that MSPipe can maintain the same convergence rate as vanilla sampling-based GNN training without applying staleness ($O(T^{-\frac{1}{2}})$), we compare the training curves of all models on all datasets in Figure 10 (the complete results can be found in Appendix E.2.3). We observe that MSPipe's training curves largely overlap with those of vanilla methods (TGL and Presample), verifying our theoretical results in Section 4. With staleness mitigation, MSPipe-S can achieve even better and more steady convergence (e.g., on WIKI and LastFM) than others.

5.4 Stall-free Minimal Staleness Bound

To further validate that MSPipe can find the minimal staleness bound without delaying the MTGNN training stage, we conduct a comparative analysis of accuracy and throughput between the minimal staleness bound computed by MSPipe and other different staleness bounds k. The results, depicted in Figure 11, consistently demonstrate that MSPipe achieves the highest throughput while maintaining the best accuracy compared to other staleness bound options. Additionally, the computed minimal staleness bounds for various datasets range from 2 to 4, providing further evidence for the

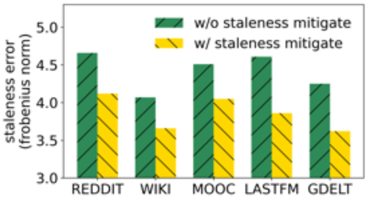


Figure 12: Staleness error comparison on TGN

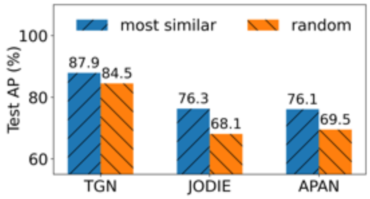


Figure 13: Staleness mitigation with most similar or random nodes on LastFM

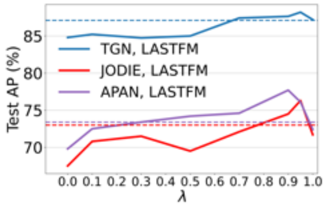


Figure 14: Hyperpara-meter analysis

Table 4: Additional memory overhead of TGN when applying staleness. Other models can be found in Appendix E.6.

Overhead\Dataset	REDDIT	WIKI	MOOC	LastFM	GDELT
Addition	48.8MB	38.4MB	42.9MB	38.1MB	1.17GB
Upperbound	51.4MB	34.3MB	44.3MB	44.3MB	1.35GB
GPU Mem (40GB) portion	0.12%	0.10%	0.11%	0.09%	2.92%

necessity of accurately determining the minimal staleness bound rather than relying on random selection. Note that $k = 1$ represents the baseline method of TGL without applying staleness.

5.5 Staleness Mitigation Mechanism

Error reduction. To better understand the accuracy enhancement and convergence speed-up achieved by MSPipe-S, we conduct a detailed analysis of the intermediate steps involved in our staleness mitigation mechanism. Specifically, we refer to Theorem 4.1, where we assume the existence of a bounded difference ϵ_s between the stale node memory vector $\tilde{s}_v^{(i)}$ and the precise node memory vector $s_v^{(i)}$. To assess the effectiveness of our staleness mitigation mechanism, we compare the mitigated staleness error $\left\|\hat{s}_v^{(i)} - s_v^{(i)}\right\|_F$ obtained after applying our mechanism with the original staleness error $\left\|\tilde{s}_v^{(i)} - s_v^{(i)}\right\|_F$. As shown in Figure 12, MSPipe-S consistently reduces the staleness error across all datasets, validating the theoretical guarantee and the effectiveness in enhancing accuracy.

Benefit of using most-similar neighbors. We further investigate our staleness mitigation mechanism by comparing using the most similar and active nodes for staleness mitigation with utilizing random active nodes, on the LastFM dataset. In Figure 13, we observe that our proposed most similar mechanism leads to better model performance, while a random selection from the active nodes would even degrade model accuracy. This can be attributed to the fact that similar nodes possess resemblant representations, enabling the stale node to acquire more updated information. Further details regarding the comparison of memory similarity between the most similar nodes and random nodes can be found in Appendix E.2.

Hyperparameter analysis. We examine the effect of hyperparameter λ on test accuracy, as depicted in Figure 14. The dashed horizontal lines in the figure denote the AP from the TGL baseline for comparison. We find that mitigating staleness with a larger λ (> 0.8) results in better model performance than TGL's results, indicating that we should retain more of the original stale memory representations and apply a small portion of mitigation from their similar ones. Notably, setting λ to 1 causes MSPipe-S to revert to the standard MSPipe configuration, thereby omitting the staleness mitigation strategy entirely.

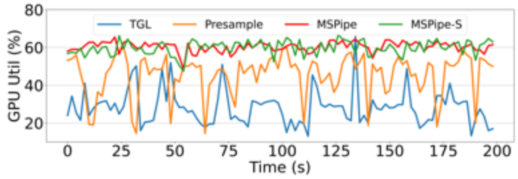


Figure 15: GPU utilization of different methods when training TGN with LastFM dataset.

5.6 GPU memory and utilization

We present an analysis of the memory overhead associated with MSPipe, as staleness-based strategies generally require additional memory to enhance training throughput. Unlike other asynchronous training frameworks [4, 17, 22, 36] that introduce staleness during DNN or GNN parameter learning, MSPipe only introduces staleness within the memory module to break temporal dependencies. Each subgraph is executed sequentially, resulting in no additional hidden states during MTGNN computation. The extra memory consumption in MSPipe comes from the prefetched subgraph, including node/edge features and memory vectors. We provide a detailed analysis to determine the upper bound of this additional memory overhead, assuming maximum neighbor size for all nodes (i.e., $\mathcal{N} = 10$). Additionally, we measure the actual memory consumption using *torch.cuda.memory_summary()* API in experiments across all datasets. Table 4 shows that the observed additional memory usage in MSPipe aligns with our analyzed upper bound. Moreover, we compare the additional memory cost with the GPU memory size in Table 4, demonstrating that it constitutes a relatively small proportion (up to 2.92%) of the modern GPU's capacity.

Figure 15 presents the GPU utilization during the training of the TGN model using the LastFM dataset. The plot showcases the average utilization of 4 A100 GPUs, with a smoothing interval of 2 seconds. The utilization data was collected throughout the training process across multiple epochs, excluding the validation stage. In Figure 15, both MSPipe and MSPipe-S demonstrate consistently high GPU utilization, outperforming the baseline methods that exhibit significant fluctuations. This notable improvement can be attributed to the minimal staleness and pipeline scheduling mechanism introduced in MSPipe, ensuring uninterrupted execution of the MTGNN training stage. In contrast, the TGL and Presample methods require the GPU to wait for data preparation, resulting in decreased GPU utilization and overall performance degradation.

6 RELATED WORKS

Sampling-based mini-batch training has become the norm for static GNN and TGNN training [10, 11, 35, 43, 45], which samples a subset of neighbors of target nodes to generate a subgraph, as input to GNN. The bottlenecks mainly lie in subgraph sampling and feature fetching due to the neighbor explosion problem [2, 42]. ByteGNN [49] and SALIENT [13] adopt pre-sampling and pre-fetching to hide sampling and feature fetching overhead in multi-layer static GNN training. These optimizations may not address the bottleneck in TGNN training, where maintaining node memories in sequential order incurs overhead while lightweight sampling and feature fetching are sufficient for single TGNN layer [15, 25, 38, 47].

Asynchronous Distributed Training. Many studies advocate asynchronous training with staleness for DNN and static GNN models. For distributed DNN training, previous works [1, 4, 9, 12, 17, 24] adopt stale weight gradients on large model parameters to eliminate communication overhead, while GNN models typically have much smaller sizes. For static GNN training, PipeGCN [36] and Sancus [22] introduce staleness in node embeddings under the full-graph training paradigm. Although these methods are effective for static GNNs, their effect is limited when applied to MTGNNs, from three aspects: **1)** they focus on full graph training and apply staleness between multiple GNN layers to overlap the significant communication overhead with computation. In MTGNN training, the communication overhead is relatively small due to subgraph sampling and the presence of only one GNN layer. **2)** all previous GNN training frameworks simply introduce a pre-defined staleness bound without explicitly analyzing the relationship between model quality and training throughput, potentially leading to sub-optimal parallelization solutions; **3)** the unique challenges arising from the temporal dependency caused by memory fetching and updating in MTGNN training have not been adequately addressed by these frameworks. Therefore, these asynchronous training frameworks for DNN and static GNN are not suitable for accelerating MTGNNs. A detailed analysis of the related work can be found in Appendix B.

7 CONCLUSION

We present MSPipe, a general and efficient memory-based TGNN training framework that improves training throughput while maintaining model accuracy. MSPipe addresses the unique challenges posed by temporal dependency in MTGNN. MSPipe strategically identifies the minimal staleness bound to adopt and proposes an online scheduler to dynamically control the staleness bound without stalling the pipeline. Moreover, MSPipe employs a lightweight staleness mitigation strategy and provides a comprehensive theoretical analysis for MTGNN training. Extensive experiments validate that MSPipe attains significant speed-up over state-of-the-art TGNN training frameworks while maintaining high model accuracy.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and area chairs for their helpful comments. This work is supported in part by grants from Hong Kong RGC under the contracts HKU 17207621, 17203522 and C7004-22G (CRF)

REFERENCES

[1] Saar Barkai, Ido Hakimi, and Assaf Schuster. 2019. Gap aware mitigation of gradient staleness. *arXiv preprint arXiv:1909.10802* (2019).

[2] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247* (2018).

[3] Jianfei Chen, Jun Zhu, and Le Song. 2017. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568* (2017).

[4] Yangrui Chen, Cong Xie, Meng Ma, Juncheng Gu, Yanghua Peng, Haibin Lin, Chuan Wu, and Yibo Zhu. 2022. SAPipe: Staleness-Aware Pipeline for Data Parallel DNN Training. In *Advances in Neural Information Processing Systems*.

[5] Jack Choquette and Wish Gandhi. 2020. Nvidia a100 gpu: Performance & innovation for gpu computing. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 1–43.

[6] Weilin Cong, Rana Forsati, Mahmut Kandemir, and Mehrdad Mahdavi. 2020. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1393–1403.

[7] Weilin Cong, Morteza Ramezani, and Mehrdad Mahdavi. 2021. On the importance of sampling in learning graph convolutional networks. *arXiv preprint arXiv:2103.02696* (2021).

[8] Weilin Cong, Si Zhang, Jian Kang, Baichuan Yuan, Hao Wu, Xin Zhou, Hanghang Tong, and Mehrdad Mahdavi. 2023. Do We Really Need Complicated Model Architectures For Temporal Networks? *arXiv preprint arXiv:2302.11636* (2023).

[9] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric Xing. 2018. Toward Understanding the Impact of Staleness in Distributed Machine Learning. In *International Conference on Learning Representations*.

[10] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale.. In *OSDI*. 551–568.

[11] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[12] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in neural information processing systems* 26 (2013).

[13] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. 2022. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems* 4 (2022), 172–189.

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

[15] 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.

[16] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2020. *Mining of massive data sets.* Cambridge university press.

[17] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. 2018. Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training. *Advances in Neural Information Processing Systems* 31 (2018).

[18] Yifei Ma, Balakrishnan Narayanaswamy, Haibin Lin, and Hao Ding. 2020. Temporal-contextual recommendation in real-time. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2291–2299.

[19] Syed Nasir Mehmood, Nazleeni Haron, Vaqar Akhtar, and Younus Javed. 2011. Implementation and experimentation of producer-consumer synchronization problem. *International Journal of Computer Applications* 975, 8887 (2011), 32–37.

[20] Giang Hoang Nguyen, John Boaz Lee, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-time dynamic network embeddings. In *Companion proceedings of the the web conference 2018*. 969–976.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[22] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1937–1950.

[23] Farimah Poursafaei, Shenyang Huang, Kellin Pelrine, , and Reihaneh Rabbany. 2022. Towards Better Evaluation for Dynamic Link Prediction. In *Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks*.

[24] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems* 24 (2011).

[25] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2021. Temporal Graph Networks for Deep Learning on Dynamic Graphs. In *Proceedings of International Conference on Learning Representations*.

[26] Polina Rozenshtein and Aristides Gionis. 2019. Mining temporal networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 3225–3226.

[27] 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.

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

[29] Junwei Su, Lingjun Mao, and Chuan Wu. 2024. BG-HGNN: Toward Scalable and Efficient Heterogeneous Graph Neural Network. *arXiv preprint arXiv:2403.08207* (2024).

[30] Junwei Su and Peter Marbach. 2022. Structure of Core-Periphery Communities. In *International Conference on Complex Networks and Their Applications*. Springer, 151–161.

[31] Junwei Su and Chuan Wu. 2023. Towards robust inductive graph incremental learning via experience replay. *arXiv preprint arXiv:2302.03534* (2023).

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

[33] Junwei Su, Difan Zou, Zijun Zhang, and Chuan Wu. 2023. Towards robust graph incremental learning on evolving graphs. In *International Conference on Machine Learning*. PMLR, 32728–32748.

[34] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. Dyrep: Learning representations over dynamic graphs. In *International conference on learning representations*.

[35] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks. In *Eighteenth European Conference on Computer Systems (EuroSys' 23)*.

[36] C Wan, Y Li, Cameron R Wolfe, A Kyrillidis, Nam S Kim, and Y Lin. 2022. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. In *The Tenth International Conference on Learning Representations (ICLR 2022)*.

[37] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[38] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. 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.

[39] 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.

[40] Wei Wei and Kathleen M Carley. 2015. Measuring temporal patterns in dynamic social networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 10, 1 (2015), 1–27.

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

[42] Sijie Yan, Yuanjun Xiong, and Dahua Lin. 2018. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

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

[44] Wenwen Ye, Shuaiqiang Wang, Xu Chen, Xuepeng Wang, Zheng Qin, and Dawei Yin. 2020. Time matters: Sequential recommendation with complex temporal information. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*. 1459–1468.

[45] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.

[46] Yao Zhang, Yun Xiong, Xiangnan Kong, Zhuang Niu, and Yangyong Zhu. 2019. IGE+: A Framework for Learning Node Embeddings in Interaction Graphs. *IEEE Transactions on Knowledge and Data Engineering* 33, 3 (2019), 1032–1044.

[47] Yao Zhang, Yun Xiong, Yongxiang Liao, Yiheng Sun, Yucheng Jin, Xuehao Zheng, and Yangyong Zhu. 2023. TIGER: Temporal Interaction Graph Embedding with Restarts. *arXiv preprint arXiv:2302.06057* (2023).

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

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

[50] 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).

[51] Hongkuan Zhou, Bingyi Zhang, Rajgopal Kannan, Viktor K. Prasanna, and Carl E. Busart. 2022. Model-Architecture Co-Design for High Performance Temporal GNN Inference on FPGA. *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2022), 1108–1117.

[52] 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).

A PROOFS

In this section, we provide the detailed proofs of the theoretical analysis.

Lemma A.1. *If $f(\cdot)$ is β-smooth, then we have,*

$$f(y) \leq f(x) + \left\langle \nabla f(x), y - x \right\rangle + \frac{\beta}{2} \|y - x\|^2$$

Proof.

$$\begin{aligned}
&\left| f(y) - f(x) - \left\langle \nabla f(x), y - x \right\rangle \right| \\
=&\left| \int_0^1 \left\langle \nabla f(x) + t(y - x), y - x \right\rangle dt - \left\langle \nabla f(x), y - x \right\rangle \right| \\
\leq& \int_0^1 \left| \left\langle \nabla f(x) + t(y - x) - \nabla f(x), y - x \right\rangle \right| dt \\
\leq& \int_0^1 \|\nabla f(x) + t(y - x) - \nabla f(x)\| \cdot \|y - x\| dt \\
\leq& \int_0^1 t\beta \|y - x\|^2 dt \\
=&\frac{\beta}{2} \|y - x\|^2
\end{aligned}$$

Lemma A.2. *if $\mathcal{L}(\cdot)$ is ρ-Lipschitz smooth, then we have*

$$\left\| \nabla \tilde{\mathcal{L}}(W) - \nabla \mathcal{L}(W) \right\|_F \leq \rho \epsilon_s$$

where $\nabla \tilde{\mathcal{L}}\left(W_t\right)$ denote the gradient when stale memoys are used.

Proof. By the assumption that there is a bounded difference between the stale node memory vector $\tilde{S_i}$ and the exact node memory vector S_i with the staleness bound ϵ_s, we have:

$$\left\| S - \tilde{S} \right\|_F \leq \epsilon_s$$

By smoothness of $\mathcal{L}(\cdot)$, we have

$$\begin{aligned}
&\left\| \nabla \mathcal{L}(S, W) - \nabla \mathcal{L}(\tilde{S}, W) \right\|_F \\
=&\left\| \nabla \tilde{\mathcal{L}}(W) - \nabla \mathcal{L}(W) \right\|_F \\
\leq&\rho \epsilon_s
\end{aligned}$$

Learning Algorithms. In the t^{th} step, we have

$$W_{t+1} - W_t = -\eta_t \nabla \tilde{\mathcal{L}}\left(W_t\right) \tag{5}$$

, where $\nabla \tilde{\mathcal{L}}\left(W_t\right)$ denote the gradient when stale memoys are used and η_t is the learning rate.

By Lemma 1 and the L_f-smoothness of $\mathcal{L}$, we have

$$\mathcal{L}(W_{t+1}) - \mathcal{L}(W_t) \leq \left\langle W_{t+1} - W_t, \nabla \mathcal{L}(W_t) \right\rangle + \frac{L_f}{2} \|W_{t+1} - W_t\|_F^2 \tag{6}$$

Use Eqn. 5 to substitute, we have

$$\mathcal{L}(W_{t+1}) - \mathcal{L}(W_t) \le \underbrace{-\eta_t \langle \nabla \tilde{\mathcal{L}}(W_t), \nabla \mathcal{L}(W_t) \rangle}_{\textcircled{1}} + \underbrace{\frac{L_f \eta_t^2}{2} \|\nabla \tilde{\mathcal{L}}(W_t)\|_F^2}_{\textcircled{2}} \tag{7}$$

We bound the terms step by step and let $\delta_t = \nabla \tilde{\mathcal{L}}(W_t) - \nabla \mathcal{L}(W_t)$ to subsitute in Equ. 7.

First, For ①, we have

$$\begin{aligned} & -\eta_t \left\langle \nabla \tilde{\mathcal{L}}(W_t), \nabla \mathcal{L}(W_t) \right\rangle \\ =& -\eta_t \left\langle \delta_t + \nabla \mathcal{L}(W_t), \nabla \mathcal{L}(W_t) \right\rangle \\ =& -\eta_t \left[\left\langle \delta_t, \nabla \mathcal{L}(W_t) \right\rangle + \left\| \nabla \mathcal{L}(W_t) \right\|_F^2 \right] \end{aligned}$$

For ②, we have

$$\begin{aligned} & \frac{L_f \eta_t^2}{2} \left\| \nabla \tilde{\mathcal{L}}(W_t) \right\|_F^2 \\ =& \frac{L_f \eta_t^2}{2} \left\| \delta_t + \nabla \mathcal{L}(W_t) \right\|_F^2 \\ =& \frac{L_f \eta_t^2}{2} \left(\left\| \delta_t \right\|_F^2 + 2 \left\langle \delta_t, \nabla \mathcal{L}(W_t) \right\rangle + \left\| \nabla \mathcal{L}(W_t) \right\|_F^2 \right) \end{aligned}$$

Combining both ① and ② together and by the choice of learning rate $\eta_t = \frac{1}{L_f}$, we have

$$\mathcal{L}(W_{t+1}) - \mathcal{L}(W_t) \le -\left(\eta_t - \frac{L_f}{2} \eta_t^2 \right) \left\| \nabla \mathcal{L}(W_t) \right\|_F^2 + \frac{L_f \eta_t^2}{2} \left\| \delta_t \right\|_F^2$$

By Lemma 2. we have $\|\delta_t\|_F^2 \le \rho \epsilon_s$

$$\mathcal{L}(W_{t+1}) - \mathcal{L}(W_t) \le -\left(\eta_t - \frac{L_f}{2} \eta_t^2 \right) \left\| \nabla \mathcal{L}(W_t) \right\|_F^2 + \frac{L_f \eta_t^2}{2} \rho \epsilon_s \tag{8}$$

Rearrange Eqn. 8 and let $c = \frac{L_f \rho \epsilon_s}{2}$, we have,

$$\left(\eta_t - \frac{L_f}{2} \eta_t^2 \right) \left\| \nabla \mathcal{L}(W_t) \right\|_F^2 \le \mathcal{L}(W_t) - \mathcal{L}(W_{t+1}) + \eta_t^2 c \tag{9}$$

Telescope sum from $t = 1...T$, we have

$$\sum_{t=1}^{T} \left(\eta_t - \frac{L_f \eta_t^2}{2} \right) \|\nabla \mathcal{L}(W_t)\|_F^2 \le \mathcal{L}(W_0) - \mathcal{L}(W_T) + \sum_{t=1}^{T} \eta_t^2 c \tag{10}$$

$$\min_{1 \le t \le T} \|\nabla \mathcal{L}(W_t)\|_F^2 \le \frac{\mathcal{L}(W_0) - \mathcal{L}(W_T)}{\sum_{t=1}^{T} \left(\eta_t - \frac{L_f \eta_t^2}{2} \right)} + \frac{\sum_{t=1}^{T} \eta_t^2 c}{\sum_{t=1}^{T} \left(\eta_t - \frac{L_f \eta_t^2}{2} \right)} \tag{11}$$

Substitute Equ. 11 with $\eta_t = min\{\frac{1}{\sqrt{t}}, \frac{1}{L_f}\}$ and $\mathcal{L}(W^*) \le \mathcal{L}(W_T)$, we have

$$\begin{aligned} & \min_{1 \le t \le T} \|\nabla \mathcal{L}(W_t)\|_F^2 \\ \le & \left(2\left(\mathcal{L}(W_0) - \mathcal{L}(W^*) \right) + \frac{c}{L_f} \right) \frac{1}{\sqrt{T}} \\ \le & \left(2\left(\mathcal{L}(W_0) - \mathcal{L}(W^*) \right) + \frac{\rho \epsilon_s}{2} \right) \frac{1}{\sqrt{T}} \end{aligned}$$

Therefore, the convergence rate of MSPipe is $O(T^{-\frac{1}{2}})$, which maintains the same convergence rate as vanilla sampling-based GNN training methods ($O(T^{-\frac{1}{2}})$ [3, 6, 7]).

B MORE DISCUSSION ON THE RELATED WORK

As discussed before, the key design space of the memory-based TGNN model lies in memory updater and memory aggregator functions. JODIE [15] updates the memory using two mutually recursive RNNs and applies MLPs to predict the future representation of a node. Similar to JODIE, TGN [25] and APAN [38] use RNN as the memory update function while incorporating an attention mechanism to capture spatial and temporal information jointly. APAN further optimizes inference speed by using asynchronous propagation. A recent work TIGER [47] improves TGN by introducing an additional memory module that stores node embeddings and proposes a restarter for warm initialization of node representations.

Moreover, some researchers focus on optimizing the inference speed of MTGNN models: [51] propose a model-architecture co-design to reduce computation complexity and external memory access. TGOpt [39] leverages redundancies to accelerate inference of the temporal attention mechanism and the time encoder.

There are several static GNN training schemes with staleness techniques, PipeGCN [36] and Sancus [22], as we have discussed the difference in Section 6, we would like to emphasize and detail the difference between those works and MSPipe:

- **Dependencies and Staleness**: PipeGCN [36] and Sancus [22] aim to eliminate inter-layer dependencies in multi-layer GNN training to enable communication-computation overlap. In contrast, MSPipe is specifically designed to tackle temporal dependencies within the memory module of MTGNN training. The dependencies and staleness in MTGNN training pose unique challenges that require distinct theoretical analysis and system designs.
- **The choice of staleness bound**: Previous staleness-based static GNN methods randomly choose a staleness bound for acceleration, which may lead to suboptimal system performance and affect model accuracy. MSPipe strategically decides the minimal staleness bound that can reach the highest throughput without sacrificing the model accuracy.
- **Bottlenecks**: In full-graph training scenarios, such as PipeGCN [36] and Sancus [22], the main bottleneck lies in communication between graph partitions on GPUs. Due to limited GPU memory, the graph is divided into multiple parts, leading to increased communication time during full graph training. Therefore, these methods aim to optimize the communication-computation overlap to improve training throughput. In contrast, in MTGNN training, the main bottleneck stems from maintaining the memory module on the CPU and the associated challenges of updating and synchronizing it with CPU storage across multiple GPUs [50]. MSPipe focuses on addressing this specific bottleneck. Furthermore, unlike full graph training where the entire graph structure needs to be stored in the GPU, MTGNN adopts a sampling-based subgraph training approach. As a result, the communication overhead in MTGNN is significantly smaller than full graph training.

- **Training Paradigm and Computation Patterns**: PipeGCN [36] and Sancus [22] are tailored for full-graph training scenarios, which differ substantially from MTGNN training in terms of training paradigm, computation patterns, and communication patterns. MTGNNs typically involve sample-based subgraph training, which presents unique challenges and constraints not addressed by full graph training approaches. Therefore, the full graph training works cannot support MTGNN training.
- **Multi-Layer GNNs vs Single-Layer MTGNNs**: PipeGCN [36] and Sancus [22] lies on the assumption that the GNN have multiple layers (e.g., GCN [14], GAT [45]) and they break the dependencies among multiple layers to overlap communication with computation. While memory-based TGNNs only have one layer with a memory module [15, 23, 25, 38, 52], which makes their methods lose efficacy for MTGNNs.

C TRAINING TIME BREAKDOWN

C.1 Profiling setups

We use TGL [52], the SOTA MTGNN training framework, on a server equipped with 4 A100 GPUs for profiling, which is the same as the experiment testbed introduced in the section 5. The local batch size for the REDDIT, WIKI, MOOC, and LastFM datasets is set to 600, while for the GDELT dataset, it is set to 4000. All the breakdown statistics are averaged over 100 epochs. All these hyperparameters are the same as the experiments. We firmly believe that, by leveraging TGL's highly optimized performance, we can evaluate bottlenecks and areas for improvement, further justifying the need for our proposed MSPipe framework.

Table 5: Training time breakdown of JODIE model

Dataset	Sample	Fetch feature	Fetch memory	Train MTGNN	Update memory
REDDIT [15]	4.14%	8.05%	7.36%	50.11%	30.34%
WIKI [15]	2.20%	1.10%	4.95%	46.70%	45.05%
MOOC [15]	3.41%	1.02%	5.80%	51.05%	38.71%
LASTFM [15]	4.29%	1.14%	6.19%	44.95%	43.43%
GDELT [52]	3.25%	8.56%	9.34%	38.75%	40.11%

Table 6: Training time breakdown of APAN model

Dataset	Sample	Fetch feature	Fetch memory	Train MTGNN	Update memory
REDDIT [15]	12.94%	5.75%	15.18%	39.14%	27.00%
WIKI [15]	6.52%	0.87%	9.13%	42.61%	40.87%
MOOC [15]	10.60%	0.83%	8.32%	45.11%	35.14%
LASTFM [15]	11.12%	1.02%	12.26%	41.77%	33.83%
GDELT [52]	14.34%	3.25%	20.31%	23.95%	38.15%

C.2 Overlap the memory update stage with MTGNN training stage

We have identified an opportunity to overlap the execution of the memory update stage with the MTGNN training stage. Although we have implemented this overlapping, the memory update overhead remains significant, as reported in Table 1, 5, and 6. There are two main reasons for this:

1. The MTGNN training stage cannot fully overlap with the memory update stage due to the dependency on the memory updater for updating the memory within the MTGNN training stage, as discussed in Section 2. Additionally, the computational overhead of the memory updater may outweigh that of the embedding modules [15, 38]. Consequently, the available time for the memory update stage to overlap with the MTGNN training stage becomes further limited.

2. The MTGNN training process can be decomposed into three steps: the memory updater computes the updated memory, the MTGNN layer computes the embeddings, and the loss and backward steps are performed (including all-reduce). The latter two stages can indeed be parallelized with the memory update stage, which we have already implemented in our experiments, aligning with TGL [52]. However, even with these overlaps, the memory update stage still accounts for up to 31.7%, 45.0%, and 40.9% of the total time, as indicated in Table 1, 5, and 6 respectively, making it impossible to completely conceal the associated overhead.

C.3 Breakdown statistics of JODIE and APAN

We provide the training time breakdowns for the JODIE and APAN models in Table 5 and Table 6, which reveal that memory operations, including memory fetching and updating, can account for up to 50.51% and 58.56% of the total training time, respectively. Notably, the significant overhead is primarily due to memory operations rather than the sampling and feature fetching stages, which distinguishes these models from static GNN models and the systems designed for static GNN models.

C.4 GPU sampler analysis

MSPipe utilizes a GPU sampler for improved resource utilization and faster sampling and we further clarify the remarkable speedup mainly comes from our pipeline mechanism not the GPU sampler. As shown in Table 7, we conducted a detailed profiling of the sampling time using TGL and found that our sampler is 24.3% faster than TGL's CPU sampler for 1-hop most recent sampling, which accounts for only 3.6% of the total training time. Therefore, the performance gain is primarily attributed to our pipeline mechanism and resource-aware minimal staleness schedule but not to the acceleration of the sampler.

C.5 Why does the memory update stage take longer time than memory fetching?

The memory update takes a longer time for two reasons: **1)**In a multi-GPU environment, the memory module is stored in the CPU, allowing multiple GPUs to read simultaneously but not write simultaneously to ensure consistency and avoid conflicts; **2)** our memory fetching implementation, aligns with TGL, utilizes non-blocking memory copy APIs for efficient transfer of memory vectors from CPU to GPU with pinned memory. However, the lack of a non-blocking API equivalent for *tensor.cpu()* can impact performance.

Table 7: Detailed training time breakdown of TGN model to illustrate the effect of the GPU sampler.

Dataset	Framework	Avg Epoch(s)	Sample(s)	Fetch feature (s)	Fetch memory(s)	Train MTGNN(s)	Update memory(s)
REDDIT	TGL	7.31	0.69	0.92	0.42	3.43	1.85
	MSPipe-NoPipe	7.05	0.44	0.88	0.41	3.42	1.90
WIKI	TGL	2.41	0.16	0.14	0.14	1.24	0.73
	MSPipe-NoPipe	2.32	0.08	0.12	0.10	1.20	0.82
MOOC	TGL	4.31	0.42	0.13	0.11	2.29	1.37
	MSPipe-NoPipe	4.20	0.31	0.31	0.21	2.13	1.41
LASTFM	TGL	13.10	1.50	1.19	1.11	5.64	3.65
	MSPipe-NoPipe	12.64	1.04	1.20	1.05	6.12	3.23
GDELT	TGL	645.46	113.62	82.39	67.62	242.61	139.22
	MSPipe-NoPipe	626.09	94.26	85.20	69.21	240.99	136.43

D IMPLEMENTATION DETAILS

D.1 Algorithm details

We clarify that $\tau^{(j)}$ is the execution time of different stages, which can be collected in a few iterations of the profiling. The $\tau^{(j)}$ and the staleness k_i can be pre-calculated for all the graph data, which can be reused for future training. It's simple and efficient to do the profiling, pre-calculation, and training with our open-source code provided in the anonymous link.

In the case of stages such as the GNN computation stage, the execution time is likely to be dependent on the number of sampled nodes or edges. This quantity not only varies across different batches but also depends on the underlying graph structure. While the training time of a static GNN can differ due to varying numbers of neighbors for each node and the utilization of random sampling, memory-based TGNNs typically employ a fixed-size neighbor sampling approach using the most recent temporal sampler. Specifically, the sampler selects a fixed number of the most recently observed neighbors to construct the subgraph. Consequently, as the timestamp increases, the number of neighboring nodes grows, and it becomes more stable, governed by the maximum number of neighbors per node constraint. Through our profiling analysis, we observed that the number of nodes in the subgraph converges after approximately 10-20 iterations, allowing the average execution time to effectively represent the true execution time.

D.2 Multi-GPU server implementation

We have provided a brief description of how MSPipe works in multi-GPU servers at Section 2 and Section 3.1 and we have provided the implementation with the anonymous link in the abstract. We will give you a more detailed analysis of the implementation details here: The graph storage is implemented with NVIDIA UVA so each GPU worker retrieves a local batch of events and performs the sampling process on GPU to generate sub-graphs. The memory module is stored in the CPU's main memory without replication to ensure consistency and exhibit the ability to store large graphs. Noted that, except for the GPU sample, the other stages align with TGL. Here is a step-by-step overview:

(1) Each GPU worker retrieves a local batch of events and performs the sampling process on the GPU to generate sub-graphs.
(2) Fetches the required features and node memory vectors from the CPU to the GPU for the subgraphs.
(3) Performs MTGNN forward and backward computations on each GPU. MSPipe implements Data Parallel training similar to TGL.
(4) The memory module is stored in the CPU's main memory without replication to ensure consistency. Each GPU transfers the updated memory vectors to the CPU and updates the corresponding elements, which ensures that the memory module remains consistent across all GPUs.

D.3 Stall-free minimal staleness scheduling

We propose a resource-aware online scheduling algorithm to decide the starting time of stages in each training iteration, as given in Algorithm 1

Algorithm 1 Online Scheduling for MTGNN training pipeline

```
Input: $E$ batches of events $\mathcal{B}_i$, Graph $\mathcal{G}$, minimum staleness iteration number $k_i$
Global: $i_{\text{upd}} \leftarrow 0$ ▷ the latest iteration whose memory update is done
for $i \in 1, 2, ..., E$ in parallel do
    if $lock(sample\_lock)$ then
        $\mathcal{G}_{\text{sub}} \leftarrow Sample(\mathcal{G}, \mathcal{B}_i)$ ▷ sample subgraph $\mathcal{G}_{\text{sub}}$ using a batch of events
    if $lock(feature\_lock \ \& \ pcie\_lock)$ then
        $fetch\_feature(\mathcal{G}_{\text{sub}})$ ▷ feature fetching for the subgraphs
    if $lock(memory\_lock \ \& \ pcie\_lock)$ then
        while $i - i_{\text{upd}} > k_i$ do
            $wait()$ ▷ delay memory fetching until staleness iteration number is smaller than $k_i$
        $fetch\_memory(\mathcal{G}_{\text{sub}})$ ▷ transfer memory vectors for the subgraphs
    if $lock(gnn\_lock)$ then
        $MTGNN(\mathcal{G}_{\text{sub}})$ ▷ train the MTGNN model using the subgraphs
    if $lock(update\_lock)$ then
        $update\_mem(\mathcal{G}_{\text{sub}}, \mathcal{B}_i)$ ▷ generate new memory vectors and write back to CPU storage
        $i_{\text{upd}} \leftarrow i$ ▷ update the last iteration with memory update done
```

To enable asynchronous and parallel execution of the stages, we utilize a thread pool and a CUDA stream pool. Each batch of data is assigned an exclusive thread and stream from the respective pools, enabling concurrent processing of multiple batches. Dedicated locks for each stage are used to resolve resource contention and enforce sequential execution (Equation 3). Figure 7 provides a schematic illustration of our online scheduling. The schedule of the memory fetching stage ensures the minimal staleness iteration requirement (Lines 8-11). As illustrated in Figure 7, the scheduling effectively fills the bubble time while minimizing staleness and avoiding resource competence. At the end of each training iteration, new memory vectors are generated based on the staled historical memories and events in the current batch (Line 15). Finally, the latest iteration whose memory update stage has been completed is recorded, enabling other parallel threads that run other training iterations to track (Line 16). Note that the first few iterations before iteration k will act as a warmup, which means they will not wait for the memory update k iterations before.

E FULL EXPERIMENTS

We first provide the details of the experiments and discuss the experiment setting. Then we provide the full version of the experiment results, including the accuracy and throughput speedup, the convergence of the JODIE and APAN model, the distribution of Δt in remaining datasets, and the analysis of the node memory similarity.

E.1 Details of the Experiments

Datasets. This paper employs several datasets, each with its unique properties and characteristics. The Reddit dataset captures the posting behavior of users on subreddits over one month, and the link feature is extracted through the conversion of post text into a feature vector. The Wikipedia dataset records the editing behavior of users on Wikipedia pages over a month, and the link feature is extracted through the conversion of the edit text into a 172-dimensional Linguistic Inquiry and Word Count (LIWC) feature vector. The MOOC dataset captures the online learning behavior of students in a MOOC course while the LastFM dataset contains information about which songs were listened to by which users over one month. The GDELT dataset is a Temporal Knowledge Graph that records global events in multiple languages every 15 minutes, which covers events from 2016 to 2020 and consists of homogeneous dynamic graphs with nodes representing actors and temporal edges representing point-time events. Furthermore, it is important to highlight that *TGNN training employs graph edges as training samples*, in contrast to static GNN training, which utilizes nodes as training samples. All the datasets are downloaded from the link in TGL [52] repository.

E.2 Full version of the Experiment Results

E.2.1 The superior AP in LastFM. The reasons why our staleness mitigation strategy outperforms the AP of the baseline TGL in the LastFM dataset is due to the unique characteristics of the LastFM datasets:

- The LastFM dataset exhibits a larger average time gap ($\frac{t_{max}-t_{min}}{E}$, where t_{max} and t_{min})represent the largest and smallest timestamps, respectively, and E denotes the number of events) compared to other datasets, as discussed by [8]. Specifically, LastFM has an average time gap of 106, whereas Reddit's average time gap is 4, Wiki's average time gap is 17, MOOC's average time gap is 3.6, and GDELT's average time gap is 0.1.
- Consequently, even without staleness in the baseline method, the node memory in the LastFM graph tends to become significantly outdated [25], as discussed in Section 3.3. Our staleness mitigation strategy eliminates the outdated node representation by aggregating the memories of the recently active nodes with the highest similarity. This approach helps mitigate the impact of the large time gap present in LastFM datasets, ultimately leading to an improvement in AP compared to the baseline methods.

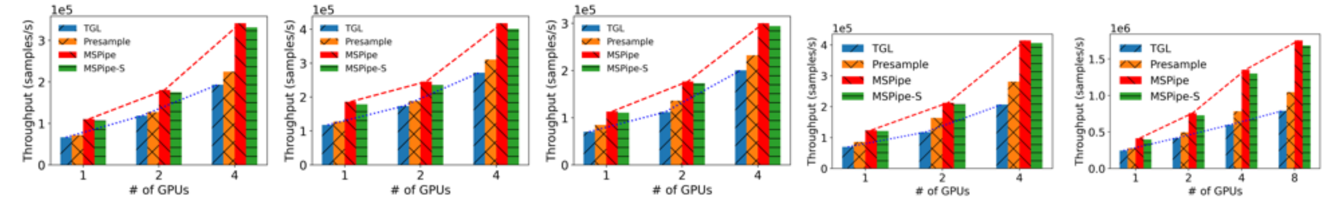
E.2.2 Scalability results on all datasets. We further provide the full scalability results discussed in Section 5.2. We show the training throughput with different numbers of GPUs of TGN models on five datasets in Figure 16. MSPipe not only achieves consistent speed-up but also demonstrates remarkable scaling efficiency, reaching up to 83.6% on a single machine. Scaling efficiency is computed as the ratio of the speed-up achieved by utilizing 4 GPUs to the ideal speed-up. These results surpass those of other baseline methods. Furthermore, when scaling TGN training on GDELT to two machines equipped with eight GPUs (as shown in Figure 16(e)), MSPipe continues to outperform the baselines and exhibits superior scalability, even without explicit optimization for inter-machine communication.

E.2.3 Convergence of the TGN, JODIE and APAN. We further provide the full results discussed in Section 5.3. We show the convergence of TGN, JODIE, and APAN models on five datasets in Figure 17, Figure 18 and Figure 19. We can see that the training curves of all models largely overlap with the baselines (TGL and Presample), demonstrating that MSPipe preserves the convergence rate. Notably, MSPipe-S achieves better performance than the other variants on the WIKI and LastFM datasets.

E.2.4 Comparison between different staleness bound. Furthermore, we present a comprehensive comparison of various staleness bounds across multiple datasets including REDDIT, WIKI, LASTFM, and GDELT, using the TGN model, in order to validate the efficacy of MSPipe. The results consistently demonstrate that MSPipe outperforms other staleness-bound options in terms of both throughput and accuracy across all datasets. As shown in Fig 23, the number of staleness k_i will soon converge to a steady minimal staleness value. To represent this minimal staleness bound, we utilize a fixed value that corresponds to the steady state. This choice allows us to showcase the minimal staleness bound effectively.

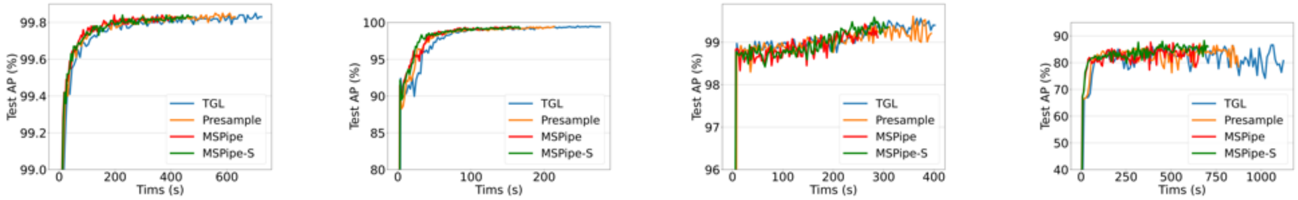
E.2.5 The distribution of Δt on other datasets. We introduce Δt as the duration since a node v's memory was last updated, which differs from the Δt in the MTGNN inference system [39, 51]. The Δt defined in TGOpt [39] and Zhou *et al.* [51] are designed for the time-encoder, which is computed by the difference between current events' timestamp and their historical events' timestamps with their neighbors. We further post the distribution of Δt of the remaining datasets in Figure 21 and observed that the Δt in all datasets follow the power-law distribution, indicating that most Δt values are small and that most node memories are not stale or constant. This observation provides insights into the occurrence patterns of nodes in different dynamic graphs. Our similarity-based staleness mitigation mechanism focuses on compensating for memory vectors with stale Δt values in the long tail of the distributions.

E.2.6 Analysis of the node memory similarity. We compensate the stale node memory by finding their most similar and recently active



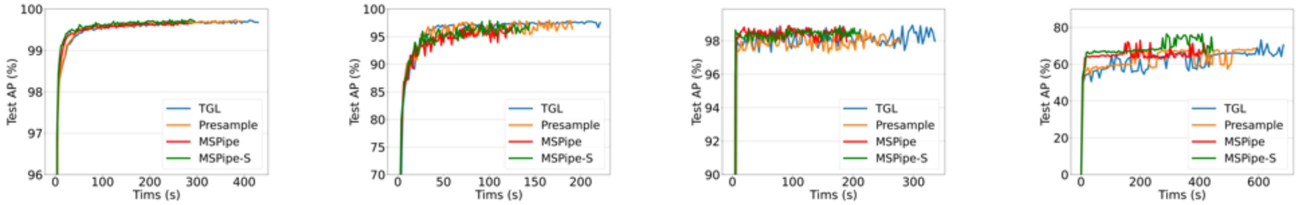
(a) REDDIT (b) WIKI (c) MOOC (d) LASTFM (e) GDELT

Figure 16: Scalability of training TGN.



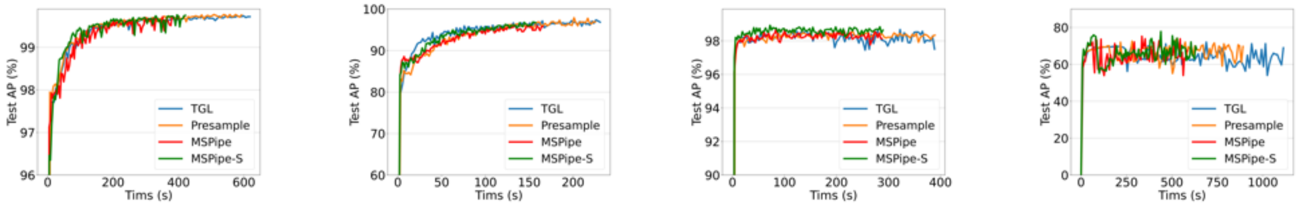
(a) REDDIT (b) WIKI (c) MOOC (d) LASTFM

Figure 17: Convergence of TGN training. x-axis is the wall-clock training time, and y-axis is the test average precision.



(a) REDDIT (b) WIKI (c) MOOC (d) LASTFM

Figure 18: Convergence of JODIE training. the x-axis is the wall-clock training time, and the y-axis is the test average precision



(a) REDDIT (b) WIKI (c) MOOC (d) LASTFM

Figure 19: Convergence of APAN training. the x-axis is the wall-clock training time, and the y-axis is the test average pricision

nodes with the intuition that similar nodes have resembling representations that facilitate the stale node to obtain more updated information. The most similar nodes are computed by counting their common neighbors to get Jaccard similarity. As illustrated in Figure 22, our mechanism for identifying the most recent similar nodes can locate those with representations that are not only similar but also more recently updated than randomly selected nodes. We use cosine similarity as the evaluation metric for similarity.

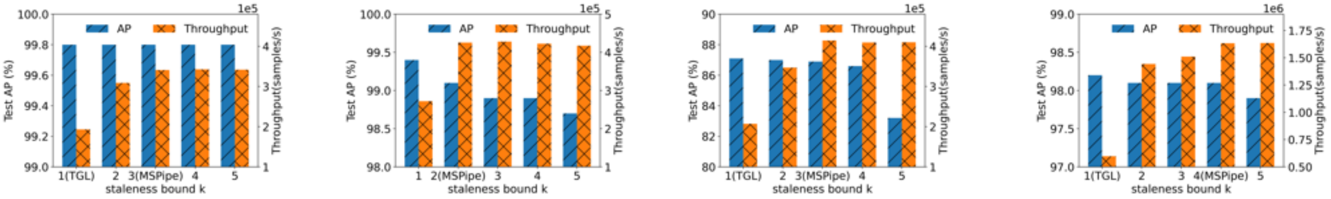
E.3 The variance of k_i with respect to i

We further evaluate the variance of k_i when the i changes. As shown in Figure 23, the number of staleness k_i will soon converge to a steadily minimal staleness value. This is because of the periodic manner of the MTGNN training as the computation time of different training stages is quite steady.

E.4 Batch size sensitivity analysis

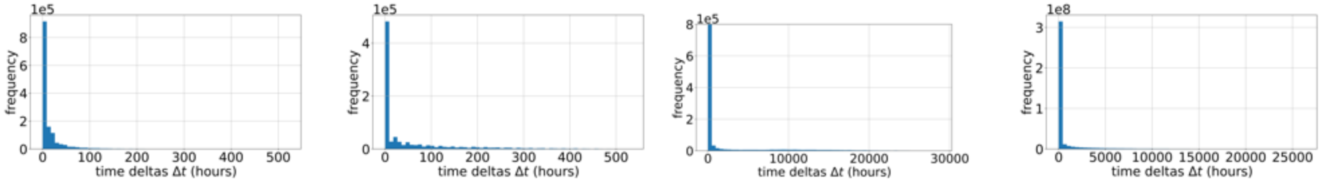
To further validate the effectiveness of MSPipe in different batch sizes, we conducted batch size sensitivity evaluations using the following local batch sizes: 300, 900, 1200, and 1600 for the small datasets, and 2000, 6000, and 8000 for the large dataset (used 600 and 4000 in the original experiments), illustrated in Table 8.

As demonstrated in Table 8, MSPipe consistently outperforms all baseline methods in varying batch sizes, achieving up to 2.01×



(a) REDDIT (b) WIKI (c) LASTFM (d) GDELT

Figure 20: Staleness error comparison on TGN. MOOC and GDELT datasets are presented in Figure 11



(a) REDDIT (b) MOOC (c) LASTFM (d) GDELT

Figure 21: Distribution of Δt on different datasets. The WIKI dataset is presented in Figure 8

Table 8: Batch size sensitive analysis. The best results are in bold, and the second-best are underlined.

Batch size	Scheme	REDDIT		WIKI		MOOC		LASTFM		GDELT	
		AP(%)	Speedup	AP(%)	Speedup	AP(%)	Speedup	AP(%)	Speedup	AP(%)	Speedup
Batch 300 (2000 for GDELT)	TGL	**99.8**	1×	**99.5**	1×	**99.4**	1×	**88.1**	1×	**98.5**	1×
	Presample	**99.8**	1.26×	**99.5**	1.08×	**99.4**	1.04×	88.0	1.51×	**98.5**	1.12×
	MSPipe	**99.8**	**1.73**×	99.4	**1.67**×	**99.4**	**1.47**×	87.2	**1.90**×	98.2	**1.93**×
	MSPipe-S	**99.8**	1.68×	**99.5**	1.65×	**99.4**	1.45×	88.0	1.86×	**98.5**	1.88×
Batch 900 (6000 for GDELT)	TGL	**99.8**	1×	**98.9**	1×	**98.6**	1×	86.9	1×	97.8	1×
	Presample	**99.8**	1.10×	**98.9**	1.12×	**98.6**	1.10×	86.9	1.37×	97.8	1.26×
	MSPipe	**99.8**	**1.62**×	98.5	**1.49**×	**98.6**	**1.58**×	86.7	**1.87**×	97.7	**2.01**×
	MSPipe-S	**99.8**	1.56×	**98.9**	1.46×	**98.6**	1.53×	**87.8**	1.80×	**98.2**	1.93×
Batch 1200 (8000 for GDELT)	TGL	**99.8**	1×	**98.5**	1×	98.3	1×	85.8	1×	97.1	1×
	Presample	**99.8**	1.34×	**98.5**	1.37×	98.3	1.32×	85.8	1.56	97.1	1.28×
	MSPipe	**99.8**	**1.64**×	**98.5**	**1.48**×	98.3	**1.69**×	85.8	**1.92**×	97.1	**1.99**×
	MSPipe-S	**99.8**	1.59×	**98.5**	1.45×	**98.8**	1.62×	**86.2**	1.84×	**98.1**	1.90×
Batch 1600	TGL	**99.8**	1×	**98.4**	1×	97.9	1×	**84.4**	1×		
	Presample	**99.8**	1.38×	**98.4**	1.39×	97.9	1.33×	**84.4**	1.51×		
	MSPipe	**99.8**	**1.66**×	98.1	**1.58**×	97.9	**1.71**×	82.7	**1.97**×		
	MSPipe-S	**99.8**	1.58×	98.3	1.53×	**98.7**	1.64×	84.2	1.88×		

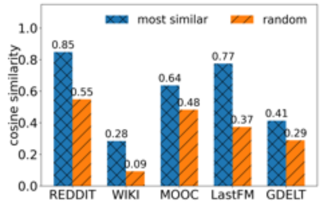


Figure 22: The cosine similarity of the memory vectors between the target nodes (with staled node memory) and their most similar nodes or random nodes

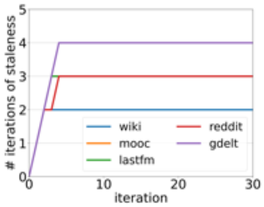


Figure 23: The minimal number of staleness k_i in different iteration i

speedup without compromising model accuracy. These results further validate the practicality of MSPipe. It is worth noting that for the same dataset, MSPipe tends to exhibit similar speedup among various batch sizes, indicating no direct correlation between batch size and speedup.

E.5 Compare with Strawman method: increase batch size

We conducted additional empirical comparisons between MSPipe and baseline methods using larger batch sizes. In Table 9, MSPipe consistently outperforms baseline methods with batch sizes increased by 1.5× and 2×, achieving speedups of up to 57% and 32% respectively. While the TGN model experiences up to 1.4% accuracy loss with larger batch sizes, MSPipe maintains high accuracy with a maximum accuracy loss of 0.3%. It is worth emphasizing

Table 9: MSPipe compares with baseline methods using larger batch size.

Scheme	REDDIT			WIKI			MOOC			LastFM		
	AP(%)	Time(s)	Speedup	AP(%)	Time(s)	Speedup	AP(%)	Time(s)	Speedup	AP(%)	Time(s)	Speedup
TGL batch 600	99.8	7.31	1×	**99.4**	2.41	1×	**99.4**	4.31	1×	**87.2**	13.10	1×
MSPipe batch 600	99.8	**4.14**	**1.77×**	99.1	**1.57**	**1.54×**	99.3	**2.88**	**1.50×**	86.9	**6.55**	**1.87×**
TGL batch 900	99.8	5.22	1.40×	98.9	2.03	1.19×	98.7	3.18	1.36×	86.9	10.10	1.30×
TGL batch 1200	99.8	4.48	1.63×	98.5	1.83	1.32×	98.3	2.99	1.44×	85.8	8.43	1.55×

Table 11: Additional memory overhead of APAN when applying staleness.

Overhead\Dataset	REDDIT	WIKI	MOOC	LastFM	GDELT
Addition	50.1MB	32.7MB	43.5MB	55.2MB	1.28GB
Upperbound	51.4MB	34.3MB	44.3MB	44.3MB	1.35GB
GPU Mem (40GB) portion	0.12%	0.08%	0.11%	0.14%	3.20%

Table 10: Additional memory overhead of JODIE when applying stalenes.

Overhead\Dataset	REDDIT	WIKI	MOOC	LastFM	GDELT
Addition	54.6MB	42.6MB	43.9MB	47.7MB	0.98GB
Upperbound	51.4MB	34.3MB	44.3MB	44.3MB	1.35GB
GPU Mem (40GB) portion	0.14%	0.11%	0.11%	0.12%	2.45%

that MSPipe can be applied with larger batch sizes to further boost training throughput as shown in Table 8.

E.6 Memory overhead analysis.

In MSPipe, we introduce staleness within the memory module to facilitate the pre-fetching of features and memory in subsequent iterations. However, unlike other asynchronous training frameworks [4, 17, 22, 36], where staleness is introduced during DNN or GNN parameter learning, our MTGNN training stage does not incorporate staleness. Each subgraph is executed sequentially, resulting in no additional hidden states during MTGNN computation.

Consequently, the additional memory consumption in MSPipe arises from the prefetched subgraph, which includes node/edge features and memory vectors. We can compute an upper bound for this memory consumption as follows:

Let the subgraph in each iteration have a batch size of B, node feature dimension of H_n, edge feature dimensions of H_e, node memory dimension of M, and an introduced staleness bound of K. During subgraph sampling, we use the maximum neighbor size of $\mathcal{N}$ (e.g. 10) to compute the memory consumption, which represents an upper bound. Within each subgraph, we have three nodes per sample, comprising a source node, destination node, and neg_sample node. Hence, a single subgraph contains a total of $3B(\mathcal{N}+1)$ nodes, where $\mathcal{N}+1$ denotes the number of neighbors per node, inclusive of the target node itself. Additionally, each graph event involves both positive and negative links, resulting in two edges per event. Consequently, the total number of links per subgraph amounts to $2B(\mathcal{N}+1)$. The memory utilization of a subgraph encompasses node IDs, edge IDs, node features, edge features, and node memory states. With the introduction of a staleness bound of K, the GPU accommodates a maximum of K additional subgraphs. Assuming a data format of Float32 (i.e., 4 bytes), the additional memory consumption for these subgraphs can be formulated as:

$$\begin{aligned}
&4 \times K \times [3B(\mathcal{N}+1)H_n + 2B(\mathcal{N}+1)H_e \\
&\qquad + 3B(\mathcal{N}+1)M + 3B(\mathcal{N}+1) + 2B(\mathcal{N}+1)] \\
&= 4 \times K \times 3B(\mathcal{N}+1)(H_n + \frac{2}{3}H_e + M + \frac{5}{3}) \\
&= 12KB(\mathcal{N}+1)(H_n + \frac{2}{3}H_e + M + \frac{5}{3})
\end{aligned}$$

Moreover, we conduct empirical experiments on all the models/datasets with the *torch.cuda.memory_summary()* API. As observed in Table4,10 and11, the additional memory usage from MSPipe strictly resembles to our analyzed upper bound. Additionally, we compare the additional memory cost with the GPU memory size, demonstrating that the additional memory overhead is a relatively small proportion (up to 3.20% for APAN) of the modern GPU's capacity.